YARR is available on the web and as a PDF. Nicole's post is also helpful.

Introduction

This is a crash course in the Rust programming language. It's written with a specific audience in mind: software engineers who know how to program in a high-level language but aren't familiar with lower-level programming and want to learn to write Rust *quickly*.

It does not aim to be a comprehensive course, and it's best done with a Rust buddy who can help you through tricky bits. The aim of this course is to **teach you just enough to pair program** so that you can learn by doing with others after this course.

This course contains exercises which should be done as you go through to check your understanding, and it also contains exercises which may be done independently or in a group. We highly encourage pairing for these with someone who's experienced or a fellow learner. Learning together is highly effective, especially for new, difficult concepts like those Rust may expose you to.

Rust has a notoriously difficult learning curve because it forces you at the language level to deal with concepts that are typically hidden (such as lifetimes). Struggling with these concepts is expected. When you come out of this course, these concepts will still be fuzzy and unclear. That's expected. This course will set you off on your journey, and your understanding and confidence in these concepts will grow over time.

This is just the beginning.

Let's get started.

Contributing, Bug Reports, and License

Please see the repo for contribution and license information.

Bug reports and feedback may be sent to the discussion mailing list.

Setting Up Your Environment

Before we can move forward to learning Rust, we need to have a working local environment. There are a lot of online tools to run Rust code, but since the target here is to start pairing, local is best.

The preferred way to setup your local Rust toolchain is with rustup. Go ahead and install that, then come back here. You can check if you have it installed by checking which version it is:

```
$ rustup --version
```

You should see something like this as the output:

```
rustup 1.25.1 (bb60b1e89 2022-07-12)
info: This is the version for the rustup toolchain manager, not the
rustc compiler.
info: The currently active `rustc` version is `rustc 1.68.0-nightly
(cc47b0699 2023-01-08)`
```

If you get an error, try closing your terminal and opening a new one; the installation adds itself to your path and that doesn't get reloaded automatically.

The Rust language moves quickly, so we can benefit from a lot of improvements by using the nightly toolchain instead of stable. Set it as your default like so:

```
$ rustup default nightly
```

And then make sure it's up to date:

```
$ rustup update
```

That's all you need, and you should be up and running now! Let's go take a look at our first Rust code.

Rust Analyzer

Before you start writing too much Rust, you will want to also install rust-analyzer. This is a tool which provides a lot of nice features in your IDE or editor, like completions and refactoring help. Crucially to me, it also can display inline types annotations for inferred types, which can *greatly* aid understanding of Rust programs.

It's by no means necessary for working with Rust, but if you're going to do any work with Rust it's an extremely helpful tool.

Instructions on how to configure it are out of scope of this guide, as it will be different for every editor.

> **Exercise**: Look up instructions for your editor of choice and set it up to use Rust Analyzer.

Hello World

In the time-worn tradition, our first Rust program will be getting our computer to greet this wide world of ours.

Let's start by making a directory to work out of[1]:

```
$ cd ~
$ mkdir yarr
$ cd yarr
```

For the rest of this course, we'll assume that you're in the `~/yarr` directory.

Now let's make our first Rust program. When we installed `rustup` and the Rust toolchain, it included a tool called `cargo`. This is the Swiss army knife for Rust. It takes care of a *lot* of things for you, including:

- Installing and updating packages
- Creating new projects
- Running your tests
- Executing your binary
- Formatting and linting your code
- Publishing your library

We'll learn by doing, but feel free to explore the other functions with `cargo help` or in the Cargo docs.

To create a new project, you can use `cargo new`:

```
$ cargo new hello_world
Created binary (application) `hello_world` package
```

If you print out the directory that was created, there are a few files:

```
hello_world/
├── Cargo.toml
└── src
    └── main.rs
```

Now we can run the program:

```
$ cd hello_world
$ cargo run
   Compiling hello_world v0.1.0 (/home/ntietz/Code/ntietz/rust-crash-
course/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.55s
     Running `target/debug/hello_world`
Hello, world!
```

And we printed it out!

Whoops, we cheated our way to "hello world" without actually writing the program! But we know our toolchain works. Let's delete the existing file and write our own:

```
$ rm src/main.rs
```

Now open `src/main.rs` as a new file in your editor of choice, and write this program:

```rust
fn main() {
    println!("Ahoy, matey!");
}
```

Now when you run it, you should see our new, corrected, greeting:

```
$ cargo run
   Compiling hello_world v0.1.0 (/home/ntietz/Code/ntietz/rust-crash-
course/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.27s
     Running `target/debug/hello_world`
Ahoy, matey!
```

A few quick notes on the program we wrote, then we'll move on:

- This declares the `main` function, which has no return value here but can have a return value in some cases; we'll cover that later in Error Handling.
- It does only one thing, which is invoking `println!`. This looks like a function, but is a macro. Macros always end in `!`, so when you see that you know it's a macro. You can think of macros as fancy powerful functions, and use them just like you would use a function. *We won't write our own in this course.*

Exercises:

1. Repeat the hello world program from scratch in a new directory, going by memory as much as possible.
2. Find and read the documentation on `println!`. What do you find surprising? What do you find normal?

And now we're ready to move on! Stretch, shake it out, and then move on to learn about variables, functions, and control flow!

---

[1] If you prefer a different working directory, feel free to do as you wish!

Variables

This chapter has a few concepts to cover:

- Creating variables
- Giving a variable a type
- Primitive types

Creating Variables

In Rust, you can create a variable much like you would in other languages:

```rust
let x = 10;
```

The base syntax is straightforward: give it a name, give it a value. This creates an **immutable** variable. If you try to change it after this, you'll get an error:

```rust
let x = 10;
x = 11;
```

If you do this in a file and try to compile it, you'll get a pretty helpful error:

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:3:5
  |
2 |     let x = 10;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     x = 11;
  |     ^^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
```

To fix this error, we learn another keyword: `mut`. This keyword is used to create *mutable* variables (or references).

If we run this program instead, it will compile:

```
let mut x = 10;
x = 11;
```

A note on `mut`

The relation between the `mut` keyword and the concept of mutability is apparent in its name and in its usage: it allows you to mutate things.

However, this hides another way of thinking about it which is very helpful in the context of references later on. How do you achieve mutability? By enforcing **exclusive access**: when there is one reference to a mutable variable, there can be no others (but you can use that one multiple times). So a `mut` reference is also an *exclusive* reference. (We'll talk about references later on.)

More on that later in the ownership and references section!

Type Annotations

Rust is a statically typed language. Everything is given a type at compile time. This does *not* mean, however, that we need to write types for everything! Like many modern languages, it employs **type inference** to figure out what types things are if you don't say. That's how Rust was able to compile our code up above without any types on it.

If you use a tool like rust-analyzer, you can see inline type hints. If you use that, you'll be able to see which types are inferred for a given variable, which is invaluable.

Sometimes, though, you just have to write the types down yourself! The compiler can't always figure out what you meant, and it can make it more clear to other programmers and the compiler what you *intended*—so if the inferred type doesn't match what you intended, now that's caught at compile time rather than run time.

The syntax for type annotations will feel somewhat familiar if you're used to Python or TypeScript, which have similar syntax. Let's say you're declaring a string and a 32-bit unsigned integer. You could write:

