Rust Crash Course

The engineering team at FP Complete has people with a variety of backgrounds, but mostly Haskell and DevOps. Over the past number of years, a number of team members started to experiment off-and-on with Rust as well, and started to share those experiences. As we began using Rust for some of our work, and as interest on the team kept going up, we decided to launch a training program on it.

For a distributed team like ours, we rely heavily on writing and text communication instead of live meetings. Therefore, we decided to approach the Rust training as a series of written lessons that could be consumed by the team when convenient, and follow up with questions on Slack and weekly sync-up calls.

We decided to go the route of our own training material to give focus to the topics that most affected our work, and which seemed to be particular pain points for our team.

Then we made one more decision: to actually publish this material for the rest of the world to consume. Throughout 2018, and a bit more in 2019, I published a series of posts on my personal blog which I called the Rust Crash Course.

Fast forward to 2019 and 2020. The material seems to overall be holding up and is still a useful reference for experienced programmers wanting to pick up Rust quickly. So we've decided to update the material to the latest Rust release and republish as an ebook. We hope that you find this format helpful and the material useful.

Interested to learn more about FP Complete and Rust? Check out our Rust home page.

Table of Contents

1. Kick the Tires . . . 4
    1.1. Tooling . . . 4
    1.2. Hello, world! . . . 5
    1.3. Macros . . . 6
    1.4. Traits and Display . . . 7
    1.5. Semicolons . . . 10
    1.6. Numeric types . . . 12
    1.7. Printing numbers . . . 14
2. Basics of Ownership . . . 19
    2.1. Format . . . 19
    2.2. Comparison with Haskell . . . 19
    2.3. Simple example . . . 20
    2.4. Dropping . . . 21

2.5. Lexical scoping . . . 22
2.6. Borrows/references (immutable) . . . 23
2.7. Multiple live references . . . 25
2.8. Challenge . . . 27
2.9. Mutable reference vs mutable variable . . . 28
2.10. Copy trait . . . 30
2.11. Lifetimes . . . 32
2.12. Notes on structs and enums . . . 33
2.13. Bouncy . . . 34
3. Iterators and Errors . . . 43
3.1. Command line arguments . . . 43
3.2. The extra datatype pattern . . . 44
3.3. CLI args via iterators . . . 45
3.4. Skipping . . . 47
3.5. Parsing integers . . . 49
3.6. Parse our command line . . . 50
3.7. Question mark . . . 54
3.8. Forgotten question marks . . . 56
3.9. Updating bouncy . . . 61
3.10. Use it! . . . 62
3.11. Mismatched types . . . 64
3.12. Public and private . . . 64
3.13. Exit code . . . 65
3.14. Better error handling . . . 67
3.15. Next time . . . 67
4. Crates and more iterators . . . 67
4.1. Finding a crate . . . 67
4.2. Starting a project . . . 68
4.3. Adding the crate . . . 68
4.4. Library docs . . . 69
4.5. Getting the frame size . . . 69
4.6. Carriage return and the border . . . 71
4.7. Double buffering . . . 71
4.8. More iterators! . . . 72
4.9. Mutable state . . . 75
4.10. Iterator adapters . . . 79
4.11. Alternative syntax: where . . . 83
4.12. Not just u32 . . . 83
4.13. Recap . . . 86
4.14. More idiomatic . . . 86
4.15. Collecting results . . . 87

4.16. Next Chapter . . . 88
5. Rule of Three - Parameters, Iterators, and Closures . . . 89
5.1. Types of parameters . . . 89
5.2. Iterators . . . 94
5.3. Closures . . . 100
5.4. The type of a closure . . . 102
5.5. Anonymous types . . . 103
5.6. Mutable variables . . . 106
5.7. Multiple traits? . . . 109
5.8. The rule of three? . . . 109
5.9. The third function trait . . . 110
5.10. Further function subtyping . . . 112
5.11. The move keyword . . . 112
5.12. Reluctant Rust . . . 116
5.13. Recap: ownership, capture, and usage . . . 116
5.14. Which trait to use? . . . 121
5.15. Summary of the rule of three for closures . . . 121
5.16. GUIs and callbacks . . . 122
5.17. Replacing the callback . . . 124
5.18. Share the file . . . 125
5.19. Move it . . . 127
5.20. Reference counting (hint: nope) . . . 127
5.21. RefCell . . . 128
5.22. Fearless concurrency! . . . 132
5.23. Next Chapter . . . 137
6. Lifetimes and Slices . . . 138
6.1. Printing a person . . . 138
6.2. Birthday! . . . 140
6.3. The single iterator . . . 143
6.4. Swap . . . 146
6.5. replace and take . . . 147
6.6. Lifetimes . . . 147
6.7. Requirement for multiple lifetime parameters . . . 151
6.8. Lifetime elision . . . 153
6.9. Static lifetime . . . 153
6.10. Arrays, slices, vectors, and String . . . 153
6.11. Deref . . . 157
6.12. Using slices . . . 158
6.13. Byte literals . . . 158
6.14. Strings . . . 159
6.15. Lifetimes in data structures . . . 160

6.16. References and slices in APIs . . . 161
7. Async, futures, and tokio . . . 161
8. Down and dirty with Future . . . 161
8.1. Sleepus Interruptus . . . 162
8.2. Introducing `async` . . . 164
8.3. `async` functions . . . 165
8.4. `.await` a minute . . . 167
8.5. Dropping `async` block . . . 168
8.6. Implement our own `Future` . . . 169
8.7. The third `async` difference . . . 172
8.8. SleepPrint . . . 173
8.9. TwoFutures . . . 175
8.10. AndThen . . . 177
8.11. `main` attribute . . . 178
8.12. Cooperative concurrency . . . 179
8.13. Summary . . . 179
8.14. Exercises . . . 180
9. Tokio 0.2 . . . 182
9.1. Hello Tokio! . . . 183
9.2. Spawning processes . . . 185
9.3. Take a break . . . 186
9.4. Time to spawn . . . 187
9.5. Synchronous code . . . 189
9.6. Let's network! . . . 190
9.7. TCP client and ownership . . . 192
9.8. Playing with `lines` . . . 194
9.9. LocalSet and `!Send` . . . 197
9.10. Conclusion . . . 200

1. Kick the Tires

In this lesson, we just want to get set up with the basics: tooling, ability to compile, basic syntax, etc. Let's start off with the tooling, you can keep reading while things download.

1.1. Tooling

Your gateway drug to Rust will be the `rustup` tool, which will install and manage your Rust toolchains. I put that in the plural, because it can manage both multiple versions of the Rust compiler, as well as cross compilers for alternative targets. For now, we'll be doing simple stuff.

- Installation page on rust-lang.org
- rustup.rs page

Both of these pages will tell you to do the same thing:

- On Unix-like systems, run `curl https://sh.rustup.rs -sSf | sh`
- Or run a Windows installer, probably the 64-bit installer

Read the instructions on the rust-lang page about setting up your `PATH` environment variable. For Unix-like systems, you'll need `~/.cargo/bin` in your `PATH`.

Along with the `rustup` executable, you'll also get:

- `cargo`, the build tool for Rust
- `rustc`, the Rust compiler

1.2. Hello, world!

Alright, this part's easy: `cargo new hello && cd hello && cargo run`.

We're *not* learning all about Cargo right now, but to give you the basics:

- `Cargo.toml` contains the metadata on your project, including dependencies. We won't be using dependencies quite yet, so the defaults will be fine.
- `Cargo.lock` is generated by `cargo` itself
- `src` contains your source files, for now just `src/main.rs`
- `target` contains generated files

We'll get to the source code itself in a bit, first a few more tooling comments.

1.2.1. Building with rustc

For something this simple, you don't need `cargo` to do the building. Instead, you can just use: `rustc src/main.rs && ./main`. If you feel like experimenting with code this way, go for it. But typically, it's a better idea to create a scratch project with `cargo new` and experiment in there. Entirely your decision.

1.2.2. Choosing the toolchain

As I mentioned, `rustup` can maintain multiple toolchains. You can use the `rustup show` command to find out which toolchains are installed, and which are active. You can change the default for your entire account with something like:

```
rustup default 1.40.0
```

You can also force a specific project to use a specific toolchain with a `rust-toolchain` file. For example:

```
$ cat rust-toolchain
1.40.0
```

1.2.3. Running tests

We won't be adding any tests to our code yet, but you can run tests in your code with `cargo test`.

1.2.4. Extra tools

Two useful utilities are the `rustfmt` tool (for automatically formatting your code) and `clippy` (for getting code advice). To get them set up, run:

```
$ rustup component add clippy rustfmt
```

And then you can run them with:

```
$ cargo fmt
$ cargo clippy
```

1.2.5. IDE

There is some IDE support for those who want it. I've heard great things about IntelliJ IDEA's Rust add-on. Personally, I haven't used it much yet, but I'm also not much of an IDE user in the first place. This crash course won't assume any IDE, just basic text editor support.

1.3. Macros

Alright, we can finally look out our source code in `src/main.rs`:

```
fn main() {
    println!("Hello, world!");
}
```

Simple enough. `fn` says we're writing a function. The name is `main`. It takes no arguments, and has no return value. (Or, more accurately, it returns the *unit type*, which is kind of like void in C/C++, but really closer to the unit type in Haskell.) String literals look pretty normal, and function calls look almost identical to other C-style languages.

Alright, here's the first "crash course" part of this: why is there an exclamation point after the `println`? I say "crash course" because when I first learned Rust, I didn't see an explanation of this, and it bothered me for a while.

`println` is *not* a function. It's a macro. This is because it takes a *format string*, which needs to be checked at compile time. To prove the point, try changing the string literal to include `{}`. You'll get

an error message along the lines of:

```
error: 1 positional argument in format string, but no arguments were given
```

This can be fixed by providing an argument to fill into the placeholder:

```
println!("Hello , world! {} {} {}", 5, true, "foobar");
```

Take a guess at what the output will be, and you'll probably be right. But that leaves us with a question: how does the `println!` macro know how to display these different types?

1.4. Traits and Display

More crash course time! To get a better idea of how displaying works, let's trigger a compile time error. To do this, we're going to define a new data type called `Person`, create a value of that type, and try to print it:

```
struct Person {
    name: String,
    age: u32,
}

fn main() {
    let alice = Person {
        name: String::from("Alice"),
        age: 30,
    };
    println!("Person: {}", alice);
}
```

We'll get into more examples on defining your own `struct`s and `enum`s later, but you can cheat and read the Rust book if you're curious.

If you try to compile that, you'll get:

```
error[E0277]: `Person` doesn't implement `std::fmt::Display`
  --> src/main.rs:11:28
   |
11 |     println!("Person: {}", alice);
   |                            ^^^^^ `Person` cannot be formatted with the default
formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Person`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print)
instead
   = note: required by `std::fmt::Display::fmt`
```

That's a bit verbose, but the important bit is `the trait `std::fmt::Display` is not implemented for `Person``. In Rust, a *trait* is similar to an interface in Java, or even better like a typeclass in Haskell. (Noticing a pattern of things being similar to Haskell concepts? Yeah, I did too.)

We'll get to all of the fun of defining our own traits, and learning about implementing them later. But we're crashing forward right now. So let's throw in an implementation of the trait right here:

```
impl Display for Person {
}
```

That didn't work:

```
error[E0405]: cannot find trait `Display` in this scope
 --> src/main.rs:6:6
  |
6 | impl Display for Person {
  |      ^^^^^^^ not found in this scope
help: possible candidates are found in other modules, you can import them into scope
  |
1 | use core::fmt::Display;
  |
1 | use std::fmt::Display;
  |

error: aborting due to previous error

For more information about this error, try `rustc --explain E0405`.
error: Could not compile `foo`.
```

We haven't imported `Display` into the local namespace. The compiler helpfully recommends two different traits that we may want, and tells us that we can use the `use` statement to import them into the local namespace. We saw in an earlier error message that we wanted `std::fmt::Display`, so adding `use std::fmt::Display;` to the top of `src/main.rs` will fix this error message. But just to prove the point, no `use` statement is necessary! We can instead use:

```
impl std::fmt::Display for Person {
}
```

Awesome, our previous error message has been replaced with something else:

```
error[E0046]: not all trait items implemented, missing: `fmt`
 --> src/main.rs:6:1
  |
6 | impl std::fmt::Display for Person {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ missing `fmt` in implementation
  |
  = note: `fmt` from trait: `fn(&Self, &mut std::fmt::Formatter<'_>) ->
std::result::Result<(), std::fmt::Error>`
```

We're quickly approaching the limit of things we're going to cover in a "kicking the tires" lesson. But hopefully this will help us plant some seeds for next time.

The error message is telling us that we need to include a `fmt` method in our implementation of the `Display` trait. It's also telling us what the type signature of this is going to be. Let's look at that signature, or at least what the error message says:

```
fn(&Self, &mut std::fmt::Formatter<'_>) -> std::result::Result<(), std::fmt::Error>
```

There's a lot to unpack there. I'm going to apply terminology to each bit, but you shouldn't expect to fully grok this yet.

- `Self` is the type of the thing getting the implementation. In this case, that's `Person`.
- Adding the `&` at the beginning makes it a *reference* to the value, not the value itself. C++ developers are used to that concept already. Many other languages talk about pass by reference too. In Rust, this plays in quite a bit with *ownership*. Ownership is a massively important topic in Rust, and we're not going to discuss it more now.
- `&mut` is a *mutable reference*. By default, everything in Rust is immutable, and you have to explicitly say that things are mutable. We'll later get into why mutability of references is important to ownership in Rust.
- Anyway, the second argument is a mutable reference to a `Formatter`. What's the `<'_>` thing after `Formatter`? That's a *lifetime parameter*. That *also* has to do with ownership. We'll get to lifetimes later as well.
- The `->` indicates that we're providing the return type of the function.
- `Result` is an `enum`, which is a *sum type*, or a *tagged union*. It's generic on two *type parameters*: the value in case of success and the value in case of error.
- In the case of success, our function returns a `()`, or unit value. This is another way of saying "I don't return any useful value if things go well." In the case of an error, we return `std::fmt::Error`.

- Rust has no runtime exceptions. Instead, when something goes wrong, you return it explicitly. Almost all code uses the `Result` type to track things going wrong. This is more explicit than exception-based languages. But unlike languages like C, where it's easy to forget to check the type of a return to see if it succeeded, or tedious to do error handling properly, Rust makes this much less painful. We'll deal with it later.

  NOTE Rust *does* have the concept of panics, which in practice behave similarly to runtime exceptions. However, there are two important differences. Firstly, by convention, code is not supposed to use the panic mechanism for signaling normal error conditions (like file not found), and instead reserve panics for completely unexpected failures (like logic errors). Secondly, panics are (mostly) unrecoverable, meaning they take down the current thread.

Awesome, that type signature all on its own gave us enough material for about 5 more lessons! Don't worry, you'll be able to write some Rust code without understanding all of those details, as we'll demonstrate in the rest of this lesson. But if you're really adventurous, feel free to explore the Rust book for more information.

1.5. Semicolons

Let's get back to our code, and actually implement our `fmt` method:

```
struct Person {
    name: String,
    age: u32,
}

impl std::fmt::Display for Person {
    fn fmt(&self, fmt: &mut std::fmt::Formatter) -> Result<(), std::fmt::Error> {
        write!(fmt, "{} ({} years old)", self.name, self.age)
    }
}

fn main() {
    let alice = Person {
        name: String::from("Alice"),
        age: 30,
    };
    println!("Person: {}", alice);
}
```

We're using the `write!` macro to write content into the `Formatter` provided to our method. Using a `Formatter` allows for more efficient construction of values and production of I/O than producing a bunch of intermediate strings. Yay efficiency.

The `&self` parameter of the method is a special way of saying "this is a method that works on this object." This is quite similar to how you'd write code in Python, though in Rust you have to deal with pass by value vs pass by reference.

The second parameter is named `fmt`, and `&mut Formatter` is its type.

The very observant among you may have noticed that, above, the error message mentioned `&Self`. In our implementation, however, we made a lower `&self`. The difference is that `&Self` refers to the *type* of the value, and the lower case `&self` is the value itself. In fact, the `&self` parameter syntax is basically sugar for `self: &Self`.

Does anyone notice something missing? You may think I made a typo. Where's the semicolon at the end of the `write!` call? Well, first of all, copy that code in and run it to prove to yourself that it's *not* a typo, and that code works. Now add the semicolon and try compiling again. You'll get something like:

```
error[E0308]: mismatched types
 --> src/main.rs:7:81
  |
7 |       fn fmt(&self, fmt: &mut std::fmt::Formatter) -> Result<(), std::fmt::Error>
{
  |
________________________________________________________________________________^
8 | |         write!(fmt, "{} ({} years old)", self.name, self.age);
  | |                                                              - help: consider
removing this semicolon
9 | |     }
  | |_____^ expected enum `std::result::Result`, found ()
  |
  = note: expected type `std::result::Result<(), std::fmt::Error>`
             found type `()`
```

This is potentially a huge point of confusion in Rust. Let me point out something else that you may have noticed, especially if you come from a C/C++/Java background: we have a return value from our method, but we never used `return`!

The answer to that second concern is easy: the last value generated in a function in Rust is taken as its return value. This is similar to Ruby and—yet again—Haskell. `return` is only needed for early termination.

But we're still left with our first question: why don't we need a semicolon here, and why does adding the semicolon break our code? Semicolons in Rust are used for terminating *statements*. A statement is something like the `let` statement we briefly demonstrated here. The value of a statement is always unit, or `()`. That's why, when we add the semicolon, the error message says `found type `()`. Leaving off the semicolon, the expression itself is the return value, which is what we want.

You'll hear Rust referred to as an *expression-oriented language*. You can see mention of this in the FAQ. Personally, I find that the usage of semicolon like this can be subtle, and I still instinctively trip up on it occasionally when my C/C++/Java habits kick in. But fortunately the compiler helps identify these pretty quickly.

1.6. Numeric types

Last concept before we just start dropping in some code. We're going to start off by playing with numeric values. There's a really good reason for this in Rust: they are *copy values*, values which the compiler automatically clones for us. Keep in mind that a big part of Rust is ownership, and tracking who owns what is non-trivial. However, with the primitive numeric types, making copies of the values is so cheap, the compiler will do it for you automatically.

To demonstrate, let's check out some code that works fine with numeric types:

```
fn main() {
    let val: i32 = 42;
    printer(val);
    printer(val);
}

fn printer(val: i32) {
    println!("The value is: {}", val);
}
```

We've used a *let statement* to create a new variable, `val`. We've explicitly stated that its type is `i32`, or a 32-bit signed integer. Typically, these kinds of type annotations are not needed in Rust, as it will usually be able to infer types. Try leaving off the type annotation here. Anyway, we then call the function `printer` on `val` twice. All good.

Now, let's use a `String` instead. A `String` is a heap-allocated value which can be created from a string literal with the `to_string` method. (Much more on the many string types later.) It's expensive to copy a `String`, so the compiler won't do it for us automatically. Therefore, this code won't compile:

```
fn main() {
    let val: String = "Hello, World!".to_string();
    printer(val);
    printer(val);
}

fn printer(val: String) {
    println!("The value is: {}", val);
}
```

You'll get this intimidating error message:

```
error[E0382]: use of moved value: `val`
 --> src/main.rs:4:13
  |
3 |     printer(val);
  |             --- value moved here
4 |     printer(val);
  |             ^^^ value used here after move
  |
  = note: move occurs because `val` has type `std::string::String`, which does not
implement the `Copy` trait

error: aborting due to previous error
```

Exercise

There are two easy ways to fix this error message: one using the `clone()` method of `String`, and one that changes `printer` to take a reference to a `String`. Implement both solutions.

Solution 1: Clone

We've moved the original `val` into the first call to `printer`, and can't use it again. One workaround is to instead move a *clone* of `val` into that call, leaving the original unaffected:

```
fn main() {
    let val: String = String::from("Hello, World!");
    printer(val.clone());
    printer(val);
}

fn printer(val: String) {
    println!("The value is: {}", val);
}
```

Notice that I only cloned `val` the first time, not the second time. We don't need `val` again after the second call, so it's safe to move it. Using an extra clone is expensive, since it requires allocating memory and performing a buffer copy.

Speaking of it being expensive...

Solution 2: Pass by reference

Instead of moving the value, we can instead pass it into the `printer` function by reference. Let's first try to achieve that by just modifying `printer`:

```
fn main() {
    let val: String = String::from("Hello, World!");
    printer(val);
    printer(val);
}

fn printer(val: &String) {
    println!("The value is: {}", val);
}
```

This doesn't work, because when we call `printer`, we're still giving it a `String` and not a reference to a `String`. Fixing that is pretty easy:

```
fn main() {
    let val: String = String::from("Hello, World!");
    printer(&val);
    printer(&val);
}
```

Note that the ampersand means both:

- A reference to this type, and
- Take a reference to this value

There's an even better way to write `printer`:

```
fn printer(val: &str) {
    println!("The value is: {}", val);
}
```

By using `&str` instead of `&String`, we can pass in string literals, and do not need to force the allocation of a heap object. We'll get to this in more detail when we discuss strings.

1.7. Printing numbers

We're going to tie off this lesson with a demonstration of three different ways of looping to print the numbers 1 to 10. I'll let readers guess which is the most idiomatic approach.

1.7.1. loop

`loop` creates an infinite loop.

```
fn main() {
    let i = 1;

    loop {
        println!("i == {}", i);
        if i >= 10 {
            break;
        } else {
            i += 1;
        }
    }
}
```

Exercise

This code doesn't quite work. Try to figure out why without asking the compiler. If you can't find the problem, try to compile it. Then fix the code.

Solution

The error message we get from the compiler is pretty informative:

```
cannot assign twice to immutable variable `i`
```

In order to fix this, we change the variable from immutable to mutable:

```
fn main() {
    let mut i = 1;
    ...
```

If you're wondering: you could equivalently use `return` or `return ()` to exit the loop, since the end of the loop is also the end of the function.

1.7.2. while

This is similar to C-style while loops: it takes a condition to check.

```
fn main() {
    let i = 1;

    while i <= 10 {
        println!("i == {}", i);
        i += 1;
    }
}
```

This has the same bug as the previous example.

1.7.3. for loops

For loops let you perform some action for each value in a collection. The collections are generated lazily using iterators, a great concept built right into the language in Rust. Iterators are somewhat similar to generators in Python.

```
fn main() {
    for i in 1..11 {
        println!("i == {}", i);
    }
}
```

Exercise: Semicolons

Can you leave out any semicolons in the examples above? Instead of just slamming code into the compiler, try to think through when you can and cannot drop the semicolons.

Solution

Typically, you can leave off a semicolon on a statement if:

1. The statement is the last statement in a block
2. The type of the expression is unit

For example, in this code, removing the semicolon is fine:

```
fn main() {
    for i in 1..11 {
        println!("i == {}", i)
    }
}
```

That said, it tends to be somewhat idiomatic to leave semicolons on expressions like these which are purely effectful.

Exercise: FizzBuzz

Implement fizzbuzz in Rust. The rules are:

- Print the numbers 1 to 100
- If the number is a multiple of 3, output fizz instead of the number
- If the number is a multiple of 5, output buzz instead of the number
- If the number is a multiple of 3 **and** 5, output fizzbuzz instead of the number

Solution

Here's one possible solution using `if`/`else` fallbacks:

```
fn main() {
    for i in 1..101 {
        if i % 3 == 0 && i % 5 == 0 {
            println!("fizzbuzz");
        } else if i % 3 == 0 {
            println!("fizz");
        } else if i % 5 == 0 {
            println!("buzz");
        } else {
            println!("{}", i);
        }
    }
}
```

This has at least one downside: it will need to test `i % 3 == 0` and `i % 5 == 0` potentially multiple times. Under the surface, the compiler may optimize this away. But still, the repeated modulus calls are just sitting in the code taunting us! We can instead use *pattern matching*:

```
fn main() {
    for i in 1..101 {
        match (i % 3 == 0, i % 5 == 0) {
            (true, true) => println!("fizzbuzz"),
            (true, false) => println!("fizz"),
            (false, true) => println!("buzz"),
            (false, false) => println!("{}", i),
        }
    }
}
```

Or, if you want to have some fun with wildcard matching:

```
fn main() {
    for i in 1..101 {
        match (i % 3, i % 5) {
            (0, 0) => println!("fizzbuzz"),
            (0, _) => println!("fizz"),
            (_, 0) => println!("buzz"),
            (_, _) => println!("{}", i),
        }
    }
}
```

I'm not going to tell you which of these is the "best" solution. And there are certainly other implementations that could be attempted. This was meant to give you a feel for some more Rust constructs.

2. Basics of Ownership

Arguably the biggest distinguishing aspect of Rust versus other popular programming languages is its ownership model. In this lesson, we're going to hit the basics of ownership in Rust. You can read much more in the Rust book chapter on ownership.

2.1. Format

I'm going to be experimenting a bit with lesson format. I want to cover both:

- More theoretical discussions of ownership
- Trying to implement an actual program

As we go further in this book, I intend to spend more time on the latter and less on the former, though we still need significant time on the former right now. I'm going to try approaching this by having the beginning of this chapter discuss ownership, and then we'll implement a first version of bouncy afterwards.

2.2. Comparison with Haskell

I'm going to start by comparing Rust with Haskell, since both languages have a strong concept of immutability. However, Haskell is a garbage collected language. Let's see how these two languages compare. In Haskell:

- Everything is immutable by default
- You use explicit mutability wrappers (like `IORef` or `MVar`) to mark mutability
- References to data can be shared however much you like
- Garbage collection frees up memory non-deterministically
- When you need deterministic resource handling (like file handles), you need to use the bracket pattern or similar

In Rust, data ownership is far more important: it's a primary aspect of the language, and allows the language to bypass garbage collection. It also allows data to often live on the stack instead of the heap, leading to better performance. Also, it runs deterministically, making it a good approach for handling other resources like file handles.

Ownership starts off with the following rules:

- Each value in Rust has a variable that's called its owner.
- There can only be one owner at a time.

- When the owner goes out of scope, the value will be dropped.

2.3. Simple example

Consider this code:

```
#[derive(Debug)]
struct Foobar(i32);

fn uses_foobar(foobar: Foobar) {
    println!("I consumed a Foobar: {:?}", foobar);
}

fn main() {
    let x = Foobar(1);
    uses_foobar(x);
}
```

Syntax note `#[...]` is a compiler pragma. `#[derive(...)]` is one example, and is similar to using `deriving` for typeclasses in Haskell. For some traits, the compiler can automatically provide an implementation if you ask it.

Syntax note `{:?}` in a format string means "use the `Debug` trait for displaying this"

`Foobar` is what's known as a newtype wrapper: it's a new data type wrapping around, and with the same runtime representation of, a signed 32-bit integer (`i32`).

In the `main` function, `x` contains a `Foobar`. When it calls `uses_foobar`, ownership of that `Foobar` passes to `uses_foobar`. Using that `x` again in `main` would be an error:

```
#[derive(Debug)]
struct Foobar(i32);

fn uses_foobar(foobar: Foobar) {
    println!("I consumed a Foobar: {:?}", foobar);
}

fn main() {
    let x = Foobar(1);
    uses_foobar(x);
    uses_foobar(x);
}
```

Results in:

```
error[E0382]: use of moved value: `x`
  --> foo.rs:11:17
   |
10 |     uses_foobar(x);
   |                 - value moved here
11 |     uses_foobar(x);
   |                 ^ value used here after move
   |
   = note: move occurs because `x` has type `Foobar`, which does not implement the
`Copy` trait

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

Side note on copy trait way below...

2.4. Dropping

When a value goes out of scope, its `Drop` trait (like a typeclass) is used, and then the memory is freed. We can demonstrate this by writing a `Drop` implementation for `Foobar`:

Challenge Guess what the output of the program below is before seeing the output.

```
#[derive(Debug)]
struct Foobar(i32);

impl Drop for Foobar {
    fn drop(&mut self) {
        println!("Dropping a Foobar: {:?}", self);
    }
}

fn uses_foobar(foobar: Foobar) {
    println!("I consumed a Foobar: {:?}", foobar);
}

fn main() {
    let x = Foobar(1);
    println!("Before uses_foobar");
    uses_foobar(x);
    println!("After uses_foobar");
}
```

Output

```
Before uses_foobar
I consumed a Foobar: Foobar(1)
Dropping a Foobar: Foobar(1)
After uses_foobar
```

Notice that the value is dropped before `After uses_foobar`. This is because the value was moved into `uses_foobar`, and when that function exits, the `drop` is called.

Exercise 1

There's a function in the standard library, `std::mem::drop`, which drops a value immediately. Implement it.

Solution

Implementing `drop` is fairly simple. We need a function that allows a value to be moved into it, and then does nothing with that value.

```
fn drop<T>(_: T) {
}
```

And surprise: check out the actual `drop` function.

2.5. Lexical scoping

Scoping in Rust is lexical. (Though there's also Non Lexical Lifetimes (NLL) which is enabled by default. There's a nice explanation on Stack Overflow of what NLL does.) We can demonstrate the lexical nature of scoping:

```
#[derive(Debug)]
struct Foobar(i32);

impl Drop for Foobar {
    fn drop(&mut self) {
        println!("Dropping a Foobar: {:?}", self);
    }
}

fn main() {
    println!("Before x");
    let _x = Foobar(1);
    println!("After x");
    {
        println!("Before y");
        let _y = Foobar(2);
        println!("After y");
    }
    println!("End of main");
}
```

Syntax note Use a leading underscore _ for variables that are not used.

The output from this program is:

```
Before x
After x
Before y
After y
Dropping a Foobar: Foobar(2)
End of main
Dropping a Foobar: Foobar(1)
```

CHALLENGE Remove the seemingly-superfluous braces and run the program. Extra points: guess what the output will be before looking at the actual output.

2.6. Borrows/references (immutable)

Sometimes you want to be able to share a reference to a value without moving ownership. Easy enough:

```
#[derive(Debug)]
struct Foobar(i32);

impl Drop for Foobar {
    fn drop(&mut self) {
        println!("Dropping a Foobar: {:?}", self);
    }
}

fn uses_foobar(foobar: &Foobar) {
    println!("I consumed a Foobar: {:?}", foobar);
}

fn main() {
    let x = Foobar(1);
    println!("Before uses_foobar");
    uses_foobar(&x);
    uses_foobar(&x);
    println!("After uses_foobar");
}
```

Things to notice:

- `uses_foobar` now takes a value of type `&Foobar`, which is "immutable reference to a `Foobar`."
- Inside `uses_foobar`, we don't need to explicitly dereference the `foobar` value, this is done automatically by Rust.
- In `main`, we can now use `x` in two calls to `uses_foobar`
- In order to create a reference from a value, we use `&` in front of the variable.

Challenge When do you think the `Dropping a Foobar:` message gets printed?

Remember from the last lesson that there is special syntax for a parameter called `self`. The signature of `drop` above looks quite different from `uses_foobar`. When you see `&mut self`, you can think of it as `self: &mut Self`. Now it looks more similar to `uses_foobar`.

Exercise 2

We'd like to be able to use object syntax for `uses_foobar` as well. Create a method `use_it` on the `Foobar` type that prints the `I consumed` message. Hint: you'll need to do this inside `impl Foobar { ... }`.

Solution

The important trick to implement this method syntax is that the first parameter *must* be some form of `self`. We want to keep this as an immutable reference, so we use `&self`.

