A Practical Guide To Rust

- Volume I -

Meade Kincke

Table Of Contents

Introduction

Personal note: I think practically. When learning programming concepts, I do well with things that have a real application. I easily grasp concepts when they have a real world use, so I wrote this guide with that intention.

In each chapter, I provide descriptive variables, real life analogies, and actual code snipets from my personal project.

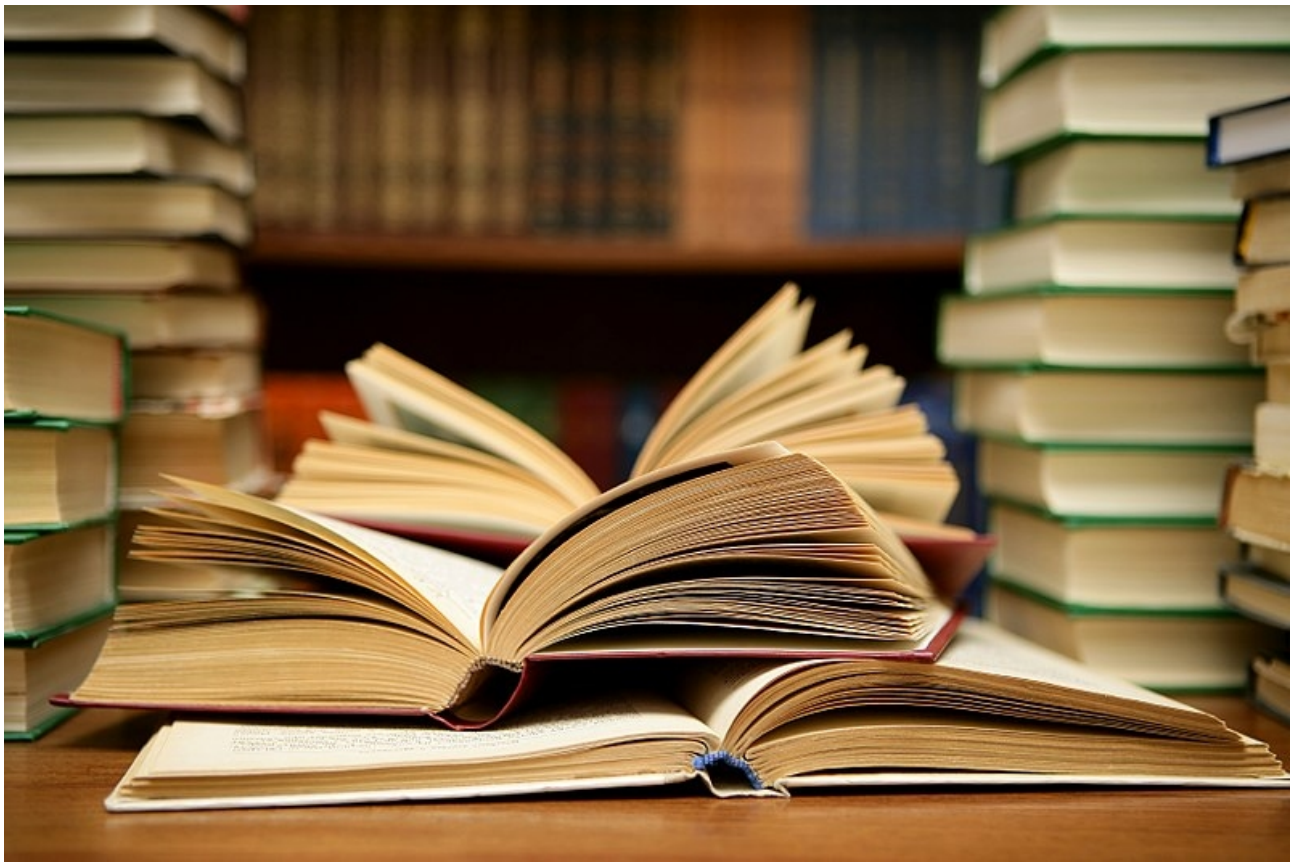
I hope it will help others understand Rust better and think with the idea of "here's a real use for that."

Cheers.

Also, standard Rust uses snake case, which `looks_like_this`. I prefer using camel case which `looksLikeThis`. You'll see `#![allow(non_snake_case, non_camel_case_types)]` which tells the compiler not to complain with a warning about my particular style.

Write however makes you happy!

Chapter 1: Stringy String



One thing that a lot of people seem to struggle with when learning Rust is the concept of ownership. It's a very human, non-mathematical concept.

If I own something, you can borrow it, but you can't change it.

Think about a library versus a bookshop. If you purchase a book, you can write in the margins, tear out pages, do whatever you want to it. If you borrow a book from a library, you aren't allowed to do anything but read it.

Rust uses the same concept for data.

Take a look here:

```
#![allow(non_snake_case, non_camel_case_types)]

// Say you're a function called borrowedFromLibrary().
// I'm main().
```

```
// I lend you a book and tell you to set it on fire.
// You take a string literal (&str).
// You cannot destroy it because of the ampersand &,
// which tells us that it is a
// borrowed value.
// You can use what's written in the book, but you
// can't destroy it.

fn borrowedFromLibrary(stringy: &str) {
    drop(stringy);
}

fn main() {
    let literal: &str = "does nothing";
    borrowedFromLibrary(literal);

    // Notice here that borrowedFromLibrary() does
    // nothing and we can still print out
    // the variable "literal".
    // This is because Drop "will not release any
    // borrows".
    // https://doc.rust-lang.org/std/mem/fn.drop.html

    println!("{}", literal);
}
```

When compiled, we see:

```
/usr/bin/cargo run --color=always
   Compiling stringyString v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in
    0.52 secs
     Running `target/debug/stringyString`

does nothing
```

So, the library made sure we followed the rules and didn’t do anything nasty to their book.