```
let name: &str = "Captain Blackbeard";
let age: u32 = 35;
```

In a variable initializer, the type annotation is the `: <type>` portion. There are a few other places you'll see type annotations, like in functions and closures and structs. We'll cover those when we get there, but they're all of this form with a colon and a type.

Primitive Types

Rust has a bunch of primitive types to help you express what you want to write! The primitives are well documented in the Rust docs, which you can look to for more details.

- Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, and `usize` (8-bit through 128-bit numbers, respectively) These can be written with bare literals like `123`, or you can append on the integer type, like `123u8`. `usize` is a pointer-sized unsigned integer; on 64-bit systems, this is often a 64-bit integer, and it's the type you use to index into collections.
- Signed integers: `i8`, `i16`, `i32`, `i64`, `i128`, and `isize`. These can be written with bare literals like `456`, or you can append on the integer type, like `456i32`. `isize` is a pointer-sized signed integer, which is handy for representing the difference between two indexes.
- Floating point numbers: `f32` and `f64`. These can be written with bare literals like `1.2`, or they can be written `1.2f32` or `1.2f64`.
- Characters: `char` is a 4-byte Unicode character. These can be written with single quotes around them, like `'a'`.
- Booleans: `bool` is `true` or `false`.

- Strings: `str` is the primitive type for a string. String handling is complicated in Rust[1], but this is the primitive type, which is a UTF-8 encoded string (or "string slice"). It's usually seen in the reference form, `&str`, but more on references later. Since strings are UTF-8 encoded, characters have variable-width encoding, which complicates accessing characters at specific indices.
- Unit: `()` is the "unit type". This can be written with a literal `()`, and it basically means... nothing. It's an empty, 0-element tuple, and it's the equivalent of `void` in other languages.

There are also a couple of other types worth mentioning here which aren't technically primitives, but are formed from them.

- Tuples: `(u8, bool)` is the type of a two-tuple of an unsigned 8-bit integer and a boolean. You can make a tuple from as many elements as you want, practically. A tuple value is of the form `(123, true)`, which would have the previous type.
- Arrays: an array in Rust is a fixed-size list of values, with the size as part of its type. The array `[0, 1, 2, 3, 4]` may have the type `[u8; 5]`, for example.

Casting Between Types

Sometimes you have a primitive of one type, but you want it in another type! This happens often with numbers, where you'll have (say) an 8-bit integer but need to use a 64-bit integer for something. Rust does not do any implicit casting for you. You must explicitly say that you want it to happen, which helps prevent overflow errors.

Casting with the `as` keyword is straightforward. You take the variable, and say `as <type>` for what you want to cast it into. If it can't be done, the compiler will tell you! You do have to be careful to ensure that if you cast to a *smaller* size value, that you won't overflow anything. The behavior you get is well-defined but may be unexpected.

Here is an examples of a cast:

```
let x: i64 = -13;
let y: u8 = 10;
let z = x + (y as i64);
```

In this example, we had to cast `y` so that we could add it to `x`. If we didn't, we'd get a compiler error. Try modifying this to remove the cast and see what error you get. Also try casting `x` to a `u8` and see what you get instead.

Exercises:

1. From memory, try to recall the different primitive types in Rust, and write them down. How many did you get right? How many did you miss?
2. Write a program that creates three integers of different types and multiplies them together. Print the result.

---

[1] Or rather, Rust forces you to acknowledge how complicated strings really are. They're among the more confusing bits coming in, I find.

Control Flow

Programs aren't very useful if they just follow a linear path forward. To do useful things, we need conditional statements and loops! Rust can do this, of course.

This section ostensibly is about control flow. It's also about **Rust as an expression-oriented language**; this is one of the things that makes Rust really ergonomic, and is also one of the things most confusing if you haven't used a language like it before! Basically, almost everything in Rust returns a value! This is super helpful, and different/confusing if you're not used to it.

Also, a note: we'll use println quite a bit through this section in a few forms. The fmt docs have thorough explanations of all the different ways you can format output with println. Please refer there if you need any help understanding the `println!` usage on this page.

Blocks

Blocks aren't really control flow *per se*, but since every control flow mechanism takes a block, here we are.

A block is just a section of code surrounded by curly braces:

```
{
    println!("I'm in a block!");
}
```

There are two main notable things about blocks:

- **Blocks delimit a scope**, so any variables you declare inside a block are out of scope outside of it; this is super handy for temporary variables.
- **Blocks return a value**, which is precisely the value of the last expression in the block. If the last expression of the block ends with a semicolon,

then it's a *statement*, which returns `()`, the unit value, so it functionally has no return value.

Let's see a couple of examples of this and how you'd use it.

```
let msg = "Hello, world";
{
    // we're in the pirate block now
    let msg = "Ahoy, matey!";
    println!("{msg}");
}
println!("{msg}");
```

> **Exercise**: What do you think this will print when you run it? Try to figure it out before executing it to test your understanding and intuition!

Since blocks declare a new scope, the `msg` variable inside the block **shadows** the `msg` variable on the outside, and does *not* change its value.

Note: Examples here are going to be **rather contrived** because we're avoiding things like structs and methods for now, trying to stick to (largely) just the syntax introduced so far.

Okay, so we saw scope delimiting. Here's an example of a block returning a value:

```
let parrots = 5;
let shipmates = 10;

let legs_on_ship = {
    let parrot_legs = 2 * parrots;
    let human_legs = 2 * shipmates;
    parrot_legs + human_legs
};
```

Here we end up with `legs_on_ship` having the value `30`, and the temporary variables (`parrot_legs` and `human_legs`) are freed when the block ends.

What's the real-world use for this usage of blocks? A common use is to do what we saw in the previous example and use it to constrain the scope of temporary variables. Doing this inside a block allows you to set things up in a readable manner without polluting the outer scope. Another common use is to release resources; you can lock a mutex at the beginning of a block, and when the block ends it will be released (like `with lock:` in Python).

Ifs

Blocks are neat, but if-else is how we really get stuff done.

It works how you'd expect from other languages, with a few notable things:

- Parentheses around the condition are *optional* (and usually considered un-idiomatic)
- if-else-expressions return a value! Just like with blocks, this is the value of the last expression in **each branch**[1].
- The branches must be surrounded by curly braces (yes, even if it's just a single statement[2])

Here's a simple one:

```
if plunder > 5 {
    println!("A good haul");
} else {
    println!("Just jetsam");
}
```

And then using it to get back a value. Since Rust doesn't have the conditional/ternary operator, this is the way you set a value conditionally:

```
let is_crew_member = true;
let greeting = if is_crew_member {
    "Ahoy!!! Welcome aboard!"
} else {
    "Yarrrrr get off me ship"
};
println!("{}", greeting);
```

This block will print a different greeting depending on whether or not you're a crew member.

Loops

The final basic control flow constructs for getting stuff done are loops. There are a few kinds of loops, so we'll just whirlwind through them.

There are three kinds of loops:

- loop-expressions
- while-expressions
- for-expressions

After those, we'll go through an extra: returning values from a loop.

loop expressions

These are the basic infinite loops. You have to exit out of them manually with the `break` keyword. Otherwise, they keep going forever.

```
// Don't do this, it will run forever
loop {
    println!("Wheeeee");
}
```

To have the loop terminate, you have to break. Usually you want that on a condition:

```rust
let mut count = 0;

loop {
    count += 1;

    println!("iteration {count}");

    if count >= 10 {
        break;
    }
}
```

while expressions

These work like loop-expressions with the added bonus of having a condition to halt, so you don't have to manually break out of them. Otherwise, they're the same: you give a body and it gets run each time until the condition evaluates to `false`.

```rust
let mut count = 0;

while count < 10 {
    count += 1;
    println!("iteration {count}");
}
```

This should behave the same as the loop above. Note that since we run *while the condition is true*, we don't have to do an awkward inversion of it to decide *when to break*. It's much more convenient.

Of course, you can also use `break` statements in these. There just usually isn't as much need to.

for expressions

The for loop in Rust is one of the ways you iterate over an iterable, which is typically a collection of things or a range ("from 0 to 10"). We'll see an example of both. There will be some syntax you're not familiar with, but we'll come back to the collections later on.

```
for count in 0..10 {
    println!("iteration {}", count+1);
}
```

This behaves the same as above. `0..10` gives us the range from 0 to 10, excluding the 10. That means we have to do the awkward `+1` to get count the same. We can specify the range as an inclusive range instead to avoid that, using `1..=10`, which ranges from 1 to 10, including the 10.

```
for count in 1..=10 {
    println!("iteration {count}");
}
```

Collections of things are similar. Let's see a basic example with an array. We'll see examples from other collections later on, when we talk about the standard library.

```
for prime in [2, 3, 5, 7, 11] {
    println!("{prime} is prime.");
}
```

And like with the other types of loops, you can use `break` in for loops! You usually won't need it, but it comes in handy occasionally.

Loop values

We mentioned before that in Rust, most expressions return values. But if you try to do that with a loop, you're going to see a compiler error:

```
let x = for count in 0..3 {
    count * 2
};
```

```
error[E0308]: mismatched types
 --> src/main.rs:4:5
  |
4 |     count * 2
  |     ^^^^^^^^^ expected `()`, found integer
  |
help: you might have meant to break the loop with this value
  |
4 |     break count * 2;
  |     +++++          +

For more information about this error, try `rustc --explain E0308`.
error: could not compile `playground` due to previous error
```

One of the really nice things with Rust is that the compiler messages are often helpful. (Sometimes, they try to be helpful... Trust yourself over the compiler, it cannot know your intent!) In this case, it tells us precisely what we need to do: use the `break` keyword.

There are two things to unpack here:

- The loop body must result in `()`, so the last statement needs to end in a semicolon.
- If you want the loop to result in a value, you have to emit that value using the `break` keyword.

Here's the previous example, but modified to work (and made a little more interesting):

```rust
let x = for count in 0..3 {
    if count > 1 {
        break count * 2;
    }
};
```

Oops, another compiler error!