```
#[derive(Debug)]
struct Foobar(i32);

impl Drop for Foobar {
    fn drop(&mut self) {
        println!("Dropping a Foobar: {:?}", self);
    }
}

impl Foobar {
    fn use_it(&self) {
        println!("I consumed a Foobar: {:?}", self);
    }
}

fn main() {
    let x = Foobar(1);
    println!("Before uses_foobar");
    x.use_it();
    x.use_it();
    println!("After uses_foobar");
}
```

You may be wondering: why does the code use `x.use_it()`? `use_it` requires a *reference* to a `Foobar`, but `x` is a `Foobar`! In fact, you may have ended up writing something like this:

```
fn main() {
    let x = Foobar(1);
    println!("Before uses_foobar");
    (&x).use_it();
    (&x).use_it();
    println!("After uses_foobar");
}
```

While that's perfectly valid code, it's also unnecessary: Rust will automatically take a reference to a value in the case of a method call.

2.7. Multiple live references

We can change our `main` function to allow two references to `x` to live at the same time. This version also adds explicit types on the local variables, instead of relying on type inference:

```
fn main() {
    let x: Foobar = Foobar(1);
    let y: &Foobar = &x;
    println!("Before uses_foobar");
    uses_foobar(&x);
    uses_foobar(y);
    println!("After uses_foobar");
}
```

This is allowed in Rust, because:

1. Multiple read-only references to a variable cannot result in any data races
2. The lifetime of the value outlives the references to it. In other words, in this case, `x` lives at least as long as `y`.

Let's see two ways to break this.

2.7.1. Reference outlives value

Remember that `std::mem::drop` from before? Check this out:

```
fn main() {
    let x: Foobar = Foobar(1);
    let y: &Foobar = &x;
    println!("Before uses_foobar");
    uses_foobar(&x);
    std::mem::drop(x);
    uses_foobar(y);
    println!("After uses_foobar");
}
```

This results in the error message:

```
error[E0505]: cannot move out of `x` because it is borrowed
  --> foo.rs:19:20
   |
16 |     let y: &Foobar = &x;
   |                       - borrow of `x` occurs here
...
19 |     std::mem::drop(x);
   |                    ^ move out of `x` occurs here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0505`.
```

2.7.2. Mutable reference with other references

You can also take *mutable* references to a value. In order to avoid data races, Rust does not allow value to be referenced mutably and accessed in any other way at the same time.

```
fn main() {
    let mut x: Foobar = Foobar(1);
    let y: &mut Foobar = &mut x;
    println!("Before uses_foobar");
    uses_foobar(&x); // will fail!
    uses_foobar(y);
    println!("After uses_foobar");
}
```

Notice how the type of `y` is now `&mut Foobar`. Like Haskell, Rust tracks mutability at the type level. Yay!

2.8. Challenge

Try to guess which lines in the code below will trigger a compilation error:

```
#[derive(Debug)]
struct Foobar(i32);

fn main() {
    let x = Foobar(1);

    foo(x);
    foo(x);

    let mut y = Foobar(2);

    bar(&y);
    bar(&y);

    let z = &mut y;
    bar(&y);
    baz(&mut y);
    baz(z);
}

// move
fn foo(_foobar: Foobar) {
}

// read only reference
fn bar(_foobar: &Foobar) {
}

// mutable reference
fn baz(_foobar: &mut Foobar) {
}
```

2.9. Mutable reference vs mutable variable

Something I didn't explain above was the `mut` before `x` in:

```
fn main() {
    let mut x: Foobar = Foobar(1);
    let y: &mut Foobar = &mut x;
    println!("Before uses_foobar");
    uses_foobar(&x);
    uses_foobar(y);
    println!("After uses_foobar");
}
```

By default, variables are immutable, and therefore do not allow any kind of mutation. You cannot take a mutable reference to an immutable variable, and therefore `x` must be marked as mutable. Here's an easier way to see this:

```
#[derive(Debug)]
struct Foobar(i32);

fn main() {
    let mut x = Foobar(1);

    x.0 = 2; // changes the 0th value inside the product

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

If you remove the `mut`, this will fail.

2.9.1. Moving into mutable

This bothered me, and I assume it will bother other Haskellers. As just mentioned, the following code will not compile:

```
#[derive(Debug)]
struct Foobar(i32);

fn main() {
    let x = Foobar(1);

    x.0 = 2; // changes the 0th value inside the product

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

Obviously you can't mutate `x`. But let's change this ever so slightly:

```
#[derive(Debug)]
struct Foobar(i32);

fn main() {
    let x = Foobar(1);
    foo(x);
}

fn foo(mut x: Foobar) {

    x.0 = 2; // changes the 0th value inside the product

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

Before learning Rust, I would have objected to this: `x` is immutable, and therefore we shouldn't be

allowed to pass it to a function that needs a mutable `x`. However, this isn't how Rust views the world. The mutability here is a feature of the variable, not the value itself. When you move the `x` into `foo`, `main` no longer has access to `x`, and doesn't care if it's mutated. Inside `foo`, we've explicitly stated that `x` can be mutated, so we're cool.

This is fairly different from how Haskell looks at things.

2.10. Copy trait

We touched on this topic last time with numeric types vs `String`. Let's hit it a little harder. Will the following code compile or not?

```
fn uses_i32(i: i32) {
    println!("I consumed an i32: {}", i);
}

fn main() {
    let x = 1;
    uses_i32(x);
    uses_i32(x);
}
```

It *shouldn't* work, right? `x` is moved into `uses_i32`, and then used again. However, it compiles just fine! What gives?

Rust has a special trait, `Copy`, which indicates that a type is so cheap that it can automatically be passed-by-value. That's exactly what happens with `i32`. You can explicitly do this with the `Clone` trait if desired:

```
#[derive(Debug, Clone)]
struct Foobar(i32);

impl Drop for Foobar {
    fn drop(&mut self) {
        println!("Dropping: {:?}", self);
    }
}

fn uses_foobar(foobar: Foobar) {
    println!("I consumed a Foobar: {:?}", foobar);
}

fn main() {
    let x = Foobar(1);
    uses_foobar(x.clone());
    uses_foobar(x);
}
```

Challenge Why don't we need to use `x.clone()` on the second `uses_foobar`? What happens if we put it in anyway?

Exercise 3

Change the code below, without modifying the `main` function at all, so that it compiles and runs successfully. Some hints: `Debug` is a special trait that can be automatically derived, and in order to have a `Copy` implementation you also need a `Clone` implementation.

```
#[derive(Debug)]
struct Foobar(i32);

fn uses_foobar(foobar: Foobar) {
    println!("I consumed a Foobar: {:?}", foobar);
}

fn main() {
    let x = Foobar(1);
    uses_foobar(x);
    uses_foobar(x);
}
```

Solution

The original code doesn't compile, since our `x` value was moved. However, if we have a `Copy` implementation, Rust will automatically create a copy of the value for us. As I hinted, you can do this easily by using automatic deriving:

```
#[derive(Debug, Clone, Copy)]
struct Foobar(i32);

fn uses_foobar(foobar: Foobar) {
    println!("I consumed a Foobar: {:?}", foobar);
}

fn main() {
    let x = Foobar(1);
    uses_foobar(x);
    uses_foobar(x);
}
```

If you want to be more explicit, you can write the implementations directly. Note that, to the best of my knowledge, there's no advantage to doing so, at least in this case:

```
impl Clone for Foobar {
    fn clone(&self) -> Self {
        Foobar(self.0)
    }
}
impl Copy for Foobar {
}
```

There's no need for a body for the `Copy` trait. Its only purpose is as a signal to the compiler that it's acceptable to copy when needed.

2.11. Lifetimes

The term that goes along most with ownership is *lifetimes*. Every value needs to be owned, and its owned for a certain lifetime until it's dropped. So far, everything we've looked at has involved implicit lifetimes. However, as code gets more sophisticated, we need to be more explicit about these lifetimes. We'll cover that another time.

Exercise 4

Add an implementation of the `double` function to get this code to compile, run, and output the number 2:

```
#[derive(Debug)]
struct Foobar(i32);

fn main() {
    let x: Foobar = Foobar(1);
    let y: Foobar = double(x);
    println!("{}", y.0);
}
```

Remember: to provide a return value from a function, put `-> ReturnType` after the parameter list.

Solution

We're not taking a reference to `x` when calling `double`, so our `double` function needs to take an actual `Foobar`, not a reference to one. It must also return a `Foobar`. One implementation we could come up with is:

```
fn double(foobar: Foobar) -> Foobar {
    Foobar(foobar.0 * 2)
}
```

This takes in an immutable `Foobar`, and then constructs a new `Foobar` from the value inside of it. Another option, however, would be to mutate the original `Foobar` and then return it:

```
fn double(mut foobar: Foobar) -> Foobar {
    foobar.0 *= 2;
    foobar
}
```

The Haskeller in me prefers the first implementation, since it has less mutation. The troll in me loves the second one, since it lets me taunt the Haskeller in me.

2.12. Notes on structs and enums

I mentioned above that `struct Foobar(i32)` is a newtype around an `i32`. That's actually a special case of a more general *positional* struct, where you can have 0 or more fields, named by their numeric position. And positions start numbering at 0, as god and Linus Torvalds intended.

There are some more examples:

```
struct NoFields; // may seem strange, we might cover examples of this later
struct OneField(i32);
struct TwoFields(i32, char);
```

You can also use record syntax:

```
struct Person {
    name: String,
    age: u32,
}
```

`struct`s are known as *product types*, which means they contain multiple values. Rust also provides `enum`s, which are sum types, or tagged unions. These are *alternatives*, where you select one of the options. A simple enum would be:

```
enum Color {
    Red,
    Blue,
    Green,
}
```

But enums variants can also take values:

```
enum Shape {
    Square(u32), // size of one side
    Rectangle(u32, u32), // width and height
    Circle(u32), // radius
}
```

2.13. Bouncy

Enough talk, let's fight! I want to create a simulation of a bouncing ball. The idea behind the game is that a ball will be bouncing inside a rectangular container.

Let's step through the process of creating such a game together. I'll provide the complete `src/main.rs` at the end of the lesson, but strongly recommend you implement this together with me throughout the sections below. Try to **avoid copy pasting**, but instead type in the code yourself to get more comfortable with Rust syntax.

2.13.1. Initialize the project

This part's easy:

```
$ cargo new bouncy
```

If you `cd` into that directory and run `cargo run`, you'll get output like this:

```
$ cargo run
   Compiling bouncy v0.1.0 (/Users/michael/Desktop/bouncy)
    Finished dev [unoptimized + debuginfo] target(s) in 1.37s
     Running `target/debug/bouncy`
Hello, world!
```

The only file we're going to touch today is `src/main.rs`, which will have the source code for our program.

2.13.2. Define data structures

To track the ball bouncing around our screen, we need to know the following information:

- The width of the box containing the ball
- The height of the box containing the ball
- The x and y coordinates of the ball
- The vertical direction of the ball (up or down)
- The horizontal direction of the ball (left or right)

We're going to define new datatypes for tracking the vertical and horizontal direction, and use `u32`s for tracking the position.

We can define `VertDir` as an `enum`. This is a simplified version of what enums can handle, since we aren't given it any payload. We'll do more sophisticated stuff later.

```
enum VertDir {
    Up,
    Down,
}
```

Go ahead and define a `HorizDir` as well that tracks whether we're moving left or right. Now, to track a ball, we need to know its `x` and `y` positions and its vertical and horizontal directions. This will be a struct, since we're tracking multiple values instead of (like an enum) choosing between different options.

```
struct Ball {
    x: u32,
    y: u32,
    vert_dir: VertDir,
    horiz_dir: HorizDir,
}
```

Define a `Frame` struct that tracks the width and height of the play area. Then tie it all together with a `Game` struct:

```
struct Game {
    frame: Frame,
    ball: Ball,
}
```

2.13.3. Create a new game

We can define a method on the `Game` type itself to create a new game. We'll assign some default width and height and initial ball position.

```
impl Game {
    fn new() -> Game {
        let frame = Frame {
            width: 60,
            height: 30,
        };
        let ball = Ball {
            x: 2,
            y: 4,
            vert_dir: VertDir::Up,
            horiz_dir: HorizDir::Left,
        };
        Game {frame, ball}
    }
}
```

Challenge Rewrite this implementation to not use any `let` statements.

Notice how we use `VertDir::Up`; the `Up` constructor is not imported into the current namespace by default. Also, we can define `Game` with `frame, ball` instead of `frame: frame, ball: ball` since the local variable names are the same as the field names.

2.13.4. Bounce

Let's implement the logic of a ball to bounce off of a wall. Let's write out the logic:

- If the `x` value is 0, we're at the left of the frame, and therefore we should move right.

- If `y` is 0, move down.
- If `x` is one less than the width of the frame, we should move left.
- If `y` is one less than the height of the frame, we should move up.
- Otherwise, we should keep moving in the same direction.

We'll want to *modify* the ball, and take the frame as a parameter. We'll implement this as a method on the `Ball` type.

```
impl Ball {
    fn bounce(&mut self, frame: &Frame) {
        if self.x == 0 {
            self.horiz_dir = HorizDir::Right;
        } else if self.x == frame.width - 1 {
            self.horiz_dir = HorizDir::Left;
        }

        ...
```

Go ahead and implement the rest of this function.

2.13.5. Move

Once we know which direction to move in by calling `bounce`, we can move the ball one position. We'll add this as another method within `impl Ball`:

```
fn mv(&mut self) {
    match self.horiz_dir {
        HorizDir::Left => self.x -= 1,
        HorizDir::Right => self.x += 1,
    }
    ...
}
```

Implement the vertical half of this as well.

2.13.6. Step

We need to add a method to `Game` to perform a step of the game. This will involve both bouncing and moving. This goes inside `impl Game`:

```
fn step(&self) {
    self.ball.bounce(self.frame);
    self.ball.mv();
}
```

There are a few bugs in that implementation which you'll need to fix.

2.13.7. Render

We need to be able to display the full state of the game. We'll see that this initial implementation has its flaws, but we're going to do this by printing the entire grid. We'll add a border, use the letter `o` to represent the ball, and put spaces for all of the other areas inside the frame. We'll use the `Display` trait for this.

Let's pull some of the types into our namespace. At the top of our source file, add:

```
use std::fmt::{Display, Formatter};
```

Now, let's make sure we got the type signature correct:

```
impl Display for Game {
    fn fmt(&self, fmt: &mut Formatter) -> std::fmt::Result {
        unimplemented!()
    }
}
```

We can use the `unimplemented!()` macro to stub out our function before we implement it. Finally, let's fill in a dummy `main` function that will print the initial game:

```
fn main () {
    println!("{}", Game::new());
}
```

If everything is set up correctly, running `cargo run` will result in a "not yet implemented" panic. If you get a compilation error, go fix it now.

2.13.8. Top border

Alright, now we can implement `fmt`. First, let's just draw the top border. This will be a plus sign, a series of dashes (based on the width of the frame), another plus sign, and a newline. We'll use the `write!` macro, range syntax (`low..high`), and a `for` loop:

```
write!(fmt, "+");
for _ in 0..self.frame.width {
    write!(fmt, "-");
}
write!(fmt, "+\n");
```

Looks nice, but we get a compilation error:

```
error[E0308]: mismatched types
  --> src/main.rs:79:60
   |
79 |       fn fmt(&self, fmt: &mut Formatter) -> std::fmt::Result {
   |  ____________________________________________________________^
80 | |         write!(fmt, "+");
81 | |         for _ in 0..self.frame.width {
82 | |             write!(fmt, "-");
83 | |         }
84 | |         write!(fmt, "+\n");
   | |                            - help: consider removing this semicolon
85 | |     }
   | |_____^ expected enum `std::result::Result`, found ()
   |
   = note: expected type `std::result::Result<(), std::fmt::Error>`
              found type `()`
```

It says "considering removing this semicolon." Remember that putting the semicolon forces our statement to evaluate to the unit value `()`, but we want a `Result` value. And it seems like the `write!` macro is giving us a `Result` value. Sure enough, if we drop the trailing semicolon, we get something that works:

```
    Finished dev [unoptimized + debuginfo] target(s) in 0.55s
     Running `target/debug/bouncy`
+-------------------------------------------------------------+
```

You may ask: what about all of the other `Result` values from the other calls to `write!`? Good question! We'll get to that a bit later.

2.13.9. Bottom border

The top and bottom border are exactly the same. Instead of duplicating the code, let's define a closure that we can call twice. We introduce a closure in Rust with the syntax `|args| { body }`. This closure will take no arguments, and so will look like this:

```
let top_bottom = || {
    write!(fmt, "+");
    for _ in 0..self.frame.width {
        write!(fmt, "-");
    }
    write!(fmt, "+\n");
};

top_bottom();
top_bottom();
```

First we're going to get an error about `Result` and `()` again. You'll need to remove the last semicolon

to fix this. Do that now. Once you're done with that, you'll get a brand new error message. Yay!

```
error[E0596]: cannot borrow `top_bottom` as mutable, as it is not declared as mutable
  --> src/main.rs:88:9
   |
80 |         let top_bottom = || {
   |             ---------- help: consider changing this to be mutable: `mut
top_bottom`
...
88 |         top_bottom();
   |         ^^^^^^^^^^ cannot borrow as mutable
```

The error message tells us exactly what to do: stick a `mut` in the middle of `let top_bottom`. Do that, and make sure it fixes things. Now the question: why? The `top_bottom` closure has captured the `fmt` variable from the environment. In order to use that, we need to call the `write!` macro, which mutates that `fmt` variable. Therefore, each call to `top_bottom` is itself a mutation. Therefore, we need to mark `top_bottom` as mutable.

There are three different types of closure traits: `Fn`, `FnOnce`, and `FnMut`. We'll get into the differences among these in a later tutorial.

Anyway, we should now have both a top and bottom border in our output.

2.13.10. Rows

Let's print each of the rows. In between the two `top_bottom()` calls, we'll stick a `for` loop:

```
for row in 0..self.frame.height {
}
```

Inside that loop, we'll want to add the left border and the right border:

```
write!(fmt, "|");
// more code will go here
write!(fmt, "|\n");
```

Go ahead and call `cargo run`, you're in for an unpleasant surprise:

```
error[E0501]: cannot borrow `*fmt` as mutable because previous closure requires unique
access
  --> src/main.rs:91:20
   |
80 |         let mut top_bottom = || {
   |                              -- closure construction occurs here
81 |             write!(fmt, "+");
   |                    --- first borrow occurs due to use of `fmt` in closure
...
91 |             write!(fmt, "|");
   |                    ^^^ borrow occurs here
...
96 |         top_bottom()
   |         ---------- first borrow used here, in later iteration of loop
```

Oh no, we're going to have to deal with the borrow checker!

2.13.11. Fighting the borrow checker

Alright, remember before that the `top_bottom` closure capture a mutable reference to `fmt`? Well that's causing us some trouble now. There can only be one mutable reference in play at a time, and `top_bottom` is holding it for the entire body of our method. Here's a simple workaround in this case: take `fmt` as a parameter to the closure, instead of capturing it:

```
let top_bottom = |fmt: &mut Formatter| {
```

Go ahead and fix the calls to `top_bottom`, and you should get output that looks like this (some extra rows removed).

```
+------------------------------------------------------------+
||
||
||
||
...
+------------------------------------------------------------+
```

Alright, now we can get back to...

2.13.12. Columns

Remember that `// more code will go here` comment? Time to replace it! We're going to use another `for` loop for each column:

```
for column in 0..self.frame.width {
    write!(fmt, " ");
}
```

Running `cargo run` will give you a complete frame, nice! Unfortunately, it doesn't include our ball. We want to write a `o` character instead of space when `column` is the same as the ball's `x`, and the same thing for `y`. Here's a partial implementation:

```
let c = if row == self.ball.y {
    'o'
} else {
    ' '
};
write!(fmt, "{}", c);
```

There's something wrong with the output (test with `cargo run`). Fix it and your render function will be complete!

2.13.13. The infinite loop

We're almost done! We need to add an infinite loop in our `main` function that:

- Prints the game
- Steps the game
- Sleeps for a bit of time

We'll target 30 FPS, so we want to sleep for 33ms. But how do we sleep in Rust? To figure that out, let's go to the Rust standard library docs and search for `sleep`. The first result is `std::thread::sleep`, which seems like a good bet. Check out the docs there, especially the wonderful example, to understand this code.

```
fn main () {
    let game = Game::new();
    let sleep_duration = std::time::Duration::from_millis(33);
    loop {
        println!("{}", game);
        game.step();
        std::thread::sleep(sleep_duration);
    }
}
```

There's one compile error in this code. Try to anticipate what it is. If you can't figure it out, ask the compiler, then fix it. You should get a successful `cargo run` that shows you a bouncing ball.

2.13.14. Problems

There are two problems I care about in this implementation:

- The output can be a bit jittery, especially on a slow terminal. We should really be using something like the `curses` library to handle double buffering of the output.
- If you ran `cargo run` before, you probably didn't see it. Run `cargo clean` and `cargo build` to force a rebuild, and you should see the following warning:

```
warning: unused `std::result::Result` which must be used
  --> src/main.rs:88:9
   |
88 |         top_bottom(fmt);
   |         ^^^^^^^^^^^^^^^^
   |
   = note: this `Result` may be an `Err` variant, which should be handled
```

I mentioned this problem above: we're ignoring failures coming from the calls to the `write!` macro in most cases but throwing away the `Result` using a semicolon. There's a nice, single character solution to this problem. This forms the basis of proper error handling in Rust. However, we'll save that for another time. For now, we'll just ignore the warning.

2.13.15. Complete source

You can find the complete source code for this implementation as a Github gist. Reminder: it's much better if you step through the code above and implement it yourself.

I've added one piece of syntax we haven't covered yet in that tutorial, at the end of the call to `top_bottom`. We'll cover that in much more detail next week.

3. Iterators and Errors

In the previous chapter, we finished off with a bouncy ball implementation with some downsides: lackluster error handling and ugly buffering. It also had another limitation: a static frame size. Today, we're going to address all of these problems, starting with that last one: let's get some command line arguments to control the frame size.

Like the previous chapter, I'm going to expect you, the reader, to be making changes to the source code along with me. Make sure to *actually type in the code while reading*!

3.1. Command line arguments

We're going to modify our application as follows:

- Accept two command line arguments: the width and the height
- Both must be valid `u32`s

- Too many or too few command line arguments will result in an error message

Sounds easy enough. In a real application, we would use a proper argument-handling library, like clap. But for now, we're going lower level. Like we did for the sleep function, let's start by searching the standard library docs for the word `args`. The first two entries both look relevant.

- `std::env::Args` An iterator over the arguments of a process, yielding a `String` value for each argument.
- `std::env::args` Returns the arguments which this program was started with (normally passed via the command line).

Now's a good time to mention that, by strong convention:

- Module names (like `std` and `env`) and function names (like `args`) are `snake_cased`
- Types (like `Args`) are `PascalCased`
  - Exception: primitives like `u32` and `str` are lower case

The `std` module has an `env` module. The `env` module has both an `Args` type and a `args` function. Why do we need both? Even more strangely, let's look at the type signature for the `args` function:

```
pub fn args() -> Args
```

The `args` *function* returns a value of type `Args`. If `Args` was a type synonym for, say, a vector of `String`s, this would make sense. But that's not the case. And if you check out its docs, there aren't any fields or methods exposed on `Args`, only trait implementations!

3.2. The extra datatype pattern

Maybe there's a proper term for this in Rust, but I haven't seen it myself yet. (If someone has, please let me know so I can use the proper term.) There's a pervasive pattern in the Rust ecosystem, which in my experience starts with iterators and continues to more advanced topics like futures and async I/O.

- We want to have composable interfaces
- We also want high performance
- Therefore, we define lots of helper data types that allow the compiler to perform some great optimizations
- And we define traits as an interface to let these types compose nicely with each other

Sound abstract? Don't worry, we'll make that concrete in a bit. Here's the practical outcome of all of this:

- We end up programming quite a bit against traits, which provide a common abstractions and lots of helper functions
- We get a matching data type for many common functions

- Often times, our type signatures will end up being massive, representing all of the different composition we performed (though the `-> impl Iterator` style helps with that significantly, see the announcement blog post for more details)

Alright, with that out of the way, let's get back to command line arguments!

3.3. CLI args via iterators

Let's play around in an empty file before coming back to bouncy. (Either use `cargo new` and `cargo run`, or use `rustc` directly, your call.) If I click on the expand button next to the `Iterator` trait on the `Args` docs page, I see this function:

```
fn next(&mut self) -> Option<String>
```

Let's play with that a bit:

```
use std::env::args;

fn main() {
    let mut args = args(); // Yes, that name shadowing works
    println!("{:?}", args.next());
    println!("{:?}", args.next());
    println!("{:?}", args.next());
    println!("{:?}", args.next());
}
```

Notice that we had to use `let mut`, since the `next` method will mutate the value. Now I'm going to run this with `cargo run foo bar`:

```
$ cargo run foo bar
   Compiling args v0.1.0 (/Users/michael/Desktop/tmp/args)
    Finished dev [unoptimized + debuginfo] target(s) in 1.60s
     Running `target/debug/args foo bar`
Some("target/debug/args")
Some("foo")
Some("bar")
None
```

Nice! It gives us the name of our executable, followed by the command line arguments, returning `None` when there's nothing left. (For pedants out there: command line arguments aren't technically *required* to have the command name as the first argument, it's just a really strong convention most tools follow.)

Let's play with this some more. Can you write a loop that prints out all of the command line arguments and then exits? Take a minute, and then I'll provide some answers.

Alright, done? Cool, let's see some examples! First, we'll `loop` with `return`.

```
use std::env::args;

fn main() {
    let mut args = args();
    loop {
        match args.next() {
            None => return,
            Some(arg) => println!("{}", arg),
        }
    }
}
```

We also don't need to use `return` here. Instead of returning from the function, we can just `break` out of the loop:

```
use std::env::args;

fn main() {
    let mut args = args();
    loop {
        match args.next() {
            None => break,
            Some(arg) => println!("{}", arg),
        }
    }
}
```

Or, if you want to save on some indentation, you can use the `if let`.

```
use std::env::args;

fn main() {
    let mut args = args();
    loop {
        if let Some(arg) = args.next() {
            println!("{}", arg);
        } else {
            break;
            // return would work too, but break is nicer
            // here, as it is more narrowly scoped
        }
    }
}
```

You can *also* use `while let`. Try to guess what that would look like before checking the next

example:

```
use std::env::args;

fn main() {
    let mut args = args();
    while let Some(arg) = args.next() {
        println!("{}", arg);
    }
}
```

Getting better! Alright, one final example:

```
use std::env::args;

fn main() {
    for arg in args() {
        println!("{}", arg);
    }
}
```

Whoa, what?!? Welcome to one of my favorite aspects of Rust. Iterators are a concept built into the language directly, via `for` loops. A `for` loop will automate the calling of `next()`. It also hides away the fact that there's some mutable state at play, at least to some extent. This is a powerful concept, and allows a lot of code to end up with a more functional style, something I happen to be a big fan of.

3.4. Skipping

It's all well and good that the first arguments in the name of the executable. But we typically don't care about that. Can we somehow skip that in our output? Well, here's one approach:

```
use std::env::args;

fn main() {
    let mut args = args();
    let _ = args.next(); // drop it on the floor
    for arg in args {
        println!("{}", arg);
    }
}
```

That works, but it's a bit clumsy, especially compared to our previous version that had no mutable variables. Maybe there's some other way to skip things. Let's search the standard library again. I see the first results as `std::iter::Skip` and `std::iter::Iterator::skip`. The former is a data type, and the latter is a method on the `Iterator` trait. Since our `Args` type implements the `Iterator` trait, we can use it. Nice!

Side note Haskellers: `skip` is like `drop` in most Haskell libraries, like `Data.List` or `vector`. `drop` has a totally different meaning in Rust (dropping owned data), so `skip` is a better name in Rust.

Let's look at some signatures from the docs above:

```
pub struct Skip<I> { /* fields omitted */ }
fn skip(self, n: usize) -> Skip<Self>
```

Hmm... deep breaths. `Skip` is a data type that is *parameterized* over some data type, `I`. This is a common pattern in iterators: `Skip` wraps around an existing data type and adds some new functionality to how it iterates. The `skip` method will *consume* an existing iterator, take the number of arguments to skip, and return a new `Skip<OrigDataType>` value. How do I know it consumes the original iterator? The first parameter is `self`, not `&self` or `&mut self`.

That seemed like a lot of concepts. Fortunately, *usage* is pretty easy:

```
use std::env::args;

fn main() {
    for arg in args().skip(1) {
        println!("{}", arg);
    }
}
```

Nice!

Exercise 1

Type inference lets the program above work just fine without any type annotations. However, it's a good idea to get used to the generated types, since you'll see them all too often in error messages. Get the program below to compile by fixing the type signature. Try to do it without using compiler at first, since the error messages will almost give the answer away.

```
use std::env::{args, Args};
use std::iter::Skip;

fn main() {
    let args: Args = args().skip(1);
    for arg in args {
        println!("{}", arg);
    }
}
```

Solution

The trick here is to "wrap" the `Args` data type with the `Skip` data type:

```
use std::env::{args, Args};
use std::iter::Skip;

fn main() {
    let args: Skip<Args> = args().skip(1);
    for arg in args {
        println!("{}", arg);
    }
}
```

You may have noticed that we didn't need to mark `args` as mutable. That's because we are moving the `args` value into the `for` loop, meaning that any mutation to it by the `for` loop cannot be seen in the `main` function.

This layering-of-datatypes approach, as mentioned above, is a real boon to performance. Iterator-heavy code will often compile down to an efficient loop, comparable with the best hand-rolled loop you could have written. However, iterator code is much higher level, more declarative, and easy to maintain and extend.

There's a lot more to iterators, but we're going to stop there for the moment, since we still want to process our command line parameters, and we need to learn one more thing first.

3.5. Parsing integers

If you search the standard library for `parse`, you'll find the `str::parse` method. The documentation does a good job of explaining things, I won't repeat that here. Please go read that now.

OK, you're back? Turbofish is a funny name, right?