Now we'll look at a bookshop example:

```
#![allow(non_snake_case, non_camel_case_types)]

// Say you're a function that's called
// purchasedFromBookshop(). I'm still main().
// I give you a book and tell you to set it on fire.
// You take a String.
// You are allowed to destroy it because string is an
// owned value.
// You can do whatever you want to the book.

fn purchasedFromBookshop(stringy: String) {
    drop(stringy);
}

fn main() {
    // Here, since the String type is owned,
    // purchasedFromBookshop() does precisely
    // what we tell it to do, and frees the value.
    // When we try to print it out, we can't because
    // it's gone from memory (we destroyed it).

    let actualString: String = String::from("owned");
    purchasedFromBookshop(actualString);

    println!("{}", actualString);
}
```

Running this one we see:

```
/usr/bin/cargo run --color=always
   Compiling stringyString v0.1.0
error[E0382]: use of moved value: `actualString`
  --> src/main.rs:28:20
   |
27 |     purchasedFromBookshop(actualString);
   |                           ------------ value moved
                                          here
28 |     println!("{}", actualString);
   |                    ^^^^^^^^^^^^ value used here
                                     after move
   |
```

```
    = note: move occurs because `actualString` has type
      `std::string::String`, which does not
      implement the `Copy` trait

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0382`.
error: Could not compile `stringyString`.

To learn more, run the command again with --verbose.
```

Yay! We broke it!

Now we can be total jackasses to the book because we own it.

Chapter 2: Easter Egg

As in everything, rule of threes. If you have a function that takes more than three arguments, we need a container to put it in. Think of a bunch of Cadbury Mini Eggs.

Hand me those one at a time.

Exhausted yet?

Now put them inside an Easter Egg and just hand me that. Much more efficient. All I have to do is open the one big egg to get what's inside.

In Rust, structs are our Easter Eggs. We use them as a container for types. We can put any type we want in them. Structs only have type declarations, no data. This is because we're telling the compiler what to expect so it knows how to allocate memory for how big or small our data will be.

Just as the Easter Egg has a bunch of minis in it, a struct has individual entries called fields. Here, we're going to take our struct and fill it with a String and an f64:

```
#![allow(non_snake_case, non_camel_case_types)]


struct easterEgg {
    ownedString: String,
    floaty: f64,
}


fn cubeFloaty(stringsAndFloat: &mut easterEgg) {
    stringsAndFloat.floaty = stringsAndFloat
      .floaty.powi(3);
}


fn main() {
    let mut stringsAndFloat: easterEgg = easterEgg {
        ownedString: String::from(
          "The number cubed is: "),
        floaty: 3.33,
    };

    cubeFloaty(&mut stringsAndFloat);


    println!("{}{}", stringsAndFloat.ownedString,
      stringsAndFloat.floaty);
}
```

Notice that we first create the struct outside of any function.

Later, inside a function, we `let` a variable which is an instance of our struct. We see this in our `stringsAndFloat` variable. This is where we fill in actual data.

As demonstrated in our `println!`, we access the fields of a struct with dot notation: `variableName.structField`.

Running this shows us:

```
/usr/bin/cargo run --color=always
    Finished dev [unoptimized + debuginfo] target(s) in
    0.0 secs
     Running `target/debug/postsScratchPad`

The number cubed is: 36.926037
```

In case you don't know, `.powi()` is a function which raises a number to an integer power. So above, we're doing `3.33^3`. There is also a `.powf()` function that allows us to raise a floating point number to a floating point number, so we could do `3.33^3.33`.

Something else to note is that our `stringsAndFloat` variable is mutatable. We need to have it this way so we can change the value of its fields like we do when we pass it to `cubeFloaty()`. We redefine the field `floaty` by cubing it.

As we've seen, structs make it beautifully simple to pass around a

bunch of types with just one variable/type.

Now go ahead and have some actual Cadbury Mini Eggs. You know you're drooling.

Chapter 3: Dead Or Alive

Stemming from the previous chapter, we can complicate things for ourselves a bit further. Now that we've got two types in our struct, let's add a third and see where that takes us.

```
#![allow(non_snake_case, non_camel_case_types)]


struct easterEgg {
    stringy: &str,
    ownedString: String,
    floaty: f64,
}

fn cubeFloaty(stringsAndFloat: &mut easterEgg) {
    stringsAndFloat.floaty = stringsAndFloat
      .floaty.powi(3);
}
```

```
fn main() {
    let mut stringsAndFloat: easterEgg = easterEgg {
        stringy: "The number ",
        ownedString: String::from("cubed is: "),
        floaty: 3.33,
    };

    cubeFloaty(&mut stringsAndFloat);


    println!("{}{}{}", stringsAndFloat.stringy,
      stringsAndFloat.ownedString,
      stringsAndFloat.floaty);
}
```

Running this, we get:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
error[E0106]: missing lifetime specifier
 --> src/main.rs:5:14
  |
5 |     stringy: &str,
  |              ^ expected lifetime parameter

error: aborting due to previous error
```

Now we've gone made the compiler unhappy :(

It's telling us that something odd is going on because of that borrowed (`&`) value. The `String` type from before didn't cause us any problems because we owned it.

The thing about Rust is that it does a whole mess of crap for us without us knowing it. For example, if we create a string literal/string slice like this: `let borrowedString = "borrowed";`, we didn't specify the type like we should, and we didn't specify a lifetime. What's going on behind the scenes is this: `let borrowedString: &'static str = "borrowed";`.

In Rust, everything has a lifetime. There is one special lifetime called `'static` which means that it lives as long as the program does.

Being that our struct field `stringy` is a borrowed value, Rust doesn't know how long that data should live. To fix our error, we give that field a custom lifetime like so:

```
#![allow(non_snake_case, non_camel_case_types)]


struct easterEgg {
    stringy: &'a str,
    ownedString: String,
    floaty: f64,
}

fn cubeFloaty(stringsAndFloat: &mut easterEgg) {
    stringsAndFloat.floaty = stringsAndFloat
      .floaty.powi(3);
}


fn main() {
    let mut stringsAndFloat: easterEgg = easterEgg {
        stringy: "The number ",
        ownedString: String::from("cubed is: "),
        floaty: 3.33,
    };

    cubeFloaty(&mut stringsAndFloat);


    println!("{}{}{}", stringsAndFloat.stringy,
      stringsAndFloat.ownedString,
      stringsAndFloat.floaty);
}
```

Which then yields:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
error[E0261]: use of undeclared lifetime name `'a`
 --> src/main.rs:5:15
```

```
  |
5 |     stringy: &'a str,
  |               ^^ undeclared lifetime

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0261`.
error: Could not compile `postsScratchPad`.
```

Great. Rust complains a lot. First it said we didn't give it a lifetime. Now it's unhappy because we didn't tell it what `'a` was. So let's fix it.

```
#![allow(non_snake_case, non_camel_case_types)]


struct easterEgg<'a> {
    stringy: &'a str,
    ownedString: String,
    floaty: f64,
}

fn cubeFloaty(stringsAndFloat: &mut easterEgg) {
    stringsAndFloat.floaty = stringsAndFloat
      .floaty.powi(3);
}


fn main() {
    let mut stringsAndFloat: easterEgg = easterEgg {
        stringy: "The number ",
        ownedString: String::from("cubed is: "),
        floaty: 3.33,
    };

    cubeFloaty(&mut stringsAndFloat);


    println!("{}{}{}", stringsAndFloat.stringy,
       stringsAndFloat.ownedString,
       stringsAndFloat.floaty);
}
```

This finally gives us:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in
    0.61 secs
     Running `target/debug/postsScratchPad`

The number cubed is: 36.926037
```

So what have we done here?

Rust doesn't know whether the borrowed value should live as long as `main() {}` or as `cubeFloatly() {}`, so, we tell it.

With the lifetime declaration (`<'a>`) when we create our struct, we say that wherever we create an instance of `easterEgg` with `let`, the borrowed value of `stringy` must live as long as that scope does. In this case, the scope is `main() {}`.

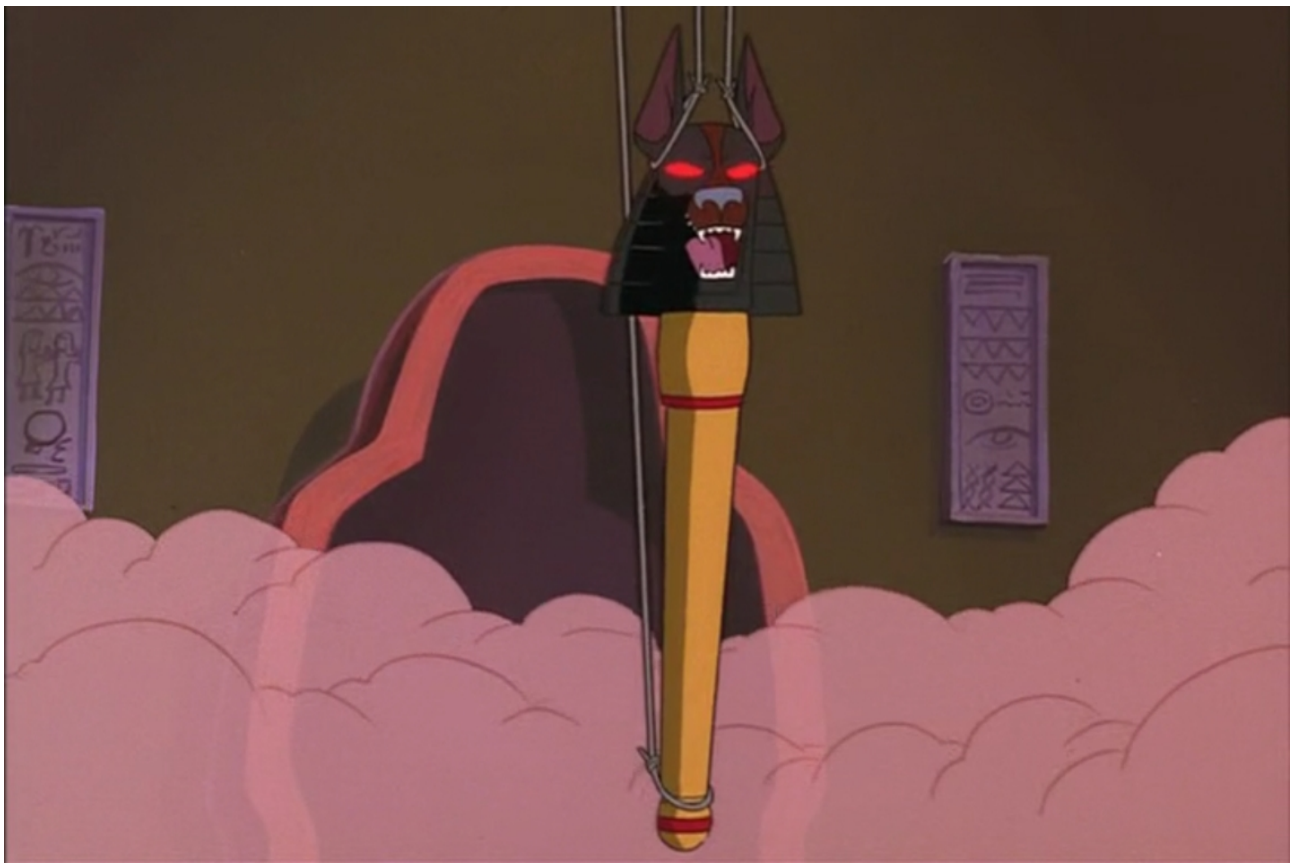
We set one lifetime at the beginning of the struct (we can create as many lifetimes as we want). The syntax for that is `<'lifetimeName>`. Typically, everyone just uses `'a`, because it's short. Then we specify that lifetime for the `&str` that we want to apply it to like: `&'a str`.

Dealing with lifetimes shows how much Rust lets us take for granted.

Just remember when dealing with borrowed values that we have to be explicit whether it is dead or alive :)

Click To Get The Joke

Chapter 4: Invisibo



Enums, unlike structs, effectively don't exist. They don't (have to) have any data at all. They don't even have any meaning except for the name of the enum and the names of its variants. So why the crap would we want to use them? Let's talk about a three way switch.