```
Compiling playground v0.0.1 (/playground)
error[E0571]: `break` with value from a `for` loop
 --> src/main.rs:5:9
  |
3 | let x = for count in 0..3 {
  |         ----------------- you can't `break` with a value in a `for`
loop
4 |     if count > 1 {
5 |         break count * 2;
  |         ^^^^^^^^^^^^^^^ can only break with a value inside `loop`
or breakable block
  |
help: use `break` on its own without a value inside this `for` loop
  |
5 |         break;
  |         ~~~~~

For more information about this error, try `rustc --explain E0571`.
error: could not compile `playground` due to previous error
```

See, the problem is that while- and for-expressions are not going to be guaranteed to have a value, because they may hit their termination before they hit the break statement.

We can do it with `loop` though. This one will compile, I promise:

```rust
let mut count = 0;

let x = loop {
    if count > 1 {
        break count * 2;
    }

    count += 1;
};
```

And there you have it: control flow! The first of many things Rust provides to do useful things.

Exercises:

1. Write fizz buzz using a while loop.
2. Now write fizz buzz using a for loop.

We'll see you in the next section for functions.

---

[1] If you use it this way, the branches must have matching types. You can't, say, return a string from one branch and an integer from the other.

[2] And thank goodness, too. This helps prevent some infamous bugs with major security implications, like goto fail.

Just as with control flow, doing useful things is a whole lot easier if you have functions available. They're there in every major high-level language, and they're there in Rust as well.

Functions

Functions in Rust are defined with the `fn` keyword (often pronounced "fun", or if you're feeling edgy, "effin'") followed by the function name, parameter list, return type, and function body. That's a few words, so let's just see an example and break it down.

Let's pretend that for some totally-not-contrived reason, we need a function which adds 10 to whatever argument we pass in. We give it 5, we want to get back 15. And let's also pretend we only care about 32-bit unsigned integers. That function would look something like this:

```rust
fn add_10(x: u32) -> u32 {
    let value = x + 10;
    value
}
```

`fn` says we're starting a function, and `add_10` gives it a name.

Then we have the parameter list. We only have one thing: `x: u32`, which says we'll accept a parameter named `x` with the type `u32`.

The `-> u32` is our return type; if you omit it, it's the same as writing `-> ()`, or returning the unit type (kind of like the return type `void` in TypeScript or C/C++).

Then we have the function body. This is a block like we talked about in control flow. Inside we have two lines. The first creates a variable named `value`, and the second returns it implicitly.

Examples

Now we have everything we need to do some more more interesting things. We'll walk through one example here, then you can work through the other one on your own.

Let's take the classic Fibonacci function. First we create our main function as every program needs, we also create a stub. We'll just return a dummy value for now.

```
fn main() {
    let x = fibonacci(10);
    println!("{x}");
}

fn fibonacci(n: u32) -> u32 {
    0
}
```

Now we need to fill in that function body. It's a straightforward translation from the algorithm:

```
fn main() {
    let x = fibonacci(10);
    println!("{x}");
}

fn fibonacci(n: u32) -> u32 {
    if n < 2 {
        return n;
    }

    fibonacci(n-1) + fibonacci(n-2)
}
```

And now it should print 55!

Now, one to try on your own. I'll give you the main:

```rust
fn main() {
    fizzbuzz_up_to(100);
}
```

Now you should define a function named `fizzbuzz_up_to` which takes a parameter as a number and does Fizzbuzz. You'll need to use loops and define at least one function to achieve this!

Exercise: Write fizz buzz using a function, taking a parameter for the max.

Good luck. Come back here when you've completed that.

Memory Management

One of the defining characteristics of Rust is that it gives you full control over memory: it doesn't have a runtime garbage collector, and it doesn't do reference counting unless you explicitly ask it to. When things get allocated and deallocated is entirely in your control (but constrained by some safety rules; you can't deallocate it then use it again).

This means to really get Rust, we have to also really get how memory management works. There's only a little bit here that's specific to Rust; it's more of a primer on memory management for those coming from higher level languages who may not have seen this before, or in a while.

Stack and Heap

The first thing to know is that there are two places memory gets allocated: on the stack and on the heap. If you're familiar with the data structures, you're halfway there. The stack is a stack like you're familiar with: things get pushed on and popped off (but also referenced from the middle of the stack). On the other hand, the heap is just a big open field of memory (not a heap data structure) where we can allocate things.

The stack is controlled by the program's execution, comprising call frames (pushed on for function calls) and local variables of those functions. If something is on the stack, it's going to only live as long as the function it's inside. That's why you can't return references to things that are local variables, because they'll get deallocated when the function ends and its call frame is popped off the stack.

The heap is different. It's this big wide open space where variables can live as long as they'd like. Well, until someone comes along and deallocates them. When programs allocate memory on the heap, then it has to be deliberately deallocated later. The heap gives us basically unbounded memory (up to what's made available by your operating system) that can live for a long time. The penalty is that that memory has to be managed (we have to deallocate it sometime, not automatically when call frames get popped) and it is often less efficient than memory on the stack[1].

Managing Memory

Different programming languages have different ways of handling heap-allocated memory.

The base level where the language gives you no help is **manual memory management**. This is what you get in C and C++, where you have to explicitly allocate and deallocate memory. In C, these functions are `malloc` to allocate memory, `free` to release it, and `realloc` to resize some allocation.

With manual memory management, you have *all* the power but you also have *all* the problems. This technique is where we get vulnerabilities like buffer overflows leading to remote code execution, or issues with use-after-free which can wreak all sorts of havoc. With great power comes great responsibility.

The highest amount of help is in languages with **automatic memory management**. This is what you see in most programming languages. There are a few forms, the two most common are **(tracing) garbage collection** (as you see in Java and Go, for example) and **reference counting** (as you see in Python and Swift, for example). These techniques largely handle memory management for you so you don't have to think about it, and you avoid some major problems with manual memory management. Notably, you *cannot* have use-after-free issues, and it's usually harder to have buffer overflows lead to arbitrary memory access, since the runtime checks for that.

Rust is a special language. It doesn't employ automatic memory management, but it also avoids the pain and problems that manual memory management brings. **You get all the power of manual memory management, and usually don't have to worry about it at all.** It does this by keeping track of "ownership" of memory (we'll talk about this in the next section) and the lifetimes of memory, and deallocating memory automatically when it won't be used anymore.

Rust also places some restrictions on what you can do to make this feasible.

References and Pointers

When we talk about memory management, we're talking about how much memory we have, where it is, and (to an extent) what it represents. This leads us to a type we haven't talked about yet: **references**. (And their cousins, **pointers**.)

References are found in most languages, but they're not usually called out as such. It's a concept hidden below the surface[2].

A reference is just a way of **referring to a variable (or memory) at another location**. So if you have a variable `x`, then a reference to `x` would be written `&x`, and is a way of letting you access `x`. To access the underlying value, you **dereference it**, which you write as `*x`.

This is a little abstract, so let's see it in action.