Take a crack at writing a program that prints the result of parsing each command line argument as a `u32`, then check my version:

```
fn main() {
    for arg in std::env::args().skip(1) {
        println!("{:?}", arg.parse::<u32>());
    }
}
```

And let's try running it:

```
$ cargo run one 2 three four 5 6 7
Err(ParseIntError { kind: InvalidDigit })
Ok(2)
Err(ParseIntError { kind: InvalidDigit })
Err(ParseIntError { kind: InvalidDigit })
Ok(5)
Ok(6)
Ok(7)
```

When the parse is successful, we get the `Ok` variant of the `Result` enum. When the parse fails, we get the `Err` variant, with a `ParseIntError` telling us what went wrong. (The type signature on `parse` itself uses some associated types to indicate this type, we're not going to get into that right now.)

This is a common pattern in Rust. Rust has no runtime exceptions, so we track potential failure at the type level with actual values.

Side note You may think of `panic`s as similar to runtime exceptions, and to some extent they are. However, you're not able to properly recover from `panic`s, making them different in practice from how runtime exceptions are used in other languages like Python.

3.6. Parse our command line

We're finally ready to get started on our actual command line parsing! We're going to be overly tedious in our implementation, especially with our data types. After we finish implementing this in a blank file, we'll move the code into the bouncy implementation itself. First, let's define a data type to hold a successful parse, which will contain the width and the height.

Challenge Will this be a struct or an enum? Can you try implementing this yourself first?

Since we want to hold onto multiple values, we'll be using a `struct`. I'd like to use named fields, so we have:

```
struct Frame {
    width: u32,
    height: u32,
}
```

Next, let's define an error type to represent all of the things that can go wrong during this parse. We have:

- Too few arguments
- Too many arguments
- Invalid integer

Challenge Are we going to use a struct or an enum this time?

This time, we'll use an enum, because we'll only detect one of these problems (whichever we notice

first). Aficionados of web forms and applicative parsing may scoff at this and say we should detect *all* errors, but we're going to be lazy.

```
enum ParseError {
    TooFewArgs,
    TooManyArgs,
    InvalidInteger(String),
}
```

Notice that the `InvalidInteger` variant takes a payload, the `String` it failed parsing. This is what makes `enum`s in Rust so much more powerful than enumerations in most other languages.

Challenge We're going to write a `parse_args` helper function. Can you guess what its type signature will be?

Combining all of the knowledge we established above, here's an implementation:

```
#[derive(Debug)]
struct Frame {
    width: u32,
    height: u32,
}

#[derive(Debug)]
enum ParseError {
    TooFewArgs,
    TooManyArgs,
    InvalidInteger(String),
}

fn parse_args() -> Result<Frame, ParseError> {
    use self::ParseError::*; // bring variants into our namespace

    let mut args = std::env::args().skip(1);

    match args.next() {
        None => Err(TooFewArgs),
        Some(width_str) => {
            match args.next() {
                None => Err(TooFewArgs),
                Some(height_str) => {
                    match args.next() {
                        Some(_) => Err(TooManyArgs),
                        None => {
                            match width_str.parse() {
                                Err(_) => Err(InvalidInteger(width_str)),
                                Ok(width) => {
                                    match height_str.parse() {
                                        Err(_) => Err(InvalidInteger(height_str)),
                                        Ok(height) => Ok(Frame {
                                            width,
                                            height,
                                        }),
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

fn main() {
    println!("{:?}", parse_args());
}
```

Holy nested blocks Batman, that is a *lot* of indentation! The pattern is pretty straightforward:

- Pattern match
- If we got something bad, stop with an `Err`
- If we got something good, keep going

Haskellers at this point are screaming about `do` notation and monads. Ignore them. We're in the land of Rust, we don't take **kind**ly to those things around here. (Someone please yell at me for that terrible pun.)

Exercise 2

Why didn't we need to use the turbofish on the call to `parse` above?

Solution

When we call `parse` in the context of that example, type inference tells us that the `width` and `height` results must be `u32`s, since they are used as fields in `Frame`. Rust is able to determine what implementation to use based on the needed return type. Cool!

Yet again, Haskellers are rolling their eyes and saying "old news."

What we want to do is *return* early from our function. You know what keyword can help with that? That's right: `return`!

```
fn parse_args() -> Result<Frame, ParseError> {
    use self::ParseError::*;

    let mut args = std::env::args().skip(1);

    let width_str = match args.next() {
        None => return Err(TooFewArgs),
        Some(width_str) => width_str,
    };

    let height_str = match args.next() {
        None => return Err(TooFewArgs),
        Some(height_str) => height_str,
    };

    match args.next() {
        Some(_) => return Err(TooManyArgs),
        None => (),
    }

    let width = match width_str.parse() {
        Err(_) => return Err(InvalidInteger(width_str)),
        Ok(width) => width,
    };

    let height = match height_str.parse() {
        Err(_) => return Err(InvalidInteger(height_str)),
        Ok(height) => height,
    };

    Ok(Frame {
        width,
        height,
    })
}
```

Much nicer to look at! However, it's still a bit repetitive, and littering those `return`s everywhere is subjectively not very nice. In fact, while typing this up, I accidentally left off a few of the `return`s and got to stare at some long error messages. (Try that for yourself.)

3.7. Question mark

Side note The trailing question mark we're about to introduce used to be the `try!` macro in Rust. If you're confused about the seeming overlap: it's simply a transition to new syntax.

The pattern above is so common that Rust has built in syntax for it. If you put a question mark after an expression, it basically does the whole match/return-on-Err thing for you. It's more powerful than we'll demonstrate right now, but we'll get to that extra power a bit later.

To start off, we're going to define some helper functions to:

- Require another argument
- Require that there are no more arguments
- Parse a `u32`

All of these need to return `Result` values, and we'll use a `ParseError` for the error case in all of them. The first two functions need to take a mutable reference to our arguments. (As a side note, I'm going to stop using the `skip` method now, because if I do it will give away the solution to exercise 1.)

```
use std::env::Args;

fn require_arg(args: &mut Args) -> Result<String, ParseError> {
    match args.next() {
        None => Err(ParseError::TooFewArgs),
        Some(s) => Ok(s),
    }
}

fn require_no_args(args: &mut Args) -> Result<(), ParseError> {
    match args.next() {
        Some(_) => Err(ParseError::TooManyArgs),
        // I think this looks a little weird myself.
        // But we're wrapping up the unit value ()
        // with the Ok variant. You get used to it
        // after a while, I guess
        None => Ok(()),
    }
}

fn parse_u32(s: String) -> Result<u32, ParseError> {
    match s.parse() {
        Err(_) => Err(ParseError::InvalidInteger(s)),
        Ok(x) => Ok(x),
    }
}
```

Now that we have these helpers defined, our `parse_args` function is much easier to look at:

```
fn parse_args() -> Result<Frame, ParseError> {
    let mut args = std::env::args();

    // skip the command name
    let _command_name = require_arg(&mut args)?;

    let width_str = require_arg(&mut args)?;
    let height_str = require_arg(&mut args)?;
    require_no_args(&mut args)?;
    let width = parse_u32(width_str)?;
    let height = parse_u32(height_str)?;

    Ok(Frame { width, height })
}
```

Beautiful!

3.8. Forgotten question marks

What do you think happens if you forget the question mark on the `let width_str` line? If you do so:

- `width_str` will contain a `Result<String, ParseError>` instead of a `String`
- The call to `parse_u32` will not type check

```
error[E0308]: mismatched types
  --> src/main.rs:50:27
   |
50 |     let width = parse_u32(width_str)?;
   |                           ^^^^^^^^^ expected struct `std::string::String`, found
enum `std::result::Result`
   |
   = note: expected type `std::string::String`
              found type `std::result::Result<std::string::String, ParseError>`
```

That's nice. But what will happen if we forget the question mark on the `require_no_args` call? We never use the output value there, so it will type check just fine. Now we have the age old problem of C: we're accidentally ignoring error codes!

Well, not so fast. Check out this wonderful warning from the compiler:

```
warning: unused `std::result::Result` which must be used
  --> src/main.rs:49:5
   |
49 |     require_no_args(&mut args);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: #[warn(unused_must_use)] on by default
   = note: this `Result` may be an `Err` variant, which should be handled
```

That's right: Rust will detect if you've ignored a potential failure. There *is* a hole in this in the current code sample:

```
let _command_name = require_arg(&mut args);
```

That doesn't trigger the warning, since in `let _name = blah;`, the leading underscore says "I know what I'm doing, I don't care about this value." Instead, it's better to write the code without the `let`:

```
require_arg(&mut args);
```

Now we get a warning, which can be solved with the added trailing question mark.

Exercise 3

It would be more convenient to use method call syntax. Let's define a helper data type to make this possible. Fill in the implementation of the code below.

```
#[derive(Debug)]
struct Frame {
    width: u32,
    height: u32,
}

#[derive(Debug)]
enum ParseError {
    TooFewArgs,
    TooManyArgs,
    InvalidInteger(String),
}

struct ParseArgs(std::env::Args);

impl ParseArgs {
    fn new() -> ParseArgs {
        unimplemented!()
    }


    fn require_arg(&mut self) -> Result<String, ParseError> {
        match self.0.next() {
        }
    }
}

fn parse_args() -> Result<Frame, ParseError> {
    let mut args = ParseArgs::new();

    // skip the command name
    args.require_arg()?;

    let width_str = args.require_arg()?;
    let height_str = args.require_arg()?;
    args.require_no_args()?;
    let width = parse_u32(width_str)?;
    let height = parse_u32(height_str)?;

    Ok(Frame { width, height })
}

fn main() {
    println!("{:?}", parse_args());
}
```

Solution

Complete source file:

```
#[derive(Debug)]
struct Frame {
    width: u32,
    height: u32,
}

#[derive(Debug)]
enum ParseError {
    TooFewArgs,
    TooManyArgs,
    InvalidInteger(String),
}

struct ParseArgs(std::env::Args);

impl ParseArgs {
    fn new() -> ParseArgs {
        ParseArgs(std::env::args())
    }


    fn require_arg(&mut self) -> Result<String, ParseError> {
        match self.0.next() {
            None => Err(ParseError::TooFewArgs),
            Some(s) => Ok(s),
        }
    }

    fn require_no_args(&mut self) -> Result<(), ParseError> {
        match self.0.next() {
            Some(_) => Err(ParseError::TooManyArgs),
            None => Ok(()),
        }
    }
}

fn parse_u32(s: String) -> Result<u32, ParseError> {
    match s.parse() {
        Err(_) => Err(ParseError::InvalidInteger(s)),
        Ok(x) => Ok(x),
    }
}

fn parse_args() -> Result<Frame, ParseError> {
    let mut args = ParseArgs::new();
```

```
    // skip the command name
    args.require_arg()?;

    let width_str = args.require_arg()?;
    let height_str = args.require_arg()?;
    args.require_no_args()?;
    let width = parse_u32(width_str)?;
    let height = parse_u32(height_str)?;

    Ok(Frame { width, height })
}

fn main() {
    println!("{:?}", parse_args());
}
```

3.9. Updating bouncy

This next bit should be done as a Cargo project, not with `rustc`. Let's start a new empty project:

```
$ cargo new bouncy-args --bin
$ cd bouncy-args
```

Next, let's get the old code and place it in `src/main.rs`. You can copy-paste manually, or run:

```
$ curl
https://gist.githubusercontent.com/snoyberg/5307d493750d7b48c1c5281961bc31d0/raw/8f467
e87f69a197095bda096cbbb71d8d813b1d7/main.rs > src/main.rs
```

Run `cargo run` and make sure it works. You can use `Ctrl-C` to kill the program.

We already wrote fully usable argument parsing code above. Instead of putting it in the same source file, let's put it in its own file. In order to do so, we're going to have to play with modules in Rust.

For convenience, you can view the full source code as a Gist. We need to put this in `src/parse_args.rs`:

```
$ curl
https://gist.githubusercontent.com/snoyberg/568899dc3ae6c82e54809efe283e4473/raw/2ee26
1684f81745b21e571360b1c5f5d77b78fce/parse_args.rs > src/parse_args.rs
```

If you run `cargo build` now, it won't even look at `parse_args.rs`. Don't believe me? Add some invalid content to the top of that file and run `cargo build` again. Nothing happens, right? We need to tell the compiler that we've got another module in our project. We do that by modifying `src/main.rs`. Add

the following line to the top of your file:

```
mod parse_args;
```

If you put in that invalid line before, running `cargo build` should now result in an error message. Perfect! Go ahead and get rid of that invalid line and make sure everything compiles and runs. We won't be accepting command line arguments yet, but we're getting closer.

3.10. Use it!

We're currently getting some dead code warnings, since we aren't using anything from the new module:

```
warning: struct is never constructed: `Frame`
 --> src/parse_args.rs:2:1
  |
2 | struct Frame {
  | ^^^^^^^^^^^^
  |
  = note: #[warn(dead_code)] on by default

warning: enum is never used: `ParseError`
 --> src/parse_args.rs:8:1
  |
8 | enum ParseError {
  | ^^^^^^^^^^^^^^^

warning: function is never used: `parse_args`
  --> src/parse_args.rs:14:1
   |
14 | fn parse_args() -> Result<Frame, ParseError> {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Let's fix that. To start off, add the following to the top of your `main` function, just to prove that we can, in fact, use our new module:

```
println!("{:?}", parse_args::parse_args());
return; // don't start the game, our output will disappear
```

Also, add a `pub` in front of the items we want to access from the `main.rs` file, namely:

- `struct Frame`
- `enum ParseError`
- `fn parse_args`

Running this gets us:

```
$ cargo run
   Compiling bouncy-args v0.1.0 (/Users/michael/Desktop/tmp/bouncy-args)
warning: unreachable statement
   --> src/main.rs:115:5
    |
115 |     let mut game = Game::new();
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^
    |
    = note: #[warn(unreachable_code)] on by default

    Finished dev [unoptimized + debuginfo] target(s) in 0.67s
     Running `target/debug/bouncy-args`
Err(TooFewArgs)
```

It's nice that we get an unreachable statement warning. But most importantly: our argument parsing is working!

Let's try to use this. We'll modify the `Game::new()` method to accept a `Frame` as input:

```
impl Game {
    fn new(frame: Frame) -> Game {
        let ball = Ball {
            x: 2,
            y: 4,
            vert_dir: VertDir::Up,
            horiz_dir: HorizDir::Left,
        };
        Game {frame, ball}
    }

    ...
}
```

And now we can rewrite our `main` function as:

```
fn main () {
    match parse_args::parse_args() {
        Err(e) => {
            // prints to stderr instead of stdout
            eprintln!("Error parsing args: {:?}", e);
        },
        Ok(frame) => {
            let mut game = Game::new(frame);
            let sleep_duration = std::time::Duration::from_millis(33);
            loop {
                println!("{}", game);
                game.step();
                std::thread::sleep(sleep_duration);
            }
        }
    }
}
```

3.11. Mismatched types

We're good, right? Not quite:

```
error[E0308]: mismatched types
   --> src/main.rs:114:38
    |
114 |             let mut game = Game::new(frame);
    |                                      ^^^^^ expected struct `Frame`, found struct
`parse_args::Frame`
    |
    = note: expected type `Frame`
               found type `parse_args::Frame`
```

We now have two different definitions of `Frame`: in our `parse_args` module, and in `main.rs`. Let's fix that. First, delete the `Frame` declaration in `main.rs`. Then add the following after our `mod parse_args;` statement:

```
use self::parse_args::Frame;
```

`self` says we're finding a module that's a child of the current module.

3.12. Public and private

Now everything will work, right? Wrong again! `cargo build` will vomit a bunch of these errors:

```
error[E0616]: field `height` of struct `parse_args::Frame` is private
  --> src/main.rs:85:23
   |
85 |         for row in 0..self.frame.height {
   |
```

By default, identifiers are private in Rust. In order to expose them from one module to another, you need to add the `pub` keyword. For example:

```
pub width: u32,
```

Go ahead and add `pub` as needed. Finally, if you run `cargo run`, you should see `Error parsing args: TooFewArgs`. And if you run `cargo run 5 5`, you should see a much smaller frame than before. Hurrah!

Exercise 4

What happens if you run `cargo run 0 0`? How about `cargo run 1 1`? Put in some better error handling in `parse_args`.

Solution

We want to ensure a minimum size for the width and height. First, let's add two more variants to the `ParseError` enum:

```
WidthTooSmall(u32),
HeightTooSmall(u32),
```

Then add the following to the `parse_args` function, just before the `Ok`:

```
if width < 2 {
    return Err(WidthTooSmall(width));
}
if height < 2 {
    return Err(HeightTooSmall(height));
}
```

3.13. Exit code

Alright, one final irritation. Let's provide some invalid arguments and inspect the exit code of the process:

```
$ cargo run 5
Error parsing args: TooFewArgs
$ echo $?
0
```

For those not familiar: a 0 exit code means everything went OK. That's clearly not the case here! If we search the standard library, it seems the `std::process::exit` can be used to address this. Go ahead and try using that to solve the problem here.

However, we've got one more option: we can return a `Result` straight from `main`!

```
fn main () -> Result<(), self::parse_args::ParseError> {
    match parse_args::parse_args() {
        Err(e) => {
            return Err(e);
        },
        Ok(frame) => {
            let mut game = Game::new(frame);
            let sleep_duration = std::time::Duration::from_millis(33);
            loop {
                println!("{}", game);
                game.step();
                std::thread::sleep(sleep_duration);
            }
        }
    }
}
```

Exercise 5

Can you do something to clean up the nesting a bit here?

Solution

This is another perfect time to pull out our trailing question mark!

```
fn main () -> Result<(), self::parse_args::ParseError> {
    let frame = parse_args::parse_args()?;
    let mut game = Game::new(frame);
    let sleep_duration = std::time::Duration::from_millis(33);
    loop {
        println!("{}", game);
        game.step();
        std::thread::sleep(sleep_duration);
    }
}
```

3.14. Better error handling

The error handling problem we had in the last chapter involved the call to `top_bottom`. I've already included a solution to that in the download of the code provided. Guess what I changed since last time and then check the code to confirm that you're right.

If you're following closely, you will also notice that there are more warnings about unused `Result` values coming from other calls to `write!`. It would be good practice to fix up those calls to `write!`. Take a stab at doing so.

3.15. Next time

We still didn't fix our double buffering problem, we'll get to that next time. We're also going to introduce some more error handling from the standard library. And maybe we'll get to play a bit more with iterators as well.

4. Crates and more iterators

I'm getting bored with bouncy, this is our third chapter talking about this program. It's time to finish this off! How do we do double buffering? It's time to learn about external libraries in Rust, or *crates*.

We're also going to introduce some more error handling from the standard library. And then we'll get to play a bit more with iterators as well.

4.1. Finding a crate

To do double buffering in the terminal, we're going to want some kind of a curses library. We're going to look for a crate on crates.io. On that page, search for "curses". Looking through the download counts and the descriptions, pancurses looks nice, especially since it supports Unix and

Windows. Let's use it!

Side note If you're wondering, this is *exactly* the process I went through when writing up bouncy. I had no prior knowledge to tell me pancurses was going to be the right choice. Also, this program happens to be the first time I've ever used a curses library.

4.2. Starting a project

We're going to create a new project for this using `cargo new`. We're going to pull in the bouncy code from the end of week 2 (before the introduction of command line parsing), since as we'll see later, we won't need that parsing anymore.

```
$ cargo new bouncy4
$ cd bouncy4
$ curl
https://gist.githubusercontent.com/psibi/3485d2fa30d755d2913d44e848fe2e53/raw/f8a5ec38
bbc40ad3223f8765867f350e91d4ddbe/main.rs > src/main.rs
$ cargo run
```

The ball should be bouncing around your terminal, right? Great!

4.3. Adding the crate

The configuration for your project lives in the file `Cargo.toml`, known as the *manifest*. On my system, it looks like this:

```
[package]
name = "bouncy4"
version = "0.1.0"
authors = ["Michael Snoyman <michael@snoyman.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
```

It will look a bit different on your machine (unless your name is also Michael Snoyman). Notice that `[dependencies]` section at the end? That's where we add information on external crates. If you go back to the pancurses page on crates.io, you'll see:

```
Cargo.toml  pancurses = "0.16.1"
```

It may be different when you're reading this. That's telling us what we can add to the dependencies section to depend on the library. There are lots of details about how to specify dependencies, which we won't get into here (and probably never will in the crash course, I've spent enough of my life

discussing dependency management already :)). You can read more in the Cargo book reference.

Anyway, go ahead and add `pancurses = "0.16.1"` to the end of your `Cargo.toml` and run `cargo build`. `cargo` should run off and do some updating, downloading, compiling, and end up with an executable that does exactly the same thing as it did before. Perfect, time to use it!

4.4. Library docs

On the crates.io page, there's a link to the documentation for pancurses. Open that in a new tab. Also, reading down the crates page, we get a nice usage example. In `main()`, the first call is to `initscr`. Jump over to the API documentation and search for `initscr`, and you'll end up at the `initscr` function.

Let's try this out. Add the following line to the top of your `main` function and compile:

```
let window = pancurses::initscr();
```

Great! We're going to use that returned `Window` struct to interact with pancurses. Go ahead and open its documentation and start browsing through.

4.5. Getting the frame size

We're currently just assigning an arbitrary frame size. Ideally, we'd like to base it off of the actual terminal size. `Window` provides a method for this, `get_max_yx`.

First, a step for you dear reader: modify the `new` method of `Game` to take a `Frame` as input. Then, let's jump back to our `main`. We can capture the maximum `y` and `x` values with:

```
let (max_y, max_x) = window.get_max_yx();
```

And now let's create a `Frame` value out of those.

```
let frame = Frame {
    width: max_x,
    height: max_y,
};
```

Then pass that value into `Game::new()`. This will only work if you already modified `new` to take an extra `Frame` argument, go back a few paragraphs if you missed that.

Challenge Before you hit compile, can you figure out what error message we're about to encounter?

When I run `cargo build`, I get the following error:

```
error[E0308]: mismatched types
   --> src/main.rs:109:16
    |
109 |         width: max_x,
    |                ^^^^^ expected u32, found i32

error[E0308]: mismatched types
   --> src/main.rs:110:17
    |
110 |         height: max_y,
    |                 ^^^^^ expected u32, found i32
```

If you remember, `width` and `height` are `u32`s, but pancurses gives us `i32`s for the maximum x and y values. How do we convert? One easy way to handle this is with `as u32`:

```
width: max_x as u32,
height: max_y as u32,
```

This is a totally unchecked cast. To demonstrate, try running this program:

```
fn main() {
    for i in -5i32..6i32 {
        println!("{} -> {}", i, i as u32)
    }
}
```

(Yes, there's syntax in Rust for following a number with its type like that, it's really nice.)

Which results in:

```
-5 -> 4294967291
-4 -> 4294967292
-3 -> 4294967293
-2 -> 4294967294
-1 -> 4294967295
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 5
```

We're going to just accept this bad behavior for now. Don't worry, it'll get worse.

4.6. Carriage return and the border

If you run this program, you'll get some really weird looking output. If you don't believe me, go run it yourself now. I'll wait.

The first problem is that our newlines aren't working as expected. We previously used `\n` to create a newline. However, with curses enabled, this create a *line feed*, moving the cursor down one row, without a *carriage return*, moving the cursor to the beginning of the line. Go ahead and replace the `\n` usages with `\r\n`.

That's better, but there's still a problem: the grid doesn't fit in the terminal! That's because we didn't take the size of the border into account. Try subtracting 4 from the maximum x and y values and see if that fixes things. (Note: there's a bit of a trick to where you put the `- 4`. Think about what value you're trying to cast.)

4.7. Double buffering

We still haven't done double buffering, but we're in a much better position to do so now! The trick is to replace our `println!` call in the `main` function's `loop` with calls to `window` methods. If you want the challenge, go read through the docs and try to figure out how to make it work. One hint: you'll need to revert the addition of the `\r` carriage returns we added above.

```
loop {
    window.clear(); // get rid of old content
    window.printw(game.to_string()); // write to the buffer
    window.refresh(); // update the screen
    game.step();
    std::thread::sleep(sleep_duration);
}
```

Hurrah, we have double buffering in place! We can finally be done with these bouncing balls.

Or can we?

Exercise 1

It's time to get more comfortable with exploring API docs, and writing some Rust code with less training wheels. There are a few problems with our implementation:

- We don't need to generate an entire string for the whole grid. Instead, we can use some `Window` methods for moving the cursor around and adding characters. Use that instead.
- Drawing the border as we have is harder than it should be. There's a method on `Window` that will help significantly.
- We have a number of assumptions about numbers, in particular that the maximum x and y are positive, and large enough to hold the ball's starting position. Put in some sane limits: both values must be at least 10.
- More advanced, but: pancurses has some built in support for input handling with timeouts. Instead of using `std::thread::sleep`, we can set a timeout on input handling and add that to our main loop. You can then also respond to two specific kinds of input:
  - Exit the program on a `q` press
  - Reset the game when the size of the window changes

Solution

You can find my complete solution as a Github Gist. If your solution looks a bit different than mine, don't worry. Also, see if there's anything interesting you can learn from my implementation, or some improvements you'd like to make to it.

4.8. More iterators!

Alright, I'm sufficiently bored with bouncing balls. Let's talk about something far more interesting: streaming data. Personally I found it easiest to understand iterators by writing a few of them myself, so that's where we're going to start.

Let's do some compiler-guided programming. We discussed previously that there's an `Iterator` trait. So a fair bet is that we need to create a new data type and provide an implementation of the `Iterator` trait. Let's start off with something simple: an iterator that produces no values at all.

```
struct Empty;

fn main() {
    for i in Empty {
        panic!("Wait, this shouldn't happen!");
    }
    println!("All done!");
}
```

`panic!`ing is a way to cause the current thread to exit due to an impossible situation. It's *kind of* like runtime exceptions in other languages, except you can't recover from them. They are only intended to be used for impossible situations.

OK, compile that and you'll get a helpful error message:

```
error[E0277]: `Empty` is not an iterator
```

Cool, let's add an empty implementation (no pun intended):

```
impl Iterator for Empty {
}
```

More help from the compiler!

```
error[E0046]: not all trait items implemented, missing: `Item`, `next`
 --> foo.rs:3:1
  |
3 | impl Iterator for Empty {
  | ^^^^^^^^^^^^^^^^^^^^^^^ missing `Item`, `next` in implementation
  |
  = note: `Item` from trait: `type Item;`
  = note: `next` from trait: `fn(&mut Self) -> std::option::Option<<Self as
std::iter::Iterator>::Item>`
```

So we need to provide two things: `Item` and `next`. `next` is a function, so we'll get to that in a second. What about that `type Item;`? It's what we call an *associated type*. It tells us what type of values will be produced by this iterator. Since we're not producing anything, we can use any type here. I'll use a `u32`:

```
type Item = u32;
```

Now we need to add in a `next`. Above, it gives the type of that function as:

```
fn(&mut Self) -> std::option::Option<<Self as std::iter::Iterator>::Item>
```

Let's simplify this bit by bit. The *type* of the input is `&mut Self`. However, the correct syntax will be `&mut self`. Find that confusing? Remember that `&mut self` is a shortcut for `self: &mut Self`.

```
fn(&mut self) -> std::option::Option<<Self as std::iter::Iterator>::Item>
```

Next, we can remove the module qualifiers for `Option` and `Iterator`, since they're already in our namespace:

```
fn(&mut self) -> Option<<Self as Iterator>::Item>
```

This `Self as Iterator` is interesting. It's saying "take the current type, and look at its `Iterator` implementation. The reason we care about specifying the implementation is because of what comes next: `::Item`. What we're *really* saying is "we want the `Item` associated type related to the `Iterator` implementation." It's possible that other traits will *also* have an associated type with the same name, so this is an unambiguous way to refer to it.

Anyway, let's see if this type signature works. Include the name of the function and give it a dummy function body:

```
fn next(&mut self) -> Option<<Self as Iterator>::Item> {
    unimplemented!()
}
```

`unimplemented!()` is a macro that uses `panic!` under the surface, and is a convenient way to stub out implementations while under active development. If you compile this, it will succeed. Yay! Then it crashes at runtime due to the `unimplemented!`. Cool.

We can simplify the signature a bit by removing the `as Iterator`, which isn't necessary:

```
fn next(&mut self) -> Option<Self::Item>
```

You can *also* replace the `Self::Item` directly with `u32` if you want. The upside is, in this case, it's shorter. The downside is that if you change the `Item` in the future, you'll have to change it in two places. This is really a subjective point, your call.

Alright, let's provide an implementation. We return an `Option`, which is an `enum` with two variants: `None` or `Some`. The former means "we don't have anything," the latter means "we have something." Given that we're implementing the empty iterator, returning `None` seems like the right move:

```
fn next(&mut self) -> Option<u32> {
    None
}
```

And just like that, we have our first `Iterator` implementation!

Exercise 2

Create an iterator that infinitely produces the number 42. Here's a `main` function to test it out:

```
fn main() {
    // only take 10 to avoid looping forever
    for i in TheAnswer.take(10) {
        println!("The answer to life, the universe, and everything is {}", i);
    }
    println!("All done!");
}
```

Solution

```
struct TheAnswer;

impl Iterator for TheAnswer {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        Some(42)
    }
}
```

4.9. Mutable state

The signature for `next` involves taking a mutable reference to `self`. Let's use it! We're going to create an iterator that counts from 1 to 10. (If you're feeling brave, try to implement it yourself before reading my solution.)

```
struct OneToTen(u32);

fn one_to_ten() -> OneToTen {
    OneToTen(1)
}

impl Iterator for OneToTen {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        if self.0 > 10 {
            None
        } else {
            let res = Some(self.0);
            self.0 += 1;
            res
        }
    }
}

fn main() {
    for i in one_to_ten() {
        println!("{}", i);
    }
}
```

Exercise 3

Create an iterator that produces the Fibonacci sequence. (Anyone who's heard me bemoaning this exercise in the functional programming world should have a hearty chuckle right now.)

Solution

Let's start with the simpler solution:

```
struct Fibs {
    x: u32,
    y: u32,
}

fn fibs() -> Fibs {
    Fibs {
        x: 0,
        y: 1,
    }
}

impl Iterator for Fibs {
    type Item = u32;

    fn next(&mut self) -> Option<u32> {
        let orig_x = self.x;
        let orig_y = self.y;

        self.x = orig_y;
        self.y = orig_x + orig_y;

        Some(orig_x)
    }
}

fn main() {
    for i in fibs().take(10) {
        println!("{}", i);
    }
}
```

However, if you bump that `take(10)` to `take(47)`, the end of your output will look like:

```
701408733
1134903170
thread 'main' panicked at 'attempt to add with overflow', foo.rs:21:18
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

One solution would be to bump to a `u64`, but that's just delaying the problem. Instead, we can use Rust's checked addition method:

```
fn next(&mut self) -> Option<u32> {
    let orig_x = self.x;
    let orig_y = self.y;

    match orig_x.checked_add(orig_y) {
        // overflow
        None => None,

        // no overflow
        Some(new_y) => {
            self.x = orig_y;
            self.y = new_y;

            Some(orig_x)
        }
    }
}
```

Now our stream will stop as soon as overflow occurs.

