The Rust Programming Language

Darth-Revan

Originally created and published by Mozilla Foundation

https://github.com/Darth-Revan/rust-lang_Doc-LaTeX

April 6, 2016

Contents

1 Introduction

Welcome! This book will teach you about the Rust Programming Language. Rust is a systems programming language focused on three goals: safety, speed, and concurrency. It maintains these goals without having a garbage collector, making it a useful language for a number of use cases other languages aren't good at: embedding in other languages, programs with specific space and time requirements, and writing low-level code, like device drivers and operating systems. It improves on current languages targeting this space by having a number of compile-time safety checks that produce no runtime overhead, while eliminating all data races. Rust also aims to achieve 'zero-cost abstractions' even though some of these abstractions feel like those of a high-level language. Even then, Rust still allows precise control like a low-level language would.

"The Rust Programming Language" is split into chapters. This introduction is the first. After this:

- Getting Started - Set up your computer for Rust development.
- Tutorial: Guessing Game - Learn some Rust with a small project.
- Syntax and Semantics - Each bit of Rust, broken down into small chunks.
- Effective Rust - Higher-level concepts for writing excellent Rust code.
- Nightly Rust - Cutting-edge features that aren't in stable builds yet.
- Glossary - A reference of terms used in the book.
- Bibliography - Background on Rust's influences, papers about Rust.

Contributing

The source files from which this book is generated can be found on GitHub.

2 Getting Started

This first chapter of the book will get us going with Rust and its tooling. First, we'll install Rust. Then, the classic 'Hello World' program. Finally, we'll talk about Cargo, Rust's build system and package manager.

2.1 Installing Rust

The first step to using Rust is to install it. Generally speaking, you'll need an Internet connection to run the commands in this section, as we'll be downloading Rust from the internet.

We'll be showing off a number of commands using a terminal, and those lines all start with `$`. We don't need to type in the `$`s, they are there to indicate the start of each command. We'll see many tutorials and examples around the web that follow this convention: `$` for commands run as our regular user, and `#` for commands we should be running as an administrator.

Platform support

The Rust compiler runs on, and compiles to, a great number of platforms, though not all platforms are equally supported. Rust's support levels are organized into three tiers, each with a different set of guarantees.

Platforms are identified by their "target triple" which is the string to inform the compiler what kind of output should be produced. The columns below indicate whether the corresponding component works on the specified platform.

Tier 1

Tier 1 platforms can be thought of as "guaranteed to build and work". Specifically they will each satisfy the following requirements:

- Automated testing is set up to run tests for the platform.
- Landing changes to the `rust-lang/rust` repository's master branch is gated on tests passing.
- Official release artifacts are provided for the platform.
- Documentation for how to use and how to build the platform is available.

Target	std	rustc	cargo	notes
`x86_64-pc-windows-msvc`	✓	✓	✓	64-bit MSVC (Windows 7+)
`i686-pc-windows-gnu`	✓	✓	✓	32-bit MinGW (Windows 7+)
`x86_64-pc-windows-gnu`	✓	✓	✓	64-bit MinGW (Windows 7+)
`i686-apple-darwin`	✓	✓	✓	32-bit OSX (10.7+, Lion+)
`x86_64-apple-darwin`	✓	✓	✓	64-bit OSX (10.7+, Lion+)
`i686-unkown-linux-gnu`	✓	✓	✓	32-bit Linux (2.6.18+)
`x86_64-unkown-linux-gnu`	✓	✓	✓	64-bit Linux (2.6.18+)

Tier 2

Tier 2 platforms can be thought of as "guaranteed to build". Automated tests are not run so it's not guaranteed to produce a working build, but platforms often work to quite a good degree and patches are always welcome! Specifically, these platforms are required to have each of the following:

- Automated building is set up, but may not be running tests.
- Landing changes to the `rust-lang/rust` repository's master branch is gated on platforms **building**. Note that this means for some platforms only the standard library is compiled, but for others the full bootstrap is run.
- Official release artifacts are provided for the platform.

Target	std	rustc	cargo	notes
`i686-pc-windows-msvc`	✓	✓	✓	32-bit MSVC (Windows 7+)
`x86_64-unkown-linzux-musl`	✓			64-bit Linux with MUSL
`arm-linux-androideabi`	✓			ARM Android
`arm-unkown-linux-gnueabi`	✓	✓		ARM Linux (2.6.18+)
`arm-unkown-linux-gnueabihf`	✓	✓		ARM Linux (2.6.18+)
`aarch64-unkown-linux-gnu`	✓			ARM64 Linux (2.6.18+)
`mips-unkown-linux-gnu`	✓			MIPS Linux (2.6.18+)
`mipsel-unkown-linux-gnu`	✓			MIPS (LE) Linux (2.6.18+)

Tier 3

Tier 3 platforms are those which Rust has support for, but landing changes is not gated on the platform either building or passing tests. Working builds for these platforms may be spotty as their reliability is often defined in terms of community contributions. Additionally, release artifacts and installers are not provided, but there may be community infrastructure producing these in unofficial locations.

Target	std	rustc	cargo	notes
`i686-linux-android`	✓			32-bit x86 Android
`aarch64-linux-android`	✓			ARM64 Android
`powerpc-unkown-linux-gnu`	✓			PowerPC Linux (2.6.18+)
`i386-apple-ios`	✓			32-bit x86 iOS
`x86_64-apple-ios`	✓			64-bit x86 iOS
`armv7-apple-ios`	✓			ARM iOS
`armv7s-apple-ios`	✓			ARM iOS
`aarch64-apple-ios`	✓			ARM64 iOS
`i686-unkown-freebsd`	✓	✓		32-bit FreeBSD
`x86_64-unkown-freebsd`	✓	✓		64-bit FreeBSD
`x86_64-unkown-openbsd`	✓	✓		64-bit OpenBSD
`x86_64-unkown-netbsd`	✓	✓		64-bit NetBSD
`x86_64-unkown-bitrig`	✓	✓		64-bit Bitrig
`x86_64-unkown-dragonfly`	✓	✓		64-bit DragonFlyBSD
`x86_64-rumprun-netbsd`	✓			64-bit NetBDS Rump Kernel
`i686-pc-windows-msvc (XP)`	✓			Windows XP support
`x86_64-pc-windows-msvc (XP)`	✓			Windows XP support

Note that this table can be expanded over time, this isn't the exhaustive set of tier 3 platforms that will ever be!

Installing on Linux or Mac

If we're on Linux or a Mac, all we need to do is open a terminal and type this:

```
$ curl -sSf https://static.rust-lang.org/rustup.sh | sh
```

This will download a script, and stat the installation. If it all goes well, you'll see this appear:

```
Welcome to Rust.

This script will download the Rust compiler and its package manager,
Cargo, and install them to /usr/local. You may install elsewhere by
running this script with the --prefix=<path> option.

The installer will run under 'sudo' and may ask you for your password.
If you do not want the script to run 'sudo' then pass it the
--disable-sudo flag.
```

```
You may uninstall later by running /usr/local/lib/rustlib/uninstall.sh,
or by running this script again with the --uninstall flag.

Continue? (y/N)
```

From here, press `y` for 'yes', and then follow the rest of the prompts.

Installing on Windows

If you're on Windows, please download the appropriate installer.

Uninstalling

Uninstalling Rust is as easy as installing it. On Linux or Mac, run the uninstall script:

```
$ sudo /usr/local/lib/rustlib/uninstall.sh
```

If we used the Windows installer, we can re-run the `.msi` and it will give us an uninstall option.

Troubleshooting

If we've got Rust installed, we can open up a shell, and type this:

```
$ rustc --version
```

You should see the version number, commit hash, and commit date.

If you do, Rust has been installed successfully! Congrats!

If you don't and you're on Windows, check that Rust is in your %PATH% system variable. If it isn't, run the installer again, select "Change" on the "Change, repair, or remove installation" page and ensure "Add to PATH" is installed on the local hard drive.

If not, there are a number of places where we can get help. The easiest is the #rust IRC channel on irc.mozilla.org, which we can access through Mibbit. Click that link, and we'll be chatting with other Rustaceans (a silly nickname we call ourselves) who can help us out. Other great resources include the user's forum, and Stack Overflow.

This installer also installs a copy of the documentation locally, so we can read it offline. On UNIX systems, `/usr/local/share/doc/rust` is the location. On Windows, it's in a `share/doc directory`, inside the directory to which Rust was installed.

2.2 Hello, World!

Now that you have Rust installed, we'll help you write your first Rust program. It's traditional when learning a new language to write a little program to print the text "Hello, world!" to the screen, and in this section, we'll follow that tradition.

The nice thing about starting with such a simple program is that you can quickly verify that your compiler is installed, and that it's working properly. Printing information to the screen is also a pretty common thing to do, so practicing it early on is good.

> Note: This book assumes basic familiarity with the command line. Rust itself makes no specific demands about your editing, tooling, or where your code lives, so if you prefer an IDE to the command line, that's an option. You may want to check out SolidOak, which was built specifically with Rust in mind. There are a number of extensions in development by the community, and the Rust team ships plugins for various editors. Configuring your editor or IDE is out of the scope of this tutorial, so check the documentation for your specific setup.

Creating a Project File

First, make a file to put your Rust code in. Rust doesn't care where your code lives, but for this book, I suggest making a *projects* directory in your home directory, and keeping all your projects there. Open a terminal and enter the following commands to make a directory for this particular project:

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

> Note: If you're on Windows and not using PowerShell, the may not work. Consult the documentation for your shell for more details.

Writing and Running a Rust Program

Next, make a new source file and call it *main.rs*. Rust files always end in a *.rs* extension. If you're using more than one word in your filename, use an underscore to separate them; for example, you'd use *hello_world.rs* rather than *helloworld.rs*.

Now open the main.rs file you just created, and type the following code:

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

Save the file, and go back to your terminal window. On Linux or OSX, enter the following commands:

```
$ rustc main.rs
$ ./main
Hello, world!
```

In Windows, replace `main` with `main.exe`. Regardless of your operating system, you should see the string `Hello, world!` print to the terminal. If you did, then congratulations! You've officially written a Rust program. That makes you a Rust programmer! Welcome.

Anatomy of a Rust Program

Now, let's go over what just happened in your "Hello, world!" program in detail. Here's the first piece of the puzzle:

```
fn main() {

}
```

These lines define a *function* in Rust. The `main` function is special: it's the beginning of every Rust program. The first line says, "I'm declaring a function named `main` that takes no arguments and returns nothing." If there were arguments, they would go inside the parentheses (`(` and `)`), and because we aren't returning anything from this function, we can omit the return type entirely.

Also note that the function body is wrapped in curly braces (`{` and `}`). Rust requires these around all function bodies. It's considered good style to put the opening curly brace on the same line as the function declaration, with one space in between.

Inside the `main()` function:

```
println!("Hello, world!");
```

This line does all of the work in this little program: it prints text to the screen. There are a number of details that are important here. The first is that it's indented with four spaces, not tabs.

The second important part is the `println!()` line. This is calling a Rust *macro*, which is how metaprogramming is done in Rust. If it were calling a function instead, it would look like this: `println()` (without the !). We'll discuss Rust macros in more detail later, but for now you only need to know that when you see a `!` that means that you're calling a macro instead of a normal function.

Next is `"Hello, world!"` which is a *string*. Strings are a surprisingly complicated topic in a systems programming language, and this is a statically allocated string. We pass this string as an argument to `println!`, which prints the string to the screen. Easy enough!

The line ends with a semicolon (`;`). Rust is an Expression-Oriented Language, which means that most things are expressions, rather than statements. The `;` indicates that this expression is over, and the next one is ready to begin. Most lines of Rust code end with a `;`.

Compiling and Running are Separate Steps

In "Writing and Running a Rust Program", we showed you how to run a newly created program. We'll break that process down and examine each step now.

Before running a Rust program, you have to compile it. You can use the Rust compiler by entering the `rustc` command and passing it the name of your source file, like this:

```
$ rustc main.rs
```

If you come from a C or C++ background, you'll notice that this is similar to `gcc` or `clang`. After compiling successfully, Rust should output a binary executable, which you can see on Linux or OSX by entering the `ls` command in your shell as follows:

```
$ ls
main main.rs
```

On Windows, you'd enter:

```
$ dir
main.exe main.rs
```

This shows we have two files: the source code, with an `.rs` extension, and the executable (`main.exe` on Windows, `main` everywhere else). All that's left to do from here is run the `main` or `main.exe` file, like this:

```
$./main  # or main.exe on Windows
```

If *main.rs* were your "Hello, world!" program, this would print `Hello, world!` to your terminal.