```
#![allow(non_snake_case, non_camel_case_types)]

 fn takesStr(stringy: &str) {
     match stringy {
         "imperialGB" => println!("The state of the
           switch is imperialGB"),
         "imperialUS" => println!("The state of the
           switch is imperialUS"),
         "metric" => println!("The state of the switch
           is metric"),
```

```
        _ => println!("We had a typo"),
    }
}

fn main() {
    let borrowed: &str = "imperial GB";

    takesStr(borrowed);
}
```

When matching on string literals, we have to match on `_`, which is effectively like an `else {}` block in an `if/else` statement.

Let's see what happens when we run it.

```
/usr/bin/cargo run --color=always
    Blocking waiting for file lock on build directory
   Compiling postsScratchPad v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in
    2.07s
     Running `target/debug/postsScratchPad`

We had a typo
```

So, why did our match statement not work? Everything looks correct.

Oh, crap. We did our `let` statement for the variable `borrowed` with the value as "`imperial GB`", not "`imperialGB`". The fact that we had a space means that none of our explicit match arms will actually get used, screwing our program over.

What if we didn't do a `println!()`, but instead had our program perform some maths? Now it will just fail silently. Our program will still compile, because strings aren't verified by the compiler. They are only evaluated at runtime.

This seems like an awful lot of responsibility on us to make sure that every character is exactly how it should be. There has to be a way for us to be able to make mistakes and have the compiler check things for us instead.

Enums to the rescuuue!

```
#![allow(non_snake_case, non_camel_case_types)]

enum imperialOrMetric {
    imperialGB,
    imperialUS,
    metric,
}

fn printsSwitchState(switch: imperialOrMetric) {
    match switch {
        imperialOrMetric::imperialGB => println!("The
          state of the switch is imperialGB"),
        imperialOrMetric::imperialUS => println!("The
          state of the switch is imperialUS"),
        imperialOrMetric::metric => println!("The state
          of the switch is metric"),
    }
}

fn main() {
    let switchy: imperialOrMetric = imperialOrMetric
      ::imperialGB;

    printsSwitchState(switchy);
}
```

Whereas struct's have fields, enum's have variants. We create an enum with a relevant name and corresponding logically named variants. The way we create an instance of an enum is with `enumName::enumVariant`.

After running this program, we get:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
warning: variant is never constructed: `imperialUS`
 --> src/main.rs:5:5
  |
5 |     imperialUS,
  |     ^^^^^^^^^^
  |
```

```
  = note: #[warn(dead_code)] on by default

warning: variant is never constructed: `metric`
 --> src/main.rs:6:5
  |
6 |     metric,
  |     ^^^^^^

    Finished dev [unoptimized + debuginfo] target(s) in
    0.67s
     Running `target/debug/postsScratchPad`

The state of the switch is imperialGB
```

First, Rust gives us a warning that we created variants that we didn't use, but it's only because this is an incomplete example. Then we see the output we expected. :)

The gorgeous thing about enums is that because they are verified by the compiler, if we make a mistake, `rustc/cargo` will not build our program.

For example:

```
#![allow(non_snake_case, non_camel_case_types)]

enum imperialOrMetric {
    imperialGB,
    imperialUS,
    metric,
}

fn printsSwitchState(switch: imperialOrMetric) {
    match switch {
        imperialOrMetric::imperialGB => println!("The
          state of the switch is imperialGB"),
        imperialOrMetric::imperialUS => println!("The
          state of the switch is imperialUS"),
        imperialOrMetric::metric => println!("The state
          of the switch is metric"),
    }
}
```

```
fn main() {
    let switchy: imperialOrMetric = imperialOrMetric
      ::imperialgB;

    printsSwitchState(switchy);
}
```

This yields an error:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
error[E0599]: no variant named `imperialgB` found for
type `imperialOrMetric` in the current scope
  --> src/main.rs:18:37
   |
3  | enum imperialOrMetric {
   | --------------------- variant `imperialgB` not
                           found here
...
18 |     let switchy: imperialOrMetric =
         imperialOrMetric::imperialgB;
   |
^^^^^^^^^^^^^^^^^^^^^^^^^^ variant not found in
                             `imperialOrMetric`
   |
   = note: did you mean `variant::imperialGB`?

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0599`.
error: Could not compile `postsScratchPad`.