If you want to get *really* advanced here, you could actually output two more values. To do so, we need to assign to a *derefenced* value and use an `enum` to track our state:

```
enum Fibs {
    Done,
    OneLeft(u32),
    Running(u32, u32),
}

fn fibs() -> Fibs {
    Fibs::Running(0, 1)
}

impl Iterator for Fibs {
    type Item = u32;
    fn next(&mut self) -> Option<u32> {
        use Fibs::*;
        match *self {
            Done => None,
            OneLeft(x) => {
                *self = Done;
                Some(x)
            }
            Running(orig_x, orig_y) => {
                *self = match orig_x.checked_add(orig_y) {
                    // overflow
                    None => OneLeft(orig_y),
                    Some(new_y) => Running(orig_y, new_y),
                };

                Some(orig_x)
            }
        }
    }
}

fn main() {
    for i in fibs().take(47) {
        println!("{}", i);
    }
}
```

4.10. Iterator adapters

We can also write an iterator that will modify an existing iterator. Let's write `Doubler`, which will double the values produced by a previous iterator. In order to make this work, we're going to capture the underlying iterator inside our new data type, which will also necessitate parameterizing our `Doubler` data type on the contained iterator:

```
struct Doubler<I> {
    iter: I,
}
```

Let's throw in a `main` function to show how this is used:

```
fn main() {
    let orig_iter = 1..11; // array indices start at 1
    let doubled_iter = Doubler {
        iter: orig_iter,
    };
    for i in doubled_iter {
        println!("{}", i);
    }
}
```

Great. If we compile this, we get an error about the missing `Iterator` implementation. Let's try to write something for this:

```
impl Iterator for Doubler {
}
```

When we compile this, we get the error message:

```
error[E0107]: wrong number of type arguments: expected 1, found 0
 --> foo.rs:5:19
  |
5 | impl Iterator for Doubler {
  |                   ^^^^^^^ expected 1 type argument
```

OK, that makes sense. `Doubler` itself isn't a type until we give it its parameters. Let's do that:

```
impl Iterator for Doubler<I> {
}
```

We get two error messages. Feel free to look at the second if you like, but it's not terribly helpful. (Recommendation: always look at the first error message first and try to solve that before moving on.) The first error message is:

```
error[E0412]: cannot find type `I` in this scope
 --> foo.rs:5:27
  |
5 | impl Iterator for Doubler<I> {
  |                           ^ not found in this scope
```

OK, what's happening? When we provide an implementation, we need to state all of the type variables we want upfront. So let's do this:

```
impl<I> Iterator for Doubler<I> {
}
```

That may look a bit redundant (it did to me at first), but eventually you'll get to cases where things are more complicated and the two sets of angle brackets don't look identical. (For Haskellers or PureScript users, this is kind of like requiring an explicit `forall`.)

Alright, now we've got something closer, and the compiler is upset that we haven't given `type Item` and `next`. Let's go ahead and return a `u32` again:

```
type Item = u32;
fn next(&mut self) -> Option<u32> {
    unimplemented!()
}
```

The compiles and runs, and then crashes because of the `unimplemented!`. Great, progress! The trick here is we want to ask the underlying `Iterator` for its next value. We'll do this with some explicit pattern matching (functional programmers: yes, there's a `map` method on `Option` we could use here).

```
fn next(&mut self) -> Option<u32> {
    match self.iter.next() {
        None => None,
        Some(x) => Some(x * 2),
    }
}
```

Nice enough, but when we compile it, we get told:

```
error[E0599]: no method named `next` found for type `I` in the current scope
 --> foo.rs:8:25
  |
8 |         match self.iter.next() {
  |                          ^^^^ method not found in `I`
  |
  = help: items from traits can only be used if the type parameter is bounded by the
trait
help: the following traits define an item `next`, perhaps you need to restrict type
parameter `I` with one of them:
  |
5 | impl<I: std::iter::Iterator> Iterator for Doubler<I> {
  |      ^^^^^^^^^^^^^^^^^^^^^^
5 | impl<I: std::str::pattern::Searcher> Iterator for Doubler<I> {
  |      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

The compiler knows that we *might* mean the `next` method from `Iterator`. But it doesn't use it. Why, you might ask? Because *we never told the compiler that the implementation exists*! We need to indicate that the `I` parameter must have an `Iterator` implementation.

```
impl<I: Iterator> Iterator for Doubler<I>
```

That's some new syntax, but pretty straightforward: `I` must have an implementation of `Iterator`. Unfortunately, we're not out of the woods quite yet:

```
error[E0369]: binary operation `*` cannot be applied to type `<I as
std::iter::Iterator>::Item`
  --> foo.rs:10:29
   |
10 |             Some(x) => Some(x * 2),
   |                             ^^^^^
   |
   = note: an implementation of `std::ops::Mul` might be missing for `<I as
std::iter::Iterator>::Item`
```

Let's talk this through. `I` is some `Iterator`, we've already established that. And we know that the `x` value we use in `x * 2` will be of whatever type the `Item` associated type for that `I` is. The problem is: we have no idea what it is, or whether or not it can be multiplied!

We've already said we're going to produce `u32`s here, so can we just enforce that the `Item` is a `u32`? Yes!

```
impl<I: Iterator<Item=u32>> Iterator for Doubler<I>
```

Whew, our code works!

4.11. Alternative syntax: where

As our constraints get more complex, shoving them all into the parameters at the beginning of `impl` starts to feel crowded. You can alternatively use `where` for this:

```
impl<I> Iterator for Doubler<I>
    where
    I: Iterator<Item=u32>
```

There's a subjective point at which people decide to make that transition. Personally, I prefer consistency over brevity, and almost always use `where`. Your mileage may vary. You should be aware of both syntaxes for reading other people's code.

4.12. Not just u32

It's a bit weird that we're tied down to `u32`s. What if we change our `main` funciton to have:

```
let orig_iter = 1..11u64;
```

We'll get a compiler error:

```
error[E0271]: type mismatch resolving `<std::ops::Range<u64> as
std::iter::Iterator>::Item == u32`
  --> foo.rs:23:14
   |
23 |     for i in doubled_iter {
   |              ^^^^^^^^^^^^ expected u64, found u32
   |
   = note: expected type `u64`
              found type `u32`
```

It's possible to relax this, but it starts to get more complicated. But let's do it! We'll need to remove all of the references to `u32` in our implementation. Here's a first stab at this:

```
impl<I> Iterator for Doubler<I>
    where
    I: Iterator
{
    type Item = ???;
    fn next(&mut self) -> Option<Self::Item> {
        match self.iter.next() {
            None => None,
            Some(x) => Some(x * 2),
        }
    }
}
```

I've replaced `Option<u32>` with `Option<Self::Item>`, and removed the `<Item = u32>` on the `I: Iterator`. But what should we use for `type Item =`? We want it to be the same type as the underlying iterator's `Item`... so let's just say that!

```
type Item = I::Item;
```

That works! We're still not compiling though, because Rust doesn't know that it can perform multiplication on `I::Item`. Fortunately, there's a trait for things that can be multiplied called `Mul`. We can add in:

```
where
I: Iterator,
I::Item: std::ops::Mul,
```

New error message:

```
error[E0308]: mismatched types
  --> foo.rs:14:29
   |
14 |             Some(x) => Some(x * 2),
   |                             ^^^^^ expected std::iter::Iterator::Item, found
std::ops::Mul::Output
   |
   = note: expected type `<I as std::iter::Iterator>::Item`
              found type `<<I as std::iter::Iterator>::Item as std::ops::Mul>::Output`
```

It turns out that `Mul` has an associated type for its output. This is useful for expressing more complicated relationships at the type level. For example, we can define types for `Force`, `Mass`, and `Acceleration`, and then define a `Mul` implementation that says `Mass` times `Acceleration` produces a `Force`.

That's a wonderful feature, but it's getting in our way here. We want to say "the output should be the same as the item itself." Fair enough:

```
impl<I> Iterator for Doubler<I>
    where
    I: Iterator,
    I::Item: std::ops::Mul<Output=I::Item>,
```

And now:

```
error[E0308]: mismatched types
  --> foo.rs:14:33
   |
14 |             Some(x) => Some(x * 2),
   |                                 ^ expected associated type, found integer
   |
   = note: expected type `<I as std::iter::Iterator>::Item`
              found type `{integer}`
   = note: consider constraining the associated type `<I as
std::iter::Iterator>::Item` to `{integer}` or calling a method that returns `<I as
std::iter::Iterator>::Item`
   = note: for more information, visit https://doc.rust-lang.org/book/ch19-03-
advanced-traits.html
```

Ugh. We have `2`, which can be *some* integral type. But we don't know that `Item` is an integral type! I'm not aware of a way to give a constraint that allows this code to work (if someone knows better, please let me know and I'll update this text). One trick that *does* work is to upcast from a `u8` using the `From` trait, which performs safe numeric conversions (which cannot over or underflow):

```
impl<I> Iterator for Doubler<I>
    where
    I: Iterator,
    I::Item: std::ops::Mul<Output=I::Item> + From<u8>,
{
    type Item = I::Item;
    fn next(&mut self) -> Option<Self::Item> {
        match self.iter.next() {
            None => None,
            Some(x) => Some(x * From::from(2u8)),
        }
    }
}
```

Whew, that's finally over!

Exercise 4

An easier approach to the above is to use `x + x` instead of `x * 2`. Rewrite the iterator to do that. One hint: the compiler won't know that it's allowed to make copies of that type unless you tell it via the appropriately-named trait.

Solution

```
impl<I> Iterator for Doubler<I>
    where
    I: Iterator,
    I::Item: std::ops::Add<Output=I::Item> + Copy,
{
    type Item = I::Item;
    fn next(&mut self) -> Option<Self::Item> {
        match self.iter.next() {
            None => None,
            Some(x) => Some(x + x),
        }
    }
}
```

4.13. Recap

That was a lot! But hopefully it gets the idea across of how iterators work. We can write highly generic adapters that work with many kinds of input. You could apply `Doubler` to the range iterator as we did. You could apply it to the `Empty` we defined earlier. Or to dozens of other things.

You may notice that the types of these iterators seem to grow as you add more things to them. That's absolutely true, and it's also by design. By representing the full type statically, the compiler is able to see all of the actions that are being performed in a pipeline of iterator operations, and optimize things very well.

4.14. More idiomatic

The `Doubler` we wrote was not idiomatic at all. Let's do it the *real* way:

```
fn main() {
    for i in (1..11).map(|x| x * 2) {
        println!("{}", i);
    }
}
```

The `Iterator` trait includes many helper methods, so you can chain up large sets of actions like that:

```
fn main() {
    for i in (1..11).skip(3).map(|x| x + 1).filter(|x| x % 2 == 0) {
        println!("{}", i);
    }
}
```

You could of course write something like this as a for loop in C/C++, but:

- It would be harder to see the logic
- It would be harder to extend in the future
- It wouldn't be faster: the Rust compiler will optimize case like this down to the same hand-rolled loop you may write

4.15. Collecting results

You can collect the results from an iterator in a vector:

```
fn main() {
    let my_vec: Vec<u32> = (1..11).collect();
    println!("{:?}", my_vec);
}
```

The type annotation is necessary here, since `collect` can work on many different datatypes.

Exercise 5

Use the `fold` method to get the sum of the numbers from 1 to 10. Extra credit: write a helper `sum` function.

Solution

The `fold` method takes two parameters: the initial value, and a function for adding the running total with the next value. One approach using closures is:

```
fn main() {
    let res = (1..11).fold(0, |x, y| x + y);
    println!("{}", res);
}
```

Another approach is to directly refer to the addition function. Remember how there was a `Mul` trait for the `*` operator? There's also an `Add` trait for addition:

```
fn main() {
    let res = (1..11).fold(0, std::ops::Add::add);
    println!("{}", res);
}
```

As for writing our own `sum` function: we'll end up back in the situation where things are generic and we have to provide appropriate traits. We'll follow a similar approach with using `From` and a `u8`:

```
fn sum<I>(iter: I) -> I::Item
    where
    I: Iterator,
    I::Item: std::ops::Add<Output=I::Item> + From<u8>,
{
    iter.fold(From::from(0u8), std::ops::Add::add)
}
```

4.16. Next Chapter

We've been dancing around closures for quite a while now, including in that last exercise. We now know enough to attack them head on. We'll do so in the next chapter.

You're now at a point where you can write some real Rust applications and start cutting your teeth. I'd recommend finding a few play tasks you want to experiment with. Exercism may be a good choice if you're looking for some challenges.

5. Rule of Three - Parameters, Iterators, and Closures

In this chapter, we're going to cover what I'm dubbing the "rule of three," which applies to function parameters, iterators, and closures. We've already seen this rule applied to function parameters, but didn't discuss it so explicitly. We'll expand on parameters, and use that to launch into new information on both iterators and closures.

5.1. Types of parameters

The first thing I want to deal with is a potential misconception. This may be one of those "my brain has been scrambled by Haskell" misconceptions that imperative programmers won't feel, so apologies if I'm just humoring myself and other Haskellers.

Do these two functions have the same type signature?

```
fn foo(mut person: Person) { unimplemented!() }
fn bar(person: Person) { unimplemented!() }
```

The Haskeller in me screams "they're different!" However, they're *exactly the same*. The *inner mutability* of the `person` variable in the function is *irrelevant* to someone calling the function. The caller of the function will move the `Person` value into the function, regardless of whether the value can be mutated or not. We've already seen a hint of this: the fact that we can pass an immutable value to a function like `foo`:

```
fn main() {
    let alice = Person { name: String::from("Alice"), age: 30 };
    foo(alice); // it works!
}
```

With that misconception out of the way, let's consider two other similar functions:

```
fn baz(person: &Person) { unimplemented!() }
fn bin(person: &mut Person) { unimplemented!() }
```

Firstly, it's pretty easy to say that both `baz` and `bin` have different signatures than `foo`. These are taking references to a `Person`, not a `Person` itself. But what about `baz` vs `bin`? Are they the same or different? You may be tempted to follow the same logic as `foo` vs `bar` and decide that the `mut` is an internal detail of the function. But this isn't true! Observe:

```
fn main() {
    let mut alice = Person { name: String::from("Alice"), age: 30 };
    baz(&alice); // this works
    bin(&alice); // this fails
    bin(&mut alice); // but this works
}
```

The first call to `bin` will not compile, because `bin` requires a *mutable* reference, and we've provided an *immutable* reference. We need to use the second version of the call. And not only does this have a *syntactic* difference, but a *semantic* difference as well: we've taken a mutable reference, which means we can have no other references at the same time (remember our borrow rules from lesson 2).

The upshot of this is that there are three different ways we can pass a value into a function which appear at the type level:

- Pass by value (move semantics), like `foo`
- Pass by immutable reference, like `baz`
- Pass by mutable reference, like `bin`

In addition, orthogonally, the variable that captures that parameters can itself be either immutable or mutable.

5.1.1. Mutable vs immutable pass-by-value

This one is relatively easy to see. What extra functionality do we get by having a mutable pass-by-value? The ability to mutate the value of course! Let's look at two different ways of implementing a birthday function, which increases someone's age by 1.

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u32,
}

fn birthday_immutable(person: Person) -> Person {
    Person {
        name: person.name,
        age: person.age + 1,
    }
}

fn birthday_mutable(mut person: Person) -> Person {
    person.age += 1;
    person
}

fn main() {
    let alice1 = Person { name: String::from("Alice"), age: 30 };
    println!("Alice 1: {:?}", alice1);
    let alice2 = birthday_immutable(alice1);
    println!("Alice 2: {:?}", alice2);
    let alice3 = birthday_mutable(alice2);
    println!("Alice 3: {:?}", alice3);
}
```

Some important takeaways:

- Our `_immutable` implementation follows a more functional idiom, creating a new `Person` value by deconstructing the original `Person` value. This works just fine in Rust, but is not idiomatic, and potentially less efficient.
- We call both versions of this function in exactly the same way, reinforcing the claim that these two functions have the same signature.
- You cannot reuse the `alice1` or `alice2` values in `main`, since they've been moved during their calls.
- `alice2` is an immutable variable, but it still gets passed in to a function which mutates it.

5.1.2. Mutable vs immutable pass-by-mutable-reference

This one already gets significantly harder to observe, which indicates a simple fact of Rust: *it's unusual to want a mutable variable for references*. The example below is very contrived, and requires playing with the more advanced concept of explicit lifetime parameters to even make it make sense. But it does demonstrate the difference between where the `mut` appears.

Before we dive in: parameters that begin with a single quote (') are *lifetime parameters*, and indicate how long a reference needs to live. In the examples below, we're saying "the two references must have the same lifetime." We won't cover this in more detail here, at least not yet. If

you want to learn about lifetimes, please check out the Rust book.

OK, let's see a difference between an immutable variable holding a mutable reference and a mutable variable holding a mutable reference!

```
#[derive(Debug)]
struct Person {
    name: String,
    age: u32,
}

fn birthday_immutable(person: &mut Person) {
    person.age += 1;
}

fn birthday_mutable<'a>(mut person: &'a mut Person, replacement: &'a mut Person) {
    person = replacement;
    person.age += 1;
}

fn main() {
    let mut alice = Person { name: String::from("Alice"), age: 30 };
    let mut bob = Person { name: String::from("Bob"), age: 20 };
    println!("Alice 1: {:?}, Bob 1: {:?}", alice, bob);
    birthday_immutable(&mut alice);
    println!("Alice 2: {:?}, Bob 2: {:?}", alice, bob);
    birthday_mutable(&mut alice, &mut bob);
    println!("Alice 3: {:?}, Bob 3: {:?}", alice, bob);
}

// does not compile
fn birthday_immutable_broken<'a>(person: &'a mut Person, replacement: &'a mut Person)
{
    person = replacement;
    person.age += 1;
}
```

`birthday_immutable` is fairly simple. We have a mutable reference, and we've stored it in an immutable variable. We've completely free to mutate the value pointed to by that reference. The takeaway is: we're mutating the value, *not* the variable, which is remaining the same.

`birthday_mutable` is a contrived, ugly mess, but it demonstrates our point. Here, we take *two* references: a `person`, and a `replacement`. They're both mutable references, but `person` is in a mutable variable. The first thing we do is `person = replacement;`. This changes what our `person` variable is pointing at, and *does not modify* the original value being pointed at by the reference at all. In fact, when compiling this, we'll get a warning that we never used the value passed to `person`:

```
warning: value passed to `person` is never read
```

Notice that we needed to mark both `alice` and `bob` as mutable in `main` in this example. That's because we pass them by mutable reference, which requires that we have the ability to mutate them. This is different from pass-by-value with move semantics, because in our `main` function, we can directly observe the effect of mutating the references we've passed in.

Also notice that we also have a `birthday_immutable_broken` version. As you may guess from the name, it doesn't compile. We cannot change what `person` points to if it is an immutable variable.

Challenge Figure out what the output of this program is going to be before you run it.

5.1.3. Mutable vs immutable pass-by-immutable-reference

I'm not actually going to cover this case, since it's basically the same as the previous one. If you mark a variable as mutable, you can change which reference it holds. Feel free to play around with an example like the one above using immutable references.

5.1.4. Mutable to immutable

Let's point out one final bit:

```
fn needs_mutable(x: &mut u32) {
    *x *= 2;
}

fn needs_immutable(x: &u32) {
    println!("{}", x);
}

fn main() {
    let mut x: u32 = 5;
    let y: &mut u32 = &mut x;
    needs_immutable(y);
    needs_mutable(y);
    needs_immutable(y);
}
```

From what I've told you so far, you should expect this program to fail to compile. `y` is of type `&mut u32`, but we're passing it to `needs_immutable` which requires a `&u32`. Type mismatch, go home!

Not so fast: since the guarantees of a mutable reference are strictly stronger than an immutable reference, you can always use a mutable reference where an immutable was needed. (Hold onto this, it will be important for closures below.)

5.1.5. Summary of the rule of three for parameters

There are three types of parameters:

- Pass by value
- Pass by immutable reference

- Pass by mutable reference

This is what I'm calling the rule of three. The captured variables within a function can either be mutable or immutable, which is orthogonal to the type of the parameter. However, it's by far most common to have a mutable variable with a pass-by-value. Also, at the call site, a variable must be mutable if it is called on a pass by mutable reference functions. Finally, you can use a mutable reference where an immutable was requested.

Exercise 1

Fix the program below so that it outputs the number 10. Ensure that there are no compiler warnings.

```
fn double(mut x: u32) {
    x *= 2;
}

fn main() {
    let x = 5;
    double(x);
    println!("{}", x);
}
```

TIP you'll need to know how to *dereference* a reference, by putting a asterisk (*) in front of the variable.

Solution

```
fn double(x: &mut u32) {
    *x *= 2;
}

fn main() {
    let mut x = 5;
    double(&mut x);
    println!("{}", x);
}
```

Notice that the variable `x` does not need to be mutable, since we're only modifying the value it references.

5.2. Iterators

What's the output of the program below?

```
fn main() {
    let nums = vec![1, 2, 3, 4, 5];
    for i in nums {
        println!("{}", i);
    }
}
```

That's right: it prints the numbers 1 to 5. How about this one?

```
fn main() {
    for i in 1..3 {
        let nums = vec![1, 2, 3, 4, 5];
        for j in nums {
            println!("{},{}", i, j);
        }
    }
}
```

It prints `1,1`, `1,2`, ..., `1,5`, `2,1`, ..., `2,5`. Cool, easy enough. Let's move `nums` a bit. What does this do?

```
fn main() {
    let nums = vec![1, 2, 3, 4, 5];
    for i in 1..3 {
        for j in nums {
            println!("{},{}", i, j);
        }
    }
}
```

Trick question: it doesn't compile!

```
error[E0382]: use of moved value: `nums`
 --> main.rs:4:18
  |
2 |     let nums = vec![1, 2, 3, 4, 5];
  |         ---- move occurs because `nums` has type `std::vec::Vec<i32>`, which does
not implement the `Copy` trait
3 |     for i in 1..3 {
4 |         for j in nums {
  |                  ^^^^
  |                  |
  |                  value moved here, in previous iteration of loop
  |                  help: consider borrowing to avoid moving into the for loop:
`&nums`

error: aborting due to previous error
```

Well, that kind of makes sense. The first time we run through the outer loop, we *move* the `nums` value into the inner loop. Then, we can't use the `nums` value again on the second pass through the loop. OK, logical.

Side note This was one of my personal "mind blown" moments with Rust, realizing how sophisticated lifetime tracking was to work through loops like this. Rust is pretty amazing.

We can go back to our previous version and put `nums` inside the first `for` loop. That means recreating the value each time we pass through that outer `for` loop. For our little vector example, it's not a big deal. But imagine constructing `nums` was expensive. This would be a major overhead!

If we want to avoid the move of `nums`, can we get away with just borrowing it instead? Yes we can!

```
fn main() {
    let nums = vec![1, 2, 3, 4, 5];
    for i in 1..3 {
        for j in &nums {
            println!("{},{}", i, j);
        }
    }
}
```

This works, but I've got a question for you: what's the type of `j`? I've got a sneaky little trick to test out different options. If you throw this in just above the `println!` call, you'll get an error message:

```
let _: u32 = j;
```

However, this will compile just fine:

```
let _: &u32 = j;
```

By iterating over a reference to `nums`, we got a reference to each value instead of the value itself. That makes sense. Can we complete our "rule of three" with mutable references? Yet again, yes!

```
fn main() {
    let nums = vec![1, 2, 3, 4, 5];
    for i in 1..3 {
        for j in &mut nums {
            let _: &mut u32 = j;
            println!("{},{}", i, j);
            *j *= 2;
        }
    }
}
```

Challenges First, there's a compilation error in the program above. Try to catch it before asking the

compiler to help. Second, guess the output of this program before running it.

Our rule of three translates into iterators as well! We have can iterators of values, iterators of references, and iterators of mutable references. Sweet!

5.2.1. New nomenclature

The `Vec` struct has three different methods on it that are relevant to our examples above. Starting with the mutable case, we can replace the line:

```
for j in &mut nums {
```

with

```
for j in nums.iter_mut() {
```

The signature of that method is:

```
pub fn iter_mut(&mut self) -> IterMut<T>
```

Similarly, we've got a `iter()` method that can replace our immutable reference case:

```
fn main() {
    let nums = vec![1, 2, 3, 4, 5];
    for i in 1..3 {
        for j in nums.iter() {
            let _: &u32 = j;
            println!("{},{}", i, j);
        }
    }
}
```

And, finally, what about the iterator of values case? There, the nomenclature is `into_iter`. The idea is that we are *converting* the existing value *into* an iterator, consuming the previous value (the `Vec` in this case) completely. This code won't compile, go ahead and fix it by moving the `let nums` statement.

```
fn main() {
    let nums = vec![1, 2, 3, 4, 5];
    for i in 1..3 {
        for j in nums.into_iter() {
            println!("{},{}", i, j);
        }
    }
}
```

5.2.2. Reexamining for loops

Here's a cool little trick I didn't mention before. `for` loops are a bit more flexible than I'd implied. The `into_iter` method I mention is actually part of a trait, appropriately named `IntoIterator`. Whenever you use `for x in y`, the compiler automatically calls `into_iter()` on `y`. This allows you to loop over types which don't actually have their own implementation of `Iterator`.

Exercise 2

Make this program compile by defining an `IntoIterator` implementation for `InfiniteUnit`. Do *not* define an `Iterator` implementation for it. You'll probably want to define an extra datatype. (Extra credit: also try to find a helper function in the standard library that repeats values.)

```
struct InfiniteUnit;

fn main() {
    let mut count = 0;
    for _ in InfiniteUnit {
        count += 1;
        println!("count == {}", count);
        if count >= 5 {
            break;
        }
    }
}
```

Solution

The (IMO) straightforward solution is:

```
struct InfiniteUnit;

impl IntoIterator for InfiniteUnit {
    type Item = ();
    type IntoIter = InfiniteUnitIter;

    fn into_iter(self) -> Self::IntoIter {
        InfiniteUnitIter
    }
}

struct InfiniteUnitIter;

impl Iterator for InfiniteUnitIter {
    type Item = ();
    fn next(&mut self) -> Option<()> {
        Some(())
    }
}

fn main() {
    let mut count = 0;
    for _ in InfiniteUnit {
        count += 1;
        println!("count == {}", count);
        if count >= 5 {
            break;
        }
    }
}
```

However, if you want to be a bit more clever, there's already a function in the standard library that creates an infinite iterator, called `repeat`. Using that, you can bypass the extra struct here:

```
struct InfiniteUnit;

impl IntoIterator for InfiniteUnit {
    type Item = ();
    type IntoIter = std::iter::Repeat<()>;

    fn into_iter(self) -> Self::IntoIter {
        std::iter::repeat(())
    }
}

fn main() {
    let mut count = 0;
    for _ in InfiniteUnit {
        count += 1;
        println!("count == {}", count);
        if count >= 5 {
            break;
        }
    }
}
```

5.2.3. Summary of the rule of three for iterators

Just like function parameters, iterators come in three flavors, corresponding to the following naming scheme:

- `into_iter()` is an iterator of values, with move semantics
- `iter()` is an iterator of immutable references
- `iter_mut()` is an iterator of mutable references

Only `iter_mut()` requires that the original variable itself be mutable.

5.3. Closures

We've danced around closures a bit throughout this book so far. Closures are like functions, in that they can be called on some arguments. Closures are unlike functions in that they can capture values from the local scope. We'll demonstrate this in an example, after a word of warning.

One word of warning If you're coming from a non-functional programming background, you'll likely find closures in Rust very powerful, and surprisingly common in library usage. If you come from a functional programming background, you'll likely be annoyed at how much you have to think about ownership of data when working with closures. As a Haskeller, this is still the aspect of Rust I most often get caught on. I promise, the trade-offs in the design are logical and necessary to achieve Rust's goals, but it can feel a bit onerous when compared to Haskell, or even compared to Javascript.

Alright, back to our function vs closure thing. Did you know that you can define a function *inside another function*?

```
fn main() {
    fn say_hi() {
        let msg: &str = "Hi!";
        println!("{}", msg);
    };
    say_hi();
    say_hi();
}
```

That's pretty nifty. Let's slightly refactor that:

```
fn main() {
    let msg: &str = "Hi!";
    fn say_hi() {
        println!("{}", msg);
    };
    say_hi();
    say_hi();
}
```

Unfortunately, Rust doesn't like that very much:

```
error[E0434]: can't capture dynamic environment in a fn item
 --> main.rs:4:24
  |
4 |         println!("{}", msg);
  |                        ^^^
  |
  = help: use the `|| { ... }` closure form instead

error: aborting due to previous error
```

Fortunately, the compiler tells us *exactly* how to fix it: use a closure! Let's rewrite that:

```
fn main() {
    let msg: &str = "Hi!";
    let say_hi = || {
        println!("{}", msg);
    };
    say_hi();
    say_hi();
}
```

We now have a closure (introduced by `||`) which takes 0 arguments. And everything just works.

Note You can shorten this a bit with `let say_hi = || println!("{}", msg);`, which is more idiomatic.

Exercise 3

Rewrite the above so that instead of taking 0 arguments, `say_hi` takes a single argument: the `msg` variable. Then try out the `fn` version again.

Solution

The closure version:

```
fn main() {
    let msg: &str = "Hi!";
    let say_hi = |msg| println!("{}", msg);
    say_hi(msg);
    say_hi(msg);
}
```

And the function version:

```
fn main() {
    let msg: &str = "Hi!";
    fn say_hi(msg: &str) {
        println!("{}", msg);
    }
    say_hi(msg);
    say_hi(msg);
}
```

Since `say_hi` is no longer referring to any variables in the local scope, it doesn't need to be a closure.

5.4. The type of a closure

What exactly is the type of `say_hi`? I'm going to use an ugly trick to get the compiler to tell us: give it the *wrong* type, and then try to compile. It's probably safe to assume that a closure isn't a `u32`, so let's try this:

```
fn main() {
    let msg: &str = "Hi!";
    let say_hi: u32 = |msg| println!("{}", msg);
}
```

And we get the error message:

```
error[E0308]: mismatched types
 --> main.rs:3:23
  |
3 |     let say_hi: u32 = |msg| println!("{}", msg);
  |                       ^^^^^^^^^^^^^^^^^^^^^^^^^ expected u32, found closure
  |
  = note: expected type `u32`
             found type `[closure@main.rs:3:23: 3:48]`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
```

`[closure@main.rs:3:23: 3:48]` looks like a weird type... but let's just give it a shot and see what happens:

```
fn main() {
    let msg: &str = "Hi!";
    let say_hi: [closure@main.rs:3:23: 3:48] = |msg| println!("{}", msg);
}
```

But the compiler shoots us down:

```
error: expected one of `!`, `(`, `+`, `::`, `;`, `<`, or `]`, found `@`
 --> main.rs:3:25
  |
3 |     let say_hi: [closure@main.rs:3:23: 3:48] = |msg| println!("{}", msg);
  |         ------          ^ expected one of 7 possible tokens here
  |         |
  |         while parsing the type for `say_hi`

error: aborting due to previous error
```

Oh well, that isn't a valid type. What exactly is the compiler telling us then?