If you come from a dynamic language like Ruby, Python, or JavaScript, you may not be used to compiling and running a program being separate steps. Rust is an *ahead-of-time compiled* language, which means that you can compile a program, give it to someone else, and they can run it even without Rust installed. If you give someone a `.rb` or `.py` or `.js` file, on the other hand, they need to have a Ruby, Python, or JavaScript implementation installed (respectively), but you only need one command to both compile and run your program. Everything is a tradeoff in language design.

Just compiling with `rustc` is fine for simple programs, but as your project grows, you'll want to be able to manage all of the options your project has, and make it easy to share your code with other people and projects. Next, I'll introduce you to a tool called Cargo, which will help you write real-world Rust programs.

2.3 Hello, Cargo!

Cargo is Rust's build system and package manager, and Rustaceans use Cargo to manage their Rust projects. Cargo manages three things: building your code, downloading the libraries your code depends on, and building those libraries. We call libraries your code needs 'dependencies' since your code depends on them.

The simplest Rust programs don't have any dependencies, so right now, you'd only use the first part of its functionality. As you write more complex Rust programs, you'll want to add dependencies, and if you start off using Cargo, that will be a lot easier to do.

As the vast, vast majority of Rust projects use Cargo, we will assume that you're using it for the rest of the book. Cargo comes installed with Rust itself, if you used the official installers. If you installed Rust through some other means, you can check if you have Cargo installed by typing:

```
$ cargo --version
```

into a terminal. If you see a version number, great! If you see an error like '`command not found`', then you should look at the documentation for the system in which you installed Rust, to determine if Cargo is separate.

Converting to Cargo

Let's convert the Hello World program to Cargo. To Cargo-fy a project, you need to do three things:

1. Put your source file in the right directory.
2. Get rid of the old executable (`main.exe` on Windows, `main` everywhere else) and make a new one.
3. Make a Cargo configuration file.

Let's get started!

Creating a new Executable and Source Directory

First, go back to your terminal, move to your *hello_world* directory, and enter the following commands:

```
$ mkdir src
$ mv main.rs src/main.rs
$ rm main  # or 'del main.exe' on Windows
```

Cargo expects your source files to live inside a *src* directory, so do that first. This leaves the top-level project directory (in this case, *hello_world*) for READMEs, license information, and anything else not related to your code. In this way, using Cargo helps you keep your projects nice and tidy. There's a place for everything, and everything is in its place.

Now, copy *main.rs* to the *src* directory, and delete the compiled file you created with `rustc`. As usual, replace `main` with `main.exe` if you're on Windows.

This example retains `main.rs` as the source filename because it's creating an executable. If you wanted to make a library instead, you'd name the file `lib.rs`. This convention is used by Cargo to successfully compile your projects, but it can be overridden if you wish.

Creating a Configuration File

Next, create a new file inside your *hello_world* directory, and call it `Cargo.toml`.

Make sure to capitalize the `C` in *Cargo.toml*, or Cargo won't know what to do with the configuration file.

This file is in the TOML (Tom's Obvious, Minimal Language) format. TOML is similar to INI, but has some extra goodies, and is used as Cargo's configuration format.

Inside this file, type the following information:

```
[package]

name = "hello_world"
version = "0.0.1"
authors = [ "Your name <you@example.com>" ]
```

The first line, `[package]`, indicates that the following statements are configuring a package. As we add more information to this file, we'll add other sections, but for now, we only have the package configuration.

The other three lines set the three bits of configuration that Cargo needs to know to compile your program: its name, what version it is, and who wrote it.

Once you've added this information to the *Cargo.toml* file, save it to finish creating the configuration file.

Building and Running a Cargo Project

With your *Cargo.toml* file in place in your project's root directory, you should be ready to build and run your Hello World program! To do so, enter the following commands:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
$ ./target/debug/hello_world
Hello, world!
```

Bam! If all goes well, `Hello, world!` should print to the terminal once more.

You just built a project with `cargo build` and ran it with `./target/debug/hello_world`, but you can actually do both in one step with `cargo` run as follows:

```
$ cargo run
     Running `target/debug/hello_world`
Hello, world!
```

Notice that this example didn't re-build the project. Cargo figured out that the file hasn't changed, and so it just ran the binary. If you'd modified your source code, Cargo would have rebuilt the project before running it, and you would have seen something like this:

```
$ cargo run
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
     Running `target/debug/hello_world`
Hello, world!
```

Cargo checks to see if any of your project's files have been modified, and only rebuilds your project if they've changed since the last time you built it.

With simple projects, Cargo doesn't bring a whole lot over just using `rustc`, but it will become useful in future. This is especially true when you start using crates; these are synonymous with a 'library' or 'package' in other programming languages. For complex projects composed of multiple crates, it's much easier to let Cargo coordinate the build. Using Cargo, you can run `cargo build`, and it should work the right way.

Building for Release

When your project is finally ready for release, you can use `cargo build --release` to compile your project with optimizations. These optimizations make your Rust code run faster, but turning them on makes your program take longer to compile. This is why there are two different profiles, one for development, and one for building the final program you'll give to a user.

Running this command also causes Cargo to create a new file called *Cargo.lock*, which looks like this:

```
[root]
name = "hello_world"
version = "0.0.1"
```

Cargo uses the *Cargo.lock* file to keep track of dependencies in your application. This is the Hello World project's *Cargo.lock* file. This project doesn't have dependencies, so the file is a bit sparse. Realistically, you won't ever need to touch this file yourself; just let Cargo handle it.

That's it! If you've been following along, you should have successfully built `hello_world` with Cargo.

Even though the project is simple, it now uses much of the real tooling you'll use for the rest of your Rust career. In fact, you can expect to start virtually all Rust projects with some variation on the following commands:

```
$ git clone someurl.com/foo
$ cd foo
$ cargo build
```

Making a new Cargo Project the Easy Way

You don't have to go through that previous process every time you want to start a new project! Cargo can quickly make a bare-bones project directory that you can start developing in right away.

To start a new project with Cargo, enter `cargo new` at the command line:

```
$ cargo new hello_world --bin
```

This command passes `--bin` because the goal is to get straight to making an executable application, as opposed to a library. Executables are often called binaries (as in `/usr/bin`, if you're on a Unix system).

Cargo has generated two files and one directory for us: a `Cargo.toml` and a *src* directory with a *main.rs* file inside. These should look familliar, they're exactly what we created by hand, above.

This output is all you need to get started. First, open `Cargo.toml`. It should look something like this:

```
[package]

name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
```

Cargo has populated *Cargo.toml* with reasonable defaults based on the arguments you gave it and your `git` global configuration. You may notice that Cargo has also initialized the `hello_world` directory as a `git` repository.

Here's what should be in `src/main.rs`:

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

Cargo has generated a “Hello World!” for you, and you’re ready to start coding!

> Note: If you want to look at Cargo in more detail, check out the official Cargo guide, which covers all of its features.

2.4 Closing Thoughts

This chapter covered the basics that will serve you well through the rest of this book, and the rest of your time with Rust. Now that you’ve got the tools down, we’ll cover more about the Rust language itself.

You have two options: Dive into a project with ’Learn Rust’, or start from the bottom and work your way up with ’Syntax and Semantics’. More experienced systems programmers will probably prefer ’Learn Rust’, while those from dynamic backgrounds may enjoy either. Different people learn differently! Choose whatever’s right for you.

3 Tutorial: Guessing Game

Let's learn some Rust! For our first project, we'll implement a classic beginner programming problem: the guessing game. Here's how it works: Our program will generate a random integer between one and a hundred. It will then prompt us to enter a guess. Upon entering our guess, it will tell us if we're too low or too high. Once we guess correctly, it will congratulate us. Sounds good?

Along the way, we'll learn a little bit about Rust. The next chapter, 'Syntax and Semantics', will dive deeper into each part.

3.1 Set up

Let's set up a new project. Go to your projects directory. Remember how we had to create our directory structure and a `Cargo.toml` for `hello_world`? Cargo has a command that does that for us. Let's give it a shot:

```
$ cd ~/projects
$ cargo new guessing_game --bin
$ cd guessing_game
```

We pass the name of our project to `cargo new`, and then the `--bin` flag, since we're making a binary, rather than a library.

Check out the generated `Cargo.toml`:

```
[package]

name = "guessing_game"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
```

Cargo gets this information from your environment. If it's not correct, go ahead and fix that.

Finally, Cargo generated a 'Hello, world!' for us. Check out `src/main.rs`:

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

Let's try compiling what Cargo gave us:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

Excellent! Open up your `src/main.rs` again. We'll be writing all of our code in this file.

Before we move on, let me show you one more Cargo command: `run`. `cargo run` is kind of like `cargo build`, but it also then runs the produced executable. Try it out:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Hello, world!
```

Great! The `run` command comes in handy when you need to rapidly iterate on a project. Our game is such a project, we need to quickly test each iteration before moving on to the next one.

3.2 Processing a Guess

Let's get to it! The first thing we need to do for our guessing game is allow our player to input a guess. Put this in your `src/main.rs`:

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");

    println!("You guessed: {}", guess);
}
```

There's a lot here! Let's go over it, bit by bit.

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

We'll need to take user input, and then print the result as output. As such, we need the `io` library from the standard library. Rust only imports a few things by default into every program, the 'prelude'. If it's not in the prelude, you'll have to `use` it directly. There is also a second 'prelude', the io prelude, which serves a similar function: you import it, and it imports a number of useful, `io`-related things.

```
fn main() {
```

As you've seen before, the `main()` function is the entry point into your program. The `fn` syntax declares a new function, the `()`s indicate that there are no arguments, and `{` starts the body of the function. Because we didn't include a return type, it's assumed to be `()`, an empty tuple.

```
    println!("Guess the number!");

    println!("Please input your guess.");
```

We previously learned that `println!()` is a macro that prints a string to the screen.

```
    let mut guess = String::new();
```

Now we're getting interesting! There's a lot going on in this little line. The first thing to notice is that this is a let statement, which is used to create 'variable bindings'. They take this form:

```
let foo = bar;
```

This will create a new binding named `foo`, and bind it to the value `bar`. In many languages, this is called a 'variable', but Rust's variable bindings have a few tricks up their sleeves.

For example, they're immutable by default. That's why our example uses `mut`: it makes a binding mutable, rather than immutable. `let` doesn't take a name on the left hand side of the assignment, it actually accepts a 'pattern'. We'll use patterns later. It's easy enough to use for now:

```
let foo = 5; // immutable.
let mut bar = 5; // mutable
```

Oh, and `//` will start a comment, until the end of the line. Rust ignores everything in comments.

So now we know that `let mut guess` will introduce a mutable binding named `guess`, but we have to look at the other side of the `=` for what it's bound to: `String::new()`.

`String` is a string type, provided by the standard library. A String is a growable, UTF-8 encoded bit of text.

The `::new()` syntax uses `::` because this is an 'associated function' of a particular type. That is to say, it's associated with `String` itself, rather than a particular instance of a `String`. Some languages call this a 'static method'.

This function is named `new()`, because it creates a new, empty `String`. You'll find a `new()` function on many types, as it's a common name for making a new value of some kind.

Let's move forward:

```
    io::stdin().read_line(&mut guess)
        .expect("Failed to read line");
```

That's a lot more! Let's go bit-by-bit. The first line has two parts. Here's the first:

```
io::stdin()
```

Remember how we `used std::io` on the first line of the program? We're now calling an associated function on it. If we didn't `use std::io`, we could have written this line as `std::io::stdin()`.

This particular function returns a handle to the standard input for your terminal. More specifically, a std::io::Stdin.

The next part will use this handle to get input from the user:

```
.read_line(&mut guess)
```

Here, we call the read_line() method on our handle. Methods are like associated functions, but are only available on a particular instance of a type, rather than the type itself. We're also passing one argument to `read_line(): &mut guess`.

Remember how we bound `guess` above? We said it was mutable. However, `read_line` doesn't take a `String` as an argument: it takes a `&mut String`. Rust has a feature called 'references', which allows you to have multiple references to one piece of data, which can reduce copying. References are a complex feature, as one of Rust's major selling points is how safe and easy it is to use references. We don't need to know a lot of those details to finish our program right now, though. For now, all we need to know is that like `let` bindings, references are immutable by default. Hence, we need to write `&mut guess`, rather than `&guess`.

Why does `read_line()` take a mutable reference to a string? Its job is to take what the user types into standard input, and place that into a string. So it takes that string as an argument, and in order to add the input, it needs to be mutable.

But we're not quite done with this line of code, though. While it's a single line of text, it's only the first part of the single logical line of code:

```
        .expect("Failed to read line");
```

When you call a method with the `.foo()` syntax, you may introduce a newline and other whitespace. This helps you split up long lines. We *could* have done:

```
    io::stdin().read_line(&mut guess).expect("failed to read line");
```

But that gets hard to read. So we've split it up, three lines for three method calls. We already talked about `read_line()`, but what about `expect()`? Well, we already mentioned that `read_line()` puts what the user types into the `&mut String` we pass it. But it also returns a value: in this case, an io::Result. Rust has a number of types named `Result` in its standard library: a generic Result, and then specific versions for sub-libraries, like `io::Result`.

The purpose of these `Result` types is to encode error handling information. Values of the `Result` type, like any type, have methods defined on them. In this case, `io::Result` has an expect() method that takes a value it's called on, and if it isn't a successful one, panic!s with a message you passed it. A `panic!` like this will cause our program to crash, displaying the message.

If we leave off calling these two methods, our program will compile, but we'll get a warning:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:10:5: 10:39 warning: unused result which must be used,
#[warn(unused_must_use)] on by default
src/main.rs:10     io::stdin().read_line(&mut guess);
                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Rust warns us that we haven't used the `Result` value. This warning comes from a special annotation that `io::Result` has. Rust is trying to tell you that you haven't handled a possible error. The right way to suppress the error is to actually write error handling. Luckily, if we want to crash if there's a problem, we can use these two little methods. If we can recover from the error somehow, we'd do something else, but we'll save that for a future project.

There's only one line of this first example left:

```
    println!("You guessed: {}", guess);
}
```

This prints out the string we saved our input in. The `{}`s are a placeholder, and so we pass it `guess` as an argument. If we had multiple `{}`s, we would pass multiple arguments:

```
let x = 5;
let y = 10;

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

Easy.