```
// First create a variable, x
let x: u32 = 10;

// Then create a reference to it:
let ref_x: &u32 = &x;

// Then print out the reference!
println!("x = {}", *ref_x);
```

Rust will also automatically dereference references for you, so usually you can omit the `*` dereference operator:

```
// Print out the value without explicitly dereferencing it
println!("x = {}", ref_x);
```

Pointers have the same underlying representation as references: both hold an address to another place in memory. But while a reference refers to a variable at another location, **a pointer is just an address in memory**. This means that unlike with references, Rust doesn't make guarantees about what a pointer can do, so you have to use **unsafe Rust** to use it and access memory at that location.

You generally don't need to use pointers in Rust unless you're doing something very specific, and using them entails using unsafe Rust. It's good to know *of* them, but we won't go into the details on *how* to use them at all.

Allocating on the Stack or on the Heap

To allocate on the stack, you don't do anything special. Any local variable is allocated on the stack, since it's part of the call frame for the function you're in.

To allocate on the heap, you have to use a type that causes heap allocation. We'll talk about that in the next section!

> **Exercises**:
>
> 1. In your own words, what is the difference between allocating on the heap and on the stack?
> 2. In your own words, what is the difference between a reference and a pointer?
>
> These exercises are particularly challenging due to how abstract the concepts are when you haven't put them in practice. Be kind to yourself!

---

[1] **Why** this is is a bit out of scope, but it's due to memory locality and how things get retrieved. The variables in the stack will likely already be in the CPU cache when the related code is executing, but you have to do a slow fetch from main memory for heap memory, since the CPU can't predict well what you're going to need.

[2] This is one of the things I greatly appreciate about Rust: it doesn't hide these fundamental concepts, so while there's more work to understand things up front before

you use the language, you can rest confident that you better understand what's happening, and there's less unexpected behavior.

Heap Allocation

Sometimes you need to allocate something on the heap. Maybe you don't know its size at compile time, or you need it to live independently of the scope it's in. When you want to allocate something on the heap, there are a few types you can use to do so[1].

Boxed values

The most basic, fundamental heap allocation is using `Box`.

A boxed value is simply one that lives on the heap instead of the stack. A `Box` is a boxed value, and it's generic: a `Box<u32>` is a u32 that's heap-allocated, and a `Box<f64>` is a 64-bit float that's heap-allocated.

To construct one, you use the `Box::new` constructor. Often the compiler can infer what type the Box has, but if not, you have to provide type annotations. Here are two ways to do that:

```
let x: Box<u32> = Box::new(42);
let y = Box::<f64>::new(4.2);
```

To move a value back onto the stack, out of the box, you can dereference it:

```
let x: Box<u32> = Box::new(42);
let y = *x; // y is now a u32 on the stack
```

Vecs

Sometimes you just need to build a list of things. You do this with `Vec`, which is the type for dynamically sized arrays in Rust. (Rust's arrays are of fixed size.)

```rust
let mut parrots: Vec<&str> = Vec::new();
parrots.push("Shivers");
parrots.push("Tweety");
parrots.push("Dinner");
println!("parrots: {:?}", parrots);
```

You can also do it inline, using the `vec!` macro:

```rust
let parrots = vec!["Shivers", "Tweety", "Dinner"];
println!("parrots: {:?}", parrots);
```

You can also iterate over the elements of a Vec:

```rust
let parrots = vec!["Shivers", "Tweety", "Dinner"];
for parrot in parrots.iter() {
    println!("{} says hi.", parrot);
}
```

Other collection types

The standard library has a variety of other collection types, which are helpful for just about anything you need. The std::collections docs have a lot of information about the types available and when you would want to use each.

As a sampling:

- `Vec`
- `HashMap` and `BTreeMap`
- `HashSet` and `BTreeSet`

We won't go into the details on this page.

Exercise: Write a small program which constructs a HashMap with pirate ship names as keys and their crew sizes as values. For example, "Black Pearl" could have 3 crew members. Iterate over the elements of the hashmap and print each ship with its crew size.

[1] You can also use raw pointers and unsafe Rust. We're not going to talk about that here, because you should **almost always avoid it**.

Ownership and Lifetimes

And now we come to one of the things that differentiates Rust from other systems languages. From other languages in general, really. And that is **ownership and the borrow checker**.

As we went over earlier, Rust does not have a garbage collector. Instead, the compiler is tracking when memory should be allocated and deallocated, and ensuring that your references remain valid.

How it does that is by keeping track of the **lifetime** of variables as well as their **ownership**. To unpack those, in short:

- A value has one **owner** at a time, and **ownership** is used for tracking when memory is valid and when it is dropped.
- The **lifetime** of a variable is the time during which references to it are valid.

These two concepts are highly related. The lifetime of a reference is linked to the lifetime of its owner.

To make it concrete, let's look at an example of a function which explains why we need lifetimes. This code example will *not* compile:

```
{   // create an outer scope

    // define a reference to a u32, but do not initialize it
    let outer_ref: &u32;

    { // create an inner scope

        // declare and initialize a u32
        let inner_val: u32 = 10;

        // try to assign a reference to the inner val to our ref
        outer_ref = &inner_val; // <-- this line will fail to compile

    } // <-- inner_val goes out of scope here

    // <-- but the compiler needs outer_ref to be valid here!
    println!("outer_ref value: {}", outer_ref);
}
```

Here's another example, where we try to return a reference to a local variable inside a function.

```
fn naughty_function() -> &u32 {
    let x: u32 = 10;
    &x
}
```

Rust rightfully presents us with an error there if we try to compile it. Whoops!

This might be your first interaction with the borrow checker. It's a vital piece of Rust machinery which helps prevent major issues. Notably, you could get both of the previous examples to compile in languages like C or C++, leading to use-after-free errors. With Rust, you can't do that[1].

Lifetimes

Every reference is a borrow, and each borrow has a lifetime. That lifetime spans from when the variable is created to when it is destroyed.

What the borrow checker does is it ensures that **each reference's lifetime is wholly contained by the borrowed value's lifetime**.

Lifetimes can be explicitly given names. These are typically `'a`, `'b`, etc. but you can also use longer descriptive names.

```rust
fn example<'a>(x: &'a u32) {
    let y: &'a u32 = &x;
}
```

This example also introduces **generics**, which we will only use for lifetimes until we cover them in more depth. But basically, the `<'a>` is for generics, and here it's giving a generic lifetime. This says that we have some lifetime, `'a`, and our parameter `x` is a reference of that lifetime. It doesn't say specifically *how long* that lifetime is, because we don't know anything about that lifetime until it's filled in as a parameter of the generic function.

The lifetime `'static` means "referred to data will live for the duration of the program". This is often used for string constants:

```rust
let msg: &'static str = "hello, world!";
```

Anywhere where you write a type annotation for a reference, you can also include an explicit lifetime. We've seen one example of this above. You'll see it often for structs, enums, and other data structures when they contain references. You will **not** see it often for functions, because of **lifetime elision**.

Lifetime elision is when we're allowed to omit the explicit lifetimes and just let the compiler take a guess (based on a few rules). The above example would be better written using implicit lifetimes:

```rust
fn example(x: &u32) {
    let y: &u32 = &x;
}
```

We won't unpack the lifetime elision rules here, but in general you can omit explicit lifetime names in most places. If you can't, the compiler will tell you, and you can try adding them!

Ownership

Ownership is related to lifetimes, and we can see a few examples of it. Each variable has an owner, and this ownership can be *moved* to another place. This happens if you pass something by value: the new place receives the value and promises that it will deallocate it when it needs to. But since the new place owns it, the old place is not allowed to use it anymore!

Here's one example that shows a value moving to a new owner, one that new Rust programmers often encounter:

```
let xs = vec![1,2,3];

for x in xs {
    println!("{x}");
}

println!("total len: {}", xs.len());
```

This looks totally reasonable, but there's a problem: When we iterated over `xs`, since we used the *value* of `xs`, we moved it, and we don't own it anymore! This example *does not compile*.

Instead, we need to use a *reference* to let the for loop **borrow** the Vec, and then we can use it in both places:

```
let xs = vec![1,2,3];

for x in &xs {
    println!("{x}");
}

println!("total len: {}", xs.len());
```

Now it works, woohoo!

That's the basics of ownership. Just remember that when you pass a value around, it *moves* the value to the other place.

Well, that's not true. It moves it **if it can't copy it**.

There are some variables, the simplest ones, which "are Copy", which means they implement the Copy trait. If something is Copy, then it will get copied instead of moved, and you can keep using it.

As an example, we can use a primitive `u32` twice:

```rust
let x: u32 = 10;
println!("x is {x}");
println!("x*2 is {}", x*2);
```

Are you a little confused? If so, you're not alone! This isn't very clear, and the same syntax doing two different things implicitly can be tricky. But rest assured that this is understandable, and *the compiler is here to help*. If we go back to the first Ownership example and try to compile it, here's part of the error message:

```
3 | let xs = vec![1,2,3];
  |     -- move occurs because `xs` has type `Vec<i32>`, which does not
implement the `Copy` trait
4 |
5 | for x in xs {
  |          -- `xs` moved due to this implicit call to `.into_iter()`
...
```

In here, it clearly tells us that the Vec was moved, why it was moved, and later on in the error message (which I cut off) it gives a suggestion for how to fix it (using a reference). We'll talk more about ownership in the next section on closures.

---

1 The borrow checker places a lot of restrictions on you. As a result, it's common to hear people refer to "fighting with the borrow checker." I like to think of it as getting a code review from an eager and extremely pedantic partner. It can be quite frustrating the first few times you run into it, but with time it becomes an invaluable part of your workflow. It's catching *legitimate* issues, and you *should* be scared working in languages without it!

Closures

Previously, we looked at named functions and we looked at ownership. In Rust, we also have **closures**, which give us the ability to make anonymous functions.

You can make a closure with explicit or inferred types:

```rust
let y: u32 = 10;
let annotated = |x: u32| -> u32 { x + y };
let inferred = |x| x + y;

println!("annotated: {}", annotated(32));
println!("inferred: {}", inferred(32));
```

The basic syntax for a closure is to use pipes around the parameter list followed by an expression for the return value. Sometimes you'll see a no-argument closure, which looks like it's using the or-operator (`||`) but that's the empty parameter list here, and could be written with spaces for clarity (`| |`).

Closures can reference values from their outer scope, which is really handy. They can also **capture** the outer values and use them. This is handy for things like counters:

```rust
let mut count = 0;
let mut increment = || {
    count += 1;
    count
};

println!("count is {}", increment());
println!("count is {}", increment());
println!("count is {}", increment());
```

Note that the closure must be `mut` if it captures a mutable variable. This is because what it captures is part of the closure, so if it's mutating it then it's mutating itself.

You can also return closures from functions! To do this, if you capture variables, you'll need to *move* the variables into the closure. You do this with the `move` keyword, which signals to the compiler that it should *take ownership* of its arguments, so that the closure cannot outlive its arguments.

Here are two examples of that in action. First, a closure which prints a message. This one has to explicitly annotate its lifetime, including on the return type.

```
fn print_msg<'a>(msg: &'a str) -> impl Fn() + 'a {
    let printer = move || {
        println!("{msg}");
    };
    printer
}
```

The way you would use it is by calling it to get a function, then calling that function.

```
// this line creates a new function, f
let f = print_msg("hello, world");
// nothing has been printed yet

// and this line invokes the function, which will print our message
f();
```

And one which makes a counter.

```
fn make_counter() -> impl FnMut() -> u32 {
    let mut count = 0;
    let increment = move || {
        count += 1;
        count
    };
    increment
}
```

Invoking it is similar, but this time, it has to be mutable.

```
let mut counter = make_counter();

println!("count is {}", counter()); // prints 1
println!("count is {}", counter()); // prints 2
println!("count is {}", counter()); // prints 3
```

You'll notice the return types for these functions are different. What they return is an `impl` of a trait. We'll get to what traits are in a later section; for now, you can think of them like interfaces, so we know what we can do with the thing, but not its specific type.

There are three traits for functions: `Fn`, `FnMut`, and `FnOnce`, which provide various restrictions on how the caller of the function can use it. The other thing you'll notice is the `impl` keyword, which is new. This says we'll return something which implements this trait (like an interface in other languages), but we don't specify exactly what it is. This is how you return closures, generally, because each closure is its own type.

There are good docs on these traits, as usual. In short, the restrictions are:

- `Fn` can be called be called multiple times, and it doesn't modify its underlying state.
- `FnMut` can be called multiple times, but it may mutate itself when you do (so it needs a mutable reference to itself)
- `FnOnce` can be called once. It consumes itself in that call, and you can't use it a second time.

If you have an `Fn`, you can use it as `FnMut` or `FnOnce`. And you can use `FnMut` as `FnOnce`. But you can't go back up the chain!

Exercises:

1. Write a closure which takes in two numbers and adds them together.
2. Write a function which takes in an initial value and an increment and returns a closure which increments the value by that amount each time it's called.

Thanks for following along so far! You've gotten through what I think are the hardest parts of Rust. The rest should be easier, and you should be able to put this into practice. Take a breather, then move on to the next section.

Structs

Structuring your data is a key part of programming, and every language worth its salt gives you a way to do that. In object-oriented languages, this is usually by creating a class. In Rust, you structure your data with **structs**.

A struct is a named grouping of fields (data belonging to the struct), which also can have methods on it. Let's unpack that.

When you're creating a struct, first you have to give it a name:

```rust
struct PirateShip {
}
```

Then you can also put fields on it, of any type:

```rust
struct PirateShip {
    captain: String,
    crew: Vec<String>,
    treasure: f64,
}
```

And you can have methods on the struct by using an `impl` block. Those methods can take a reference to self (`&self`) if they are just reading fields, or they can use a mutable reference (`&mut self`) if they will be *changing* any data.

```rust
impl PirateShip {
    pub fn count_treasure(&self) -> f64 {
        // some computations probably
        self.treasure
    }

    pub fn mutiny(&mut self) {
        if self.crew.len() > 0 {
            // replace the captain with one of the crew
            self.captain = self.crew.pop().unwrap();
        } else {
            println!("there's no crew to perform mutiny");
        }
    }
}
```

To create an instance of a struct, you give the name of the struct along with a value for each of the fields, specified by name (like `treasure: 64.0`). There is also some shorthand to use: if you have a variable in scope with the same name as one of the fields, you can specify that just by name. That's confusing without an example, so let's see it in action.

```rust
let blackbeard = "Blackbeard".to_owned();
let crew = vec!["Scurvy".to_owned(), "Rat".to_owned(),
"Polly".to_owned()];
let ship = PirateShip {
    captain: blackbeard,
    crew,
    treasure: 64.0,
};
```

In this example, we can see both forms of specifying fields. The captain and treasure are specified with the `<name>: <value>` form, while the crew is specified with the shorthand that means `crew: crew`.

Note that in this example, we used the method `to_owned` a few times. This takes a reference to a string (`&str`) and creates an owned string (`String`), so that we don't have to worry about lifetimes. The precise details of this aren't particularly relevant in this chapter, but it's a nice thing to keep in mind: if you want to avoid including lifetimes, you can use owned instances by cloning (or a method like `to_owned`). There's more complexity with strings in Rust than in other languages due to references and lifetimes, but further treatment of them is beyond the scope of this course.

Exercises:

1. Define a struct for a crew member with a name, age, and any other attributes you would like.
2. Implement a few methods on this struct, such as one to say who it is.

Enums

Many languages let you create enumeration types. Typically, these are shorthand for constants, so you can have a few different values that you know it's coming from. A common example would be days of the week.

In Typescript, an enum for days of the week would look something like this[1]:

```
enum WeekDay {
    Monday = 0,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

These are pretty straightforward in most languages, and useful but not earth-shattering.

Rust enums are **far more powerful** and are a key tool in structuring data and programs well. You use them almost everywhere (indirectly, through `Option` and `Result`) for error handling and type safety.

What makes Rust enums more powerful is that they capture more than just a constant. Each variant of the enum can also *have data*.

If you're familiar with C, a Rust enum is equivalent to a tagged union. If you're *not* familiar with C, which is probably more likely if you're reading this, we'll unpack that and explain how enums are represented in memory. But first we'll see some examples of *using* them.

Defining and Using Enums

An enum has a name and it has variants. Each variant can either be a "unit" enum or it can have data associated with it. Additionally, there are two ways you can add data for a variant: positionally, or with named fields.

Let's define an enum with one of each of these types of variants to see them in action. This will just be a silly made-up example.

```rust
enum LanguageResource {
    // This variant has a field by position; in this case, it's
probably the URL
    Website(String),

    // This variant has fields by name
    Book { title: String, pages: u64 },

    // This is a unit variant, with no data contained
    SelfTeaching,
}
```

To write create an instance of an enum, you instantiate one of the variants. You refer to it using the enum name and the variant name, separated with `::`.