5.5. Anonymous types

The types of closures are anonymous in Rust. We cannot directly refer to them at all. But this leaves

us in a bit of a pickle. What if we want to pass a closure into another function? For example, let's try out this program:

```
fn main() {
    let say_message = |msg: &str| println!("{}", msg);
    call_with_hi(say_message);
    call_with_hi(say_message);
}

fn call_with_hi<F>(f: F) {
    f("Hi!");
}
```

We've added a type annotation on the `msg` parameter in the closure. These are generally optional in closures, unless type inference fails. And with our current broken code, type inference is definitely failing. We're including it now to get better error messages later.

We also now have a type parameter, called `F`, for the closure we're passing in. We don't know anything about `F` right now, but we're going to just try using it in a function call manner. If we compile this, we get:

```
error[E0618]: expected function, found `F`
 --> main.rs:8:5
  |
7 | fn call_with_hi<F>(f: F) {
  |                    - `F` defined here
8 |     f("Hi!");
  |     ^-------
  |     |
  |     call expression requires function

error: aborting due to previous error

For more information about this error, try `rustc --explain E0618`.
```

OK, fair enough: the compiler doesn't know that `F` is a function. It's time to finally introduce the magic that will make this compile: the `Fn` trait!

```
fn call_with_hi<F>(f: F)
    where F: Fn(&str) -> ()
{
    f("Hi!");
}
```

We've now put a constraint on `F` that it must be a function, which takes a single argument of type `&str`, and returns a unit value. Actually, returning unit values is the default, so we can just omit that bit:

```
fn call_with_hi<F>(f: F)
    where F: Fn(&str)
{
    f("Hi!");
}
```

Another nifty thing about the `Fn` trait is that it doesn't just apply to closures. It works on regular ol' functions too:

Exercise 4

Rewrite `say_message` as a function *outside* of `main` and make the program above compile.

Solution

```
fn main() {
    call_with_hi(say_message);
    call_with_hi(say_message);
}

fn say_message(msg: &str) {
    println!("{}", msg);
}

fn call_with_hi<F>(f: F)
    where F: Fn(&str)
{
    f("Hi!");
}
```

This was a bit boring, since `say_message` isn't actually a closure. Let's change that a bit.

```
fn main() {
    let name = String::from("Alice");
    let say_something = |msg: &str| println!("{}, {}", msg, name);
    call_with_hi(say_something);
    call_with_hi(say_something);
    call_with_bye(say_something);
    call_with_bye(say_something);
}

fn call_with_hi<F>(f: F)
    where F: Fn(&str)
{
    f("Hi");
}

fn call_with_bye<F>(f: F)
    where F: Fn(&str)
{
    f("Bye");
}
```

5.6. Mutable variables

Remember the good old days of visitor counters on webpages? Let's recreate that beautiful experience!

```
fn main() {
    let mut count = 0;

    for _ in 1..6 {
        count += 1;
        println!("You are visitor #{}", count);
    }
}
```

That works, but it's so boring! Let's make it more interesting with a closure.

```
fn main() {
    let mut count = 0;
    let visit = || {
        count += 1;
        println!("You are visitor #{}", count);
    };

    for _ in 1..6 {
        visit();
    }
}
```

The compiler disagrees:

```
error[E0596]: cannot borrow `visit` as mutable, as it is not declared as mutable
 --> main.rs:9:9
  |
3 |     let visit = || {
  |         ----- help: consider changing this to be mutable: `mut visit`
...
9 |         visit();
  |         ^^^^^ cannot borrow as mutable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0596`.
```

Huh... what? Apparently calling a function counts as borrowing it. Fine, that explains why we're allowed to call it multiple times. But now we need to borrow it *mutably* for some reason. How come?

That reason is fairly simple: `visit` has captured and is mutating a local variable, `count`. Therefore, any borrow of it is implicitly mutably borrowing `count` as well. Logically, this makes sense. But how about at the type level? How is the compiler tracking this mutability? To see that, let's extend this a bit further with a helper function:

```
fn main() {
    let mut count = 0;
    let visit = || {
        count += 1;
        println!("You are visitor #{}", count);
    };

    call_five_times(visit);
}

fn call_five_times<F>(f: F)
where
    F: Fn(),
{
    for _ in 1..6 {
        f();
    }
}
```

We get the error message:

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only
implements `FnMut`
```

Nice! Rust has two different traits for functions: one which covers functions that don't mutate their environment (`Fn`), and one for functions which do mutate their environment (`FnMut`). Let's try modifying our `where` to use `FnMut` instead. We get one more error message:

```
error[E0596]: cannot borrow `f` as mutable, as it is not declared as mutable
  --> main.rs:16:9
   |
11 | fn call_five_times<F>(f: F)
   |                        - help: consider changing this to be mutable: `mut f`
...
16 |         f();
   |         ^ cannot borrow as mutable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0596`.
```

Calling this mutating function requires taking a mutable borrow of the variable, and that requires defining the variable as mutable. Go ahead and stick a `mut` in front of the `f: F`, and you'll be golden.

5.7. Multiple traits?

Is this closure a `Fn` or a `FnMut`?

```
|| println!("Hello World!");
```

Well, it doesn't modify any variables in the local scope, so presumably it's an `Fn`. Therefore, passing it to `call_five_times`--which expects a `FnMut`--should fail, right? Not so fast, it works just fine! Go ahead and add this line to the program above and prove it to yourself:

```
call_five_times(|| println!("Hello World!"));
```

Every value which is a `Fn` is *automatically* an `FnMut`. This is similar to what happens with a function parameter: if you have a mutable reference, you can automatically use it as an immutable reference, since the guarantees of a mutable reference are stronger. Similarly, if we're using a function in such a way that it's safe even if the function is mutable (`FnMut`), it's certainly safe to do the same thing with an immutable function (`Fn`).

Does this sound a bit like subtyping? Good, it should :)

5.8. The rule of three?

If you've noticed, we now have two different types of functions, in a lesson entitled "the rule of three." What could possibly be coming next? We've seen functions that can be called multiple times in an immutable context, kind of like immutable references. We've seen functions that can be called multiple times in a mutable context, kind of like mutable references. That just leaves one thing... call by value/move semantics!

We're going to define a closure that moves a local variable around. We're going to go back to use a `String` instead of a `u32`, to avoid the fact that a `u32` is `Copy`able. And we're going to use a weird bit of magic in the middle to force things to be moved instead of being treated as references. We'll go into gory detail on that trick later, and see alternatives.

```
fn main() {
    let name = String::from("Alice");

    let welcome = || {
        let name = name; // here's the magic
        println!("Welcome, {}", name);
    };

    welcome();
}
```

Alright, `name` is moved into the `welcome` closure. This is forced with the `let name = name;` bit. Still not 100% convinced that `name` was actually moved in? Watch this:

```
fn main() {
    let name1 = String::from("Alice");

    let welcome = || {
        let mut name2 = name1;
        name2 += " and Bob";
        println!("Welcome, {}", name2);
    };

    welcome();
}
```

`name1` is defined as *immutable*. But `name2` is mutable, and we do in fact successfully mutate it. This can only happen if we pass by value instead of by reference. Want further proof? Try to use `name1` again after we've defined `welcome`.

5.9. The third function trait

Let's complete our rule of three. Remember our `call_five_times`? Let's use it on `welcome`:

```
fn main() {
    let name = String::from("Alice");

    let welcome = || {
        let mut name = name;
        name += " and Bob";
        println!("Welcome, {}", name);
    };

    call_five_times(welcome);
}

fn call_five_times<F>(f: F)
where
    F: Fn(),
{
    for _ in 1..6 {
        f();
    }
}
```

And we get a brand new error message, this time referencing `FnOnce`:

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only
implements `FnOnce`
  --> main.rs:4:19
   |
4  |     let welcome = || {
   |                   ^^ this closure implements `FnOnce`, not `Fn`
5  |         let mut name = name;
   |                        ---- closure is `FnOnce` because it moves the variable
`name` out of its environment
...
10 |     call_five_times(welcome);
   |     --------------- the requirement to implement `Fn` derives from here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0525`.
```

Replacing `Fn()` with `FnOnce()` should fix the compilation, right? Wrong!

```
error[E0382]: use of moved value: `f`
  --> clo4.rs:18:9
   |
13 | fn call_five_times<F>(f: F)
   |                    -  - move occurs because `f` has type `F`, which does not
implement the `Copy` trait
   |                    |
   |                    consider adding a `Copy` constraint to this type argument
...
18 |         f();
   |         ^ value moved here, in previous iteration of loop

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

Our loop ends up calling `f` multiple times. But each time we call `f`, we're moving the value. Therefore, the function can *only be called once*. Maybe that's why they named it `FnOnce`.

Let's rewrite this to have a helper function that only calls things once:

```
fn main() {
    let name = String::from("Alice");

    let welcome = || {
        let mut name = name;
        name += " and Bob";
        println!("Welcome, {}", name);
    };

    call_once(welcome);
}

fn call_once<F>(f: F)
where
    F: FnOnce(),
{
    f();
}
```

That works just fine. Hurrah!

5.10. Further function subtyping

Previously, we said that every `Fn` is also an `FnMut`, since anywhere you can safely call a mutable function, you can also call an immutable function. It turns out that every `Fn` and every `FnMut` are also `FnOnce`s, because any context you can guarantee the function will only be called once is safe for running functions with mutable or immutable environments.

5.11. The move keyword

There's a subtle point we're about to get into, which I didn't understand till I wrote this lesson (thanks to Sven Marnach for the explanation there). The Rust by Example section on closures was the best resource for helping it all click for me. I'll do my best here explaining it myself.

Functions accept parameters explicitly, complete with type signatures. You're able to explicitly state whether a parameter is pass by value, mutable reference, or immutable reference. Then, when you use it, you're able to choose any of the weaker forms available. For example, if you pass a parameter by mutable reference, you can later use it by immutable reference. However, you *cannot* use it by value:

```
fn pass_by_value(_x: String) {}
fn pass_by_ref(_x: &String) {}

fn pass_by_mut_ref(x: &mut String) {
    pass_by_ref(x); // that's fine
    pass_by_value(*x); // that's a paddlin'
}

fn main() {}
```

Closures accept parameters, but they make the type annotations optional. If you omit them, they are implicit. In addition, closures allow you to capture variables. These never take a type annotation; they are *always* implicit. Nonetheless, there needs to be some concept of how these values were captured, just like we need to know how parameters are passed into a function.

How a value is captured will imply the same set of borrow rules we're used to in Rust, in particular:

- If by reference, then other references can live concurrently with the closure
- If by mutable reference, then as long as the closure is alive, no other references to the values can exist. However, once the closure is dropped, other references can exist again.
- If by value, then the value cannot be used by anything ever again. (This automatically implies that the closure *owns* the value.)

However, there's an important and (dare I repeat myself) subtle distinction between closures and functions:

Closures can own data, functions cannot

Sure, if you pass by value to a function, the function call takes ownership of the data during execution. But closures are different: the closure *itself* can own data, and use it while it is being called. Let's demonstrate:

```
fn main() {
    // owned by main
    let name_outer = String::from("Alice");

    let say_hi = || {
        // force a move, again, we'll get smarter in a second
        let name_inner = name_outer;
        println!("Hello, {}", name_inner);
    };

    // main no longer owns name_outer, try this:
    println!("Using name from main: {}", name_outer); // error!

    // but name_inner lives on, in say_hi!
    say_hi(); // success
}
```

Try as you might, you could not achieve the same thing with a plain old function, you'd need to keep `name_outer` alive separately and then pass it in.

Alright, let's get to that smarter way to force a move. In the closure above, we have `let name_inner = name_outer;`. This forces the closure to use `name_outer` by value. Since we use by value, we can only call this closure once, since it fully consumes `name_outer` on the first call. (Go ahead and try adding a second `say_hi()` call.) But in reality, we're only using the name by immutable reference inside the closure. We *should* be able to call it multiple times. If we skip the forced use by value, we can use by reference, leaving the `name_outer` in the original scope:

```
fn main() {
    // owned by main
    let name_outer = String::from("Alice");

    let say_hi = || {
        // use by reference
        let name_inner = &name_outer;
        println!("Hello, {}", name_inner);
    };

    // main still owns name_outer, this is fine
    println!("Using name from main: {}", name_outer); // success

    // but name_inner lives on, in say_hi!
    say_hi(); // success
    say_hi(); // success
}
```

However, if we change things around a bit, so that `name_outer` goes out of scope before `say_hi`, everything falls apart!

```
fn main() {
    let say_hi = { // forcing the creation of a smaller scope
        // owned by the smaller scope
        let name_outer = String::from("Alice");

        // doesn't work, closure outlives captured values
        || {
            // use by reference
            let name_inner = &name_outer;
            println!("Hello, {}", name_inner);
        }
    };

    // syntactically invalid, name_outer isn't in this scope
    //println!("Using name from main: {}", name_outer); // error!

    say_hi();
    say_hi();
}
```

What we need is some way to say: I'd like the closure to own the values it captures, but I don't want to have to force a use by value to do it. That will allow a closure to outlive the original scope of the value, but still allow a closure to be called multiple times. And to do that, we introduce the `move` keyword:

```
fn main() {
    let say_hi = { // forcing the creation of a smaller scope
        // owned by the smaller scope
        let name_outer = String::from("Alice");

        // now it works!
        move || {
            // use by reference
            let name_inner = &name_outer;
            println!("Hello, {}", name_inner);
        }
    };

    // syntactically invalid, name_outer isn't in this scope
    //println!("Using name from main: {}", name_outer); // error!

    say_hi();
    say_hi();
}
```

The ownership of `name_outer` passes from the original scope to the closure itself. We still only use it by reference, and therefore we can call it multiple times. Hurrah!

One final bit here. Using the `move` keyword like this moves all captured variables into the closure, and therefore they cannot be used after the closure. For example, this will fail to compile:

```
fn main() {
    let name = String::from("Alice");
    let _ = move || println!("Hello, {}", name);
    println!("Using name from main: {}", name); // error!
}
```

5.12. Reluctant Rust

Alright, one final point before we sum things up and dive into examples. The type of capture is implicit in a closure. How does Rust decide whether to capture by value, mutable reference, or immutable reference. I like to think of Rust as being reluctant here: it strives to capture the weakest way possible. To paraphrase the Rust by Example book:

Closures will preferentially capture by immutable reference, then by mutable reference, and only then by value.

In our previous examples with `let name_inner = name_outer;`, we forced Rust to capture by value. However, it doesn't like doing that, and will instead capture by reference (mutable or immutable) if it can get away with that. It does this based on the strongest kind of usage for that value. That is:

- If any part of the closure uses a variable by value, it must be captured by value.
- Otherwise, if any part of the closure uses a variable by mutable reference, it must be captured by mutable reference.
- Otherwise, if any part of the closure uses a variable by immutable reference, it must be captured by immutable reference.

It does this reluctant capturing *even if it causes the program to fail to compile*. Capturing by reference instead of value can cause lifetime issues, as we've seen previously. However, Rust does not look at the full context of the usage of the closure to determine how to capture, it only looks at the body of the closure itself.

But, since there are many legitimate cases where we want to force a capture by value to solve lifetime issues, we have the `move` keyword to force the issue.

Side note It may be a little annoying at times that Rust doesn't just look at your program as a whole and guess that you want that `move` added. However, I think it's a great decision in the language: that kind of "do what I mean" logic is fragile and often times surprising.

5.13. Recap: ownership, capture, and usage

To recap the salient points:

- Within a closure, a variable can be used by value, mutable reference, or immutable reference
- In addition, all variables captured by a closure can be captured by value, by mutable reference,

or by immutable reference

- We cannot use a variable in a stronger way than it was captured. If it was captured by mutable reference, it can be used by immutable reference, but not by value.
- To solve lifetime issues, we can force a closure to capture by value with the `move` keyword.
- Short of the `move` keyword, Rust will be reluctant, and capture in the weakest way allowed by the body of the closure.
- Regarding the traits of closures:
  - If a closure uses anything by value, then the closure is a `FnOnce`
  - Otherwise, if a closure uses anything by mutable reference, then the closure is a `FnMut`, which automatically implies `FnOnce` as well
  - Otherwise, a closure is a `Fn`, which automatically implies both `FnMut` and `FnOnce`

I consider the points above complicated enough that I'm included a number of further examples to help hammer the points home. These are inspired heavily by the Rust by Example examples.

For all of the examples below, I'm going to assume the presence of the following three helper functions in the source:

```
fn call_fn<F>(f: F) where F: Fn() {
    f()
}

fn call_fn_mut<F>(mut f: F) where F: FnMut() {
    f()
}

fn call_fn_once<F>(f: F) where F: FnOnce() {
    f()
}
```

5.13.1. Examples

Consider this `main` function:

```
fn main() {
    let name = String::from("Alice");
    let say_hi = || println!("Hello, {}", name);
    call_fn(say_hi);
    call_fn_mut(say_hi);
    call_fn_once(say_hi);
}
```

`name` lives longer than `say_hi`, and therefore there's no problem with the closure keeping an immutable reference to `name`. Since it only has immutable references to the environment and consumes no values, `say_hi` is a `Fn`, `FnMut`, and `FnOnce`, and the code above compiles.

```
// bad!
fn main() {
    let say_hi = {
        let name = String::from("Alice");
        || println!("Hello, {}", name)
    };
}
```

By contrast, this example won't compile. `name` will go out of scope once we leave the curly braces. However, our closure is capturing it by reference, and so the reference outlives value. We could do our trick from before of forcing it to capture by value:

```
fn main() {
    let say_hi = {
        let name = String::from("Alice");
        || {
            let name = name;
            println!("Hello, {}", name)
        }
    };
    //call_fn(say_hi);
    //call_fn_mut(say_hi);
    call_fn_once(say_hi);
}
```

But this only implements a `FnOnce`, since the value is captured and consumed, preventing it from being run again. There's a better way! Instead, we can force the closure to take ownership of `name`, but still capture by reference:

```
fn main() {
    let say_hi = {
        let name = String::from("Alice");
        move || println!("Hello, {}", name)
    };
    call_fn(&say_hi);
    call_fn_mut(&say_hi);
    call_fn_once(&say_hi);
}
```

Now we're back to having a `Fn`, `FnMut`, and `FnOnce`! To avoid the `say_hi` value itself from being moved with each call, we now pass a reference to the `call_fn` functions. I believe (though am not 100% certain) that this wasn't necessary in the first example since, above, there was no captured environment, and therefore the closure could be `Copy`ed. This closure, with a captured environment, cannot be `Copy`ed`.

```
fn main() {
    let say_hi = {
        let name = String::from("Alice");
        || std::mem::drop(name)
    };
    //call_fn(say_hi);
    //call_fn_mut(say_hi);
    call_fn_once(say_hi);
}
```

This example uses the `drop` function to consume `name`. Since we use by value, we must capture by value, and therefore must take ownership of the value. As a result, sticking `move` at the front of the closure is unnecessary, though it will do no harm.

```
fn main() {
    let mut say_hi = {
        let mut name = String::from("Alice");
        move || {
            name += " and Bob";
            println!("Hello, {}", name);
        }
    };
    //call_fn(say_hi);
    call_fn_mut(&mut say_hi);
    call_fn_once(&mut say_hi);
}
```

Using the `+=` operator on a `String` requires a mutable reference, so we're out of the territory of immutable reference capturing. Rust will fall back to capturing via mutable reference. That requires that the `name` also be declared mutable. And since `name` will go out of scope before the closure, we need to `move` ownership to the closure. And since calling `say_hi` will mutate data, we need to put a `mut` on its declaration too.

When we pass `say_hi` to the call functions, we need to use `&mut` to ensure (1) the value isn't moved, and (2) the value can be mutated. Also, `call_fn` is invalid here, since our closure is `FnMut` and `FnOnce`, but *not* `Fn`.

Challenge What will the output of this program be? How many times do we add the string `" and Bob"` to `name`?

```
fn main() {
    let mut name = String::from("Alice");
    let mut say_hi = || {
        name += " and Bob";
        println!("Hello, {}", name);
    };
    //call_fn(say_hi);
    call_fn_mut(&mut say_hi);
    call_fn_once(&mut say_hi);
}
```

We can also avoid the capture by letting the `name` live longer than the closure.

```
fn main() {
    let mut name = String::from("Alice");
    let mut say_hi = || {
        name += " and Bob";
        println!("Hello, {}", name);
    };
    //call_fn(say_hi);
    call_fn_mut(&mut say_hi);
    call_fn_once(&mut say_hi);

    println!("And now name is: {}", name);
}
```

Adding the `println!` at the end, which references `name`, used to be invalid, since `say_hi` is still in scope. This is due to *lexical lifetimes*. But non-lexical lifetimes have been added to the compiler and is enabled by default which makes the above code compile fine. Note that you can also explicitly use braces to denote the scope of the closure:

```
fn main() {
    let mut name = String::from("Alice");
    {
        let mut say_hi = || {
            name += " and Bob";
            println!("Hello, {}", name);
        };
        //call_fn(say_hi);
        call_fn_mut(&mut say_hi);
        call_fn_once(&mut say_hi);
    }

    println!("And now name is: {}", name);
}
```

You can also (perhaps somewhat obviously) use a value in multiple different ways:

```
fn main() {
    let mut name = String::from("Alice");
    let mut say_hi = || {
        println!("Hello, {}", name); // use by ref
        name += " and Bob"; // use by mut ref
        std::mem::drop(name); // use by value
    };
    //call_fn(say_hi);
    //call_fn_mut(say_hi);
    call_fn_once(say_hi);
}
```

In these cases, the most powerful use determines the kind of capture we need. Since we used by value above, we must also capture by value, and therefore must take ownership.

5.14. Which trait to use?

It may be intimidating to try and think through which of these three traits you need. You can usually punt on this and let the compiler yell at you. To quote the Rust book:

> Most of the time when specifying one of the `Fn` trait bounds, you can start with `Fn` and the compiler will tell you if you need `FnMut` or `FnOnce` based on what happens in the closure body.

I'd give a slightly different piece of advice, following the dictum of "be lenient in what you accept." When receiving functions as arguments, the most lenient thing to start with is a `FnOnce`. If your usage turns out to be more restrictive, then listen to the compiler.

For more information on closures as output parameters, see Rust by Example's chapter.

5.15. Summary of the rule of three for closures

Both functions and closures are annotated using the `Fn` family of trait bounds. These form a subtyping relationship, where every `Fn` is also an `FnMut`, and every `FnMut` is also an `FnOnce`.

- `FnOnce` works like pass by value
- `FnMut` works like pass by mutable reference
- `Fn` works like pass by immutable reference

How these captured variables are used by the closure determines which of these three it is. Since functions, by definition, never capture local variables, they are always `Fn`.

Exercise 5

Putting together what we've learned about iterators and closures, modify line 5 below (the one starting with `for i in`) so that the program prints the numbers `2,4,6,..,20` twice.

```
fn main() {
    let nums: Vec<u32> = (1..11).collect();

    for _ in 1..3 {
        for i in nums.map(unimplemented!()) {
            println!("{}", i);
        }
    }
}
```

Solution

The first error message we get is:

```
error[E0599]: no method named `map` found for type `std::vec::Vec<u32>` in the
current scope
 --> main.rs:5:23
  |
5 |         for i in nums.map(unimplemented!()) {
  |                       ^^^
  |
  = note: the method `map` exists but the following trait bounds were not
satisfied:
          `&mut std::vec::Vec<u32> : std::iter::Iterator`
          `&mut [u32] : std::iter::Iterator`
```

Looks like we need to get an `Iterator` out of our `nums`. We have three different choices: `into_iter()`, `iter()`, and `iter_mut()`. Since we need to use the result multiple times, and don't need any mutation, `iter()` seems like the right call. Once we replace `nums.map` with `nums.iter().map`, we can move on to the `unimplemented!()` bit.

We need a closure that will double a number. That's pretty easy: `|x| x * 2`. Plugging that in works! Extra challenge: is that closure a `FnOnce`, `FnMut`, or `Fn`?

5.16. GUIs and callbacks

What better way to tie this all off than by writing a GUI and some callbacks? I'm going to use GTK+ and the wonderful gtk-rs set of crates. Our goal ultimately is to create a GUI with a single button on it. When that button is clicked, a message will be written to a file that says "I was clicked."

For this example, you'll definitely want to use a cargo project. Go ahead and run:

```
$ cargo new clicky
$ cd clicky
```

Now add gtk as a dependency. Within the `[dependencies]` section of the `Cargo.toml`, add the line:

```
gtk = "0.8"
gio = "0.8"
```

And now we're going to rip off the sample code from gtk-rs's website. Put this in your `main.rs` (bonus points if you type it yourself instead of copy-pasting):

```
extern crate gio;
extern crate gtk;

use gio::prelude::*;
use gtk::prelude::*;

use gtk::{Application, ApplicationWindow, Button};

fn main() {
    let application =
        Application::new(Some("com.github.gtk-rs.examples.basic"), Default::default())
            .expect("failed to initialize GTK application");

    application.connect_activate(|app| {
        let window = ApplicationWindow::new(app);
        window.set_title("First GTK+ Program");
        window.set_default_size(350, 70);

        let button = Button::new_with_label("Click me!");
        button.connect_clicked(|_| {
            println!("Clicked!");
        });
        window.add(&button);

        window.show_all();
    });

    application.run(&[]);
}
```

Assuming you've got all of your system libraries set up correctly, running `cargo run` should get you a nice, simple GUI.

If you do have trouble installing the crates, check out gtk-rs's requirements page first.

5.17. Replacing the callback

You may have noticed that sample code already includes a callback, which prints `Clicked!` to stdout each time the button is clicked. That certainly makes our life a little bit easier. Now, inside of that callback, we need to:

- Open up a file
- Write some data to the file

We're going to take a first stab at this without doing any error handling. Instead, we'll use `.unwrap()` on all of the `Result` values, causing our program to `panic!` if something goes wrong. We'll clean that up a bit later.

Searching the standard library for file quickly finds `std::fs::File`, which seems promising. It also seems like the `create` function will be the easiest way to get started. We'll write to `mylog.txt`. The example at the top of that page shows `write_all` (thanks Rust for awesome API docs!). First, try out this bit of code:

```
let mut file = std::fs::File::create("mylog.txt");
file.write_all(b"I was clicked.\n");
```

After addressing exercise 6 below, you'll see this error message:

```
error[E0599]: no method named `write_all` found for type `std::fs::File` in the
current scope
  --> src/main.rs:22:18
   |
22 |             file.write_all(b"I was clicked.\n");
   |                  ^^^^^^^^^ method not found in `std::fs::File`
   |
   = note: the method `write_all` exists but the following trait bounds were not
satisfied:
           `std::fs::File : gio::prelude::OutputStreamExtManual`
   = help: items from traits can only be used if the trait is in scope
help: the following trait is implemented but not in scope; perhaps add a `use` for it:
   |
4  | use std::io::Write;
```

Oh, that's something new. In order to use items from a trait, the trait has to be in scope. Easy enough, we can just add `use std::io::Write;` to our closure:

```
use std::io::Write;
let mut file = std::fs::File::create("mylog.txt");
file.write_all(b"I was clicked.\n");
```

Exercise 6

If you're following along with the code like you should be, you probably got a different error message above, and the code I've provided here doesn't actually fix everything. You need to add an extra method call to convert a `Result<File, Error>` into a `File`. Hint: I mentioned it above.

Solution

You need to add a `.unwrap()` call on the `create` call:

Go ahead and run this program (via `cargo run`), click the button a few times, and close the window. Then look at the contents of `mylog.txt`. No matter how many times you clicked, you'll only get one line of output.

The problem is that each time the callback is called, we call `create` from `File`, which overwrites the old file. One approach here would be to create an appending file handle (awesome bonus exercise for anyone who wants to take it on). We're going to take another approach.

5.18. Share the file

Let's move our `create` call to *outside* of the closure definition. We'll open the file inside the closure passed to the `connect_activate` method, the closure can capture a mutable reference to the `file`, and all will be well in the world.

Unfortunately, the compiler really dislikes this:

```
error[E0596]: cannot borrow `file` as mutable, as it is a captured variable in a `Fn`
closure
  --> src/main.rs:24:13
   |
24 |             file.write_all(b"I was clicked.\n");
   |             ^^^^ cannot borrow as mutable
   |
help: consider changing this to accept closures that implement `FnMut`
  --> src/main.rs:23:32
   |
23 |           button.connect_clicked(|_| {
   |  ________________________________^
24 | |             file.write_all(b"I was clicked.\n");
25 | |         });
   | |_________^

error[E0373]: closure may outlive the current function, but it borrows `file`, which
is owned by the current function
  --> src/main.rs:23:32
   |
23 |         button.connect_clicked(|_| {
   |                                ^^^ may outlive borrowed value `file`
24 |             file.write_all(b"I was clicked.\n");
   |             ---- `file` is borrowed here
   |
```

Or more briefly:

```
cannot borrow `file` as mutable, as it is a captured variable in a `Fn` closure
    help: consider changing this to accept closures that implement `FnMut`
closure may outlive the current function, but it borrows `file`, which is owned by the
current function
```

We can understand both of these by looking at the signature for `connect_clicked`:

```
fn connect_clicked<F: Fn(&Self) + 'static>(&self, f: F) -> SignalHandlerId
```

`connect_clicked` is a method which takes some function `f` of type `F` and returns a `SignalHandlerId`. We're not using that return value, so just ignore it. The function is a `Fn`. Therefore, we're *not* allowed to pass in a `FnMut` or an `FnOnce`. GTK must be allowed to call that function multiple times without the restrictions of a mutable context. So keeping a mutable reference won't work.

The other interesting thing is `+ 'static`. We briefly mentioned lifetime parameters above. `'static` is a special lifetime parameter, which means "can live for the entire lifetime of the program." As one nice example of this, all string literals have type `&'static str`, though we usually just write `&str`.

The problem is that our `file` does *not* have `'static` lifetime. It is created inside the closure passed to

the `connect_activate` method.

So what we're left with is: we need a closure which does not have a mutable reference to local data. How do we do that?

5.19. Move it

We can get the compiler to stop complaining about the lifetime by moving the variable into the closure. Now we're guaranteed that the `file` will live as long as the closure itself, meeting the guarantees demanded by `'static`. To accomplish this, stick `move` in front of the closure.

This still doesn't solve our `Fn` issue, however. How can we allow our callback to be called multiple times after moving the value in?

5.20. Reference counting (hint: nope)

We've reached a point where the normal borrow rules of Rust simply aren't enough. We cannot prove to the compiler that our callback will obey the mutable reference rules: exactly one mutable reference will exist at a given time. These kinds of situations occur often enough that the standard library provides built in support for reference counted types.

Add the following statement to the top of your `main.rs`:

```
use std::rc::Rc;
```

An `Rc` is a single threaded reference counted value. There's also an `Arc` type, which is atomic, and can be used in multithreaded applications. Since GTK is a single-threaded library, we're safe using an `Rc` instead of an `Arc`. One really awesome thing about Rust is that if you make a mistake about this, the compiler can catch you. This is because `Rc` does not implement the `Sync` and `Send` traits. See more in the `Send` documentation.

Anyway, back to our example. We can wrap up our original `file` with reference counting with this:

```
let file = std::fs::File::create("mylog.txt").unwrap();
let file = Rc::new(file);
```

How do we then get access to the underlying `File` to use it? Turns out: we don't need to do anything special. Keeping our original `file.write_all` does what we want. This is because `Rc` implements the `Deref` trait:

```
impl<T> Deref for Rc<T> {
    type Target = T;
    ...
}
```

This means that you can get a reference to a `T` from a `Rc<T>`. Since method call syntax automatically

takes a reference, everything works. Nice.

Well, almost everything:

```
error[E0596]: cannot borrow data in an `Rc` as mutable
  --> src/main.rs:26:13
   |
26 |             file.write_all(b"I was clicked.\n");
   |             ^^^^ cannot borrow as mutable
   |
   = help: trait `DerefMut` is required to modify through a dereference, but it is not
implemented for `std::rc::Rc<std::fs::File>`
```

Reference counting allows us to have multiple references to a value, but they're all *immutable* references. Looks like we haven't actually made our situation any better than before, where we had ensured that the single owner of our data was the closure.

5.21. RefCell

`RefCell` is designed to exactly solve this problem. I'm not going to go into detail explaining it, because the API docs for `std::cell` do that better than I could. I recommend you go read that intro now, come back and work on this code, and then go read the docs again. Personally, I had to read that explanation about 4 or 5 times and bash my head against some broken code before it finally sank in correctly.

Anyway, add `use std::cell::RefCell;`, and then wrap a `RefCell` around the original `File`:

```
let file = std::fs::File::create("mylog.txt").unwrap();
let file = RefCell::new(file);
```

Now our code will fail to compile with a different message:

```
error[E0596]: cannot borrow data in an `Rc` as mutable
  --> src/main.rs:26:13
   |
26 |             file.write_all(b"I was clicked.\n");
   |             ^^^^ cannot borrow as mutable
   |
   = help: trait `DerefMut` is required to modify through a dereference, but it is not
implemented for `std::rc::Rc<std::fs::File>`

e
error[E0599]: no method named `write_all` found for type
`std::cell::RefCell<std::fs::File>` in the current scope
  --> src/main.rs:26:18
   |
26 |             file.write_all(b"I was clicked.\n");
   |                  ^^^^^^^^^ method not found in
`std::cell::RefCell<std::fs::File>`
   |
   = note: the method `write_all` exists but the following trait bounds were not
satisfied:
           `std::cell::RefCell<std::fs::File> : gio::prelude::OutputStreamExtManual`
```

Unlike `Rc`, with `RefCell` we cannot rely on the `Deref` implementation to get us a `File`. Instead, we'll need to use a method on `RefCell` to get a reference to the `File`:

```
file.borrow().write_all(b"I was clicked.\n");
```

But that doesn't quite work:

```
error[E0596]: cannot borrow data in a dereference of `std::cell::Ref<'_,
std::fs::File>` as mutable
```

Fortunately, that fix is as easy as using `borrow_mut()` instead. And now our program works, hurray!

NOTE Often, reference counting (`Rc` or `Arc`) and cells (`Cell`, `RefCell`, or `Mutex`) go hand in hand, which is why my first instinct in writing this lesson was to use both an `Rc` and a `RefCell`. However, in this case, it turns out that just the `RefCell` is sufficient.

Exercise 7

The error handling in this program is lackluster. There are three problems:

1. If `Application::new()` fails, we panic.
2. If opening `mylog.txt` fails, we panic.
3. If writing to the file fails, we panic.

To fix this, have `main` return a value of type `Result<(), Box<std::error::Error>>`. Most other errors can be automatically coerced via `From::from` into `Box<std::error::Error>`. For problems (1) and (2), use the standard error handling mechanisms we discussed back in lesson 3. For problem (3), print an error message with `eprintln!` when an error occurs.

Solution

```
extern crate gio;
extern crate gtk;

use gio::prelude::*;
use gtk::prelude::*;
use std::cell::RefCell;
use std::error::Error;
use std::io::Write;
use std::rc::Rc;

use gtk::{Application, ApplicationWindow, Button};

fn main() -> Result<(), Box<dyn Error>> {
    let application =
        Application::new(Some("com.github.gtk-rs.examples.basic"), Default
::default())?;

    let file = std::fs::File::create("mylog.txt")?;
    let file = Rc::new(RefCell::new(file));

    application.connect_activate(move |app| {
        let window = ApplicationWindow::new(app);
        window.set_title("First GTK+ Program");
        window.set_default_size(350, 70);

        let button = Button::new_with_label("Click me!");

        let file = file.clone();
        button.connect_clicked(
            move |_| match file.borrow_mut().write_all(b"I was clicked.\n") {
                Ok(()) => (),
                Err(e) => eprintln!("Error writing to file: {}", e),
            },
        );
        window.add(&button);

        window.show_all();
    });

    application.run(&[]);
    Ok(())
}
```

5.22. Fearless concurrency!

It's finally time to do some fearless concurrency. We're going to write a program which will:

- Allocate a string containing the word "Fearless"
- Fork a thread every second for 10 iterations
- In the forked thread:
  - Add another exclamation point to the string
  - Print the string

Before we begin, you can probably identify some complex pieces of ownership that are going to go on here:

- Multiple threads will have access to some mutable data
- We need to ensure only one writer at a time
- We need to ensure that the data is released when everyone is done with it

Instead of trying to design a great solution to this from the beginning, we'll try to come up with a solution and iterate on it. We'll do the most naive stuff possible, look at the error messages, and try to improve. If you think you can implement the complete program yourself now, definitely give it a shot! Even if you don't think you can implement it yourself, it's worth trying. The effort will make the explanation below more helpful.

5.22.1. Introducing the functions

We're going to use the following three functions:

`std::thread::spawn` to spawn a thread. It has an interesting signature:

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T> where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

The `Send` trait means that both the provided function and its return value must be values which can be sent to a different thread. The `'static` bit says that we cannot retain any references to local variables. And the `FnOnce()` bit says that *any* closure will work.

`std::thread::sleep` to have the main thread sleep. It takes a value of type `Duration`, which brings us to our last function:

`std::time::Duration::new` takes the number of seconds and nanoseconds in a duration.

Before we introduce the great fun which is spawning a new thread, let's try a single threaded version:

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

fn main() {
    let mut msg: String = String::from("Fearless");
    for _ in 1..11 {
        msg.push('!');
        println!("{}", msg);
        sleep(Duration::new(1, 0));
    }
}
```

We can even wrap up that `msg.push` and `println!` in a closure to get a bit closer to the call to `spawn`:

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

fn main() {
    let mut msg: String = String::from("Fearless");
    for _ in 1..11 {
        let inner = || {
            msg.push('!');
            println!("{}", msg);
        };
        inner();
        sleep(Duration::new(1, 0));
    }
}
```

That gives us an error message:

```
error[E0596]: cannot borrow `inner` as mutable, as it is not declared as mutable
```

Go ahead and fix that and make this compile.

5.22.2. Introducing spawn

The simplest way to introduce spawn is to replace the `inner()` call with `spawn(inner)`. Replace:

```
use std::thread::sleep;
```

with

```
use std::thread::{sleep, spawn};
```

And add the `spawn` call. We get the error message:

```
error[E0373]: closure may outlive the current function, but it borrows `msg`, which is
owned by the current function
 --> main.rs:7:25
  |
7 |         let mut inner = || {
  |                         ^^ may outlive borrowed value `msg`
8 |             msg.push('!');
  |             --- `msg` is borrowed here
help: to force the closure to take ownership of `msg` (and any other referenced
variables), use the `move` keyword
  |
7 |         let mut inner = move || {
  |                         ^^^^^^^

error: aborting due to previous error
```

Seems simple enough: we have to have a self contained closure to pass to `spawn`, which can't refer to values from the parent thread. Let's just add a `move` in front of the closure. We get an error message:

```
error[E0382]: use of moved value: `msg`
 --> main.rs:7:25
  |
5 |     let mut msg: String = String::from("Fearless");
  |         ------- move occurs because `msg` has type `std::string::String`, which
does not implement the `Copy` trait
6 |     for _ in 1..11 {
7 |         let mut inner = move || {
  |                         ^^^^^^^ value moved into closure here, in previous
iteration of loop
8 |             msg.push('!');
  |             --- use occurs due to use in closure
```

I still don't find these error messages particularly enlightening. But it's telling us that we're trying to capture a moved value. This is happening because we're moving the value into the closure in the first iteration of the loop, and then trying to move it in again. That clearly won't work!

5.22.3. A broken solution

Let's just cheat and create a new copy of the string for each iteration. That's easy enough: add the following above `let mut inner`:

```
let mut msg = msg.clone();
```

This will compile (with a warning) and run, but it has the wrong output. We aren't adding extra exclamation points each time. We're not actually dealing with shared mutable data. Darn.

But that cloning gives me another idea...

5.22.4. Reference counting

Maybe we can throw in that reference counting we mentioned previously, and let each thread keep a pointer to the same piece of data.

```
use std::thread::{sleep, spawn};
use std::time::Duration;
use std::rc::Rc;

fn main() {
    let msg = Rc::new(String::from("Fearless"));
    for _ in 1..11 {
        let mut msg = msg.clone();
        let mut inner = move || {
            msg.push('!');
            println!("{}", msg);
        };
        spawn(inner);
        sleep(Duration::new(1, 0));
    }
}
```

Well, *that's* a new one:

```
error[E0277]: `std::rc::Rc<std::string::String>` cannot be sent between threads safely
  --> main.rs:13:9
   |
13 |         spawn(inner);
   |         ^^^^^ `std::rc::Rc<std::string::String>` cannot be sent between threads
safely
   |
```

There's that fearless concurrency we've heard so much about! The compiler is preventing us from sending an `Rc` value between threads. It would be nice if the compiler mentioned it, but we already know that for multithreaded applications, we need an atomic reference counter, or `std::sync::Arc`. Go ahead and switch over to that. You should get a new error message:

```
error[E0596]: cannot borrow immutable borrowed content as mutable
  --> main.rs:10:13
   |
10 |             msg.push('!');
   |             ^^^ cannot borrow as mutable

error: aborting due to previous error
```

5.22.5. Inner mutability

Above, I mentioned that `Rc` and `RefCell` usually go together. The `Rc` provides reference counting, and the `RefCell` provides mutability. Maybe we can combine `Arc` and `RefCell` too?

```
use std::thread::{sleep, spawn};
use std::time::Duration;
use std::sync::Arc;
use std::cell::RefCell;

fn main() {
    let msg = Arc::new(RefCell::new(String::from("Fearless")));
    for _ in 1..11 {
        let mut msg = msg.clone();
        let mut inner = move || {
            let msg = msg.borrow_mut();
            msg.push('!');
            println!("{}", msg);
        };
        spawn(inner);
        sleep(Duration::new(1, 0));
    }
}
```

More fearless concurrency:

```
error[E0277]: `std::cell::RefCell<std::string::String>` cannot be shared between
threads safely
  --> main.rs:15:9
   |
15 |         spawn(inner);
   |         ^^^^^ `std::cell::RefCell<std::string::String>` cannot be shared between
threads safely
   |
   = help: the trait `std::marker::Sync` is not implemented for
`std::cell::RefCell<std::string::String>`
   = note: required because of the requirements on the impl of `std::marker::Send` for
`std::sync::Arc<std::cell::RefCell<std::string::String>>`
   = note: required because it appears within the type `[closure@main.rs:10:25: 14:10
msg:std::sync::Arc<std::cell::RefCell<std::string::String>>]`
   = note: required by `std::thread::spawn`
```

You could go search for more info, but the normal way to have a mutable, multithreaded cell is a `Mutex`. Instead of `borrow_mut()`, we have a `lock()` method, which ensures that only one thread at a time is using the mutex. Let's try that out:

```
use std::thread::{sleep, spawn};
use std::time::Duration;
use std::sync::{Arc, Mutex};

fn main() {
    let msg = Arc::new(Mutex::new(String::from("Fearless")));
    for _ in 1..11 {
        let mut msg = msg.clone();
        let mut inner = move || {
            let msg = msg.lock();
            msg.push('!');
            println!("{}", msg);
        };
        spawn(inner);
        sleep(Duration::new(1, 0));
    }
}
```

We get the error:

```
error[E0599]: no method named `push` found for type
`std::result::Result<std::sync::MutexGuard<'_, std::string::String>,
std::sync::PoisonError<std::sync::MutexGuard<'_, std::string::String>>>` in the
current scope
```

Oh, right. `lock`ing can fail, due to something called poisoning. (Check out the docs for more information.) To quote the docs:

> Most usage of a mutex will simply `unwrap()` these results, propagating panics among threads to ensure that a possibly invalid invariant is not witnessed.

This is the closest to runtime exceptions I've seen the Rust docs mention, nice. If we add that `.unwrap()`, we get told that `msg` needs to be mutable. And if we add `mut`, we've written our first multithreaded Rust application using shared mutable state.

Notice how the compiler prevented us from making some serious concurrency mistakes? That's pretty awesome.

As a final step, see which `mut`s and `move`s you can and cannot remove from the final program. Make sure you can justify to yourself why the compiler does or does not accept each change.

5.23. Next Chapter

You're now deep into the hard parts of Rust. What comes now is getting more comfortable with the hairy bits of ownership and closures, and to get more comfortable with the library ecosystem. We're ready to get much more real world in the next chapter, and learn about tokio, the de facto standard async I/O framework in Rust.

6. Lifetimes and Slices

We've glossed over some details of lifetimes and sequences of values so far. It's time to dive in and learn about lifetimes and *slices* correctly.

6.1. Printing a person

Let's look at some fairly unsurprising code:

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
    age: Option<u32>,
}

fn print_person(person: Person) {
    match person.name {
        Some(name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }

    match person.age {
        Some(age) => println!("Age is {}", age),
        None => println!("No age provided"),
    }
}

fn main() {
    print_person(Person {
        name: Some(String::from("Alice")),
        age: Some(30),
    });
}
```

Fairly simple, and a nice demonstration of pattern matching. However, let's throw in one extra line. Try adding this at the beginning of the `print_person` function:

```
println!("Full Person value: {:?}", person);
```

All good. We're printing the full contents of the `Person` and then pattern matching. But try adding that line to the *end* of the function, and you'll get a compilation error:

```
error[E0382]: borrow of moved value: `person`
  --> main.rs:17:41
   |
9  |         Some(name) => println!("Name is {}", name),
   |              ---- value moved here
...
17 |     println!("Full Person value: {:?}", person);
   |                                         ^^^^^^ value borrowed here after partial
move
   |
   = note: move occurs because value has type `std::string::String`, which does not
implement the `Copy` trait

error: aborting due to previous error
```

The problem is that we've consumed a part of the `person` value, and therefore cannot display it. We can fix that by setting it again. Let's make the `person` argument `mut`able, and then fill in the moved `person.name` with a default `None` value:

```
fn print_person(mut person: Person) {
    match person.name {
        Some(name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }

    match person.age {
        Some(age) => println!("Age is {}", age),
        None => println!("No age provided"),
    }

    person.name = None;

    println!("Full Person value: {:?}", person);
}
```

That compiles, but now the output is confusingly:

```
Name is Alice
Age is 30
Full Person value: Person { name: None, age: Some(30) }
```

Notice how the `name` in the last line in `None`, when ideally it should be `Some(Alice)`. We can do better, by returning the name from the `match`:

```
person.name = match person.name {
    Some(name) => {
        println!("Name is {}", name);
        Some(name)
    },
    None => {
        println!("No name provided");
        None
    }
};
```

But that's decidedly inelegant. Let's take a step back. Do we actually need to consume/move the `person.name` at all? Not really. It should work to do everything by reference. So let's go back and avoid the move entirely, by using a borrow:

```
match &person.name {
    Some(name) => println!("Name is {}", name),
    None => println!("No name provided"),
}
```

Much better! We don't need to put the borrow on `person.age` though, since the `u32` is `Copy`able. Here, we're pattern matching on a reference, and therefore the `name` is *also* a reference.

However, we can be more explicit about that with the `ref` keyword. This keyword says that, when pattern matching, we want the pattern to be a reference, *not* a move of the original value. (More info in the Rust by Example.) We end up with:

```
match person.name {
    Some(ref name) => println!("Name is {}", name),
    None => println!("No name provided"),
}
```

In our case, this is the same basic result as `&person.name`.

6.2. Birthday!

Let's modify our code so that, when printing the age, we also increase the age by 1. First stab is below. Note that the code won't compile, try to predict why:

```
match person.age {
    Some(age) => {
        println!("Age is {}", age);
        age += 1;
    }
    None => println!("No age provided"),
}
```

We're trying to mutate the local `age` binding, but it's immutable. Well, that's easy enough to fix, just replace `Some(age)` with `Some(mut age)`. That compiles, but with a warning:

```
warning: value assigned to `age` is never read
  --> src/main.rs:16:13
   |
16 |             age += 1;
   |             ^^^
   |
   = note: #[warn(unused_assignments)] on by default
```

And then the output is:

```
Name is Alice
Age is 30
Full Person value: Person { name: Some("Alice"), age: Some(30) }
```

Notice how on the last line, the age is still 30, not 31. Take a minute and try to understand what's happening here... Done? Cool.

1. We pattern match on `person.age`
2. If it's `Some`, we need to move the age into the local `age` binding
3. But since the type is `u32`, it will make a copy and move the copy
4. When we increment the age, we're incrementing a copy, which is never used.

We can try solving this by taking a mutable reference to `person.age`:

```
fn print_person(person: Person) {
    match person.name {
        Some(ref name) => println!("Name is {}", name),
        None => println!("No name provided"),
    }

    match &mut person.age {
        Some(age) => {
            println!("Age is {}", age);
            age += 1;
        }
        None => println!("No age provided"),
    }

    println!("Full Person value: {:?}", person);
}
```

The compiler complains: `age` is a `&mut u32`, but we're trying to use `+=` on it:

```
error[E0368]: binary assignment operation `+=` cannot be applied to type `&mut u32`
  --> src/main.rs:16:13
   |
16 |             age += 1;
   |             ---^^^^^
   |             |
   |             cannot use `+=` on type `&mut u32`
   |
   = help: `+=` can be used on 'u32', you can dereference `age`: `*age`
```

The compiler taketh, and the compiler giveth as well: we just need to dereference the `age` reference. Close, but one more error:

```
error[E0596]: cannot borrow field `person.age` of immutable binding as mutable
  --> src/main.rs:13:16
   |
7  | fn print_person(person: Person) {
   |                 ------ consider changing this to `mut person`
...
13 |     match &mut person.age {
   |                ^^^^^^^^^^ cannot mutably borrow field of immutable binding

error: aborting due to previous error
```

Again, the compiler tells us exactly how to solve the problem: make `person` mutable. Go ahead and make that change, and everything should work.

Exercise 1

In the case of `person.name`, we came up with two solutions: borrow the `person.name`, or use the `ref` keyword. The same two styles of solutions will work for our current problem. We've just demonstrated the borrow approach. Try to solve this instead using the `ref` keyword.

Solution

If you try just throwing in the `ref` keyword like this:

```
match person.age {
    Some(ref age) => {
        println!("Age is {}", age);
        *age += 1;
    }
    None => println!("No age provided"),
}
```

You'll get an error message:

```
error[E0594]: cannot assign to immutable borrowed content `*age`
  --> src/main.rs:16:13
   |
14 |         Some(ref age) => {
   |              ------- help: use a mutable reference instead: `ref mut age`
15 |             println!("Age is {}", age);
16 |             *age += 1;
   |             ^^^^^^^^ cannot borrow as mutable
```

Instead, you need to say `ref mut age`. And if you're like me and regularly type in `mut ref age` instead of `ref mut age`, don't worry, the compiler's got your back:

```
error: the order of `mut` and `ref` is incorrect
  --> src/main.rs:14:14
   |
14 |         Some(mut ref age) => {
   |              ^^^^^^^ help: try switching the order: `ref mut`

error: aborting due to previous error
```

6.3. The single iterator

Let's make a silly little iterator which produces a single value. We'll track whether or not the value

has been produced by using an `Option`:

```
struct Single<T> {
    next: Option<T>,
}
```

Let's make a helper function to create `Single` values:

```
fn single<T>(t: T) -> Single<T> {
    Single {
        next: Some(t),
    }
}
```

And let's write a `main` that tests that this works as expected:

```
fn main() {
    let actual: Vec<u32> = single(42).collect();
    assert_eq!(vec![42], actual);
}
```

If you try to compile that, you'll get an error message:

```
error[E0599]: no method named `collect` found for type `Single<{integer}>` in the
current scope
  --> src/main.rs:12:39
   |
1  | struct Single<T> {
   | ---------------- method `collect` not found for this
...
12 |     let actual: Vec<u32> = single(42).collect();
   |                                       ^^^^^^^
   |
   = note: the method `collect` exists but the following trait bounds were not
satisfied:
           `&mut Single<{integer}> : std::iter::Iterator`
   = help: items from traits can only be used if the trait is implemented and in scope
   = note: the following trait defines an item `collect`, perhaps you need to
implement it:
           candidate #1: `std::iter::Iterator`
```

We need to provide an `Iterator` implementation in order to use `collect()`. The `Item` is going to be `T`. And we've already got a great `Option<T>` available for the return value from the `next` function:

```
impl<T> Iterator for Single<T> {
    type Item = T;

    fn next(&mut self) -> Option<T> {
        self.next
    }
}
```

Unfortunately this doesn't work:

```
error[E0507]: cannot move out of `self.next` which is behind a mutable reference
  --> src/main.rs:21:9
   |
18 |         self.next
   |         ^^^^^^^^^
   |         |
   |         move occurs because `self.next` has type `std::option::Option<T>`, which
does not implement the `Copy` trait
   |         help: consider borrowing the `Option`'s content: `self.next.as_ref()`

error: aborting due to previous error
```

Oh, right. We can't move the result value out, since our `next` function only mutable borrows `self`. Let's try some pattern matching:

```
match self.next {
    Some(next) => Some(next),
    None => None,
}
```

Except this *also* involves moving out of a borrow, so it fails. Let's try one more time: we'll move into a local variable, set `self.next` to `None` (so it doesn't repeat the value again), and return the local variable:

```
fn next(&mut self) -> Option<T> {
    let res = self.next;
    self.next = None;
    res
}
```

Nope, the compiler is *still* not happy! I guess we'll just have to give up on our grand vision of a `Single` iterator. We could of course just cheat:

```
fn next(&mut self) -> Option<T> {
    None
}
```

But while that compiles, it fails our test at runtime:

```
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `[42]`,
 right: `[]`', src/main.rs:13:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

6.4. Swap

What we did above was attempt to swap the `self.next` with a local variable. However, the borrow checker wasn't a fan of the approach we took. However, there's a helper function in the standard library, `std::mem::swap`, which may be able to help us. It looks like:

```
pub fn swap<T>(x: &mut T, y: &mut T)
```

And sure enough, we can use it to solve our problem:

```
fn next(&mut self) -> Option<T> {
    let res = None;
    std::mem::swap(res, self.next);
    res
}
```

Exercise 2

The code above doesn't quite compile, though the compiler can guide you to a correct solution. Try to identify the problems above and fix them yourself. Failing that, ask the compiler to help you out.

Solution

You need to provide mutable references for the two arguments to `swap`. Additionally, in order to get a mutable reference to `res`, `res` itself needs to be `mut`able.

```
fn next(&mut self) -> Option<T> {
    let mut res = None;
    std::mem::swap(&mut res, &mut self.next);
    res
}
```

6.5. replace and take

Did you find that whole create-a-temp-variable thing a bit verbose? Yeah, it does to the authors of the Rust standard library too. There's a helper function that bypasses that temporary variable:

```
fn next(&mut self) -> Option<T> {
    std::mem::replace(&mut self.next, None)
}
```

Much nicer! However, that **still** seems like more work for something that should be really easy. And fortunately, yet again, it does to the authors of the Rust standard library too. This pattern of replacing the value in an `Option` with `None` and then working with the original value is common enough that they've given it a name and a method: `take`.

```
fn next(&mut self) -> Option<T> {
    self.next.take()
}
```

And we're done!

6.6. Lifetimes

We've briefly mentioned lifetimes in previous lessons, but it's time to get a bit more serious about them. Let's look at a simple usage of references:

```
struct Person {
    name: String,
    age: u32,
}

fn get_name(person: &Person) -> String {
    person.name
}

fn main() {
    let alice = Person {
        name: String::from("Alice"),
        age: 30,
    };
    let name = get_name(&alice);
    println!("Name: {}", name);
}
```

This code doesn't compile. Our `get_name` function takes a reference to a `Person`, and then tries to move that person's `name` in its result. This isn't possible. One solution would be to clone the name:

```
fn get_name(person: &Person) -> String {
    person.name.clone()
}
```

While this works, it's relatively inefficient. We like to avoid making copies when we can. Instead, let's simply return a reference to the name:

```
fn get_name(person: &Person) -> &String {
    &person.name
}
```

Hurrah! But let's make our function a little bit more complicated. We now want a function that will take *two* `Person`s, and return the name of the older one. That sounds fairly easy to write:

```
struct Person {
    name: String,
    age: u32,
}

fn get_older_name(person1: &Person, person2: &Person) -> &String {
    if person1.age >= person2.age {
        &person1.name
    } else {
        &person2.name
    }
}

fn main() {
    let alice = Person {
        name: String::from("Alice"),
        age: 30,
    };
    let bob = Person {
        name: String::from("Bob"),
        age: 35,
    };
    let name = get_older_name(&alice, &bob);
    println!("Older person: {}", name);
}
```

Unfortunately, the compiler is quite cross with us:

```
error[E0106]: missing lifetime specifier
 --> src/main.rs:6:58
  |
6 | fn get_older_name(person1: &Person, person2: &Person) -> &String {
  |                                                          ^ expected lifetime
parameter
  |
  = help: this function's return type contains a borrowed value, but the signature
does not say whether it is borrowed from `person1` or `person2`
```

That error message is remarkably clear. Our function is returning a borrowed value. That value must be borrowed from *somewhere*. The only two options* are `person1` and `person2`. And it seems that Rust needs to know this for some reason.

- This is a small fib, see "static lifetime" below.

Remember how we have some rules about references? References cannot outlive the original values they come from. We need to track how long the result value is allowed to live, which must be less than or equal to the time the value it came from lives. This whole concept is *lifetimes*.

For reasons we'll get to in a bit (under "lifetime elision"), we can often bypass the need to explicitly talk about lifetimes. However, sometimes we do need to be explicit. To do this, we introduce some new parameters. But this time, they are *lifetime parameters*, which begin with a single quote and are lower case. Usually, they are just a single letter. For example:

```
fn get_older_name<'a, 'b>(person1: &'a Person, person2: &'b Person) -> &String
```

We still get an error from the compiler because our return value doesn't have a lifetime. Should we choose `'a` or `'b`? Or maybe we should create a new `'c` and try that? Let's start off with `'a`. We get the error message:

```
error[E0623]: lifetime mismatch
  --> src/main.rs:10:9
   |
6  | fn get_older_name<'a, 'b>(person1: &'a Person, person2: &'b Person) -> &'a String
{
   |                                                 ----------     ----------
   |                                                 |
   |                                                 this parameter and the
return type are declared with different lifetimes...
...
10 |         &person2.name
   |         ^^^^^^^^^^^^^ ...but data from `person2` is returned here
```

That makes sense: since our result may come from `person2`, we have no guarantee that the `'a` lifetime parameter is less than or equal to the `'b` lifetime parameter. Fortunately, we can explicitly state that, in the same way that we state that types implement some traits:

```
fn get_older_name<'a, 'b: 'a>(person1: &'a Person, person2: &'b Person) -> &'a String
{
```

And this actually compiles! Alternatively, in this case, we can just completely bypass the second lifetime parameter, and say that `person1` and `person2` must have the same lifetime, which must be the same as the return value:

```
fn get_older_name<'a>(person1: &'a Person, person2: &'a Person) -> &'a String {
```

If you're like me, you may think that this would be overly limiting. For example, I initially thought that with the signature above, this code wouldn't compile:

```
fn main() {
    let alice = Person {
        name: String::from("Alice"),
        age: 30,
    };
    foo(&alice);
}

fn foo(alice: &Person) {
    let bob = Person {
        name: String::from("Bob"),
        age: 35,
    };
    let name = get_older_name(&alice, &bob);
    println!("Older person: {}", name);
}
```

After all, the lifetime for `alice` is demonstrably bigger than the lifetime for `bob`. However, the semantics for lifetimes in functions signatures is that all of the values have at least the same lifetime. If they happen to live a bit longer, no harm, no foul.

6.7. Requirement for multiple lifetime parameters

So can we cook up an example where multiple lifetime parameters are absolutely necessary? Sure!

```
fn message_and_return(msg: &String, ret: &String) -> &String {
    println!("Printing the message: {}", msg);
    ret
}

fn main() {
    let name = String::from("Alice");
    let msg = String::from("This is the message");
    let ret = message_and_return(&msg, &name);
    println!("Return value: {}", ret);
}
```

This code won't compile, because we need some lifetime parameters. So let's use our trick from above, and use the same parameter:

```
fn message_and_return<'a>(msg: &'a String, ret: &'a String) -> &'a String {
```

That compiles, but let's make our calling code a bit more complicated:

```
fn main() {
    let name = String::from("Alice");
    let ret = foo(&name);
    println!("Return value: {}", ret);
}

fn foo(name: &String) -> &String {
    let msg = String::from("This is the message");
    message_and_return(&msg, &name)
}
```

Now the compiler isn't happy:

```
error[E0515]: cannot return value referencing local variable `msg`
  --> main.rs:14:5
   |
14 |     message_and_return(&msg, &name)
   |     ^^^^^^^^^^^^^^^^^^^^----^^^^^^^^
   |     |                   |
   |     |                   `msg` is borrowed here
   |     returns a value referencing data owned by the current function

error: aborting due to previous error
```

We've stated that the return value must live the same amount of time as the `msg` parameter. But we return the return value *outside* of the `foo` function, while the `msg` value will not live beyond the end of `foo`.

The calling code should be fine, we just need to tell Rust that it's OK if the `msg` parameter has a shorter lifetime than the return value.

Exercise 3

Modify the signature of `message_and_return` so that the code compiles and runs.

Solution

We need to have two different parameters, and ensure that `ret` and the return value have the same lifetime parameter:

```
fn message_and_return<'a, 'b>(msg: &'a String, ret: &'b String) -> &'b String {
    println!("Printing the message: {}", msg);
    ret
}
```

6.8. Lifetime elision

Why do we sometimes get away with skipping the lifetimes, and sometimes we need to include them? There are rules in the language called "lifetime elision." Instead of trying to cover this myself, I'll refer to the Nomicon:

https://doc.rust-lang.org/nomicon/lifetime-elision.html

6.9. Static lifetime

Above, I implied that if you return a reference, then it must have the same lifetime as one of its input parameters. This mostly makes sense, because otherwise we'd have to conjure some arbitrary lifetime out of thin air. However, it's also a lie. There's one special lifetime that survives the entire program, called `'static`. And here's some fun news: you've implicitly used it since the first Hello World we wrote together!

Every single string literal is in fact a reference with the lifetime of `'static`.