Anyway, that's the tour. We can run what we have with `cargo run`:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
Please input your guess.
6
You guessed: 6
```

All right! Our first part is done: we can get input from the keyboard, and then print it back out.

3.3 Generating a secret number

Next, we need to generate a secret number. Rust does not yet include random number functionality in its standard library. The Rust team does, however, provide a rand crate. A 'crate' is a package of Rust code. We've been building a 'binary crate', which is an executable. `rand` is a 'library crate', which contains code that's intended to be used with other programs.

Using external crates is where Cargo really shines. Before we can write the code using `rand`, we need to modify our `Cargo.toml`. Open it up, and add these few lines at the bottom:

```
[dependencies]

rand="0.3.0"
```

The `[dependencies]` section of `Cargo.toml` is like the `[package]` section: everything that follows it is part of it, until the next section starts. Cargo uses the dependencies section to know what dependencies on external crates you have, and what versions you require. In this case, we've specified version `0.3.0`, which Cargo understands to be any release that's compatible with this specific version. Cargo understands Semantic Versioning, which is a standard for writing version numbers. A bare number like above is actually shorthand for `0ˆ3.0`, meaning "anything compatible with 0.3.0". If we wanted to use only `0.3.0` exactly, we could say `rand="=0.3.0"` (note the two equal signs). And if we wanted to use the latest version we could use `*`. We could also use a range of versions. Cargo's documentation contains more details.

Now, without changing any of our code, let's build our project:

```
$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading rand v0.3.8
 Downloading libc v0.1.6
   Compiling libc v0.1.6
   Compiling rand v0.3.8
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

(You may see different versions, of course.)

Lots of new output! Now that we have an external dependency, Cargo fetches the latest versions of everything from the registry, which is a copy of data from Crates.io. Crates.io is where people in the Rust ecosystem post their open source Rust projects for others to use.

After updating the registry, Cargo checks our `[dependencies]` and downloads any we don't have yet. In this case, while we only said we wanted to depend on `rand`, we've also grabbed a copy of `libc`. This is because `rand` depends on `libc` to work. After downloading them, it compiles them, and then compiles our project.

If we run `cargo build` again, we'll get different output:

```
$ cargo build
```

That's right, no output! Cargo knows that our project has been built, and that all of its dependencies are built, and so there's no reason to do all that stuff. With nothing to do, it simply exits.

If we open up `src/main.rs` again, make a trivial change, and then save it again, we'll only see one line:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
```

So, we told Cargo we wanted any `0.3.x` version of `rand`, and so it fetched the latest version at the time this was written, `v0.3.8`. But what happens when next week, version `v0.3.9` comes out, with an important bugfix? While getting bugfixes is important, what if `0.3.9` contains a regression that breaks our code?

The answer to this problem is the `Cargo.lock` file you'll now find in your project directory. When you build your project for the first time, Cargo figures out all of the versions that fit your criteria, and then writes them to the `Cargo.lock` file. When you build your project in the future, Cargo will see that the `Cargo.lock` file exists, and then use that specific version rather than do all the work of figuring out versions again. This lets you have a repeatable build automatically. In other words, we'll stay at `0.3.8` until we explicitly upgrade, and so will anyone who we share our code with, thanks to the lock file.

What about when we do want to use `v0.3.9`? Cargo has another command, `update`, which says 'ignore the lock, figure out all the latest versions that fit what we've specified. If that works, write those versions out to the lock file'. But, by default, Cargo will only look for versions larger than `0.3.0` and smaller than `0.4.0`. If we want to move to `0.4.x`, we'd have to update the `Cargo.toml` directly. When we do, the next time we `cargo build`, Cargo will update the index and re-evaluate our `rand` requirements.

There's a lot more to say about Cargo and its ecosystem, but for now, that's all we need to know. Cargo makes it really easy to re-use libraries, and so Rustaceans tend to write smaller projects which are assembled out of a number of sub-packages.

Let's get on to actually *using* `rand`. Here's our next step:

```
extern crate rand;

use std::io;
use rand::Rng;

fn main() {
    println!("Guess the number!");

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

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    println!("You guessed: {}", guess);
}
```

The first thing we've done is change the first line. It now says `extern crate rand`. Because we declared `rand` in our `[dependencies]`, we can use `extern crate` to let Rust know we'll be making use of it. This also does the equivalent of a `use rand`; as well, so we can make use of anything in the `rand` crate by prefixing it with `rand::`.

Next, we added another `use` line: `use rand::Rng`. We're going to use a method in a moment, and it requires that `Rng` be in scope to work. The basic idea is this: methods are defined on something called 'traits', and for the method to work, it needs the trait to be in scope. For more about the details, read the traits section.

There are two other lines we added, in the middle:

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

    println!("The secret number is: {}", secret_number);
```

We use the `rand::thread_rng()` function to get a copy of the random number generator, which is local to the particular thread of execution we're in. Because we `use rand::Rng`'d above, it has a `gen_range()` method available. This method takes two arguments, and generates a number between them. It's inclusive on the lower bound, but exclusive on the upper bound, so we need 1 and 101 to get a number ranging from one to a hundred.

The second line prints out the secret number. This is useful while we're developing our program, so we can easily test it out. But we'll be deleting it for the final version. It's not much of a game if it prints out the answer when you start it up!

Try running our new program a few times:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ cargo run
     Running `target/debug/guessing_game`
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
```

Great! Next up: comparing our guess to the secret number.

3.4 Comparting guesses

Now that we've got user input, let's compare our guess to the secret number. Here's our next step, though it doesn't quite compile yet:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

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

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

A few new bits here. The first is another `use`. We bring a type called `std::cmp::Ordering` into scope. Then, five new lines at the bottom that use it:

```
match guess.cmp(&secret_number) {
    Ordering::Less    => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal   => println!("You win!"),
}
```

The `cmp()` method can be called on anything that can be compared, and it takes a reference to the thing you want to compare it to. It returns the `Ordering` type we `use`d earlier. We use a match statement to determine exactly what kind of `Ordering` it is. `Ordering` is an enum, short for 'enumeration', which looks like this:

```
enum Foo {
  Bar,
  Baz,
}
```

With this definition, anything of type `Foo` can be either a `Foo::Bar` or a `Foo::Baz`. We use the `::` to indicate the namespace for a particular `enum` variant.

The Ordering `enum` has three possible variants: `Less`, `Equal`, and `Greater`. The `match` statement takes a value of a type, and lets you create an 'arm' for each possible value. Since we have three types of `Ordering`, we have three arms:

```
match guess.cmp(&secret_number) {
    Ordering::Less    => println!("Too small!"),
    Ordering::Greater => println!("Too big!"),
    Ordering::Equal   => println!("You win!"),
}
```

If it's `Less`, we print `Too small!`, if it's `Greater`, `Too big!`, and if `Equal`, `You win!`. `match` is really useful, and is used often in Rust.

I did mention that this won't quite compile yet, though. Let's try it:

```
$ cargo build
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
src/main.rs:28:21: 28:35 error: mismatched types:
 expected `&collections::string::String`,
    found `&_`
(expected struct `collections::string::String`,
    found integral variable) [E0308]
src/main.rs:28     match guess.cmp(&secret_number) {
                                   ^~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `guessing_game`.
```

Whew! This is a big error. The core of it is that we have 'mismatched types'. Rust has a strong, static type system. However, it also has type inference. When we wrote `let guess`

`= String::new()`, Rust was able to infer that `guess` should be a `String`, and so it doesn't make us write out the type. And with our `secret_number`, there are a number of types which can have a value between one and a hundred: `i32`, a thirty-two-bit number, or `u32`, an unsigned thirty-two-bit number, or `i64`, a sixty-four-bit number or others. So far, that hasn't mattered, and so Rust defaults to an `i32`. However, here, Rust doesn't know how to compare the `guess` and the `secret_number`. They need to be the same type. Ultimately, we want to convert the `String` we read as input into a real number type, for comparison. We can do that with three more lines. Here's our new program:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

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

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let mut guess = String::new();

    io::stdin().read_line(&mut guess)
        .expect("failed to read line");

    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");

    println!("You guessed: {}", guess);

    match guess.cmp(&secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}
```

The new three lines:

```
    let guess: u32 = guess.trim().parse()
        .expect("Please type a number!");
```

Wait a minute, I thought we already had a `guess`? We do, but Rust allows us to 'shadow' the previous `guess` with a new one. This is often used in this exact situation, where `guess` starts as a `String`, but we want to convert it to an `u32`. Shadowing lets us re-use the `guess` name, rather than forcing us to come up with two unique names like `guess_str` and `guess`, or something else.

We bind `guess` to an expression that looks like something we wrote earlier:

```
guess.trim().parse()
```

Here, `guess` refers to the old `guess`, the one that was a `String` with our input in it. The `trim()` method on `String`s will eliminate any white space at the beginning and end of our string. This is important, as we had to press the 'return' key to satisfy `read_line()`. This means that if we type `5` and hit return, `guess` looks like this: `5`
`n`. The
`n` represents 'newline', the enter key. `trim()` gets rid of this, leaving our string with only the `5`. The parse() method on strings parses a string into some kind of number. Since it can parse a variety of numbers, we need to give Rust a hint as to the exact type of number we want. Hence, `let guess: u32`. The colon (`:`) after `guess` tells Rust we're going to annotate its type. `u32` is an unsigned, thirty-two bit integer. Rust has a number of built-in number types, but we've chosen `u32`. It's a good default choice for a small positive number.

Just like `read_line()`, our call to `parse()` could cause an error. What if our string contained `A%`? There'd be no way to convert that to a number. As such, we'll do the same thing we did with `read_line()`: use the `expect()` method to crash if there's an error.

Let's try our program out!

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! You can see I even added spaces before my guess, and it still figured out that I guessed 76. Run the program a few times, and verify that guessing the number works, as well as guessing a number too small.

Now we've got most of the game working, but we can only make one guess. Let's change that by adding loops!

3.5 Looping

The `loop` keyword gives us an infinite loop. Let's add that in:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

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

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => println!("You win!"),
        }
    }
}
```

And try it out. But wait, didn't we just add an infinite loop? Yup. Remember our discussion about `parse()`? If we give a non-number answer, we'll `panic!` and quit. Observe:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
```

```
You guessed: 59
You win!
Please input your guess.
quit
thread '<main>' panicked at 'Please type a number!'
```

Ha! `quit` actually quits. As does any other non-number input. Well, this is suboptimal to say the least. First, let's actually quit when you win the game:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

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

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = guess.trim().parse()
            .expect("Please type a number!");

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

By adding the `break` line after the `You win!`, we'll exit the loop when we win. Exiting the loop also means exiting the program, since it's the last thing in `main()`. We have only one more tweak to make: when someone inputs a non-number, we don't want to quit, we want to ignore it. We can do that like this:

```
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

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

    println!("The secret number is: {}", secret_number);

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

These are the lines that changed:

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};
```

This is how you generally move from 'crash on error' to 'actually handle the returned by `parse()` is an `enum` like `Ordering`, but in this case, each variant has some data associated with it: `Ok` is a success, and `Err` is a failure. Each contains more information: the successfully parsed integer, or an error type. In this case, we `match` on `Ok(num)`, which sets the inner value of the `Ok` to the name `num`, and then we return it on the right-hand side. In the `Err` case, we don't care what kind of error it is, so we use `_` instead of a name. This ignores the error, and `continue` causes us to go to the next iteration of the `loop`.