To learn more, run the command again with --verbose.
```

Not only does Rust tell us that we made a mistake, it is super helpful. It analyses the variants of our enum, looks at similar spellings, and suggests to us what it thinks the proper variant might be.

And in this case it is correct :)

Now we can program away and not worry about silly misspellings because we can rely on the compiler to verify everything for us.

Chapter 5: Unwrap Me

Now that we've established precisely what enums are(n't), we'll look at another feature that enums have: nested types. We can actually put anything we want inside an enum:

```
enum unwrapMe {
    hasFloat(f64),
}
```

Now we'll create a function to do things with it:

```
#![allow(non_snake_case, non_camel_case_types)]

enum unwrapMe {
    hasFloat(f64),
}
```

```
fn main() {
    let nestedNumber: unwrapMe = unwrapMe
      ::hasFloat(3.33);

    println!("{}", nestedNumber * 3.33);
}
```

That didn't work as intended:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
error[E0369]: binary operation `*` cannot be applied to
type `unwrapMe`
  --> src/main.rs:10:20
   |
10 |     println!("{}", nestedNumber * 3.33);
   |                    ^^^^^^^^^^^^^^^^^^^
   |
   = note: an implementation of `std::ops::Mul` might be
     missing for `unwrapMe`

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0369`.
error: Could not compile `postsScratchPad`.

To learn more, run the command again with --verbose.
```

As per usual, we have to be completely explicit for the compiler. We've told it to multiply an instance of our enum with an `f64`, not the `f64` inside of it. To be able to get to the nested value, we need to use another one of Rust's features: the `match` statement.

```
#![allow(non_snake_case, non_camel_case_types)]

enum unwrapMe {
    hasFloat(f64),
}
```

```
fn stripOut(enumInstance: unwrapMe) -> f64 {
    match enumInstance {
        unwrapMe::hasFloat(number) => number,
    }
}

fn main() {
    let nestedNumber: unwrapMe = unwrapMe
      ::hasFloat(3.33);

    println!("{}", stripOut(nestedNumber) * 3.33);
}
```

This gives us what we wanted:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in
    1.96s
     Running `target/debug/postsScratchPad`

11.0889
```

What we've done, is created a function that is the equivalent of `.unwrap()`. We can use `.unwrap()` to accomplish the same thing as our `stripOut()` function, but only for standard types. Since we created our type `unwrapMe`, we have to create something from scratch.

If we search the Rust standard library for `unwrap`, and pick one of the `unwrap` methods for an enum we'll see some very similar code. As it turns out, all of the `.unwrap()` code is basically the same as what we've written in `stripOut()`.

We `match`ed on the whole enum declaration. The syntax by itself looks like:

```
enum::variant(nestedType) => nestedType,
```

This is referred to as a match arm. A `match` statement is like a really powerful `if` statement. It allows us to get the nested type of our `enum` out so we can do

fun things with it. In our program above, we just returned it. But, we could have done something like:

```
unwrapMe::hasFloat(number) =>
  { println!("The number is: {}, number) },
```

We could have called any function there.

Next time we'll talk about how to fancy our function up. :)

Chapter 6: Unwrap Me Nicely

We're going to get into implementations and functions that take `self`. A function that takes `self` as an argument changes the order of a function call from `functionName(argumentVariable)` to `argumentVariable.functionName()`. This is a huge benefit because we can make our function calls read more like a sentence.

The example above is pretty easy to read, but let's say we need to do three function calls on one variable:

```
ouncesToGrams(poundsUSToOunces(poundsGBToUS(
  weightInGBPounds)));
```

Now this is just ridiculous.

We have to start at the center of the nesting dolls and read outwards. If we change this to an implementation that takes `self`, we can make our calls look

like this:

```
weightInGBPounds.poundsGBToUS().poundsUSToOunces()
  .ouncesToGrams()
```

This reads clear as day. We start with a GB pound, then change that to US pounds, then to ounces, then to grams.

Here's how our program changes to use an implementation. Also, our function changes to `self` as the argument:

```
#![allow(non_snake_case, non_camel_case_types)]

enum unwrapMe {
    hasFloat(f64),
}

impl unwrapMe {
    fn stripOut(self) -> f64 {
        match self {
            unwrapMe::hasFloat(number) => number,
        }
    }
}

fn main() {
    let nestedNumber: unwrapMe = unwrapMe
      ::hasFloat(3.33);

    println!("{}", nestedNumber.stripOut() * 3.33);
}
```

Which outputs:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in
    5.90s
     Running `target/debug/postsScratchPad`

11.0889
```

`self` takes the place of whatever we implement our function(s) for. In this case, it is our `enum` `unwrapMe`.

We can also add a `new()` function to our implementation:

```
#![allow(non_snake_case, non_camel_case_types)]

enum unwrapMe {
    hasFloat(f64),
}

impl unwrapMe {
    fn stripOut(self) -> f64 {
        match self {
            unwrapMe::hasFloat(number) => number,
        }
    }

    fn new(number: f64) -> Self {
        unwrapMe::hasFloat(number)
    }
}

fn main() {
    let nestedNumber: unwrapMe = unwrapMe
      ::hasFloat(3.33);

    println!("{}", nestedNumber.stripOut() * 3.33);

    println!("{}", unwrapMe::new(3.33).stripOut());
}
```

This outputs:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in
    0.62s
     Running `target/debug/postsScratchPad`

11.0889
3.33
```

Our `new()` function takes one argument, an `f64`. It also returns `Self`, which is different than `self`. `Self` here refers to whatever we have an `impl` for, in this case `unwrapMe`. It's a shortcut way of saying, "go back up to the top of the `impl` and return whatever it is".

Essentially, use `self` as an argument, `Self` as a return type.

The way we call a non-`self` function in an `impl` block is with the `::` syntax. So we say `unwrapMe::new(3.33)`. The great part is that now we can rewrite our program using our `impl` block instead of creating the `nestedNumber` variable:

```
#![allow(non_snake_case, non_camel_case_types)]

enum unwrapMe {
    hasFloat(f64),
}

impl unwrapMe {
    fn stripOut(self) -> f64 {
        match self {
            unwrapMe::hasFloat(number) => number,
        }
    }

    fn new(number: f64) -> Self {
        unwrapMe::hasFloat(number)
    }
}

fn main() {
    println!("{}", unwrapMe::new(3.33)
      .stripOut() * 3.33);
}
```

Which outputs:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in
    0.57s
     Running `target/debug/postsScratchPad`

11.0889
```

Here, we've done everything in one line. We handled our enum creation, extracting the nested value, and performing maths all in one clean, readable line.