Let's see examples of creating each of the previous variants, and then accessing their fields:

```rust
let site = LanguageResource::Website("https://yet-another-rust-
resource.pages.dev/".to_owned());
let book = LanguageResource::Book { title: "The Rust Programming
Language".to_owned(), pages: 300 };
let independent = LanguageResource::SelfTeaching;

println!("the site is at {}", site.0);
println!("the book {} is {} pages long", book.title, book.pages);
```

When you have an instance of an enum, you can use it with pattern matching (to directly look at the fields and use them), or you can use methods on the enum. One of the most common enums you use in Rust is `Option`, which is defined something like this:

```
// the `<T>` just says we take a generic type, so the Option can
contain anything
// we want instead of having to define a new type by hand for each
contained type.
enum MyOption<T> {
    Some(T),
    None,
}
```

Then to use it you can either use methods (in this case, defined by the standard library) or pattern matching. Here is an example using both ways to accomplish the same thing:

```
let x = Some(3);

if x.is_some() {
    println!("x = {}", x.unwrap());
} else {
    println!("x is empty :(");
}

match x {
    Some(v) => println!("x = {v}"),
    None => println!("x is empty :("),
}
```

And just like with structs, you can define your *own* methods on an enum you create.

Let's wrap this up by implementing the `is_some` function for our own Option type.

```
impl<T> MyOption<T> {
    pub fn is_some(&self) -> bool {
        match self {
            MyOption::Some(_) => true,
            MyOption::None => false,
        }
    }
}
```

Enum Memory Representation

As mentioned above, enums are akin to tagged unions in C. The memory layout of these is pretty straightforward. Each enum has a "tag", which represents which variant it is holding, and then it has enough bytes to hold *the largest variant*.

The upshot of this is that no matter what variant your instance is, it will be as large as the largest variant, plus a bit extra for the tag.

How large is the tag? It can be as small as one byte, if you have fewer than 255 variants (if you have more, you need a larger tag) and if the alignment of the type is 1. We won't go into how to determine what the alignment of your type is but it's a great topic to explore on your own.

Why not just use a struct?

What's the reason we'd like to have an enum with data, instead of just a struct? What's special about doing `Result` as an enum?

There are a few main reasons you want to do it as an enum, and why you cannot always use a struct:

1. It allows a more compact memory representation. A struct requires the memory of *all* its fields, and if you have fields that are present in some arms of the enum but not others, you'd pay for those always. In contrast, an enum only requires as much memory as its largest variant, plus a byte (or word) for its tag. So this can save a serious amount of memory! And for things like `Option` of a reference, it can even be *free*, because the compiler can do tricks to reuse memory in the pointer itself.
2. It allows you to do pattern matching, which you cannot do on a struct with private members.
3. You can enforce things at compile time, like which fields are or are not set, rather than enforcing populating fields through logic at run time.

Exercises:

1. Write an enum which represents days of the week.
2. Look up the standard library `Result` type and write a basic implementation of it. Refer to our `Option` implementation for a similar example.

---

[1] Weeks start on Mondays, not Sundays. Fight me.

More Control Flow

There are a couple of control flow constructs we skipped over earlier, because they rely on some concepts we didn't have. Namely, you can combine pattern matching with `if` and `while`. This comes in really handy when you want to do something *only* for a particular variant.

Here's a contrived example of retrieving something from a map, and doing something different in each case.

```
// let's just pretend we got this from a map, okay?
let value = Some(42);

if let Some(inner) = value {
    println!("inner was {inner}");
} else {
    println!("this is the failure case!");
}
```

The general setup is that you put a variant on the left hand side inside an `if let`, and if the match can be satisfied (if it works), then it'll fill in the placeholder variables. If that match doesn't succeed, then you fall through to the else case.

The same thing works in loops. One common use is with iterators, but you can use it with any enum. (I see this used a lot less than `if let`.)

```
let values = vec![1, 2, 3, 4, 5];
let mut iter = values.iter();

while let Some(v) = iter.next() {
    println!("v = {v}");
}
```

Modules

Modules allow you to structure code in a maintainable way, so you don't end up with everything in one giant file. They also let you hide details so that consumers of your code don't rely on internal implementation details.

Creating modules

You declare modules with the `mod` keyword, and then you can include them from one of three places:

- Directly with a block delimited by curly braces
- In a file at the same level with the name of the module
- Inside a folder with the module name, in the file `mod.rs`

Modules can also be either public (`pub mod`) or private (`mod`). If they're public, anyone consuming your crate can use the module and its pub members. Otherwise, private members are accessible only to themselves and their descendants.

Let's say you're creating a module called `math` . Then inside your current module (starting from `main.rs` or `lib.rs` , unless you're nested inside another module already) you would create this module. Let's assume you're working in `main.rs` for the sake of these examples, but it's the same either way.

```rust
// in main.rs
pub mod math {
    // look, I didn't say this was a *useful* module
    pub fn add(x: u32, y: u32) -> u32 {
        x + y
    }
}
```

If you want to break it out into other files, you can do that. You still declare the module in `main.rs` or your other file:

```rust
pub mod math;
```

And then you put the contents in either `math.rs` or `math/mod.rs`, depending on your preference. Usually it's clearer to use the `math.rs` style over `math/mod.rs`, because otherwise your editor can be filled with a lot of `mod.rs` files and it's not clear which one you're working on!

```rust
pub fn add(x: u32, y: u32) -> u32 {
    x + y
}
```

And that's all there is to creating modules!

Using modules

To use what's declared in another module, you use the `use` keyword. This brings things into scope. (If you want to re-export them, you can `pub use` them.)

```rust
use std::collections::HashMap;
```

If you refer to `super::thing`, that gets `thing` from the parent module.

If you refer to `crate::thing`, that gets `thing` from the root of the crate you're in.

Testing

Everyone's favorite topic! Writing tests!

It's actually pretty exciting with Rust, since you have good tools to make testing pretty easy. It's all built-in out of the box.

Unit tests

The typical way to write tests with Rust is to create a child module called `test`, import from the parent, and test things. These are written in a way that they're only compiled if a flag is enabled for tests, so they're *not* included in your release builds. Handy!

Let's see an example.

We'll go back to the same example we had of a super useful addition function. Then we can write a test to make sure it adds things correctly.

```rust
pub fn plus(x: i32, y: i32) -> i32 {
    x + y
}

// this cfg says "only compile this if the test compile option is on"
#[cfg(test)]
// by convention we call this "tests" but it doesn't need to be
mod tests {
    // usually you avoid importing all, but it's idiomatic for tests
    use super::*;

    // this is how we mark this function as a test
    #[test]
    fn adds_small_numbers() {
        let x = 10;
        let y = 20;
        let expected = 30;
        assert_eq!(plus(x, y), expected, "should add up correctly");
    }

    // this test WILL FAIL because the integers roll over
    #[test]
    fn adds_big_numbers() {
        let x = 2_000_000_000;
        let y = 2_000_000_000;
        assert!(plus(x, y) > 0, "result should be positive");
    }
}
```

Then to run the tests, you run `cargo test` and it should find and execute all your tests (or your compile errors).

Integration tests

Just as with unit tests, you can write integration tests. The main difference is that these tests can only consume the public API that you provide.

Any file you put in the `tests/` directory will be treated as an integration test.

Let's say you have your `plus` function in a crate, then you could open `tests/my_test.rs` and write:

```rust
use my_library::plus;

#[test]
fn test_addition() {
    assert_eq!(plus(10, 20), 30);
}
```

Doc tests

Rust also gives you facilities to write tests directly in your documentation. You can put a docstring on a function, module, etc. with `///` (as opposed to the usual `//` to start a comment). If you put code blocks (with the markdown style ```` ``` ````), it will be compiled and run when you run `cargo test`. (This also has the neat side effect of making your documentation code examples automatically break the build if they are out of date!)

Here's an example for, again, our `plus` function.

````rust
/// Adds together two numbers, and doesn't handle rollover.
///
/// ```
/// use playground::plus;
/// assert_eq!(30, plus(10, 20));
/// ```
pub fn plus(x: i32, y: i32) -> i32 { x + y }
````

Linting and Formatting

There are two main canonical tool for formatting and linting Rust code. The first is `cargo-fmt`, the formatter. The second is `clippy`, the linter.

To run the formatter, just run `cargo fmt`. That's it.

To run the linter, run `cargo clippy` after installing it one time. Clippy is *very* helpful at suggesting idiomatic code, and it improves over time (read: it will sometimes come back and find things in code that you thought was fine last month). It's invaluable as both a Rust learner and as an experienced Rust programmer, and it's usually good practice to treat Clippy warnings as errors (with specific exceptions annotated in the code if merited).

There are various options to configure these tools. I suggest using them vanilla for a while, if not forever. It keeps things consistent across the Rust ecosystem, and the suggestions are usually there for a reason. But if you do want to configure things, you can dive in!