Now we should be good! Let's try:

```
$ cargo run
   Compiling guessing_game v0.1.0 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
You guessed: 99
Too big!
Please input your guess.
foo
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny last tweak, we have finished the guessing game. Can you think of what it is? That's right, we don't want to print out the secret number. It was good for testing, but it kind of ruins the game. Here's our final source:

```rust
extern crate rand;

use std::io;
use std::cmp::Ordering;
use rand::Rng;

fn main() {
    println!("Guess the number!");

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

    loop {
        println!("Please input your guess.");

        let mut guess = String::new();

        io::stdin().read_line(&mut guess)
            .expect("failed to read line");

        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };

        println!("You guessed: {}", guess);

        match guess.cmp(&secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                break;
            }
        }
    }
}
```

3.6 Complete

At this point, you have successfully built the Guessing Game! Congratulations!

This first project showed you a lot: `let`, `match`, methods, associated functions, using external crates, and more. Our next project will show off even more.

4 Syntax and Semantics

This chapter breaks Rust down into small chunks, one for each concept.

If you'd like to learn Rust from the bottom up, reading this in order is a great way to do that.

These sections also form a reference for each concept, so if you're reading another tutorial and find something confusing, you can find it explained somewhere in here.

4.1 Variable Bindings

Virtually every non-'Hello World' Rust program uses *variable bindings*. They bind some value to a name, so it can be used later. `let` is used to introduce a binding, like this:

```
fn main() {
    let x = 5;
}
```

Putting `fn main() {` in each example is a bit tedious, so we'll leave that out in the future. If you're following along, make sure to edit your `main()` function, rather than leaving it off. Otherwise, you'll get an error.

Patterns

In many languages, a variable binding would be called a *variable*, but Rust's variable bindings have a few tricks up their sleeves. For example the left-hand side of a `let` expression is a 'pattern', not a variable name. This means we can do things like:

```
let (x, y) = (1, 2);
```

After this expression is evaluated, `x` will be one, and `y` will be two. Patterns are really powerful, and have their own section in the book. We don't need those features for now, so we'll keep this in the back of our minds as we go forward.

Type annotations

Rust is a statically typed language, which means that we specify our types up front, and they're checked at compile time. So why does our first example compile? Well, Rust has this thing called 'type inference'. If it can figure out what the type of something is, Rust doesn't require you to actually type it out.

We can add the type if we want to, though. Types come after a colon (`:`):

```
let x: i32 = 5;
```

If I asked you to read this out loud to the rest of the class, you'd say "x is a binding with the type i32 and the value five."

In this case we chose to represent x as a 32-bit signed integer. Rust has many different primitive integer types. They begin with i for signed integers and u for unsigned integers. The possible integer sizes are 8, 16, 32, and 64 bits.

In future examples, we may annotate the type in a comment. The examples will look like this:

```
fn main() {
    let x = 5; // x: i32
}
```

Note the similarities between this annotation and the syntax you use with let. Including these kinds of comments is not idiomatic Rust, but we'll occasionally include them to help you understand what the types that Rust infers are.

Mutability

By default, bindings are *immutable*. This code will not compile:

```
let x = 5;
x = 10;
```

It will give you this error:

```
error: re-assignment of immutable variable `x`
     x = 10;
     ^~~~~~~
```

If you want a binding to be mutable, you can use mut:

```
let mut x = 5; // mut x: i32
x = 10;
```

There is no single reason that bindings are immutable by default, but we can think about it through one of Rust's primary focuses: safety. If you forget to say mut, the compiler will catch it, and let you know that you have mutated something you may not have intended to mutate. If bindings were mutable by default, the compiler would not be able to tell you this. If you *did* intend mutation, then the solution is quite easy: add mut.

There are other good reasons to avoid mutable state when possible, but they're out of the scope of this guide. In general, you can often avoid explicit mutation, and so it is preferable in Rust. That said, sometimes, mutation is what you need, so it's not verboten.

Initializing bindings

Rust variable bindings have one more aspect that differs from other languages: bindings are required to be initialized with a value before you're allowed to use them.

Let's try it out. Change your `src/main.rs` file to look like this:

```
fn main() {
    let x: i32;

    println!("Hello world!");
}
```

You can use `cargo build` on the command line to build it. You'll get a warning, but it will still print "Hello, world!":

```
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)]
   on by default
src/main.rs:2     let x: i32;
                      ^
```

Rust warns us that we never use the variable binding, but since we never use it, no harm, no foul. Things change if we try to actually use this `x`, however. Let's do that. Change your program to look like this:

```
fn main() {
    let x: i32;

    println!("The value of x is: {}", x);
}
```

And try to build it. You'll get an error:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4     println!("The value of x is: {}", x);
                                                    ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.
```

Rust will not let us use a value that has not been initialized. Next, let's talk about this stuff we've added to `println!`.

If you include two curly braces (`{}`, some call them moustaches...) in your string to print, Rust will interpret this as a request to interpolate some sort of value. *String interpolation* is a computer science term that means ”stick in the middle of a string.” We add a comma, and then `x`, to indicate that we want `x` to be the value we’re interpolating. The comma is used to separate arguments we pass to functions and macros, if you’re passing more than one.

When you use the curly braces, Rust will attempt to display the value in a meaningful way by checking out its type. If you want to specify the format in a more detailed manner, there are a wide number of options available. For now, we’ll stick to the default: integers aren’t very complicated to print.

Scope and shadowing

Let’s get back to bindings. Variable bindings have a scope - they are constrained to live in a block they were defined in. A block is a collection of statements enclosed by `{` and `}`. Function definitions are also blocks! In the following example we define two variable bindings, `x` and `y`, which live in different blocks. `x` can be accessed from inside the `fn main() {}` block, while `y` can be accessed only from inside the inner block:

```
fn main() {
    let x: i32 = 17;
    {
        let y: i32 = 3;
        println!("The value of x is {} and value of y is {}", x, y);
    }
    println!("The value of x is {} and value of y is {}", x, y); // This
↪   won't work
}
```

The first `println!` would print ”The value of x is 17 and the value of y is 3”, but this example cannot be compiled successfully, because the second `println!` cannot access the value of `y`, since it is not in scope anymore. Instead we get this error:

```
$ cargo build
   Compiling hello v0.1.0 (file:///home/you/projects/hello_world)
main.rs:7:62: 7:63 error: unresolved name `y`. Did you mean `x`? [E0425]
main.rs:7     println!("The value of x is {} and value of y is {}", x, y); // This won't wor
                                                                       ^
note: in expansion of format_args!
<std macros>:2:25: 2:56 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
main.rs:7:5: 7:65 note: expansion site
main.rs:7:62: 7:63 help: run `rustc --explain E0425` to see a detailed explanation
error: aborting due to previous error
Could not compile `hello`.

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

Additionally, variable bindings can be shadowed. This means that a later variable binding with the same name as another binding, that’s currently in scope, will override the previous binding.

```
let x: i32 = 8;
{
    println!("{}", x); // Prints "8"
    let x = 12;
    println!("{}", x); // Prints "12"
}
println!("{}", x); // Prints "8"
let x =  42;
println!("{}", x); // Prints "42"
```

Shadowing and mutable bindings may appear as two sides of the same coin, but they are two distinct concepts that can't always be used interchangeably. For one, shadowing enables us to rebind a name to a value of a different type. It is also possible to change the mutability of a binding.

```
let mut x: i32 = 1;
x = 7;
let x = x; // x is now immutable and is bound to 7

let y = 4;
let y = "I can also be bound to text!"; // y is now of a different type
```

4.2 Functions

Every Rust program has at least one function, the `main` function:

```
fn main() {
}
```

This is the simplest possible function declaration. As we mentioned before, `fn` says 'this is a function', followed by the name, some parentheses because this function takes no arguments, and then some curly braces to indicate the body. Here's a function named `foo`:

```
fn foo() {
}
```

So, what about taking arguments? Here's a function that prints a number:

```
fn print_number(x: i32) {
  println!("x is: {}", x);
}
```

Here's a complete program that uses `print_number`:

```
fn main() {
    print_number(5);
}

fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

As you can see, function arguments work very similar to `let` declarations: you add a type to the argument name, after a colon.

Here's a complete program that adds two numbers together and prints them:

```
fn main() {
    print_sum(5, 6);
}

fn print_sum(x: i32, y: i32) {
    println!("sum is: {}", x + y);
}
```

You separate arguments with a comma, both when you call the function, as well as when you declare it.

Unlike `let`, you must declare the types of function arguments. This does not work:

```
fn print_sum(x, y) {
    println!("sum is: {}", x + y);
}
```

You get this error:

```
expected one of `!`, `:`, or `@`, found `)`
fn print_number(x, y) {
```

This is a deliberate design decision. While full-program inference is possible, languages which have it, like Haskell, often suggest that documenting your types explicitly is a best-practice. We agree that forcing functions to declare types while allowing for inference inside of function bodies is a wonderful sweet spot between full inference and no inference.

What about returning a value? Here's a function that adds one to an integer:

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

Rust functions return exactly one value, and you declare the type after an 'arrow', which is a dash (`-`) followed by a greater-than sign (`>`). The last line of a function determines what it returns. You'll note the lack of a semicolon here. If we added it in:

```rust
fn add_one(x: i32) -> i32 {
    x + 1;
}
```

We would get an error:

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
     x + 1;
}

help: consider removing this semicolon:
     x + 1;
          ^
```

This reveals two interesting things about Rust: it is an expression-based language, and semi-colons are different from semicolons in other 'curly brace and semicolon'-based languages. These two things are related.

Expressions vs. Statements

Rust is primarily an expression-based language. There are only two kinds of statements, and everything else is an expression.

So what's the difference? Expressions return a value, and statements do not. That's why we end up with 'not all control paths return a value' here: the statement `x + 1;` doesn't return a value. There are two kinds of statements in Rust: 'declaration statements' and 'expression statements'. Everything else is an expression. Let's talk about declaration statements first.

In some languages, variable bindings can be written as expressions, not statements. Like Ruby:

```ruby
x = y = 5
```

In Rust, however, using `let` to introduce a binding is *not* an expression. The following will produce a compile-time error:

```rust
let x = (let y = 5); // expected identifier, found keyword `let`
```

The compiler is telling us here that it was expecting to see the beginning of an expression, and a `let` can only begin a statement, not an expression.

Note that assigning to an already-bound variable (e.g. `y = 5`) is still an expression, although its value is not particularly useful. Unlike other languages where an assignment evaluates to the assigned value (e.g. `5` in the previous example), in Rust the value of an assignment is an empty tuple `()` because the assigned value can have only one owner, and any other returned value would be too surprising:

```
let mut y = 5;

let x = (y = 6);  // x has the value `()`, not `6`
```

The second kind of statement in Rust is the *expression statement*. Its purpose is to turn any expression into a statement. In practical terms, Rust's grammar expects statements to follow other statements. This means that you use semicolons to separate expressions from each other. This means that Rust looks a lot like most other languages that require you to use semicolons at the end of every line, and you will see semicolons at the end of almost every line of Rust code you see.

What is this exception that makes us say ”almost”? You saw it already, in this code:

```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

Our function claims to return an `i32`, but with a semicolon, it would return `()` instead. Rust realizes this probably isn't what we want, and suggests removing the semicolon in the error we saw before.

Early returns

But what about early returns? Rust does have a keyword for that, `return`:

```
fn foo(x: i32) -> i32 {
    return x;

    // we never run this code!
    x + 1
}
```

Using a `return` as the last line of a function works, but is considered poor style:

```
fn foo(x: i32) -> i32 {
    return x + 1;
}
```

The previous definition without `return` may look a bit strange if you haven't worked in an expression-based language before, but it becomes intuitive over time.

Diverging functions

Rust has some special syntax for 'diverging functions', which are functions that do not return:

```
fn diverges() -> ! {
    panic!("This function never returns!");
}
```

`panic!` is a macro, similar to `println!()` that we've already seen. Unlike `println!()`, `panic!()` causes the current thread of execution to crash with the given message. Because this function will cause a crash, it will never return, and so it has the type '`!`', which is read 'diverges'.