```
fn name() -> &'static str {
    "Alice"
}

fn main() {
    println!("{}", name());
}
```

6.10. Arrays, slices, vectors, and String

Here's another place where we've been cheeky. What's the difference between `String` and `str`? Both of these have popped up quite a bit. We'll get to those in a little bit. First, we need to talk about arrays, slices, and vectors.

6.10.1. Arrays

To my knowledge, the best official documentation on arrays is in the API docs themselves. Arrays are contiguous blocks of memory containing a single type of data with a fixed length. The type is represented as `[T; N]`, where `T` is the type of value, and `N` is the length of the array. And like any sane programming language, arrays are 0-indexed in Rust.

There are two syntaxes for initiating arrays. List literal syntax (like Javascript, Python, or Haskell):

```
fn main() {
    let nums: [u32; 5] = [1, 2, 3, 4, 5];
    println!("{:?}", nums);
}
```

And a repeat expression:

```
fn main() {
    let nums: [u32; 5] = [42; 5];
    println!("{:?}", nums);
}
```

You can make arrays mutable and then, well, mutate them:

```
fn main() {
    let mut nums: [u32; 5] = [42; 5];
    nums[2] += 1;
    println!("{:?}", nums);
}
```

That's very nice, but what if you need something more dynamic? For that, we have...

6.10.2. Vec

A `Vec` is a "contiguous, growable array type." You can `push` and `pop`, check its length, and access via index in O(1) time. We also have a nifty `vec!` macro for constructing them:

```
fn main() {
    let mut v: Vec<u32> = vec![1, 2, 3];
    v.push(4);
    assert_eq!(v.pop(), Some(4));
    v.push(4);
    v.push(5);
    v.push(6);
    assert_eq!(v.pop(), Some(6));
    assert_eq!(v[2], 3);
    println!("{:?}", v); // 1, 2, 3, 4, 5
}
```

6.10.3. Slices

I'm going to write a helper function that prints all the values in a `Vec`:

```
fn main() {
    let v: Vec<u32> = vec![1, 2, 3];
    print_vals(v);
}

fn print_vals(v: Vec<u32>) {
    for i in v {
        println!("{}", i);
    }
}
```

Of course, since this is a pass-by-value, the following doesn't compile:

```
fn main() {
    let mut v: Vec<u32> = vec![1, 2, 3];
    print_vals(v);
    v.push(4);
    print_vals(v);
}
```

Easy enough to fix: have `print_vals` take a reference to a `Vec`:

```
fn main() {
    let mut v: Vec<u32> = vec![1, 2, 3];
    print_vals(&v);
    v.push(4);
    print_vals(&v);
}

fn print_vals(v: &Vec<u32>) {
    for i in v {
        println!("{}", i);
    }
}
```

Unfortunately, this doesn't generalize to, say, an array:

```
fn main() {
    let a: [u32; 5] = [1, 2, 3, 4, 5];
    print_vals(&a);
}
```

This fails since `print_vals` takes a `&Vec<u32>`, but we've provided a `&[u32; 5]`. But this is pretty disappointing. A dynamic vector and a fixed length array behave the same for so many things. Wouldn't it be nice if there was something that generalized both of these?

Enter slices. To quote the Rust book:

> Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection.

To make this all work, we need to change the signature of `print_vals` to:

```
fn print_vals(v: &[u32]) {
```

`&[u32]` is a reference to a *slice* of `u32`s. A slice can be created from an array *or* a `Vec`, not to mention some other cases as well. (We'll discuss how the `&` borrow operator works its magic in a bit.) As a general piece of advice, if you're receiving a parameter which is a sequence of values, try to use a slice, as it will give the caller much more control about where the data comes from.

I played a bit of a word game above, switching between "reference to a slice" and "a slice." Obviously we're using a reference. Can we dereference a slice reference and get the slice itself? Let's try!

```
fn print_vals(vref: &[u32]) {
    let v = *vref;
    for i in v {
        println!("{}", i);
    }
}
```

The compiler is cross with us again:

```
error[E0277]: the size for values of type `[u32]` cannot be known at compilation time
 --> src/main.rs:7:9
  |
7 |     let v = *vref;
  |         ^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `[u32]`
  = note: to learn more, visit <https://doc.rust-lang.org/book/second-edition/ch19-04-
advanced-types.html#dynamically-sized-types-and-sized>
  = note: all local variables must have a statically known size

error[E0277]: the size for values of type `[u32]` cannot be known at compilation time
 --> src/main.rs:8:14
  |
8 |     for i in v {
  |              ^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `[u32]`
  = note: to learn more, visit <https://doc.rust-lang.org/book/second-edition/ch19-04-
advanced-types.html#dynamically-sized-types-and-sized>
  = note: required by `std::iter::IntoIterator::into_iter`

error[E0277]: the trait bound `[u32]: std::iter::Iterator` is not satisfied
 --> src/main.rs:8:14
  |
8 |     for i in v {
  |              ^ `[u32]` is not an iterator; maybe try calling `.iter()` or a
similar method
  |
  = help: the trait `std::iter::Iterator` is not implemented for `[u32]`
  = note: required by `std::iter::IntoIterator::into_iter`
```

Basically, there's no way to dereference a slice. It logically makes sense in any event to just keep a reference to the block of memory holding the values, whether it's on the stack, heap, or the executable itself (like string literals, which we'll get to later).

6.11. Deref

There's something fishy; why does the ampersand/borrow operator give us different types? The following compiles just fine!

```
fn main() {
    let v = vec![1, 2, 3];
    let _: &Vec<u32> = &v;
    let _: &[u32] = &v;
}
```

It turns out that the borrow operator interacts with "Deref coercion." If you're curious about this,

please check out the docs for the `Deref` trait. As an example, I can create a new struct which can be borrowed into a slice:

```
use std::ops::Deref;

struct MyArray([u32; 5]);

impl MyArray {
    fn new() -> MyArray {
        MyArray([42; 5])
    }
}

impl Deref for MyArray {
    type Target = [u32];

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

fn main() {
    let ma = MyArray::new();
    let _: &MyArray = &ma;
    let _: &[u32] = &ma;
}
```

Thanks to udoprog for answering this question. Also, just because you *can* do this doesn't necessarily mean you should.

6.12. Using slices

Slices are data types like any others. You can check out the `std::slice` module documentation and the `slice` primitive type.

Some common ways to interact with them include:

- Using them as `Iterator`s
- Indexing them with `slice[idx]` syntax
- Taking subslices with `slice[start..end]` syntax

6.13. Byte literals

If you put a lower case `b` in front of a string literal, you'll get a byte array. You can either treat this as a fixed length array or, more commonly, as a slice:

```
fn main() {
    let bytearray1: &[u8; 22] = b"Hello World in binary!";
    let bytearray2: &[u8] = b"Hello World in binary!";
    println!("{:?}", bytearray1);
    println!("{:?}", bytearray2);
}
```

Note that you always receive a *reference* to the value, not the value itself. The data is stored in the program executable itself, and therefore cannot be modified (thus always receiving an immutable reference).

Exercise 4

Add lifetime parameters to the `bytearray1` and `bytearray2` types above.

Solution

Since the data is stored in the program executable itself, it lives for the entire program execution. Therefore, the lifetime is `'static`:

```
fn main() {
    let bytearray1: &'static [u8; 22] = b"Hello World in binary!";
    let bytearray2: &'static [u8] = b"Hello World in binary!";
    println!("{:?}", bytearray1);
    println!("{:?}", bytearray2);
}
```

6.14. Strings

And finally we can talk about strings! You may think that a string literal would be a fixed length array of `char`s. You can in fact create such a thing:

```
fn main() {
    let char_array: [char; 5] = ['H', 'e', 'l', 'l', 'o'];
    println!("{:?}", char_array);
}
```

However, this is *not* what a `str` is. The representation above is highly inefficient. Since a `char` in Rust has full Unicode support, it takes up 4 bytes in memory (32 bits). However, for most data, this is overkill. A character encoding like UTF-8 will be far more efficient.

NOTE If you're not familiar with Unicode and character encodings, it's safe to gloss over these details here. It's not vitally important to understanding how strings work in Rust.

Instead, a string slice (`&str`) is essentially a newtype wrapper around a byte slice (`&[u8]`), which is guaranteed to be in UTF-8 encoding. This has some important trade-offs:

- You can cheaply (freely?) convert from a `&str` to a `&[u8]`, which can be great for making system calls
- You cannot get O(1) random access within strings, since the UTF-8 encoding doesn't allow for this. Instead, you need to work with a character iterator to view the individual characters.

Exercise 5

Use `std::env::args` and the `chars()` method on `String` to print out the number of characters in each command line arguments. Bonus points: also print out the number of bytes. Sample usage:

```
$ cargo run 
arg: target/debug/foo, characters: 16, bytes: 16
arg: , characters: 4, bytes: 8
```

Don't forget, the first argument is the name of the executable.

Solution

This is a great use case for the iterator method `count`:

```
fn main() {
    for arg in std::env::args() {
        println!(
            "arg: {}, characters: {}, bytes: {}",
            arg,
            arg.chars().count(),
            arg.bytes().count(),
            );
    }
}
```

6.15. Lifetimes in data structures

One final topic for today is lifetimes in data structures. It's entirely possible to keep references in your data structures. However, when you do so, you need to be explicit about their lifetimes. For example, this will fail to compile:

```
struct Person {
    name: &str,
    age: u32,
}
```

Instead, you would need to write it as:

```
struct Person<'a> {
    name: &'a str,
    age: u32,
}
```

The general recommendation I've received, and which I'd pass on, is avoid this when possible. Things end up getting significantly more complicated when dealing with lifetime parameters in data structures. Typically, you should use owned versions of values (e.g. `String` instead of `&str`, or `Vec` or array instead of a slice) inside your data structures. In such a case, you need to ensure that the lifetime of the reference within the structure outlives the structure itself.

There are times when you can avoid some extra cloning and allocation if you use references in your data structure, and the time will probably come when you need to do it. But I'd recommend waiting until your profiling points you at a specific decision being the bottleneck. For more information, see the Rust book.

6.16. References and slices in APIs

Some general advice which I received and has mostly steered me correctly is:

> When receiving parameters, prefer slices when possible

However, there are times when this is overly simplistic. If you want a deeper dive, there a great blog post covering some trade-offs in public APIs: On dealing with owning and borrowing in public interfaces. The Reddit discussion is also great.

7. Async, futures, and tokio

The original Rust Crash Course series included a section explaining the original futures and Tokio crates. However, beginning with Rust 1.39 in late 2019, the language has support for the new `async/.await` syntax, which drastically changes and simplifies all of this. Therefore, this ebook does not include that original lesson. If you really want to read about it, you can still see the content online.

8. Down and dirty with Future

Much has happened since Rust 1.31. Importantly: the `Future` trait has moved into the standard library itself and absorbed a few modifications. And then to tie that up in a nicer bow, there's a new

`async/.await` syntax. It's hard for me to overstate just how big a quality of life difference this is when writing asynchronous code in Rust.

I wrote an article on the FP Complete tech site that demonstrates the `Future` and `async/.await` stuff in practice. But here, I want to give a more thorough analysis of what's going on under the surface. I'm going to skip the motivation for why we want to write asynchronous code, and break this up into more digestible chunks.

NOTE I'm going to use the `async-std` library in this example instead of `tokio`. My only real reason for this is that I started using `async-std` before `tokio` released support for the new `async/.await` syntax. I'm not ready to weigh in on, in general, which of the libraries I prefer.

You should start a Cargo project to play along. Try `cargo new --bin sleepus-interruptus`. If you want to ensure you're on the same compiler version, add a `rust-toolchain` file with the string `1.41.1` in it. Run `cargo run` to make sure you're all good to go.

8.1. Sleepus Interruptus

I want to write a program which will print the message `Sleepus` 10 times, with a delay of 0.5 seconds. And it should print the message `Interruptus` 5 times, with a delay of 1 second. This is some fairly easy Rust code:

```
use std::thread::{sleep};
use std::time::Duration;

fn sleepus() {
    for i in 1..=10 {
        println!("Sleepus {}", i);
        sleep(Duration::from_millis(500));
    }
}

fn interruptus() {
    for i in 1..=5 {
        println!("Interruptus {}", i);
        sleep(Duration::from_millis(1000));
    }
}

fn main() {
    sleepus();
    interruptus();
}
```

However, as my clever naming implies, this isn't my real goal. This program runs the two operations *synchronously*, first printing `Sleepus`, then `Interruptus`. Instead, we would want to have these two sets of statements printed in an interleaved way. That way, the interruptus actually does some interrupting.

Exercise 1

Use the `std::thread::spawn` function to spawn an operating system thread to make these printed statements interleave.

Solution

There are two basic approaches to this. One—maybe the more obvious—is to spawn a separate thread for each function, and then wait for each of them to complete:

```
use std::thread::{sleep, spawn};

fn main() {
    let sleepus = spawn(sleepus);
    let interruptus = spawn(interruptus);

    sleepus.join().unwrap();
    interruptus.join().unwrap();
}
```

Two things to notice:

- We call `spawn` with `spawn(sleepus)`, *not* `spawn(sleepus())`. The former passes in the function `sleepus` to `spawn` to be run. The latter would immediately run `sleepus()` and pass its result to `spawn`, which is not what we want.
- I use `join()` in the main function/thread to wait for the child thread to end. And I use `unwrap` to deal with any errors that may occur, because I'm being lazy.

Another approach would be to spawn one helper thread instead, and call one of the functions in the main thread:

```
fn main() {
    let sleepus = spawn(sleepus);
    interruptus();

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

This is more efficient (less time spawning threads and less memory used for holding them), and doesn't really have a downside. I'd recommend going this way.

QUESTION What would be the behavior of this program if we didn't call `join` in the two-spawn version? What if we didn't call `join` in the one-spawn version?

But this isn't an asynchronous approach to the problem at all! We have two threads being handled

by the operating system which are both acting synchronously and making blocking calls to `sleep`. Let's build up a bit of intuition towards how we could have our two tasks (printing `Sleepus` and printing `Interruptus`) behave more cooperatively in a single thread.

8.2. Introducing async

We're going to start at the highest level of abstraction, and work our way down to understand the details. Let's rewrite our application in an `async` style. Add the following to your `Cargo.toml`:

```
async-std = { version = "1.5.0", features = ["attributes"] }
```

And now we can rewrite our application as:

```
use async_std::task::{sleep, spawn};
use std::time::Duration;

async fn sleepus() {
    for i in 1..=10 {
        println!("Sleepus {}", i);
        sleep(Duration::from_millis(500)).await;
    }
}

async fn interruptus() {
    for i in 1..=5 {
        println!("Interruptus {}", i);
        sleep(Duration::from_millis(1000)).await;
    }
}

#[async_std::main]
async fn main() {
    let sleepus = spawn(sleepus());
    interruptus().await;

    sleepus.await;
}
```

Let's hit the changes from top to bottom:

- Instead of getting `sleep` and `spawn` from `std::thread`, we're getting them from `async_std::task`. That probably makes sense.
- Both `sleepup` and `interruptus` now say `async` in front of `fn`.
- After the calls to `sleep`, we have a `.await`. Note that this is *not* a `.await()` method call, but instead a new syntax.
- We have a new attribute `#[async_std::main]` on the `main` function.

- The `main` function also has `async` before `fn`.
- Instead of `spawn(sleepus)`, passing in the function itself, we're now calling `spawn(sleepus())`, immediately running the function and passing its result to `spawn`.
- The call to `interruptus()` is now followed by `.await`.
- Instead of `join()`ing on the `sleepus JoinHandle`, we use the `.await` syntax.

Exercise 2

Run this code on your own machine and make sure everything compiles and runs as expected. Then try undoing some of the changes listed above and see what generates a compiler error, and what generates incorrect runtime behavior.

That may look like a large list of changes. But in reality, our code is almost identical structural to the previous version, which is a real testament to the `async/.await` syntax. And now everything works under the surface the way we want: a single operating system thread making non-blocking calls.

Let's analyze what each of these changes actually means.

8.3. `async` functions

Adding `async` to the beginning of a function definition does three things:

1. It allows you to use `.await` syntax inside. We'll get to the meaning of that in a bit.
2. It modified the return type of the function. `async fn foo() -> Bar` actually returns `impl std::future::Future<Output=Bar>`.
3. Automatically wraps up the result value in a new `Future`. We'll demonstrate that better later.

Let's unpack that second point a bit. There's a trait called `Future` defined in the standard library. It has an associated type `Output`. What this trait means is: I promise that, when I complete, I will give you a value of type `Output`. You could imagine, for instance, an asynchronous HTTP client that looks something like:

```
impl HttpRequest {
    fn perform(self) -> impl Future<Output=HttpResponse> { ... }
}
```

There will be some non-blocking I/O that needs to occur to make that request. We don't want to block the calling thread while those things happen. But we do want to somehow eventually get the resulting response.

We'll play around with `Future` values more directly later. For now, we'll continue sticking with the high-level `async/.await` syntax.

Exercise 3

Rewrite the signature of `sleepus` to not use the `async` keyword by modifying its result type. Note that the code will not compile when you get the type right. Pay attention to the error message you get.

Solution

The result type of `async fn sleepus()` is the implied unit value `()`. Therefore, the `Output` of our `Future` should be unit. This means we need to write our signature as:

```
fn sleepus() -> impl std::future::Future<Output=()>
```

However, with only that change in place, we get the following error messages:

```
error[E0728]: `await` is only allowed inside `async` functions and blocks
 --> src/main.rs:7:9
  |
4 | fn sleepus() -> impl std::future::Future<Output=()> {
  |    ------- this is not `async`
...
7 |         sleep(Duration::from_millis(500)).await;
  |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ only allowed inside `async`
functions and blocks

error[E0277]: the trait bound `(): std::future::Future` is not satisfied
 --> src/main.rs:4:17
  |
4 | fn sleepus() -> impl std::future::Future<Output=()> {
  |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait
`std::future::Future` is not implemented for `()`
  |
  = note: the return type of a function must have a statically known size
```

The first message is pretty direct: you can only use the `.await` syntax inside an `async` function or block. We haven't seen an `async` block yet, but it's exactly what it sounds like:

```
async {
    // async noises intensify
}
```

The second error message is a result of the first: the `async` keyword causes the return type to be an `impl Future`. Without that keyword, our `for` loop evaluates to `()`, which isn't an `impl Future`.

Exercise 4

Fix the compiler errors by introducing an `await` block inside the `sleepus` function. Do *not* add `async` to the function signature, keep using `impl Future`.

Solution

Wrapping the entire function body with an `async` block solves the problem:

```
fn sleepus() -> impl std::future::Future<Output=()> {
    async {
        for i in 1..=10 {
            println!("Sleepus {}", i);
            sleep(Duration::from_millis(500)).await;
        }
    }
}
```

8.4. .await a minute

Maybe we don't need all this `async/.await` garbage though. What if we remove the calls to `.await` usage in `sleepus`? Perhaps surprisingly, it compiles, though it does give us an ominous warning:

```
warning: unused implementer of `std::future::Future` that must be used
 --> src/main.rs:8:13
  |
8 |             sleep(Duration::from_millis(500));
  |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  |
  = note: `#[warn(unused_must_use)]` on by default
  = note: futures do nothing unless you `.await` or poll them
```

We're generating a `Future` value but not using it. And sure enough, if you look at the output of our program, you can see what the compiler means:

```
Interruptus 1
Sleepus 1
Sleepus 2
Sleepus 3
Sleepus 4
Sleepus 5
Sleepus 6
Sleepus 7
Sleepus 8
Sleepus 9
Sleepus 10
Interruptus 2
Interruptus 3
Interruptus 4
Interruptus 5
```

All of our `Sleepus` messages print without delay. Intriguing! The issue is that the call to `sleep` no longer actually puts our current thread to sleep. Instead, it generates a value which implements `Future`. And when that promise is eventually fulfilled, we know that the delay has occurred. But in our case, we're simply ignoring the `Future`, and therefore never actually delaying.

To understand what the `.await` syntax is doing, we're going to implement our function with much more direct usage of the `Future` values. Let's start by getting rid of the `async` block.

8.5. Dropping `async` block

If we drop the `async` block, we end up with this code:

```
fn sleepus() -> impl std::future::Future<Output=()> {
    for i in 1..=10 {
        println!("Sleepus {}", i);
        sleep(Duration::from_millis(500));
    }
}
```

This gives us an error message we saw before:

```
error[E0277]: the trait bound `(): std::future::Future` is not satisfied
 --> src/main.rs:4:17
  |
4 | fn sleepus() -> impl std::future::Future<Output=()> {
  |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ the trait
`std::future::Future` is not implemented for `()`
  |
```

This makes sense: the `for` loop evaluates to `()`, and unit does not implement `Future`. One way to fix

this is to add an expression after the `for` loop that evaluates to something that implements `Future`. And we already know one such thing: `sleep`.

Exercise 5

Tweak the `sleepus` function so that it compiles.

Solution

One implementation is:

```
fn sleepus() -> impl std::future::Future<Output=()> {
    for i in 1..=10 {
        println!("Sleepus {}", i);
        sleep(Duration::from_millis(500));
    }
    sleep(Duration::from_millis(0))
}
```

We still get a warning about the unused `Future` value inside the `for` loop, but not the one afterwards: that one is getting returned from the function. But of course, sleeping for 0 milliseconds is just a wordy way to do nothing. It would be nice if there was a "dummy" `Future` that more explicitly did nothing. And fortunately, there is.

Exercise 6

Replace the `sleep` call after the `for` loop with a call to `ready`.

Solution

```
fn sleepus() -> impl std::future::Future<Output=()> {
    for i in 1..=10 {
        println!("Sleepus {}", i);
        sleep(Duration::from_millis(500));
    }
    async_std::future::ready(())
}
```

8.6. Implement our own `Future`

To unpeel this onion a bit more, let's make our life harder, and *not* use the `ready` function. Instead, we're going to define our own `struct` which implements `Future`. I'm going to call it `DoNothing`.

```
use std::future::Future;

struct DoNothing;

fn sleepus() -> impl Future<Output=()> {
    for i in 1..=10 {
        println!("Sleepus {}", i);
        sleep(Duration::from_millis(500));
    }
    DoNothing
}
```

Exercise 7

This code won't compile. Without looking below or asking the compiler, what do you think it's going to complain about?

Solution

The problem here is that `DoNothing` does not provide a `Future` implementation.

We're going to do some Compiler Driven Development and let `rustc` tell us how to fix our program. Our first error message is:

```
the trait bound `DoNothing: std::future::Future` is not satisfied
```

So let's add in a trait implementation:

```
impl Future for DoNothing {
}
```

Which fails with:

```
error[E0046]: not all trait items implemented, missing: `Output`, `poll`
 --> src/main.rs:7:1
  |
7 | impl Future for DoNothing {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^ missing `Output`, `poll` in implementation
  |
  = note: `Output` from trait: `type Output;`
  = note: `poll` from trait: `fn(std::pin::Pin<&mut Self>, &mut
std::task::Context<'_>) -> std::task::Poll<<Self as std::future::Future>::Output>`
```

We don't really know about the Pin<&mut Self> or Context thing yet, but we do know about Output. And since we were previously returning a () from our ready call, let's do the same thing here.

```
use std::pin::Pin;
use std::task::{Context, Poll};

impl Future for DoNothing {
    type Output = ();

    fn poll(self: Pin<&mut Self>, ctx: &mut Context) -> Poll<Self::Output> {
        unimplemented!()
    }
}
```

Woohoo, that compiles! Of course, it fails at runtime due to the unimplemented!() call:

```
thread 'async-std/executor' panicked at 'not yet implemented', src/main.rs:13:9
```

Now let's try to implement poll. We need to return a value of type Poll<Self::Output>, or Poll<()>. Let's look at the definition of Poll:

```
pub enum Poll<T> {
    Ready(T),
    Pending,
}
```

Using some basic deduction, we can see that Ready means "our Future is complete, and here's the output" while Pending means "it's not done yet." Given that our DoNothing wants to return the output of () immediately, we can just use the Ready variant here.

Exercise 8

Implement a working version of poll.

Solution

```
fn poll(self: Pin<&mut Self>, _ctx: &mut Context) -> Poll<Self::Output> {
    Poll::Ready(())
}
```

Congratulations, you've just implemented your first Future struct!

8.7. The third async difference

Remember above we said that making a function async does a third thing:

> Automatically wraps up the result value in a new Future. We'll demonstrate that better later.

Now is later. Let's demonstrate that better.

Let's simplify the definition of sleepus to:

```
fn sleepus() -> impl Future<Output=()> {
    DoNothing
}
```

The compiles and runs just fine. Let's try switching back to the async way of writing the signature:

```
async fn sleepus() {
    DoNothing
}
```

This now gives us an error:

```
error[E0271]: type mismatch resolving `<impl std::future::Future as
std::future::Future>::Output == ()`
  --> src/main.rs:17:20
   |
17 | async fn sleepus() {
   |                    ^ expected struct `DoNothing`, found ()
   |
   = note: expected type `DoNothing`
              found type `()`
```

You see, when you have an async function or block, the result is automatically wrapped up in a Future. So instead of returning a DoNothing, we're returning a `impl Future<Output=DoNothing>`. And our type wants `Output=()`.

Exercise 9

Try to guess what you need to add to this function to make it compile.

Solution

Working around this is pretty easy: you simply append `.await` to `DoNothing`:

```
async fn sleepus() {
    DoNothing.await
}
```

This gives us a little more intuition for what `.await` is doing: it's extracting the `()` `Output` from the `DoNothing` `Future`... somehow. However, we still don't really know how it's achieving that. Let's build up a more complicated `Future` to get closer.

8.8. SleepPrint

We're going to build a new `Future` implementation which:

- Sleeps for a certain amount of time
- Then prints a message

This is going to involve using pinned pointers. I'm not going to describe those here. The specifics of what's happening with the pinning isn't terribly enlightening to the topic of `Future`s. If you want to let your eyes glaze over at that part of the code, you won't be missing much.

Our implementation strategy for `SleepPrint` will be to wrap an existing `sleep` `Future` with our own implementation of `Future`. Since we don't know the exact type of the result of a `sleep` call (it's just an `impl Future`), we'll use a parameter:

```
struct SleepPrint<Fut> {
    sleep: Fut,
}
```

And we can call this in our `sleepus` function with:

```
fn sleepus() -> impl Future<Output=()> {
    SleepPrint {
        sleep: sleep(Duration::from_millis(3000)),
    }
}
```

Of course, we now get a compiler error about a missing `Future` implementation. So let's work on that. Our `impl` starts with:

```
impl<Fut: Future<Output=()>> Future for SleepPrint<Fut> {
    ...
}
```

This says that `SleepPrint` is a `Future` if the `sleep` value it contains is a `Future` with an `Output` of type `()`. Which, of course, is true in the case of the `sleep` function, so we're good. We need to define `Output`:

```
type Output = ();
```

And then we need a `poll` function:

```
fn poll(self: Pin<&mut Self>, ctx: &mut Context) -> Poll<Self::Output> {
    ...
}
```

The next bit is the eyes-glazing part around pinned pointers. We need to *project* the `Pin<&mut Self>` into a `Pin<&mut Fut>` so that we can work on the underlying sleep `Future`. We could use a helper crate to make this a bit prettier, but we'll just do some `unsafe` mapping:

```
let sleep: Pin<&mut Fut> = unsafe { self.map_unchecked_mut(|s| &mut s.sleep) };
```

Alright, now the important bit. We've got our underlying `Future`, and we need to do something with it. The only thing we *can* do with it is call `poll`. `poll` requires a `&mut Context`, which fortunately we've been provided. That `Context` contains information about the currently running task, so it can be woken up (via a `Waker`) when the task is ready.

NOTE We're not going to get deeper into how `Waker` works in this post. If you want a real life example of how to call `Waker` yourself, I recommend reading my pid1 in Rust post.

For now, let's do the only thing we can reasonably do:

```
match sleep.poll(ctx) {
    ...
}
```

We've got two possibilities. If `poll` returns a `Pending`, it means that the `sleep` hasn't completed yet. In that case, we want our `Future` to also indicate that it's not done. To make that work, we just propagate the `Pending` value:

```
Poll::Pending => Poll::Pending,
```

However, if the `sleep` is already complete, we'll receive a `Ready(())` variant. In that case, it's finally

time to print our message and then propagate the `Ready`:

```
Poll::Ready(()) => {
    println!("Inside SleepPrint");
    Poll::Ready(())
},
```

And just like that, we've built a more complex `Future` from a simpler one. But that was pretty ad-hoc.