Dependency Management

While Rust has a relatively early ecosystem of libraries, it is a *strong* ecosystem for a variety of domains. The main place you find a crate is on, fittingly, crates.io.

After you search for a crate on there, and you want to add it, you can run a simple command. For example, to add the popular serialization/deserialization crate serde, you would run

```
cargo add serde
```

This would add the latest version. If you want to add it manually, you can specify the version in `Cargo.toml` with a line like `serde = "1.0.154"`.

The version you specify is matched with semver. To avoid just repeating the docs, I refer you to the official docs if you want to do something more advanced or want to understand how it's handling version matching.

Traits

Traits are much like interfaces in other languages. They give a way of defining shared behavior and a way of *using* said shared behavior.

At its base level, a trait is a collection of methods. The type that the methods belong to is unknown, because that's part of the implementation: the trait gets implemented for a given type.

The methods can either be abstract (must be implemented in order to implement the trait) or they can be implemented, using only the type information you have. Namely, these would use other methods on that trait.

Defining a trait

Since a trait is a collection of methods, we give those methods as the definition of the trait. For example, we can define a trait for a key-value store.

We know that any key-value store will be able to set a value and get it again. Those are the base primitives, but what if we want another operation, like get-and-set? As long as it's expressible in terms of just other methods on the KeyValueStore, we can do an implementation of that on the trait itself!

```
trait KeyValueStore {
    // These ones have to be implemented by structs which impl
KeyValueStore
    fn set(&mut self, key: &str, value: Vec<u8>);
    fn get(&self, key: &str) -> Option<Vec<u8>>;
    fn lock(&mut self, key: &str);
    fn unlock(&mut self, key: &str);

    // This one is defined for all KeyValueStores
    fn get_and_set(&mut self, key: &str, value: Vec<u8>) ->
Option<Vec<u8>> {
        self.lock(key);

        let old_value = self.get(key);
        self.set(key, value);

        self.unlock(key);
        old_value
    }
}
```

One thing to note is that the methods on a trait are all public! You don't have to put a visibility modifier, `pub`, on them because they're by default visible.

Implementing a trait on a type

Now let's look at how to implement a trait *on* a type. After you've defined a trait, and a struct, you `impl` the trait.

For example, let's say we make a trait called `Printable` with a `print` method. We will also create a struct to implement the trait.

```rust
trait Printable {
    fn print(&self);
}

struct Ship {
    name: String
}

impl Printable for Ship {
    fn print(&self) {
        println!("<Ship name=\"{}\">", self.name);
    }
}
```

There is one other way you can impl a trait, and it's pretty incredible. You can tell the compiler to derive the implementation. This is only doable for traits that implement some auto derive functionality.

A lot of the built-in traits, like `Debug` and `PartialEq`, can be derived. Here's how you derive those for a simple struct containing a string and a number:

```rust
#[derive(Debug, PartialEq)]
struct PirateShip {
    name: String,
    masts: f32, // float since you can have part of a mast
}
```

Note: You can impl *any* trait on a type that you define. And you can impl a trait that you define on *any* type. But you cannot impl a trait that you didn't define on a type that you didn't define. This prevents having multiple impls of the same trait on the same type if different crates both impl it.

Using traits

There are a few ways you can use a trait. You can use methods from the trait on a struct that impls the trait. Or you can accept the trait as a parameter to a function, or return it as the return type. (You can also use them as part of generics; we'll cover that with generics.)

Calling methods from traits

To call a method that's on a trait, you have to `use` that trait to make it visible. This is mostly so that you don't have collisions from traits with the same methods on the same type.

As an example, there's a trait in the standard library called `Read`. It lets you read bytes from a source. Lots of types impl Read, and one that does is `&[u8]`, but you can't use it unless you `use` it.

This example won't compile:

```
let mut s: Vec<u8> = "sad example".into();
let mut buf: [u8; 32] = [0; 32];
(&s[..]).read(&mut buf);
```

But by bringing `std::io::Read` into scope, it does compile!

```
use std::io::Read;

let mut s: Vec<u8> = "sad example".into();
let mut buf: [u8; 32] = [0; 32];
(&s[..]).read(&mut buf);
```

If you run into a situation where a method seems like it should be on a type but your tooling or the compiler are saying it isn't, look at if you're missing a `use` somewhere.

Traits as parameters

If we know that a function only needs something that implements a trait, we can pass it in with `impl`. Let's say we're writing a function which needs a key-value store, but we don't care which *one*. Then we can write this function to accept any key-value store:

```
fn save_record(kv: &impl KeyValueStore) {
    // use the key value store somehow
}
```

Code that passes in a type that doesn't impl KeyValueStore will not compile, and you can be sure that this will work. At compile time, the type is resolved to be the concrete type.

Traits as return types

You can also use the impl keyword for return types to specify that you're returning a value that impls the trait. For example, you could define something that returns a KeyValueStore:

```
fn create_in_memory_kvstore(config: Config) -> impl KeyValueStore {
    // create and return the KeyValueStore
    todo!()
}
```

The thing to note is that the function can only return *one type*. If you have multiple implementations of the trait, you cannot have one branch which returns type A and one which returns type B. At compile time, the compiler needs to be able to swap out `impl KeyValueStore` for the one specific type which you're going to return.

Exercises:

1. Go back to our pirate ship struct. Based on this, define a trait for any sort of `Vessel`, which has one method: `mutiny`.
2. Implement `Vessel` for `PirateShip`.
3. Create a new kind of vessel (`NavalShip`?) and implement `Vessel` for it.

Generics

Generics are an essential feature of Rust, allowing you to write reusable code that works with multiple types without sacrificing performance. They let you write code that works with a variety of types without duplicating code.

What are Generics?

Generics are a way to write code that accepts one or more type parameters, which can then be used within the code as actual types. This allows the code to work with different types, while still being type-safe and efficient.

In Rust, generics are similar to templates in C++ or generics in Java, TypeScript, or other languages.

Using Generics

To illustrate how generics work in Rust, let's start with a simple example. Suppose you have a function that takes two arguments and returns the larger of the two. Without generics, you'd need to write a separate function for each type you want to support, e.g., one for integers and one for floating-point numbers.

However, using generics, you can write a single function that works with any type that implements the `PartialOrd` trait. Here's an example:

```rust
fn max<T: PartialOrd>(x: T, y: T) -> T {
    if x > y {
        x
    } else {
        y
    }
}

fn main() {
    let a = 5;
    let b = 10;
    let c = 3.14;
    let d = 6.28;

    println!("Larger of {} and {}: {}", a, b, max(a, b));
    println!("Larger of {} and {}: {}", c, d, max(c, d));
}
```

In the max function definition, we introduce a generic type parameter T using angle brackets (`<>`). We also specify the trait bound `PartialOrd for T` using the colon syntax (`:`). This constraint ensures that the max function only works with types that implement the `PartialOrd` trait, which is necessary for comparing values using the `>` operator.

Tip: It can be hard to know what trait bound you need, especially when new to Rust. One of the things I like to do is leave it out entirely, then let the compiler tell me which trait bound it thinks is missing. This works a surprising amount of the time, especially for simple cases.

Now, the max function works with both integers and floating-point numbers. As an added bonus, you can call it with any two values of the same type that implement the PartialOrd trait. So it will even work for strings, or types that you don't even know about! Pretty neat and pretty powerful.

Generic Structs

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer_point = Point { x: 5, y: 10 };
    let float_point = Point { x: 3.14, y: 6.28 };

    println!("Integer point: ({}, {})", integer_point.x,
integer_point.y);
    println!("Floating point: ({}, {})", float_point.x, float_point
}
```

In the `Point` struct definition, we introduce a generic type parameter `T` using angle brackets (`<>`). This allows us to use the same `Point` struct with different types for the `x` and `y` coordinates.

Note that in this example, both coordinates must have the same type. If you want to allow different types for `x` and `y`, you can introduce multiple generic type parameters:

```
struct Point2<X, Y> {
    x: X,
    y: Y,
}

fn main() {
    let mixed_point = Point2 { x: 5, y: 6.28 };

    println!("Mixed point: ({}, {})", mixed_point.x, mixed_point.y);
}
```

Here we have left the types unbounded, but you would likely want some trait bounds for these generic parameters. PartialOrd, PartialEq, and Debug are common choices.

Generic Enums and Traits

You can use generics with enums and traits in a similar way as with structs and functions. Here's an example of a generic `Result` enum that can be used to represent the success or failure of a computation (in fact, this is how the standard library type is defined):

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn divide(x: f64, y: f64) -> Result<f64, String> {
    if y == 0.0 {
        Result::Err("Cannot divide by zero.".to_string())
    } else {
        Result::Ok(x / y)
    }
}

fn main() {
    let result = divide(5.0, 2.0);
    match result {
        Result::Ok(value) => println!("Result: {}", value),
        Result::Err(error) => println!("Error: {}", error),
    }

    let result = divide(5.0, 0.0);
    match result {
        Result::Ok(value) => println!("Result: {}", value),
        Result::Err(error) => println!("Error: {}", error),
    }
}
```

In this example, we define a generic `Result` enum with two type parameters: `T` for the success value and `E` for the error value. The `Result` enum has two variants: `Ok(T)` for success and `Err(E)` for failure.