If you add a main function that calls `diverges()` and run it, you'll get some output that looks like this:

```
thread '<main>' panicked at 'This function never returns!', hello.rs:2
```

If you want more information, you can get a backtrace by setting the `RUST_BACKTRACE` environment variable:

```
$ RUST_BACKTRACE=1 ./diverges
thread '<main>' panicked at 'This function never returns!', hello.rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447423903650
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

`RUST_BACKTRACE` also works with Cargo's `run` command:

```
$ RUST_BACKTRACE=1 cargo run
     Running `target/debug/diverges`
thread '<main>' panicked at 'This function never returns!', hello.rs:2
stack backtrace:
   1:     0x7f402773a829 - sys::backtrace::write::h0942de78b6c02817K8r
   2:     0x7f402773d7fc - panicking::on_panic::h3f23f9d0b5f4c91bu9w
   3:     0x7f402773960e - rt::unwind::begin_unwind_inner::h2844b8c5e81e79558Bw
   4:     0x7f4027738893 - rt::unwind::begin_unwind::h4375279447423903650
   5:     0x7f4027738809 - diverges::h2266b4c4b850236beaa
   6:     0x7f40277389e5 - main::h19bb1149c2f00ecfBaa
   7:     0x7f402773f514 - rt::unwind::try::try_fn::h13186883479104382231
   8:     0x7f402773d1d8 - __rust_try
   9:     0x7f402773f201 - rt::lang_start::ha172a3ce74bb453aK5w
  10:     0x7f4027738a19 - main
  11:     0x7f402694ab44 - __libc_start_main
  12:     0x7f40277386c8 - <unknown>
  13:                0x0 - <unknown>
```

A diverging function can be used as any type:

```
let x: i32 = diverges();
let x: String = diverges();
```

Function pointers

We can also create variable bindings which point to functions:

```
let f: fn(i32) -> i32;
```

`f` is a variable binding which points to a function that takes an `i32` as an argument and returns an `i32`. For example:

```
fn plus_one(i: i32) -> i32 {
    i + 1
}

// without type inference
let f: fn(i32) -> i32 = plus_one;

// with type inference
let f = plus_one;
```

We can then use `f` to call the function:

```
let six = f(5);
```

4.3 Primitive Types

The Rust language has a number of types that are considered 'primitive'. This means that they're built-in to the language. Rust is structured in such a way that the standard library also provides a number of useful types built on top of these ones, as well, but these are the most primitive.

Booleans

Rust has a built in boolean type, named `bool`. It has two values, `true` and `false`:

```
let x = true;

let y: bool = false;
```

A common use of booleans is in if conditionals.

You can find more documentation for `bool`s in the standard library documentation.

char

The `char` type represents a single Unicode scalar value. You can create `char`s with a single tick: (`'`)

```
let x = 'x';
let two_hearts = ' ⊠';
```

Unlike some other languages, this means that Rust's `char` is not a single byte, but four.

You can find more documentation for `char`s in the standard library documentation.

Numeric types

Rust has a variety of numeric types in a few categories: signed and unsigned, fixed and variable, floating-point and integer.

These types consist of two parts: the category, and the size. For example, `u16` is an unsigned type with sixteen bits of size. More bits lets you have bigger numbers.

If a number literal has nothing to cause its type to be inferred, it defaults:

```
let x = 42; // x has type i32

let y = 1.0; // y has type f64
```

Here's a list of the different numeric types, with links to their documentation in the standard library:

- i8
- i16
- i32
- i64
- u8
- u16
- u32
- u64
- isize
- usize
- f32
- f64

Let's go over them by category:

Signed and Unsigned

Integer types come in two varieties: signed and unsigned. To understand the difference, let's consider a number with four bits of size. A signed, four-bit number would let you store numbers from `-8` to `+7`. Signed numbers use "two's complement representation". An unsigned four bit number, since it does not need to store negatives, can store values from `0` to `+15`.

Unsigned types use a `u` for their category, and signed types use `i`. The `i` is for 'integer'. So `u8` is an eight-bit unsigned number, and `i8` is an eight-bit signed number.

Fixed size types

Fixed size types have a specific number of bits in their representation. Valid bit sizes are `8`, `16`, `32`, and `64`. So, `u32` is an unsigned, 32-bit integer, and `i64` is a signed, 64-bit integer.

Variable sized types

Rust also provides types whose size depends on the size of a pointer of the underlying machine. These types have 'size' as the category, and come in signed and unsigned varieties. This makes for two types: `isize` and `usize`.

Floating-point types

Rust also has two floating point types: `f32` and `f64`. These correspond to IEEE-754 single and double precision numbers.

Arrays

Like many programming languages, Rust has list types to represent a sequence of things. The most basic is the *array*, a fixed-size list of elements of the same type. By default, arrays are immutable.

```
let a = [1, 2, 3]; // a: [i32; 3]
let mut m = [1, 2, 3]; // m: [i32; 3]
```

Arrays have type `[T; N]`. We'll talk about this `T` notation in the generics section. The `N` is a compile-time constant, for the length of the array.

There's a shorthand for initializing each element of an array to the same value. In this example, each element of a will be initialized to `0`:

```
let a = [0; 20]; // a: [i32; 20]
```

You can get the number of elements in an array `a` with `a.len()`:

```
let a = [1, 2, 3];

println!("a has {} elements", a.len());
```

You can access a particular element of an array with *subscript notation*:

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

Subscripts start at zero, like in most programming languages, so the first name is `names[0]` and the second name is `names[1]`. The above example prints `The second name is: Brian`. If you try to use a subscript that is not in the array, you will get an error: array access is bounds-checked at run-time. Such errant access is the source of many bugs in other systems programming languages.

You can find more documentation for `array`s in the standard library documentation.

Slices

A 'slice' is a reference to (or "view" into) another data structure. They are useful for allowing safe, efficient access to a portion of an array without copying. For example, you might want to reference only one line of a file read into memory. By nature, a slice is not created directly, but from an existing variable binding. Slices have a defined length, can be mutable or immutable.

Slicing syntax

You can use a combo of `&` and `[]` to create a slice from various things. The `&` indicates that slices are similar to references, which we will cover in detail later in this section. The `[]`s, with a range, let you define the length of the slice:

```
let a = [0, 1, 2, 3, 4];
let complete = &a[..]; // A slice containing all of the elements in a
let middle = &a[1..4]; // A slice of a: only the elements 1, 2, and 3
```

Slices have type `&[T]`. We'll talk about that `T` when we cover generics.

You can find more documentation for slices in the standard library documentation.

str

Rust's `str` type is the most primitive string type. As an unsized type, it's not very useful by itself, but becomes useful when placed behind a reference, like `&str`. We'll elaborate further when we cover Strings and references.

You can find more documentation for `str` in the standard library documentation.

Tuples

A tuple is an ordered list of fixed size. Like this:

```
let x = (1, "hello");
```

The parentheses and commas form this two-length tuple. Here's the same code, but with the type annotated:

```
let x: (i32, &str) = (1, "hello");
```

As you can see, the type of a tuple looks like the tuple, but with each position having a type name rather than the value. Careful readers will also note that tuples are heterogeneous: we have an `i32` and a `&str` in this tuple. In systems programming languages, strings are a bit more complex than in other languages. For now, read `&str` as a *string slice*, and we'll learn more soon.

You can assign one tuple into another, if they have the same contained types and Arity. Tuples have the same arity when they have the same length.

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3); // y: (i32, i32)

x = y;
```

You can access the fields in a tuple through a *destructuring let*. Here's an example:

```
let (x, y, z) = (1, 2, 3);

println!("x is {}", x);
```

Remember before when I said the left-hand side of a `let` statement was more powerful than assigning a binding? Here we are. We can put a pattern on the left-hand side of the `let`, and if it matches up to the right-hand side, we can assign multiple bindings at once. In this case, `let` "destructures" or "breaks up" the tuple, and assigns the bits to three bindings.

This pattern is very powerful, and we'll see it repeated more later.

You can disambiguate a single-element tuple from a value in parentheses with a comma:

```
(0,); // single-element tuple
(0); // zero in parentheses
```

Tuple Indexing

You can also access fields of a tuple with indexing syntax:

```
let tuple = (1, 2, 3);

let x = tuple.0;
let y = tuple.1;
let z = tuple.2;

println!("x is {}", x);
```

Like array indexing, it starts at zero, but unlike array indexing, it uses a `.`, rather than `[]`s.

You can find more documentation for tuples in the standard library documentation.

Functions

Functions also have a type! They look like this:

```
fn foo(x: i32) -> i32 { x }

let x: fn(i32) -> i32 = foo;
```

In this case, `x` is a 'function pointer' to a function that takes an `i32` and returns an `i32`.

4.4 Comments

Now that we have some functions, it's a good idea to learn about comments. Comments are notes that you leave to other programmers to help explain things about your code. The compiler mostly ignores them.

Rust has two kinds of comments that you should care about: *line comments* and *doc comments*.

```
// Line comments are anything after '//' and extend to the end of the line.

let x = 5; // this is also a line comment.

// If you have a long explanation for something, you can put line comments
↪  next
// to each other. Put a space between the // and your comment so that it's
// more readable.
```

The other kind of comment is a doc comment. Doc comments use `///` instead of `//`, and support Markdown notation inside:

```
/// Adds one to the number given.
///
/// # Examples
///
/// '''
/// let five = 5;
///
/// assert_eq!(6, add_one(5));
/// # fn add_one(x: i32) -> i32 {
/// #     x + 1
/// # }
/// '''
fn add_one(x: i32) -> i32 {
    x + 1
}
```

There is another style of doc comment, `//!`, to comment containing items (e.g. crates, modules or functions), instead of the items following it. Commonly used inside crates root (lib.rs) or modules root (mod.rs):

```
//! # The Rust Standard Library
//!
//! The Rust Standard Library provides the essential runtime
//! functionality for building portable Rust software.
```

When writing doc comments, providing some examples of usage is very, very helpful. You'll notice we've used a new macro here: `assert_eq!`. This compares two values, and `panic!`s if they're not equal to each other. It's very helpful in documentation. There's another macro, `assert!`, which `panic!`s if the value passed to it is `false`.

You can use the rustdoc tool to generate HTML documentation from these doc comments, and also to run the code examples as tests!

4.5 If

Rust's take on `if` is not particularly complex, but it's much more like the `if` you'll find in a dynamically typed language than in a more traditional systems language. So let's talk about it, to make sure you grasp the nuances.

`if` is a specific form of a more general concept, the 'branch'. The name comes from a branch in a tree: a decision point, where depending on a choice, multiple paths can be taken.

In the case of `if`, there is one choice that leads down two paths:

```
let x = 5;

if x == 5 {
    println!("x is five!");
}
```

If we changed the value of `x` to something else, this line would not print. More specifically, if the expression after the `if` evaluates to `true`, then the block is executed. If it's `false`, then it is not.

If you want something to happen in the `false` case, use an `else`:

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else {
    println!("x is not five :(");
}
```

If there is more than one case, use an `else if`:

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else if x == 6 {
    println!("x is six!");
} else {
    println!("x is not five or six :(");
}
```

This is all pretty standard. However, you can also do this:

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

Which we can (and probably should) write like this:

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

This works because `if` is an expression. The value of the expression is the value of the last expression in whichever branch was chosen. An `if` without an `else` always results in `()` as the value.