Rule of thumb, always avoid doing things like this:

Click To Get The Joke

Chapter 7: Traity Trait

In the last chapter, we implemented methods (functions in an `impl` block) for our custom enum. What happens if we want to implement something for a type we didn't create?

This is where traits come into play.

We're going to start with a program like this:

```
#![allow(non_snake_case, non_camel_case_types)]

fn cubeIt(inputNumber: f64) -> f64 {
    inputNumber.powi(3)
}

fn main() {
    let cubeMe: f64 = 3.33;
```

```
    println!("{}", cubeIt(cubeMe));
}
```

This outputs:

```
/usr/bin/cargo run --color=always
    Finished dev [unoptimized + debuginfo] target(s) in
    0.01s
     Running `target/debug/postsScratchPad`

36.926037
```

Let's just do what we did previously and change `cubeIt()` into a `self` method:

```
#![allow(non_snake_case, non_camel_case_types)]

impl f64 {
    fn cubeIt(self) -> f64 {
        self.powi(3)
    }
}

fn main() {
    let cubeMe: f64 = 3.33;

    println!("{}", cubeMe.cubeIt());
}
```

When we run this, we get:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
error[E0390]: only a single inherent implementation
marked with `#[lang = "f64"]` is allowed for the `f64`
primitive
 --> src/main.rs:3:1
  |
3 | / impl f64 {
4 | |     fn cubeIt(self) -> f64 {
5 | |         self.powi(3)
6 | |     }
```

```
7 | | }
  | |_^
  |
help: consider using a trait to implement these methods
 --> src/main.rs:3:1
  |
3 | / impl f64 {
4 | |     fn cubeIt(self) -> f64 {
5 | |         self.powi(3)
6 | |     }
7 | | }
  | |_^

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0390`.
error: Could not compile `postsScratchPad`.

To learn more, run the command again with --verbose.
```

So, we've tried to just straight implement something for an `f64`, which isn't a type that we created. It comes from the standard library.

`rustc` is telling us that we need to create a trait. A trait is a block just like an `impl`, but it only contains a method's signature; no body.

Like this:

```
trait mathyMaths {
    fn cubeIt(&self) -> f64;
}
```

We created a trait called `mathyMaths` (name it whatever you want, just keep it as logical as you can for what it does). The method `cubeIt()` inside the `trait` block is stripped down. It just tells `rustc` what to expect when we do implement it *for a type*.

```
#![allow(non_snake_case, non_camel_case_types)]

trait mathyMaths {
    fn cubeIt(self) -> f64;
}

impl mathyMaths for f64 {
    fn cubeIt(self) -> f64 {
        self.powi(3)
    }
}

fn main() {
    let cubeMe: f64 = 3.33;

    println!("{}", cubeMe.cubeIt());
}
```

This yields:

```
/usr/bin/cargo run --color=always
    Finished dev [unoptimized + debuginfo] target(s) in
    0.00s
     Running `target/debug/postsScratchPad`

36.926037
```

We've implemented our trait *for* `f64`. The cool thing is, we can implement it for other types as well:

```
#![allow(non_snake_case, non_camel_case_types)]

trait mathyMaths {
    fn cubeIt(self) -> f64;
}

impl mathyMaths for i64 {
    fn cubeIt(self) -> f64 {
        (self as f64).powi(3)
    }
}
```

```
impl mathyMaths for f32 {
    fn cubeIt(self) -> f64 {
        (self as f64).powi(3)
    }
}

impl mathyMaths for f64 {
    fn cubeIt(self) -> f64 {
        self.powi(3)
    }
}

fn main() {
    let cubeMei64: i64 = 3;

    println!("{}", cubeMei64.cubeIt());

    let cubeMef32: f32 = 3.33;

    println!("{}", cubeMef32.cubeIt());

    let cubeMef64: f64 = 3.33;

    println!("{}", cubeMef64.cubeIt());
}
```

This gives us:

```
/usr/bin/cargo run --color=always
    Finished dev [unoptimized + debuginfo] target(s) in
    0.01s
     Running `target/debug/postsScratchPad`

27
36.92603446195227
36.926037
```

You'll notice we've introduced something new here: `as`. This is what is called type casting (insert acting reference here :)).

In Rust, we can cast one type as another type. So in our `i64` and `f32` implementations, we change the input to be an `f64`, because like dissolves like:

we can only do maths on the same type. Since we're outputting an `f64`, we need to convert all types to that.

Remember, you can type cast in programming, but don't do it to actors. They're real people.

Chapter 8: Dereference What?

Continuing on from the last chapter, let's say we wanted to use a borrowed value for whatever reason (like we want our original variable to live after the function call):

```
#![allow(non_snake_case, non_camel_case_types)]

trait mathyMaths {
    fn cubeIt(&self) -> f64;
}

impl mathyMaths for f32 {
    fn cubeIt(&self) -> f64 {
        (self as f64).powi(3)
    }
}
```

```
impl mathyMaths for f64 {
    fn cubeIt(&self) -> f64 {
        self.powi(3)
    }
}

fn main() {
    let cubeMe: f32 = 3.33;

    println!("{}", cubeMe.cubeIt());
}
```

When we implement `mathyMaths` for `f32`, we get yelled at:

```
/usr/bin/cargo run --color=always
   Compiling postsScratchPad v0.1.0
error[E0606]: casting `&f32` as `f64` is invalid
 --> src/main.rs:9:9
  |
9 |         (self as f64).powi(3)
  |         ^^^^^^^^^^^^^ cannot cast `&f32` as `f64`
  |
help: did you mean `*self`?
 --> src/main.rs:9:10
  |
9 |         (self as f64).powi(3)
  |          ^^^^

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0606`.
error: Could not compile `postsScratchPad`.