We then define a divide function that returns a `Result<f64, String>`. The function takes two `f64` arguments and either returns the result of the division or an error message if the divisor is zero.

In the main function, we call the divide function and pattern match on the returned `Result` to handle both the success and error cases.

This `Result` enum is a simplified version of the Result type that is part of the Rust standard library, which is used extensively for error handling.

> **Exercise**: Write a generic `divide` function. (Hint: look up the `std::ops::Div` trait.)

Error Handling

Error handling is an essential aspect of any programming language, and Rust is no exception. Rust provides robust error handling mechanisms, like the `Result` type and the `?` operator, which allow you to deal with errors in a clean and idiomatic way. And you can also bail out for unrecoverable errors with panics.

The Result Type

Rust has a built-in `Result` enum for handling errors in a type-safe manner. The `Result` type is defined as follows:

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

The `Result` enum has two variants:

- `Ok`, which represents a successful operation and contains the result value
- `Err`, which represents a failed operation and contains the error value

The `Result` type encourages you to handle errors explicitly and provides a clear separation between the success and error cases. This is one of the things that makes Rust so powerful for writing robust code: You can't forget to handle errors, the type system will require that you address them.

You can use pattern matching to handle the different variants of the `Result` type:

```rust
fn main() {
    let result = some_function();

    match result {
        Ok(value) => println!("Success: {}", value),
        Err(error) => println!("Error: {}", error),
    }
}

fn some_function() -> Result<String, String> {
    // ...
}
```

You can also use `unwrap` or `expect` if you know the returned value is an Ok. But you have to be very sure: if you unwrap an Err, your program will panic! This is okay in instances like missing configuration where you don't want the program to run at all if it isn't there, but you have to be deliberate. As a general rule, don't use `unwrap` or `expect` unless you *want* the program to crash if it fails.

The ? Operator

While pattern matching on `Result` values is powerful, it can become verbose when dealing with multiple operations that may return errors. To simplify error handling in these situations, Rust provides the `?` operator. It's only usable when you're expecting to return a Result value, and in those cases is tremendously helpful.

The `?` operator, when placed after an expression that returns a `Result`, automatically handles the error case. If the expression returns an `Err`, the function immediately returns the error value. If the expression returns an `Ok`, the `?` operator unwraps the `Ok` value and continues executing the function.

Here's an example of using the `?` operator:

```
fn main() {
    match read_config() {
        Ok(config) => println!("Config: {:?}", config),
        Err(error) => println!("Error: {}", error),
    }
}

fn read_config() -> Result<String, String> {
    let file = read_file("config.txt")?;
    let parsed = parse_config(&file)?;
    Ok(parsed)
}

fn read_file(path: &str) -> Result<String, String> {
    // ...
}

fn parse_config(file_contents: &str) -> Result<String, String> {
    // ...
}
```

In this example, the `read_config` function calls two other functions that return `Result` values. Instead of using pattern matching to handle the errors, the `?` operator is used to simplify the code. It elevates the "happy path" so that you can read through that directly, and you know that errors will trickle up.

Panics

Rust has another mechanism for error handling called panics. A panic is a runtime error that results in the immediate termination of the program. Panics are reserved for exceptional situations where it's impossible or undesirable to continue executing the program, such as an unrecoverable error or a broken invariant.

You can cause a panic explicitly using the `panic!` macro:

```rust
fn main() {
    let index = 10;
    if index > 5 {
        panic!("Index is out of bounds!");
    }

    // ...
}
```

When a panic occurs, Rust unwinds the stack, running destructors for all objects in scope and then terminating the program. Alternatively, Rust can be configured to abort the program directly, without unwinding the stack.

Panics should be used sparingly and only in exceptional circumstances. In most cases, you should prefer using the `Result` type for error handling, as it promotes explicit and robust error handling.

Async/await

Asynchronous programming is an essential technique for writing efficient and responsive code, especially when dealing with I/O-bound tasks or tasks that may take a long time to complete. Rust provides an asynchronous programming model through the `async` and `await` keywords[1], which allow you to write non-blocking code cleanly. The async and await pattern in Rust is similar to that in other languages like Python and TypeScript.

In Python, you use the `async def` syntax to define an asynchronous function, and the `await` keyword to call an asynchronous function:

```python
import asyncio

async def fetch_data():
    # ...

async def main():
    data = await fetch_data()
    print(data)

asyncio.run(main())
```

In TypeScript, you use the `async` keyword before the function definition, and the `await` keyword to call an asynchronous function:

```typescript
async function fetchData(): Promise<string> {
    // ...
}

async function main() {
    const data = await fetchData();
    console.log(data);
}

main();
```

Rust's async and await model works similarly to Python and TypeScript.

Async and await

In Rust, the `async` keyword is used to define asynchronous functions. An asynchronous function is a function that can be paused and resumed later, allowing other tasks to run concurrently[2]. Asynchronous functions in Rust always return a `Future`. A `Future` is a trait that represents a value that may not be available yet but will be at some point in the future.

Here's an example of an asynchronous function in Rust:

```rust
async fn fetch_data() -> Result<String, String> {
    // ...
}
```

If you invoke this function, you get back a Future. But the return type is Result, not Future. That's because Rust hides that detail from us a little bit. Instead of making us put `Future` everywhere, we have a little friendlier syntax.

Just remember that when you invoke the function, it doesn't execute *anything* until you `await` it.

The `await` keyword is used to pause the execution of an asynchronous function and wait for a `Future` to resolve. When a `Future` is awaited, the current task is suspended, allowing other tasks (such as the one you awaited) to run concurrently. Once the awaited `Future` resolves, the execution of the suspended task resumes.

Here's an example of how to use the `await` keyword to call an asynchronous function:

```rust
async fn main() {
    match fetch_data().await {
        Ok(data) => println!("Data: {}", data),
        Err(error) => println!("Error: {}", error),
    }
}
```

In this example, the `fetch_data` asynchronous function is called with the `.await` syntax. The `main` function is also defined as asynchronous using the `async` keyword.

Note that you can only use the `await` keyword inside an asynchronous function. If you try to use `await` in a non-async function, you'll get a compile-time error.

Async runtimes

To use async code, you'll need some sort of async runtime! The runtime is responsible for scheduling tasks onto threads. The typical one people go for these days is Tokio, but there are other options. And if you're ever feeling ambitious, you can write your own!

---

[1] `await` *looks* like a method or field when you use it, but it's a keyword and is used in syntax as such. It's a keyword the same way that `match` is, but often feels like you're calling a method, so it can be kind of confusing.

[2] Even with a single-threaded async runtime, tasks are *concurrent*, although they may not run *in parallel*.

Other Resources

This course has just scratched the surface with Rust. You should be able to pair, with some difficulty, with someone who is more experienced. But this is not the end of the road. By the nature of this course and the size of Rust, we've skipped a lot and skimmed over what we did cover.

Now you must go forth and keep learning and writing Rust code.

Here are a few paths forward from here.

- Books!
  - The Rust Programming Language
  - Programming Rust, 2nd Edition
  - Rust in Action
- Rust by Example
- Rustlings, a set of small exercises

And nothing replaces getting your hands on the keyboard and writing code.

Credits

These resources were part of the inspiration for this course:

- Rust by Example
- Comprehensive Rust

This content was authored by Nicole, aka ntietz.

Acknowledgements

Huge thanks to these people who have provided invaluable feedback and more throughout this process:

- Cole Brossart
- Jake Weiner
- Dan Reich
- Miccah Castorina