4.6 Loops

Rust currently provides three approaches to performing some kind of iterative activity. They are: `loop`, `while` and `for`. Each approach has its own set of uses.

loop

The infinite `loop` is the simplest form of loop available in Rust. Using the keyword `loop`, Rust provides a way to loop indefinitely until some terminating statement is reached. Rust's infinite `loop`s look like this:

```
loop {
    println!("Loop forever!");
}
```

while

Rust also has a `while` loop. It looks like this:

```
let mut x = 5; // mut x: i32
let mut done = false; // mut done: bool

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

`while` loops are the correct choice when you're not sure how many times you need to loop.

If you need an infinite loop, you may be tempted to write this:

```
while true {
```

However, `loop` is far better suited to handle this case:

```
loop {
```

Rust's control-flow analysis treats this construct differently than a `while true`, since we know that it will always loop. In general, the more information we can give to the compiler, the better it can do with safety and code generation, so you should always prefer `loop` when you plan to loop infinitely.

for

The `for` loop is used to loop a particular number of times. Rust's `for` loops work a bit differently than in other systems languages, however. Rust's `for` loop doesn't look like this "C-style" `for` loop:

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

Instead, it looks like this:

```
for x in 0..10 {
    println!("{}", x); // x: i32
}
```

In slightly more abstract terms,

```
for var in expression {
    code
}
```

The expression is an item that can be converted into an iterator using . The iterator gives back a series of elements. Each element is one iteration of the loop. That value is then bound to the name `var`, which is valid for the loop body. Once the body is over, the next value is fetched from the iterator, and we loop another time. When there are no more values, the for loop is over.

In our example, `0..10` is an expression that takes a start and an end position, and gives an iterator over those values. The upper bound is exclusive, though, so our loop will print `0` through `9`, not `10`.

Rust does not have the "C-style" `for` loop on purpose. Manually controlling each element of the loop is complicated and error prone, even for experienced C developers.

Enumerate

When you need to keep track of how many times you already looped, you can use the `.enumerate()` function.

On ranges:

```
for (i,j) in (5..10).enumerate() {
    println!("i = {} and j = {}", i, j);
}
```

Outputs:

```
i = 0 and j = 5
i = 1 and j = 6
i = 2 and j = 7
i = 3 and j = 8
i = 4 and j = 9
```

Don't forget to add the parentheses around the range.

On iterators:

```
for (linenumber, line) in lines.enumerate() {
    println!("{}: {}", linenumber, line);
}
```

Outputs:

```
0: Content of line one
1: Content of line two
2: Content of line three
3: Content of line four
```

Ending iteration early

Let's take a look at that `while` loop we had earlier:

```
let mut x = 5;
let mut done = false;

while !done {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 {
        done = true;
    }
}
```

We had to keep a dedicated `mut` boolean variable binding, `done`, to know when we should exit out of the loop. Rust has two keywords to help us with modifying iteration: `break` and `continue`.

In this case, we can write the loop in a better way with `break`:

```
let mut x = 5;

loop {
    x += x - 3;

    println!("{}", x);

    if x % 5 == 0 { break; }
}
```

We now loop forever with `loop` and use `break` to break out early. Issuing an explicit `return` statement will also serve to terminate the loop early.

`continue` is similar, but instead of ending the loop, goes to the next iteration. This will only print the odd numbers:

```
for x in 0..10 {
    if x % 2 == 0 { continue; }

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

Loop labels

You may also encounter situations where you have nested loops and need to specify which one your `break` or `continue` statement is for. Like most other languages, by default a `break` or `continue` will apply to innermost loop. In a situation where you would like to a `break` or `continue` for one of the outer loops, you can use labels to specify which loop the `break` or `continue` statement applies to. This will only print when both `x` and `y` are odd:

```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // continues the loop over x
        if y % 2 == 0 { continue 'inner; } // continues the loop over y
        println!("x: {}, y: {}", x, y);
    }
}
```

4.7 Ownership

This guide is one of three presenting Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership, which you're reading now
- borrowing, and their associated feature 'references'
- lifetimes, an advanced concept of borrowing

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero-cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about ownership.

Ownership

Variable Bindings have a property in Rust: they 'have ownership' of what they're bound to. This means that when a binding goes out of scope, Rust will free the bound resources. For example:

```
fn foo() {
    let v = vec![1, 2, 3];
}
```

When `v` comes into scope, a new [vector] is created, and it allocates space on the heap for each of its elements. When `v` goes out of scope at the end of `foo()`, Rust will clean up everything related to the vector, even the heap-allocated memory. This happens deterministically, at the end of the scope.

We'll cover vectors in detail later in this chapter; we only use them here as an example of a type that allocates space on the heap at runtime. They behave like arrays, except their size may change by `push()`ing more elements onto them.

Vectors have a generic type `Vec<T>`, so in this example `v` will have type `Vec<i32>`. We'll cover generics in detail later in this chapter.

Move semantics

There's some more subtlety here, though: Rust ensures that there is *exactly one* binding to any given resource. For example, if we have a vector, we can assign it to another binding:

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

let v2 = v;
```

But, if we try to use `v` afterwards, we get an error:

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

let v2 = v;

println!("v[0] is: {}", v[0]);
```

It looks like this:

```
error: use of moved value: `v`
println!("v[0] is: {}", v[0]);
                        ^
```

A similar thing happens if we define a function which takes ownership, and try to use something after we've passed it as an argument:

```
fn take(v: Vec<i32>) {
    // what happens here isn't important.
}

let v = vec![1, 2, 3];

take(v);

println!("v[0] is: {}", v[0]);
```

Same error: 'use of moved value'. When we transfer ownership to something else, we say that we've 'moved' the thing we refer to. You don't need some sort of special annotation here, it's the default thing that Rust does.

The details

The reason that we cannot use a binding after we've moved it is subtle, but important. When we write code like this:

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

let v2 = v;
```

The first line allocates memory for the vector object, `v`, and for the data it contains. The vector object is stored on the stack and contains a pointer to the content (`[1, 2, 3]`) stored on the heap. When we move `v` to `v2`, it creates a copy of that pointer, for `v2`. Which means that there would be two pointers to the content of the vector on the heap. It would violate Rust's safety guarantees by introducing a data race. Therefore, Rust forbids using `v` after we've done the move.

It's also important to note that optimizations may remove the actual copy of the bytes on the stack, depending on circumstances. So it may not be as inefficient as it initially seems.

`Copy` types

We've established that when ownership is transferred to another binding, you cannot use the original binding. However, there's a trait that changes this behavior, and it's called `Copy`. We haven't discussed traits yet, but for now, you can think of them as an annotation to a particular type that adds extra behavior. For example:

```
let v = 1;

let v2 = v;

println!("v is: {}", v);
```

In this case, `v` is an `i32`, which implements the `Copy` trait. This means that, just like a move, when we assign `v` to `v2`, a copy of the data is made. But, unlike a move, we can still use `v` afterward. This is because an `i32` has no pointers to data somewhere else, copying it is a full copy.

All primitive types implement the `Copy` trait and their ownership is therefore not moved like one would assume, following the 'ownership rules'. To give an example, the two following snippets of code only compile because the `i32` and `bool` types implement the `Copy` trait.

```
fn main() {
    let a = 5;

    let _y = double(a);
    println!("{}", a);
}

fn double(x: i32) -> i32 {
    x * 2
}


fn main() {
    let a = true;

    let _y = change_truth(a);
    println!("{}", a);
}

fn change_truth(x: bool) -> bool {
    !x
}
```

If we had used types that do not implement the `Copy` trait, we would have gotten a compile error because we tried to use a moved value.

```
error: use of moved value: `a`
println!("{}", a);
               ^
```

We will discuss how to make your own types `Copy` in the traits section.

More than ownership

Of course, if we had to hand ownership back with every function we wrote:

```
fn foo(v: Vec<i32>) -> Vec<i32> {
    // do stuff with v

    // hand back ownership
    v
}
```

This would get very tedious. It gets worse the more things we want to take ownership of:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

Ugh! The return type, return line, and calling the function gets way more complicated.

Luckily, Rust offers a feature, borrowing, which helps us solve this problem. It's the topic of the next section!

4.8 References and Borrowing

This guide is one of three presenting Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership, which you're reading now
- borrowing, and their associated feature 'references'
- lifetimes, an advanced concept of borrowing

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero-cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about borrowing.

Borrowing

At the end of the Ownership section, we had a nasty function that looked like this:

```
fn foo(v1: Vec<i32>, v2: Vec<i32>) -> (Vec<i32>, Vec<i32>, i32) {
    // do stuff with v1 and v2

    // hand back ownership, and the result of our function
    (v1, v2, 42)
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let (v1, v2, answer) = foo(v1, v2);
```

This is not idiomatic Rust, however, as it doesn't take advantage of borrowing. Here's the first step:

```
fn foo(v1: &Vec<i32>, v2: &Vec<i32>) -> i32 {
    // do stuff with v1 and v2

    // return the answer
    42
}

let v1 = vec![1, 2, 3];
let v2 = vec![1, 2, 3];

let answer = foo(&v1, &v2);

// we can use v1 and v2 here!
```

Instead of taking `Vec<i32>`s as our arguments, we take a reference: `&Vec<i32>`. And instead of passing `v1` and `v2` directly, we pass `&v1` and `&v2`. We call the `&T` type a 'reference', and rather than owning the resource, it borrows ownership. A binding that borrows something does not deallocate the resource when it goes out of scope. This means that after the call to `foo()`, we can use our original bindings again.

References are immutable, like bindings. This means that inside of `foo()`, the vectors can't be changed at all:

```
fn foo(v: &Vec<i32>) {
     v.push(5);
}

let v = vec![];

foo(&v);
```

errors with:

```
error: cannot borrow immutable borrowed content `*v` as mutable
v.push(5);
^
```

Pushing a value mutates the vector, and so we aren't allowed to do it.

&mut references

There's a second kind of reference: `&mut T`. A 'mutable reference' allows you to mutate the resource you're borrowing. For example:

```
let mut x = 5;
{
    let y = &mut x;
    *y += 1;
}
println!("{}", x);
```

This will print 6. We make y a mutable reference to x then add one to the thing y points at. You'll notice that x had to be marked mut as well. If it wasn't, we couldn't take a mutable borrow to an immutable value.

You'll also notice we added an asterisk (*) in front of y, making it *y, this is because y is a &mut reference. You'll also need to use them for accessing the contents of a reference as well.

Otherwise, &mut references are like references. There is a large difference between the two, and how they interact, though. You can tell something is fishy in the above example, because we need that extra scope, with the { and }. If we remove them, we get an error:

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
note: previous borrow of `x` occurs here; the mutable borrow prevents
subsequent moves, borrows, or modification of `x` until the borrow ends
        let y = &mut x;
                     ^
note: previous borrow ends here
fn main() {

}
^
```

As it turns out, there are rules.

The Rules

Here's the rules about borrowing in Rust:

First, any borrow must last for a scope no greater than that of the owner. Second, you may have one or the other of these two kinds of borrows, but not both at the same time:

- one or more references (&T) to a resource,
- exactly one mutable reference (&mut T).

You may notice that this is very similar, though not exactly the same as, to the definition of a data race:

> There is a 'data race' when two or more pointers access the same memory location at the same time, where at least one of them is writing, and the operations are not synchronized.

With references, you may have as many as you'd like, since none of them are writing. However, as we can only have one &mut at a time, it is impossible to have a data race. This is how Rust prevents data races at compile time: we'll get errors if we break the rules.

With this in mind, let's consider our example again.

Thinking in scopes

Here's the code:

```
let mut x = 5;
let y = &mut x;

*y += 1;

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

This code gives us this error:

```
error: cannot borrow `x` as immutable because it is also borrowed as mutable
    println!("{}", x);
                   ^
```

This is because we've violated the rules: we have a `&mut T` pointing to `x`, and so we aren't allowed to create any `&T`s. One or the other. The note hints at how to think about this problem:

```
note: previous borrow ends here
fn main() {

}
^
```

In other words, the mutable borrow is held through the rest of our example. What we want is for the mutable borrow to end before we try to call `println!` and make an immutable borrow. In Rust, borrowing is tied to the scope that the borrow is valid for. And our scopes look like this:

```
let mut x = 5;

let y = &mut x;    // -+ &mut borrow of x starts here
                   //  |
*y += 1;           //  |
                   //  |
println!("{}", x); // -+ - try to borrow x here
                   // -+ &mut borrow of x ends here
```

The scopes conflict: we can't make an `&x` while `y` is in scope.

So when we add the curly braces:

```
let mut x = 5;

{
    let y = &mut x; // -+ &mut borrow starts here
    *y += 1;        //  |
}                   // -+ ... and ends here

println!("{}", x);  // <- try to borrow x here
```

There's no problem. Our mutable borrow goes out of scope before we create an immutable one. But scope is the key to seeing how long a borrow lasts for.

Issues borrowing prevents

Why have these restrictive rules? Well, as we noted, these rules prevent data races. What kinds of issues do data races cause? Here's a few.

Iterator invalidation

One example is 'iterator invalidation', which happens when you try to mutate a collection that you're iterating over. Rust's borrow checker prevents this from happening:

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

for i in &v {
    println!("{}", i);
}
```

This prints out one through three. As we iterate through the vector, we're only given references to the elements. And `v` is itself borrowed as immutable, which means we can't change it while we're iterating:

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

for i in &v {
    println!("{}", i);
    v.push(34);
}
```

Here's the error:

```
error: cannot borrow `v` as mutable because it is also borrowed as immutable
    v.push(34);
    ^
note: previous borrow of `v` occurs here; the immutable borrow prevents
subsequent moves or mutable borrows of `v` until the borrow ends
for i in &v {
          ^
note: previous borrow ends here
for i in &v {
    println!( "{}" , i);
    v.push(34);
}
^
```

We can't modify `v` because it's borrowed by the loop.

use after free

References must not live longer than the resource they refer to. Rust will check the scopes of your references to ensure that this is true.

If Rust didn't check this property, we could accidentally use a reference which was invalid. For example:

```
let y: &i32;
{
    let x = 5;
    y = &x;
}

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

We get this error:

```
error: `x` does not live long enough
    y = &x;
         ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
let y: &i32;
{
    let x = 5;
    y = &x;
}

note: ...but borrowed value is only valid for the block suffix following
statement 0 at 4:18
    let x = 5;
    y = &x;
}
```

In other words, y is only valid for the scope where x exists. As soon as x goes away, it becomes invalid to refer to it. As such, the error says that the borrow 'doesn't live long enough' because it's not valid for the right amount of time.