8.9. TwoFutures

`SleepPrint` is pretty ad-hoc: it hard codes a specific action to run after the `sleep Future` completes. Let's up our game, and sequence the actions of two different `Future`s. We're going to define a new `struct` that has three fields:

- The first `Future` to run
- The second `Future` to run
- A `bool` to tell us if we've finished running the first `Future`

Since the `Pin` stuff is going to get a bit more complicated, it's time to reach for that helper crate to ease our implementation and avoid `unsafe` blocks ourself. So add the following to your `Cargo.toml`:

```
pin-project-lite = "0.1.4"
```

And now we can define a `TwoFutures` struct that allows us to project the first and second `Future`s into pinned pointers:

```
use pin_project_lite::pin_project;

pin_project! {
    struct TwoFutures<Fut1, Fut2> {
        first_done: bool,
        #[pin]
        first: Fut1,
        #[pin]
        second: Fut2,
    }
}
```

Using this in `sleepus` is easy enough:

```
fn sleepus() -> impl Future<Output=()> {
    TwoFutures {
        first_done: false,
        first: sleep(Duration::from_millis(3000)),
        second: async { println!("Hello TwoFutures"); },
    }
}
```

Now we just need to define our `Future` implementation. Easy, right? We want to make sure both `Fut1` and `Fut2` are `Future`s. And our `Output` will be the output from the `Fut2`. (You could also return both the first and second output if you wanted.) To make all that work:

```
impl<Fut1: Future, Fut2: Future> Future for TwoFutures<Fut1, Fut2> {
    type Output = Fut2::Output;

    fn poll(self: Pin<&mut Self>, ctx: &mut Context) -> Poll<Self::Output> {
        ...
    }
}
```

In order to work with the pinned pointer, we're going to get a new value, `this`, which projects all of the pointers:

```
let this = self.project();
```

With that out of the way, we can interact with our three fields directly in `this`. The first thing we do is check if the first `Future` has already completed. If not, we're going to poll it. If the poll is `Ready`, then we'll ignore the output and indicate that the first `Future` is done:

```
if !*this.first_done {
    if let Poll::Ready(_) = this.first.poll(ctx) {
        *this.first_done = true;
    }
}
```

Next, if the first `Future` is done, we want to poll the second. And if the first `Future` is *not* done, then we say that we're pending:

```
if *this.first_done {
    this.second.poll(ctx)
} else {
    Poll::Pending
}
```

And just like that, we've composed two `Future`s together into a bigger, grander, brighter `Future`.

Exercise 10

Get rid of the usage of an `async` block in `second`. Let the compiler errors guide you.

Solution

The error message you get says that `()` is not a `Future`. Instead, you need to return a `Future` value after the call to `println!`. We can use our handy `async_std::future::ready`:

```
second: {
    println!("Hello TwoFutures");
    async_std::future::ready(())
},
```

8.10. AndThen

Sticking together two arbitrary `Future`s like this is nice. But it's even nicer to have the second `Future`s depend on the result of the first `Future`. To do this, we'd want a function like `and_then`. (Monads FTW to my Haskell buddies.) I'm not going to bore you with the gory details of an implementation here, but feel free to read the Gist if you're interested. Assuming you have this method available, we can begin to write the `sleepus` function ourselves as:

```
fn sleepus() -> impl Future<Output = ()> {
    println!("Sleepus 1");
    sleep(Duration::from_millis(500)).and_then(|()| {
        println!("Sleepus 2");
        sleep(Duration::from_millis(500)).and_then(|()| {
            println!("Sleepus 3");
            sleep(Duration::from_millis(500)).and_then(|()| {
                println!("Sleepus 4");
                async_std::future::ready(())
            })
        })
    })
}
```

And before Rust 1.39 and the `async/.await` syntax, this is basically how async code worked. This is far from perfect. Besides the obvious right-stepping of the code, it's not actually a loop. You *could* recursively call `sleepus`, except that creates an infinite type which the compiler isn't too fond of.

But fortunately, we've now finally established enough background to easily explain what the `.await` syntax is doing: exactly what `and_then` is doing, but without the fuss!

Exercise 11

Rewrite the sleepus function above to use .await instead of and_then.

Solution

The rewrite is really easy. The body of the function becomes the non-right-stepping, super flat:

```
println!("Sleepus 1");
sleep(Duration::from_millis(500)).await;
println!("Sleepus 2");
sleep(Duration::from_millis(500)).await;
println!("Sleepus 3");
sleep(Duration::from_millis(500)).await;
println!("Sleepus 4");
```

And then we also need to change the signature of our function to use async, or wrap everything in an async block. Your call.

Besides the obvious readability improvements in the above solution, there are some massive usability improvements with .await as well. One that sticks out here is how easily it ties in with loops. This was a real pain with the older futures stuff. Also, chaining together multiple await calls is really easy, e.g.:

```
let body = make_http_request().await.get_body().await;
```

And not only that, but it plays in with the ? operator for error handling perfectly. The above example would more likely be:

```
let body = make_http_request().await?.get_body().await?;
```

8.11. main attribute

One final mystery remains. What exactly is going on with that weird attribute on main:

```
#[async_std::main]
async fn main() {
    ...
}
```

Our sleepus and interruptus functions do not actually do anything. They return Futures which

provide instructions on how to do work. Something has to actually perform those actions. The thing that runs those actions is an **executor**. The `async-std` library provides an executor, as does `tokio`. In order to run any `Future`, you need an executor.

The attribute above automatically wraps the `main` function with `async-std`'s executor. The attribute approach, however, is totally optional. Instead, you can use `async_std::task::block_on`.

Exercise 12

Rewrite `main` to not use the attribute. You'll need to rewrite it from `async fn main` to `fn main`.

Solution

Since we use `.await` inside the body of `main`, we get an error when we simply remove the `async` qualifier. Therefore, we need to use an `async` block inside `main` (or define a separate helper `async` function). Putting it all together:

```
fn main() {
    async_std::task::block_on(async {
        let sleepus = spawn(sleepus());
        interruptus().await;

        sleepus.await;
    })
}
```

Each executor is capable of managing multiple tasks. Each task is working on producing the output of a single `Future`. And just like with threads, you can `spawn` additional tasks to get concurrent running. Which is exactly how we achieve the interleaving we wanted!

8.12. Cooperative concurrency

One word of warning. `Future`s and `async`/`.await` implement a form of cooperative concurrency. By contrast, operating system threads provide preemptive concurrency. The important different is that in cooperative concurrency, you have to cooperate. If one of your tasks causes a delay, such as by using `std::thread::sleep` or by performing significant CPU computation, it will not be interrupted.

The upshot of this is that you should ensure you do not perform blocking calls inside your tasks. And if you have a CPU-intensive task to perform, it's probably worth spawning an OS thread for it, or at least ensuring your executor will not starve your other tasks.

8.13. Summary

I don't think the behavior under the surface of `.await` is too big a reveal, but I think it's useful to understand exactly what's happening here. In particular, understanding the difference between a

value of `Future` and actually chaining together the outputs of `Future` values is core to using `async/.await` correctly. Fortunately, the compiler errors and warnings do a great job of guiding you in the right direction.

In the next chapter, we can start using our newfound knowledge of `Future` and the `async/.await` syntax to build some asynchronous applications. We'll be diving into writing some async I/O, including networking code, using Tokio 0.2.

8.14. Exercises

Here are some take-home exercises to play with. You can base them on the code in this Gist.

1. Modify the `main` function to call `spawn` twice instead of just once.
2. Modify the `main` function to not call `spawn` at all. Instead, use `join`. You'll need to add a `use async_std::prelude::FutureExt;` and add the `"unstable"` feature to the `async-std` dependency in `Cargo.toml`.
3. Modify the `main` function to get the non-interleaved behavior, where the program prints `Sleepus` multiple times before `Interruptus`.
4. Write a function `foo` such that the following assertion passes: `assert_eq!(42, async_std::task::block_on(async { foo().await.await }));`

8.14.1. Solution 1

The original program spawned a new task to run `sleepus`, ran `interruptus` in the original task, and then after `interruptus` finished `await`ed for `sleepus`. Instead, we can run *both* as separate tasks and then `await` for both of them:

```
fn main() {
    async_std::task::block_on(async {
        let sleepus = spawn(sleepus());
        let interruptus = spawn(interruptus());

        sleepus.await;
        interruptus.await;
    })
}
```

8.14.2. Solution 2

Using the `join` method implicits creates the extra task, so this works:

```
fn main() {
    use async_std::prelude::FutureExt;
    async_std::task::block_on(async {
        sleepus().join(interruptus()).await;
    })
}
```

However, in this case, we're just using a single `.await` at the end of an `async` block, so we can simplify even further:

```
fn main() {
    use async_std::prelude::FutureExt;
    async_std::task::block_on(sleepus().join(interruptus()));
}
```

Note that we *do* need the semicolon at the end of that expression. The `block_on` is returning a pair of the two results values from the `sleepus` and `interruptus` tasks, and that doesn't match the unit return type of `main`.

8.14.3. Solution 3

```
fn main() {
    async_std::task::block_on(async {
        sleepus().await;
        interruptus().await;
    })
}
```

8.14.4. Solution 4

The trick here is that we need to have two layers of `Future` here. One way to do that is to use an `async` function with an `async` block inside of it:

```
use std::future::Future;

async fn foo() -> impl Future<Output=i32> {
    async {
        42
    }
}

fn main() {
    assert_eq!(42, async_std::task::block_on(async { foo().await.await }));
}
```

Another, perhaps stranger looking one, is to nest our `async` blocks and have one `impl Future` embedded inside another:

```
use std::future::Future;

fn foo() -> impl Future<Output=impl Future<Output=i32>> {
    async {
        async {
            42
        }
    }
}

fn main() {
    assert_eq!(42, async_std::task::block_on(async { foo().await.await }));
}
```

Or, perhaps a bit more directly, we can use a helper function like `ready` to more explicitly create a `Future`:

```
use std::future::Future;
use async_std::future::ready;

fn foo() -> impl Future<Output=impl Future<Output=i32>> {
    ready(ready(42))
}

fn main() {
    assert_eq!(42, async_std::task::block_on(async { foo().await.await }));
}
```

9. Tokio 0.2

In the previous chapter in this book, we covered the new `async/.await` syntax stabilized in Rust 1.39, and the `Future` trait which lives underneath it. Now it's time to look into Tokio 0.2 library. For those not familiar with it, let me quote the project's overview:

> Tokio is an event-driven, non-blocking I/O platform for writing asynchronous applications with the Rust programming language.

If you want to write an efficient, concurrent network service in Rust, you'll want to use something like Tokio. That's not to say that this is the only use case for Tokio; you can do lots of great things with an event driven scheduler outside of network services. It's also not to say that Tokio is the only solution; the `async-std` library provides similar functionality.

However, network services are likely the most common domain agitating for a non-blocking I/O

system. And Tokio is the most popular and established of these systems today. So this combination is where we're going to get started.

9.1. Hello Tokio!

Let's kick this off. Go ahead and create a new Rust project for experimenting:

```
$ cargo new --bin usetokio
```

If you want to make sure you're using the same compiler version as me, set up your `rust-toolchain` correctly:

```
$ echo 1.41.1 > rust-toolchain
```

And then set up Tokio as a dependency. For simplicity, we'll install all the bells and whistles. In your `Cargo.toml`:

```
[dependencies]
tokio = { version = "0.2", features = ["full"] }
```

PROTIP You can run `cargo build` now to kick off the download and build of crates while you keep reading...

And now we're going to write an asynchronous hello world application. Type this into your `src/main.rs`:

```
use tokio::io;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut stdout = io::stdout();
    let mut hello: &[u8] = b"Hello, world!\n";
    io::copy(&mut hello, &mut stdout).await?;
    Ok(())
}
```

NOTE I specifically said "type this in" instead of "copy and paste." For getting comfortable with this stuff, I recommend manually typing in the code.

A lot of this should look familiar from our previous lesson. To recap:

- Since we'll be `await`ing something and generating a `Future`, our `main` function is `async`.
- Since `main` is `async`, we need to use an executor to run it. That's why we use the `#[tokio::main]` attribute.

- Since performing I/O can fail, we return a `Result`.

The first really new thing since last lesson is this little bit of syntax:

```
.await?
```

I mentioned it last time, but now we're seeing it in real life. This is just the combination of our two pieces of prior art: `.await` for chaining together `Future`s, and `?` for error handling. The fact that these work together so nicely is really awesome. I'll probably mention this a few more times, because I love it that much.

The next thing to note is that we use `tokio::io::stdout()` to get access to some value that lets us interact with standard output. If you're familiar with it, this looks really similar to `std::io::stdout()`. That's by design: a large part of the `tokio` API is simply async-ifying things from `std`.

And finally, we can look at the actual `tokio::io::copy` call. As you may have guessed, and as stated in the API docs:

> This is an asynchronous version of `std::io::copy`.

However, instead of working with the `Read` and `Write` traits, this works with their async cousins: `AsyncRead` and `AsyncWrite`. A byte slice (`&[u8]`) is a valid `AsyncRead`, so we're able to store our input there. And as you may have guessed, `Stdout` is an `AsyncWrite`.

NOTE You can simplify this code using `stdout.write_all` after `use`ing `tokio::io::AsyncWriteExt`, but we'll stick to `tokio::io::copy`, since we'll be using it throughout. But if you're curious:

```
use tokio::io::{self, AsyncWriteExt};

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut stdout = io::stdout();
    stdout.write_all(b"Hello, world!\n").await?;
    Ok(())
}
```

Exercise 1

Modify this application so that instead of printing "Hello, world!", it copies the entire contents of standard input to standard output.

Solution

```
use tokio::io;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut stdin = io::stdin();
    let mut stdout = io::stdout();
    io::copy(&mut stdin, &mut stdout).await?;
    Ok(())
}
```

9.2. Spawning processes

Tokio provides a `tokio::process` module which resembles the `std::process` module. We can use this to implement Hello World once again:

```
use tokio::process::Command;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    Command::new("echo").arg("Hello, world!").spawn()?.await?;
    Ok(())
}
```

Notice how the `?` and `.await` bits can go in whatever order they are needed. You can read this line as:

- Create a new `Command` to run `echo`
- Give it the argument `"Hello, world!"`
- Spawn this, which may fail
- Using the first `?`: if it fails, return the error. Otherwise, return a `Future`
- Using the `.await`: wait until that `Future` completes, and capture its `Result`
- Using the second `?`: if that `Result` is `Err`, return that error.

Pretty nice for a single line!

One of the great advantages of `async`/`.await` versus the previous way of doing async with callbacks is how easily it works with looping.

Exercise 2

Extend this example so that it prints `Hello, world!` 10 times.

Solution

```
use tokio::process::Command;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    for _ in 1..=10 {
        Command::new("echo").arg("Hello, world!").spawn()?.await?;
    }
    Ok(())
}
```

9.3. Take a break

So far we've only really done a single bit of `.await`ing. But it's easy enough to `.await` on multiple things. Let's use `delay_for` to pause for a bit.

```
use tokio::time;
use tokio::process::Command;
use std::time::Duration;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    Command::new("date").spawn()?.await?;
    time::delay_for(Duration::from_secs(1)).await;
    Command::new("date").spawn()?.await?;
    time::delay_for(Duration::from_secs(1)).await;
    Command::new("date").spawn()?.await?;
    Ok(())
}
```

We can also use the `tokio::time::interval` function to create a stream of "ticks" for each time a certain amount of time has passed. For example, this program will keep calling `date` once per second until it is killed:

```rust
use tokio::time;
use tokio::process::Command;
use std::time::Duration;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut interval = time::interval(Duration::from_secs(1));
    loop {
        interval.tick().await;
        Command::new("date").spawn()?.await?;
    }
}
```

Exercise 3

Why isn't there a `Ok(())` after the `loop`?

Solution

Since the `loop` will either run forever or be short circuited by an error, any code following `loop` will never actually be called. Therefore, code placed there will generate a warning.

9.4. Time to spawn

This is all well and good, but we're not really taking advantage of asynchronous programming at all. Let's fix that! We've seen two different interesting programs:

1. Infinitely pausing 1 seconds and calling `date`
2. Copying all input from `stdin` to `stdout`

It's time to introduce `spawn` so that we can combine these two into one program. First, let's demonstrate a trivial usage of `spawn`:

```
use std::time::Duration;
use tokio::process::Command;
use tokio::task;
use tokio::time;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    task::spawn(dating()).await??;
    Ok(())
}

async fn dating() -> Result<(), std::io::Error> {
    let mut interval = time::interval(Duration::from_secs(1));
    loop {
        interval.tick().await;
        Command::new("date").spawn()?.await?;
    }
}
```

You may be wondering: what's up with that `??` operator? Is that some special super-error handler? No, it's just the normal error handling `?` applied twice. Let's look at some type signatures to help us out here:

```
pub fn spawn<T>(task: T) -> JoinHandle<T::Output>;

impl<T> Future for JoinHandle<T> {
    type Output = Result<T, JoinError>;
}
```

Calling `spawn` gives us back a `JoinHandle<T::Output>`. In our case, the `Future` we provide as input is `dating()`, which has an output of type `Result<(), std::io::Error>`. So that means the type of `task::spawn(dating())` is `JoinHandle<Result<(), std::io::Error>>`.

We also see that `JoinHandle` implements `Future`. So when we apply `.await` to this value, we end up with whatever that `type Output = Result<T, JoinError>` thing is. Since we know that `T` is `Result<(), std::io::Error>`, this means we end up with `Result<Result<(), std::io::Error>, JoinError>`.

The first `?` deals with the outer `Result`, exiting with the `JoinError` on an `Err`, and giving us a `Result<(), std::io::Error>` value on `Ok`. The second `?` deals with the `std::io::Error`, giving us a `()` on `Ok`. Whew!

Exercise 4

Now that we've seen `spawn`, you should modify the program so that it calls both `date` in a loop, and copies `stdin` to `stdout`.

Solution

```
use std::time::Duration;
use tokio::process::Command;
use tokio::{io, task, time};

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let dating = task::spawn(dating());
    let copying = task::spawn(copying());

    dating.await??;
    copying.await??;

    Ok(())
}

async fn dating() -> Result<(), std::io::Error> {
    let mut interval = time::interval(Duration::from_secs(1));
    loop {
        interval.tick().await;
        Command::new("date").spawn()?.await?;
    }
}

async fn copying() -> Result<(), std::io::Error> {
    let mut stdin = io::stdin();
    let mut stdout = io::stdout();
    io::copy(&mut stdin, &mut stdout).await?;
    Ok(())
}
```

9.5. Synchronous code

You may not have the luxury of interacting exclusively with `async`-friendly code. Maybe you have some really nice library you want to leverage, but it performs blocking calls internally. Fortunately, Tokio's got you covered with the `spawn_blocking` function. Since the docs are so perfect, let me quote them:

> The `task::spawn_blocking` function is similar to the `task::spawn` function discussed in the previous section, but rather than spawning an `non-blocking` future on the Tokio runtime, it instead spawns a `blocking` function on a dedicated thread pool for blocking tasks.

Exercise 5

Rewrite the `dating()` function to use `spawn_blocking` and `std::thread::sleep` so that it calls `date` approximately once per second.

Solution

```
async fn dating() -> Result<(), std::io::Error> {
    loop {
        task::spawn_blocking(|| { std::thread::sleep(Duration::from_secs(1))
}).await?;
        Command::new("date").spawn()?.await?;
    }
}
```

9.6. Let's network!

I could keep stepping through the other cools functions in the Tokio library. I encourage you to poke around at them yourself. But I promised some networking, and by golly, I'm gonna deliver!

I'm going to slightly extend the example from the `TcpListener` docs to (1) make it compile and (2) implement an echo server. This program has a pretty major flaw in it though, I recommend trying to find it.

```
use tokio::io;
use tokio::net::{TcpListener, TcpStream};

#[tokio::main]
async fn main() -> io::Result<()> {
    let mut listener = TcpListener::bind("127.0.0.1:8080").await?;

    loop {
        let (socket, _) = listener.accept().await?;
        echo(socket).await?;
    }
}

async fn echo(socket: TcpStream) -> io::Result<()> {
    let (mut recv, mut send) = io::split(socket);
    io::copy(&mut recv, &mut send).await?;
    Ok(())
}
```

We use `TcpListener` to bind a socket. The binding itself is asynchronous, so we use `.await` to wait for

the listening socket to be available. And we use `?` to deal with any errors while binding the listening socket.

Next, we loop forever. Inside the loop, we accept new connections, using `.await?` like before. We capture the `socket` (ignoring the address as the second part of the tuple). Then we call our `echo` function and `.await` it.

Within `echo`, we use `tokio::io::split` to split up our `TcpStream` into its constituent read and write halves, and then pass those into `tokio::io::copy`, as we've done before.

Awesome! Where's the bug? Let me ask you a question: what *should* the behavior be if a second connection comes in while the first connection is still active? Ideally, it would be handled. However, our program has just one task. And that task `.await`s on each call to `echo`. So our second connection won't be serviced until the first one closes.

Exercise 6

Modify the program above so that it handles concurrent connections correctly.

Solution

The simplest tweak is to wrap the `echo` call with `tokio::spawn`:

```
loop {
    let (socket, _) = listener.accept().await?;
    tokio::spawn(echo(socket));
}
```

There is a downside to this worth noting, however: we're ignoring the errors produced by the spawned tasks. Likely the best behavior in this case is to handle the errors inside the spawned task:

```
#[tokio::main]
async fn main() -> io::Result<()> {
    let mut listener = TcpListener::bind("127.0.0.1:8080").await?;

    let mut counter = 1u32;
    loop {
        let (socket, _) = listener.accept().await?;
        println!("Accepted connection #{}", counter);
        tokio::spawn(async move {
            match echo(socket).await {
                Ok(()) => println!("Connection #{} completed successfully",
counter),
                Err(e) => println!("Connection #{} errored: {:?}", counter, e),
            }
        });
        counter += 1;
    }
}
```

9.7. TCP client and ownership

Let's write a poor man's HTTP client. It will establish a connection to a hard-coded server, copy all of `stdin` to the server, and then copy all data from the server to `stdout`. To use this, you'll manually type in the HTTP request and then hit `Ctrl-D` for end-of-file.

```
use tokio::io;
use tokio::net::TcpStream;

#[tokio::main]
async fn main() -> io::Result<()> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;
    let (mut recv, mut send) = io::split(stream);
    let mut stdin = io::stdin();
    let mut stdout = io::stdout();

    io::copy(&mut stdin, &mut send).await?;
    io::copy(&mut recv, &mut stdout).await?;

    Ok(())
}
```

That's all well and good, but it's limited. It only handles half-duplex protocols like HTTP, and doesn't actually support keep-alive in any way. We'd like to use `spawn` to run the two `copy`s in different tasks. Seems easy enough:

```
let send = spawn(io::copy(&mut stdin, &mut send));
let recv = spawn(io::copy(&mut recv, &mut stdout));

send.await??;
recv.await??;
```

Unfortunately, this doesn't compile. We get four nearly-identical error messages. Let's look at the first:

```
error[E0597]: `stdin` does not live long enough
  --> src/main.rs:12:31
   |
12 |     let send = spawn(io::copy(&mut stdin, &mut send));
   |                      ---------^^^^^^^^^^------------
   |                      |        |
   |                      |        borrowed value does not live long enough
   |                      argument requires that `stdin` is borrowed for `'static`
...
19 | }
   | - `stdin` dropped here while still borrowed
```

Here's the issue: our `copy` `Future` does not *own* the `stdin` value (or the `send` value, for that matter). Instead, it has a (mutable) reference to it. That value remains in the `main` function's `Future`. Ignoring error cases, we know that the `main` function will wait for `send` to complete (thanks to `send.await`), and therefore the lifetimes appear to be correct. However, Rust doesn't recognize this lifetime information. (Also, and I haven't thought this through completely, I'm fairly certain that `send` may be dropped earlier than the `Future` using it in the case of `panic`s.)

In order to fix this, we need to convince the compiler to make a `Future` that owns `stdin`. And the easiest way to do that here is to use an `async move` block.

Exercise 7

Make the code above compile using two `async move` blocks.

Solution

```
use tokio::io;
use tokio::spawn;
use tokio::net::TcpStream;

#[tokio::main]
async fn main() -> io::Result<()> {
    let stream = TcpStream::connect("127.0.0.1:8080").await?;
    let (mut recv, mut send) = io::split(stream);
    let mut stdin = io::stdin();
    let mut stdout = io::stdout();

    let send = spawn(async move {
        io::copy(&mut stdin, &mut send).await
    });
    let recv = spawn(async move {
        io::copy(&mut recv, &mut stdout).await
    });

    send.await??;
    recv.await??;

    Ok(())
}
```

9.8. Playing with `lines`

This section will have a series of modifications to a program. I recommend you solve each challenge before looking at the solution.

Let's build an async program that counts the number of lines on standard input. You'll want to use the `lines` method for this. Read the docs and try to figure out what `use`s and wrappers will be necessary to make the types line up.

```
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let stdin = io::stdin();
    let stdin = io::BufReader::new(stdin);
    let mut count = 0u32;
    let mut lines = stdin.lines();
    while let Some(_) = lines.next_line().await? {
        count += 1;
    }
    println!("Lines on stdin: {}", count);
    Ok(())
}
```

OK, bumping this up one more level. Instead of standard input, let's take a list of file names as command line arguments, and count up the total number of lines in all the files. Initially, it's OK to read the files one at a time. In other words: don't bother calling `spawn`. Give it a shot, and then come back here:

```
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut count = 0u32;

    for filename in args {
        let file = tokio::fs::File::open(filename).await?;
        let file = io::BufReader::new(file);
        let mut lines = file.lines();
        while let Some(_) = lines.next_line().await? {
            count += 1;
        }
    }

    println!("Total lines: {}", count);
    Ok(())
}
```

But now it's time to make this properly asynchronous, and process the files in separate `spawn`ed tasks. In order to make this work, we need to spawn all of the tasks, and then `.await` each of them. I used a `Vec` of `Future<Output=Result<u32, std::io::Error>>`s for this. Give it a shot!

```rust
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut tasks = vec![];

    for filename in args {
        tasks.push(tokio::spawn(async {
            let file = tokio::fs::File::open(filename).await?;
            let file = io::BufReader::new(file);
            let mut lines = file.lines();
            let mut count = 0u32;
            while let Some(_) = lines.next_line().await? {
                count += 1;
            }
            Ok(count) as Result<u32, std::io::Error>
        }));
    }

    let mut count = 0;
    for task in tasks {
        count += task.await??;
    }

    println!("Total lines: {}", count);
    Ok(())
}
```

And finally in this progression: let's change how we handle the `count`. Instead of `.await`ing the count in the second `for` loop, let's have each individual task update a shared mutable variable. You should use an `Arc<Mutex<u32>>` for that. You'll still need to keep a `Vec` of the tasks though to ensure you wait for all files to be read.

```rust
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;
use std::sync::Arc;

// avoid thread blocking by using Tokio's mutex
use tokio::sync::Mutex;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut tasks = vec![];
    let count = Arc::new(Mutex::new(0u32));

    for filename in args {
        let count = count.clone();
        tasks.push(tokio::spawn(async move {
            let file = tokio::fs::File::open(filename).await?;
            let file = io::BufReader::new(file);
            let mut lines = file.lines();
            let mut local_count = 0u32;
            while let Some(_) = lines.next_line().await? {
                local_count += 1;
            }

            let mut count = count.lock().await;
            *count += local_count;
            Ok(()) as Result<(), std::io::Error>
        }));
    }

    for task in tasks {
        task.await??;
    }

    let count = count.lock().await;
    println!("Total lines: {}", *count);
    Ok(())
}
```

9.9. LocalSet and !Send

Thanks to @xudehseng for the inspiration on this section.

OK, did that last exercise seem a bit contrived? It was! In my opinion, the previous approach of `.await`ing the counts and summing in the `main` function itself was superior. However, I wanted to teach you something else.

What happens if you replace the `Arc<Mutex<u32>>` with a `Rc<RefCell<u32>>`? With this code:

```rust
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;
use std::rc::Rc;
use std::cell::RefCell;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut tasks = vec![];
    let count = Rc::new(RefCell::new(0u32));

    for filename in args {
        let count = count.clone();
        tasks.push(tokio::spawn(async {
            let file = tokio::fs::File::open(filename).await?;
            let file = io::BufReader::new(file);
            let mut lines = file.lines();
            let mut local_count = 0u32;
            while let Some(_) = lines.next_line().await? {
                local_count += 1;
            }

            *count.borrow_mut() += local_count;
            Ok(()) as Result<(), std::io::Error>
        }));
    }

    for task in tasks {
        task.await??;
    }

    println!("Total lines: {}", count.borrow());
    Ok(())
}
```

You get an error:

```
error[E0277]: `std::rc::Rc<std::cell::RefCell<u32>>` cannot be shared between threads
safely
  --> src/main.rs:15:20
   |
15 |         tasks.push(tokio::spawn(async {
   |                    ^^^^^^^^^^^^ `std::rc::Rc<std::cell::RefCell<u32>>` cannot be
shared between threads safely
   |
  ::: /Users/michael/.cargo/registry/src/github.com-1ecc6299db9ec823/tokio-
0.2.2/src/task/spawn.rs:49:17
   |
49 |     T: Future + Send + 'static,
   |                 ---- required by this bound in `tokio::task::spawn::spawn`
```

Tasks can be scheduled to multiple different threads. Therefore, your `Future` must be `Send`. And `Rc<RefCell<u32>>` is definitely `!Send`. However, in our use case, using multiple OS threads is unlikely to speed up our program; we're going to be doing lots of blocking I/O. It would be nice if we could insist on spawning all our tasks on the same OS thread and avoid the need for `Send`. And sure enough, Tokio provides such a function: `tokio::task::spawn_local`. Using it (and adding back in `async move` instead of `async`), our program compiles, but breaks at runtime:

```
thread 'main' panicked at '`spawn_local` called from outside of a local::LocalSet!',
src/libcore/option.rs:1190:5
```

Uh-oh! Now I'm personally not a big fan of this detect-it-at-runtime stuff, but the concept is simple enough: if you want to spawn onto the current thread, you need to set up your runtime to support that. And the way we do that is with `LocalSet`. In order to use this, you'll need to ditch the `#[tokio::main]` attribute.

Exercise 8

Follow the documentation for `LocalSet` to make the program above work with `Rc<RefCell<u32>>`.

Solution

```
use tokio::prelude::*;
use tokio::io::AsyncBufReadExt;
use std::rc::Rc;
use std::cell::RefCell;

fn main() -> Result<(), std::io::Error> {
    let mut rt = tokio::runtime::Runtime::new()?;
    let local = tokio::task::LocalSet::new();
    local.block_on(&mut rt, main_inner())
}

async fn main_inner() -> Result<(), std::io::Error> {
    let mut args = std::env::args();
    let _me = args.next(); // ignore command name
    let mut tasks = vec![];
    let count = Rc::new(RefCell::new(0u32));

    for filename in args {
        let count = count.clone();
        tasks.push(tokio::task::spawn_local(async move {
            let file = tokio::fs::File::open(filename).await?;
            let file = io::BufReader::new(file);
            let mut lines = file.lines();
            let mut local_count = 0u32;
            while let Some(_) = lines.next_line().await? {
                local_count += 1;
            }

            *count.borrow_mut() += local_count;
            Ok(()) as Result<(), std::io::Error>
        }));
    }

    for task in tasks {
        task.await??;
    }

    println!("Total lines: {}", count.borrow());
    Ok(())
}
```

9.10. Conclusion

That chapter felt short. Definitely compared to the previous optiional Tokio chapter which seemed to go on forever. I think this is a testament to how easy to use the new `async/.await` syntax is.

There's obviously a lot more that can be covered in asynchronous programming, but hopefully this establishes the largest foundations you need to understand to work with the `async`/`.await` syntax and the Tokio library itself.