To learn more, run the command again with --verbose.
```

As we see, we `cannot cast '&f32' as 'f64'`. Rust is telling us that in order to cast one primitive type as another, it must be an owned value. We now can learn about a special operator in Rust: the asterisk `*`. We use the asterisk, `*`, to *dereference* a value.

Think of it like this: the `*` operator is equal to `-&`. So our `&self` in `fn cubeIt(&self) -> f64;` becomes just `self`, an owned value.

We can modify our program to look like this:

```
#![allow(non_snake_case, non_camel_case_types)]

trait mathyMaths {
    fn cubeIt(&self) -> f64;
}

impl mathyMaths for f32 {
    fn cubeIt(&self) -> f64 {
        (*self as f64).powi(3)
    }
}

impl mathyMaths for f64 {
    fn cubeIt(&self) -> f64 {
        self.powi(3)
    }
}

fn main() {
    let cubeMe: f32 = 3.33;

    println!("{}", cubeMe.cubeIt());
}
```

Which gives us:

```
/usr/bin/cargo run --color=always
    Finished dev [unoptimized + debuginfo] target(s) in
    0.01s
     Running `target/debug/postsScratchPad`

36.92603446195227
```

If Rust ever complains about being given a reference, slap a `*` on it and tell it:

Click To Get The Joke

Chapter 9: A Faster Loop-The-Loop

Let's make things better today. We're humans. We're horrible at repetition. We like entertainment. For repetitive tasks, we have tools like loops that we can utilise.

An example from BrewStillery looked like this originally:

```
pub fn increaseABVMaths(allInputs: &increaseABVData) ->
finalSugarFloat {
    let mut newStartingBrix: f64 = allInputs
      .startingBrix;

    let mut newEstimatedABV: f64 =
      realABVAndAttenuation(newStartingBrix,
      FINAL_BRIX_IDEAL).0;

    while newEstimatedABV < allInputs.desiredABV {
        newStartingBrix = newStartingBrix + 0.001;
```

```
            newEstimatedABV =
              realABVAndAttenuation(newStartingBrix,
              FINAL_BRIX_IDEAL).0;
        }
        ...
```

What's going on is `newStartingBrix` is a user inputted `f64`.

`newEstimatedABV` is set to be the ABV return value of the function `realABVAndAttenuation()`.

Then, we say `while` `newEstimatedABV` is less than the user's desired ABV, increment `newStartingBrix` by `0.001`, use that value as an input in `realABVAndAttenuation()`, and store the resultant ABV in `newEstimatedABV`.

To our perception, this works basically instantly on modern hardware. We can still improve things though. Rust has a fantastic crate called Rayon, which allows us to parallelise Rust's `.iter()` method. We do that by changing it from `.iter()` to `.par_iter()`. It's as straightforward as it can be.

The issue is that nowhere in the Rust documentation does it say anything about parallelising a while loop. There are great examples of `for` loops, but that doesn't help us here.

What to do?

Well, we have to work with some goofy bounds. In Rust, iterators are only allowed to use integers. So we can't create one like `0.0..33.0`.

In this particular example, I know the range that brix can be. It is a sugar density which is measured by a refractometer. It's maximum value is `32`.

If we employ a bit of mathy inversion trickery, we can come up with something like this: `(newStartingBrix * 1000.0) as u32)..33000)`.

All we're doing here is getting rid of the decimal (`f64`) and turning it into a large integer (`u32`).

Then we convert it back to a float inside the `.map()` by doing this: `let tempBrix: f64 = ( mapBrix as f64 / 1000.0 ) + 0.0001;`

I changed the range from `32` to `33` because I wanted a little more buffer on the top end, and I love 3's.

Something to note, is that if our iterator goes past `33000`, this will crash our program. There are input guards in a preceeding function that rejects any input greater than `32`.

The final code looks like this:

```
pub fn increaseABVMaths(allInputs: &increaseABVData) ->
finalSugarFloat {
    let mut newStartingBrix: f64 = allInputs
      .startingBrix;

    newStartingBrix = (((newStartingBrix * 1000.0)
      as u32)..33000)
        .into_par_iter()
        .map(|mapBrix| {
            let tempBrix: f64 = (mapBrix as f64
              / 1000.0) + 0.0001;
            let tempABV: f64 = realABVAndAttenuation(
              tempBrix,FINAL_BRIX_IDEAL).0;

            (tempBrix, tempABV)
        })
        .find_first(|(_tempBrix, tempABV)| {
            allInputs.desiredABV < *tempABV
        })
        .expect("increaseABVPrep(), newEstimatedABV")
        .0;
    ...
```

To run through two `for` loop examples, we can parallelise our `twoArraySum()`:

```
pub fn twoArraySum(firstArray: [f64; 81], secondArray:
[f64; 81]) -> f64 {
    let mut sum: f64 = 0.0;

    // this does the weird spreadsheet thing of
    // (array1[0] * array2[0]) + (array1[0] *
    // array2[0]) ...
```

```
    for index in 0..81 {
        sum = sum + firstArray[index] *
          secondArray[index];
    }

    sum
}
```

Which becomes:

```
pub fn twoArraySum(firstArray: [f64; 81], secondArray:
[f64; 81]) -> f64 {
// this does the weird spreadsheet thing of (array1[0]
// * array2[0]) + (array1[0] * array2[0]) ...
    firstArray
        .par_iter()
        .zip(secondArray.par_iter())
        .map(|(first, second)|{
            first * second
        }).sum()
}
```

If you're not familiar with it, the `.zip()` method pairs up iterators. We then throw that into `.map()` and just replicate the maths from the original `for` loop.

And finally, our `threeArraySum()`:

```
pub fn threeArraySum(firstArray: [f64; 81],
secondArray: [f64; 81], thirdArray: [f64; 81]) -> f64 {
    let mut sum: f64 = 0.0;

    // this does the weird spreadsheet thing of
    // (array1[0] * array2[0]) + (array1[0] *
    // array2[0]) ...
    for index in 0..81 {
        sum = sum + firstArray[index] *
          secondArray[index] * thirdArray[index];
    }

    sum
```

Becomes:

```
pub fn threeArraySum(firstArray: [f64; 81],
secondArray: [f64; 81], thirdArray: [f64; 81]) -> f64 {
    // this does the weird spreadsheet thing of
    // (array1[0] * array2[0]) + (array1[0] *
    // array2[0]) ...
    firstArray
        .par_iter()
        .zip(secondArray.par_iter())
        .zip(thirdArray.par_iter())
        .map(|((first, second), third)|{
            first * second * third
        }).sum()
```

The unique thing here is what's going in `.map()`. The syntax for mapping multiple `.zip()`'s is a bit weird. It only accepts a tuple, so we have to group them in nested pairs. If we were going to do a fourth iterator, it would look like this: `(((first, second), third), fourth)` for the `.map()` arguments.

Now, we've *safely*

Chapter 10: Have A Seat

Previously we learned how to parallelise our functions. Now let's see how we can measure the performance increase for our different examples. While version:

```
pub fn increaseABVMaths(allInputs: &increaseABVData) ->
finalSugarFloat {
    let mut newStartingBrix: f64 = allInputs
      .startingBrix;

    let mut newEstimatedABV: f64 =
      realABVAndAttenuation(newStartingBrix,
      FINAL_BRIX_IDEAL).0;

    while newEstimatedABV < allInputs.desiredABV {
        newStartingBrix = newStartingBrix + 0.001;
```

```
        newEstimatedABV =
          realABVAndAttenuation(newStartingBrix,
          FINAL_BRIX_IDEAL).0;
    }
    ...
```

Parallelised version:

```
pub fn increaseABVMaths(allInputs: &increaseABVData) ->
finalSugarFloat {
    let mut newStartingBrix: f64 = allInputs
      .startingBrix;

    newStartingBrix = (((newStartingBrix * 1000.0)
      as u32)..33000)
        .into_par_iter()
        .map(|mapBrix| {
            let tempBrix: f64 = (mapBrix as f64
              / 1000.0) + 0.0001;
            let tempABV: f64 =
              RealABVAndAttenuation(
              tempBrix,FINAL_BRIX_IDEAL).0;

            (tempBrix, tempABV)
        })
        .find_first(|(_tempBrix, tempABV)| {
            allInputs.desiredABV < *tempABV
        })
        .expect("increaseABVPrep(), newEstimatedABV")
        .0;
    ...
```

For benchmarking purposes, I changed the range on the parallel version to be like this: `newStartingBrix = (((newStartingBrix * 1000.0) as u32)..4294967000)` The end range is the largest number a `u32` can hold. That way it would take quite a bit more maths to finish.

To do the benchmark, I wrote the following in the same module/file:

```
#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[bench]
    fn bench(bencher: &mut Bencher) {
        let allInputs: increaseABVData = increaseABVData
{
            startingBrix: 12.0,
            desiredABV: 4294967.0,
            currentVolume: 45.0,
            increaseABVUnitsEnum: imperialOrMetric
              ::Metric,
        };

        bencher.iter(|| increaseABVMaths(&allInputs));
    }
}
```

And then to `main.rs`, I added:

```
#![feature(test)]
extern crate test;
```

to the top of the file.

`cargo bench` currently requires rust nightly to run. The best thing to do is something like the following:

```
rustup run nightly cargo bench > while
```

and

```
rustup run nightly cargo bench > rayon
```

And comment/uncomment each respective version between running each command.

Then run

```
cargo install cargo-benchcmp
```

Following that, you can do

```
cargo benchcmp rayon while
```

Which gave us the following result:

```
name
functions::increaseABV::tests::bench

rayon ns/iter       while ns/iter       diff ns/iter
6,254,142           2,788,014           -3,466,128

Diff %              speedup
-55.42%             x 2.24
```

This shows that our parallelised version is 2.24 times faster than the plain while loop.

For `reconstructedTransmissionData()` we got this:

```
name
light::lightFunctions::tests::bench

rayon ns/iter       while ns/iter       diff ns/iter
17,530              10,450              -7,080

Diff %              speedup
-40.39%             x 1.68
```

For `twoArraySum()`, we got this:

```
name
light::lightFunctions::tests::bench

rayon ns/iter       while ns/iter       diff ns/iter
12,455              104                 -12,351

Diff %              speedup
-99.16%             x 119.76
```

For `threeArraySum()`, we got this:

```
name
light::lightFunctions::tests::bench

rayon ns/iter       while ns/iter       diff ns/iter
13,832              78                  -13,754

Diff %              speedup
-99.44%             x 177.33
```

For `computedIlluminantData()`, we got this:

```
name
light::colour::computedIlluminantData::tests::bench

rayon ns/iter       while ns/iter       diff ns/iter
14,170              86                  -14,084

Diff %              speedup
-99.39%             x 164.77
```

Now you can sit and chill, because we not only know we improved our functions, but we can really see how much :)

About The Author

Meade Kincke is the CTO for Chainetix and has been writing in Rust since January 2017. He has built a beer, wine, and spirits tool called BrewStillery. Meade is also a contributer to the Rust compiler. At the time of publishing this book, Meade is working on stabilising `const fn`.

Look out for Volume II and visit the Code Artistry website for regular posts.