The same problem occurs when the reference is declared before the variable it refers to. This is because resources within the same scope are freed in the opposite order they were declared:

```
let y: &i32;
let x = 5;
y = &x;

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

We get this error:

```
error: `x` does not live long enough
y = &x;
     ^
note: reference must be valid for the block suffix following statement 0 at
2:16...
    let y: &i32;
    let x = 5;
    y = &x;

    println!("{}", y);
}

note: ...but borrowed value is only valid for the block suffix following
statement 1 at 3:14
    let x = 5;
    y = &x;

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

In the above example, `y` is declared before `x`, meaning that `y` lives longer than `x`, which is not allowed.

4.9 Lifetimes

This guide is one of three presenting Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. There are a few distinct concepts, each with its own chapter:

- ownership, which you're reading now
- borrowing, and their associated feature 'references'
- lifetimes, an advanced concept of borrowing

These three chapters are related, and in order. You'll need all three to fully understand the ownership system.

Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many 'zero-cost abstractions', which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero-cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call 'fighting with the borrow checker', where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about lifetimes.

Lifetimes

Lending out a reference to a resource that someone else owns can be complicated. For example, imagine this set of operations:

1. I acquire a handle to some kind of resource.
2. I lend you a reference to the resource.
3. I decide I'm done with the resource, and deallocate it, while you still have your reference.
4. You decide to use the resource.

Uh oh! Your reference is pointing to an invalid resource. This is called a dangling pointer or 'use after free', when the resource is memory.

To fix this, we have to make sure that step four never happens after step three. The ownership system in Rust does this through a concept called lifetimes, which describe the scope that a reference is valid for.

When we have a function that takes a reference by argument, we can be implicit or explicit about the lifetime of the reference:

```
// implicit
fn foo(x: &i32) {
}

// explicit
fn bar<'a>(x: &'a i32) {
}
```

The `'a` reads 'the lifetime a'. Technically, every reference has some lifetime associated with it, but the compiler lets you elide (i.e. omit, see "Lifetime Elision" below) them in common cases. Before we get to that, though, let's break the explicit example down:

```
fn bar<'a>(...)
```

We previously talked a little about function syntax (Functions), but we didn't discuss the `<>`s after a function's name. A function can have 'generic parameters' between the `<>`s, of which lifetimes are one kind. We'll discuss other kinds of generics later in the book, but for now, let's focus on the lifetimes aspect.

We use `<>` to declare our lifetimes. This says that bar has one lifetime, `'a`. If we had two reference parameters, it would look like this:

```
fn bar<'a, 'b>(...)
```

Then in our parameter list, we use the lifetimes we've named:

```
...(x: &'a i32)
```

If we wanted a `&mut` reference, we'd do this:

```
...(x: &'a mut i32)
```

If you compare `&mut i32` to `&'a mut i32`, they're the same, it's that the lifetime `'a` has snuck in between the `&` and the `mut i32`. We read `&mut i32` as 'a mutable reference to an `i32`' and `&'a mut i32` as 'a mutable reference to an `i32` with the lifetime `'a`'.

In `structs`

You'll also need explicit lifetimes when working with structs that contain references:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5; // this is the same as `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

As you can see, `struct`s can also have lifetimes. In a similar way to functions,

```
struct Foo<'a> {
```

declares a lifetime, and

```
x: &'a i32,
```

uses it. So why do we need a lifetime here? We need to ensure that any reference to a `Foo` cannot outlive the reference to an `i32` it contains.

`impl` blocks

Let's implement a method on `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

impl<'a> Foo<'a> {
    fn x(&self) -> &'a i32 { self.x }
}

fn main() {
    let y = &5; // this is the same as `let _y = 5; let y = &_y;`
    let f = Foo { x: y };

    println!("x is: {}", f.x());
}
```

As you can see, we need to declare a lifetime for `Foo` in the `impl` line. We repeat `'a` twice, like on functions: `impl<'a>` defines a lifetime `'a`, and `Foo<'a>` uses it.

Multiple lifetimes

If you have multiple references, you can use the same lifetime multiple times:

```
fn x_or_y<'a>(x: &'a str, y: &'a str) -> &'a str {
```

This says that `x` and `y` both are alive for the same scope, and that the return value is also alive for that scope. If you wanted `x` and `y` to have different lifetimes, you can use multiple lifetime parameters:

```
fn x_or_y<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
```

In this example, `x` and `y` have different valid scopes, but the return value has the same lifetime as `x`.

Thinking in scopes

A way to think about lifetimes is to visualize the scope that a reference is valid for. For example:

```
fn main() {
    let y = &5;     // -+ y goes into scope
                    //  |
    // stuff        //  |
                    //  |
}                   // -+ y goes out of scope
```

Adding in our `Foo`:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let y = &5;             // -+ y goes into scope
    let f = Foo { x: y }; // -+ f goes into scope
    // stuff                //  |
                            //  |
}                           // -+ f and y go out of scope
```

Our `f` lives within the scope of `y`, so everything works. What if it didn't? This code won't work:

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                    // -+ x goes into scope
                              //  |
    {                         //  |
        let y = &5;           // ---+ y goes into scope
        let f = Foo { x: y }; // ---+ f goes into scope
        x = &f.x;             //  | | error here
    }                         // ---+ f and y go out of scope
                              //  |
    println!("{}", x);        //  |
}                             // -+ x goes out of scope
```

Whew! As you can see here, the scopes of `f` and `y` are smaller than the scope of `x`. But when we do `x = &f.x`, we make `x` a reference to something that's about to go out of scope.

Named lifetimes are a way of giving these scopes a name. Giving something a name is the first step towards being able to talk about it.

'static

The lifetime named 'static' is a special lifetime. It signals that something has the lifetime of the entire program. Most Rust programmers first come across `'static` when dealing with strings:

```
let x: &'static str = "Hello, world.";
```

String literals have the type `&'static str` because the reference is always alive: they are baked into the data segment of the final binary. Another example are globals:

```
static FOO: i32 = 5;
let x: &'static i32 = &FOO;
```

This adds an `i32` to the data segment of the binary, and `x` is a reference to it.

Lifetime Elision

Rust supports powerful local type inference in function bodies, but it's forbidden in item signatures to allow reasoning about the types based on the item signature alone. However, for ergonomic reasons a very restricted secondary inference algorithm called "lifetime elision" applies in function signatures. It infers only based on the signature components themselves and not based on the body of the function, only infers lifetime parameters, and does this with only three easily memorizable and unambiguous rules. This makes lifetime elision a shorthand for writing an item signature, while not hiding away the actual types involved as full local inference would if applied to it.

When talking about lifetime elision, we use the term *input lifetime* and *output lifetime*. An *input lifetime* is a lifetime associated with a parameter of a function, and an *output lifetime* is a lifetime associated with the return value of a function. For example, this function has an input lifetime:

```
fn foo<'a>(bar: &'a str)
```

This one has an output lifetime:

```
fn foo<'a>() -> &'a str
```

This one has a lifetime in both positions:

```
fn foo<'a>(bar: &'a str) -> &'a str
```

Here are the three rules:

- Each elided lifetime in a function's arguments becomes a distinct lifetime parameter.
- If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.
- If there are multiple input lifetimes, but one of them is `&self` or `&mut self`, the lifetime of self is assigned to all elided output lifetimes.

Otherwise, it is an error to elide an output lifetime.

Examples

Here are some examples of functions with elided lifetimes. We've paired each example of an elided lifetime with its expanded form.

```rust
fn print(s: &str); // elided
fn print<'a>(s: &'a str); // expanded

fn debug(lvl: u32, s: &str); // elided
fn debug<'a>(lvl: u32, s: &'a str); // expanded

// In the preceding example, `lvl` doesn't need a lifetime because it's not
↪  a
// reference (`&`). Only things relating to references (such as a `struct`
// which contains a reference) need lifetimes.

fn substr(s: &str, until: u32) -> &str; // elided
fn substr<'a>(s: &'a str, until: u32) -> &'a str; // expanded

fn get_str() -> &str; // ILLEGAL, no inputs

fn frob(s: &str, t: &str) -> &str; // ILLEGAL, two inputs
fn frob<'a, 'b>(s: &'a str, t: &'b str) -> &str; // Expanded: Output
↪  lifetime is ambiguous

fn get_mut(&mut self) -> &mut T; // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T; // expanded

fn args<T: ToCStr>(&mut self, args: &[T]) -> &mut Command; // elided
fn args<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command;
↪  // expanded

fn new(buf: &mut [u8]) -> BufWriter; // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>; // expanded
```

4.10 Mutability

Mutability, the ability to change something, works a bit differently in Rust than in other languages. The first aspect of mutability is its non-default status:

```rust
let x = 5;
x = 6; // error!
```

We can introduce mutability with the `mut` keyword:

```rust
let mut x = 5;

x = 6; // no problem!
```

This is a mutable Variable Bindings. When a binding is mutable, it means you're allowed to change what the binding points to. So in the above example, it's not so much that the value at `x` is changing, but that the binding changed from one `i32` to another.

If you want to change what the binding points to, you'll need a mutable reference (see References and Borrowing):

```
let mut x = 5;
let y = &mut x;
```

`y` is an immutable binding to a mutable reference, which means that you can't bind `y` to something else (`y = &mut z`), but you can mutate the thing that's bound to `y` (`*y = 5`). A subtle distinction.

Of course, if you need both:

```
let mut x = 5;
let mut y = &mut x;
```

Now `y` can be bound to another value, and the value it's referencing can be changed.

It's important to note that `mut` is part of a pattern, so you can do things like this:

```
let (mut x, y) = (5, 6);

fn foo(mut x: i32) {
```

Interior vs. Exterior Mutability

However, when we say something is 'immutable' in Rust, that doesn't mean that it's not able to be changed: we mean something has 'exterior mutability'. Consider, for example, Arc<T>:

```
use std::sync::Arc;

let x = Arc::new(5);
let y = x.clone();
```

When we call `clone()`, the `Arc<T>` needs to update the reference count. Yet we've not used any muts here, `x` is an immutable binding, and we didn't take `&mut 5` or anything. So what gives?

To understand this, we have to go back to the core of Rust's guiding philosophy, memory safety, and the mechanism by which Rust guarantees it, the ownership system (see Ownership), and more specifically, borrowing (see References and Borrowing):

> You may have one or the other of these two kinds of borrows, but not both at the same time:
>
> - one or more references (`&T`) to a resource,
> - exactly one mutable reference (`&mut T`).

So, that's the real definition of 'immutability': is this safe to have two pointers to? In `Arc<T>`'s case, yes: the mutation is entirely contained inside the structure itself. It's not user facing. For this reason, it hands out `&T` with `clone()`. If it handed out `&mut T`s, though, that would be a problem.

Other types, like the ones in the std::cell module, have the opposite: interior mutability. For example:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
```

RefCell hands out `&mut` references to what's inside of it with the `borrow_mut()` method. Isn't that dangerous? What if we do:

```
use std::cell::RefCell;

let x = RefCell::new(42);

let y = x.borrow_mut();
let z = x.borrow_mut();
```

This will in fact panic, at runtime. This is what `RefCell` does: it enforces Rust's borrowing rules at runtime, and `panic!`s if they're violated. This allows us to get around another aspect of Rust's mutability rules. Let's talk about it first.

Field-level mutability

Mutability is a property of either a borrow (`&mut`) or a binding (`let mut`). This means that, for example, you cannot have a struct with some fields mutable and some immutable:

```
struct Point {
    x: i32,
    mut y: i32, // nope
}
```

The mutability of a struct is in its binding:

```
struct Point {
    x: i32,
    y: i32,
}

let mut a = Point { x: 5, y: 6 };

a.x = 10;

let b = Point { x: 5, y: 6};

b.x = 10; // error: cannot assign to immutable field `b.x`
```

However, by using Cell<T>, you can emulate field-level mutability:

```
use std::cell::Cell;

struct Point {
    x: i32,
    y: Cell<i32>,
}

let point = Point { x: 5, y: Cell::new(6) };

point.y.set(7);

println!("y: {:?}", point.y);
```

This will print `y: Cell  value: 7 `. We've successfully updated `y`.

4.11 Structs

`struct`s are a way of creating more complex data types. For example, if we were doing calculations involving coordinates in 2D space, we would need both an `x` and a `y` value:

```
let origin_x = 0;
let origin_y = 0;
```

A `struct` lets us combine these two into a single, unified datatype with `x` and `y` as field labels:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

There's a lot going on here, so let's break it down. We declare a `struct` with the `struct` keyword, and then with a name. By convention, `struct`s begin with a capital letter and are camel cased: `PointInSpace`, not `Point_In_Space`.

We can create an instance of our `struct` via `let`, as usual, but we use a `key: value` style syntax to set each field. The order doesn't need to be the same as in the original declaration.

Finally, because fields have names, we can access them through dot notation: `origin.x`.

The values in `struct`s are immutable by default, like other bindings in Rust. Use `mut` to make them mutable:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    println!("The point is at ({}, {})", point.x, point.y);
}
```

This will print `The point is at (5, 0)`.

Rust does not support field mutability at the language level, so you cannot write something like this:

```
struct Point {
    mut x: i32,
    y: i32,
}
```

Mutability is a property of the binding, not of the structure itself. If you're used to field-level mutability, this may seem strange at first, but it significantly simplifies things. It even lets you make things mutable on a temporary basis:

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    let point = point; // now immutable

    point.y = 6; // this causes an error
}
```

Your structure can still contain `&mut` pointers, which will let you do some kinds of mutation:

```
struct Point {
    x: i32,
    y: i32,
}

struct PointRef<'a> {
    x: &'a mut i32,
    y: &'a mut i32,
}

fn main() {
    let mut point = Point { x: 0, y: 0 };

    {
        let r = PointRef { x: &mut point.x, y: &mut point.y };

        *r.x = 5;
        *r.y = 6;
    }

    assert_eq!(5, point.x);
    assert_eq!(6, point.y);
}
```

Update syntax

A `struct` can include `..` to indicate that you want to use a copy of some other `struct` for some of the values. For example:

```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut point = Point3d { x: 0, y: 0, z: 0 };
point = Point3d { y: 1, .. point };
```

This gives `point` a new `y`, but keeps the old `x` and `z` values. It doesn't have to be the same `struct` either, you can use this syntax when making new ones, and it will copy the values you don't specify:

```
let origin = Point3d { x: 0, y: 0, z: 0 };
let point = Point3d { z: 1, x: 2, .. origin };
```

Tuple structs

Rust has another data type that's like a hybrid between a tuple and a `struct`, called a 'tuple struct'. Tuple structs have a name, but their fields don't. They are declared with the `struct` keyword, and then with a name followed by a tuple:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Here, `black` and `origin` are not equal, even though they contain the same values.

It is almost always better to use a `struct` than a tuple struct. We would write `Color` and `Point` like this instead:

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

struct Point {
    x: i32,
    y: i32,
    z: i32,
}
```

Good names are important, and while values in a tuple struct can be referenced with dot notation as well, a `struct` gives us actual names, rather than positions.

There is one case when a tuple struct is very useful, though, and that is when it has only one element. We call this the ’newtype’ pattern, because it allows you to create a new type that is distinct from its contained value and also expresses its own semantic meaning:

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

As you can see here, you can extract the inner integer type through a destructuring `let`, as with regular tuples. In this case, the `let Inches(integer_length)` assigns `10` to `integer_length`.

Unit-like structs

You can define a `struct` with no members at all:

```
struct Electron;

let x = Electron;
```

Such a `struct` is called 'unit-like' because it resembles the empty tuple, `()`, sometimes called 'unit'. Like a tuple struct, it defines a new type.

This is rarely useful on its own (although sometimes it can serve as a marker type), but in combination with other features, it can become useful. For instance, a library may ask you to create a structure that implements a certain trait to handle events. If you don't have any data you need to store in the structure, you can create a unit-like `struct`.

4.12 Enums

An `enum` in Rust is a type that represents data that is one of several possible variants. Each variant in the `enum` can optionally have data associated with it:

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}
```

The syntax for defining variants resembles the syntaxes used to define structs: you can have variants with no data (like unit-like structs), variants with named data, and variants with unnamed data (like tuple structs). Unlike separate struct definitions, however, an `enum` is a single type. A value of the enum can match any of the variants. For this reason, an enum is sometimes called a 'sum type': the set of possible values of the enum is the sum of the sets of possible values for each variant.

We use the `::` syntax to use the name of each variant: they're scoped by the name of the `enum` itself. This allows both of these to work:

```
let x: Message = Message::Move { x: 3, y: 4 };

enum BoardGameTurn {
    Move { squares: i32 },
    Pass,
}

let y: BoardGameTurn = BoardGameTurn::Move { squares: 1 };
```

Both variants are named `Move`, but since they're scoped to the name of the `enum` they can both be used without conflict.

A value of an `enum` type contains information about which variant it is, in addition to any data associated with that variant. This is sometimes referred to as a 'tagged union', since the data includes a 'tag' indicating what type it is. The compiler uses this information to enforce that you're accessing the data in the enum safely. For instance, you can't simply try to destructure a value as if it were one of the possible variants:

```
fn process_color_change(msg: Message) {
    let Message::ChangeColor(r, g, b) = msg; // compile-time error
}
```

Not supporting these operations may seem rather limiting, but it's a limitation which we can overcome. There are two ways: by implementing equality ourselves, or by pattern matching variants with match expressions, which you'll learn in the next section. We don't know enough about Rust to implement equality yet, but we'll find out in the traits section.

Constructors as functions

An `enum` constructor can also be used like a function. For example:

```
let m = Message::Write("Hello, world".to_string());
```

is the same as

```
fn foo(x: String) -> Message {
    Message::Write(x)
}

let x = foo("Hello, world".to_string());
```

This is not immediately useful to us, but when we get to closures, we'll talk about passing functions as arguments to other functions. For example, with iterators, we can do this to convert a vector of `String`s into a vector of `Message::Writes`:

```
let v = vec!["Hello".to_string(), "World".to_string()];

let v1: Vec<Message> = v.into_iter().map(Message::Write).collect();
```

4.13 Match

Often, a simple `if/else` (see If) isn't enough, because you have more than two possible options. Also, conditions can get quite complex. Rust has a keyword, `match`, that allows you to replace complicated `if/else` groupings with something more powerful. Check it out:

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

match takes an expression and then branches based on its value. Each 'arm' of the branch is of the form val => expression. When the value matches, that arm's expression will be evaluated. It's called match because of the term 'pattern matching', which match is an implementation of. There's a separate section on patterns that covers all the patterns that are possible here.

One of the many advantages of match is it enforces 'exhaustiveness checking'. For example if we remove the last arm with the underscore _, the compiler will give us an error:

```
error: non-exhaustive patterns: `_` not covered
```

Rust is telling us that we forgot a value. The compiler infers from x that it can have any positive 32bit value; for example 1 to 2,147,483,647. The _ acts as a 'catch-all', and will catch all possible values that aren't specified in an arm of match. As you can see with the previous example, we provide match arms for integers 1-5, if x is 6 or any other value, then it is caught by _.

match is also an expression, which means we can use it on the right-hand side of a let binding or directly where an expression is used:

```
let x = 5;

let number = match x {
    1 => "one",
    2 => "two",
    3 => "three",
    4 => "four",
    5 => "five",
    _ => "something else",
};
```

Sometimes it's a nice way of converting something from one type to another; in this example the integers are converted to String.

Matching on enums

Another important use of the match keyword is to process the possible variants of an enum:

```
enum Message {
    Quit,
    ChangeColor(i32, i32, i32),
    Move { x: i32, y: i32 },
    Write(String),
}

fn quit() { /* ... */ }
fn change_color(r: i32, g: i32, b: i32) { /* ... */ }
fn move_cursor(x: i32, y: i32) { /* ... */ }

fn process_message(msg: Message) {
    match msg {
        Message::Quit => quit(),
        Message::ChangeColor(r, g, b) => change_color(r, g, b),
        Message::Move { x: x, y: y } => move_cursor(x, y),
        Message::Write(s) => println!("{}", s),
    };
}
```

Again, the Rust compiler checks exhaustiveness, so it demands that you have a match arm for every variant of the enum. If you leave one off, it will give you a compile-time error unless you use `_` or provide all possible arms.

Unlike the previous uses of `match`, you can't use the normal `if` statement to do this. You can use the `if let` (see) statement, which can be seen as an abbreviated form of `match`.

4.14 Patterns

Patterns are quite common in Rust. We use them in Variable Bindings, match statements (see Match), and other places, too. Let's go on a whirlwind tour of all of the things patterns can do!

A quick refresher: you can match against literals directly, and `_` acts as an 'any' case:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This prints `one`.

There's one pitfall with patterns: like anything that introduces a new binding, they introduce shadowing. For example:

```
let x = 1;
let c = 'c';

match c {
    x => println!("x: {} c: {}", x, c),
}

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

This prints:

```
x: c c: c
x: 1
```

In other words, `x =>` matches the pattern and introduces a new binding named `x`. This new binding is in scope for the match arm and takes on the value of `c`. Notice that the value of `x` outside the scope of the match has no bearing on the value of `x` within it. Because we already have a binding named `x` this new `x` shadows it.

Multiple patterns

You can match multiple patterns with `|`:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This prints `one or two`.

Destructuring

If you have a compound data type, like a `struct` (see Structs), you can destructure it inside of a pattern:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, y } => println!("({},{})", x, y),
}
```

We can use `:` to give a value a different name.

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x: x1, y: y1 } => println!("({},{})", x1, y1),
}
```

If we only care about some of the values, we don't have to give them all names:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, .. } => println!("x is {}", x),
}
```

This prints `x is 0`.
You can do this kind of match on any member, not only the first:

```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { y, .. } => println!("y is {}", y),
}
```

This prints `y is 0`.

This 'destructuring' behavior works on any compound data type, like tuples or enums.

Ignoring bindings

You can use `_` in a pattern to disregard the type and value. For example, here's a `match` against a `Result<T, E>`:

```
match some_value {
    Ok(value) => println!("got a value: {}", value),
    Err(_) => println!("an error occurred"),
}
```

In the first arm, we bind the value inside the `Ok` variant to value. But in the `Err` arm, we use `_` to disregard the specific error, and print a general error message.

`_` is valid in any pattern that creates a binding. This can be useful to ignore parts of a larger structure:

```
fn coordinate() -> (i32, i32, i32) {
    // generate and return some sort of triple tuple
}

let (x, _, z) = coordinate();
```

Here, we bind the first and last element of the tuple to `x` and `z`, but ignore the middle element.

Similarly, you can use `..` in a pattern to disregard multiple values.

```
enum OptionalTuple {
    Value(i32, i32, i32),
    Missing,
}

let x = OptionalTuple::Value(5, -2, 3);

match x {
    OptionalTuple::Value(..) => println!("Got a tuple!"),
    OptionalTuple::Missing => println!("No such luck."),
}
```

This prints `Got a tuple!`.

ref and ref mut

If you want to get a reference (see References and Borrowing), use the `ref` keyword:

```
let x = 5;

match x {
    ref r => println!("Got a reference to {}", r),
}
```

This prints `Got a reference to 5`.

Here, the `r` inside the `match` has the type `&i32`. In other words, the `ref` keyword creates a reference, for use in the pattern. If you need a mutable reference, `refmut` will work in the same way:

```
let mut x = 5;

match x {
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

Ranges

You can match a range of values with `...`:

```
let x = 1;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("anything"),
}
```

This prints `one through five`.

Ranges are mostly used with integers and `char`s:

```
let x = 'ä';

match x {
    'a' ... 'j' => println!("early letter"),
    'k' ... 'z' => println!("late letter"),
    _ => println!("something else"),
}
```

This prints `something else`.

Bindings

You can bind values to names with `@`:

```
let x = 1;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

This prints `got a range element 1`. This is useful when you want to do a complicated match of part of a data structure:

```
#[derive(Debug)]
struct Person {
    name: Option<String>,
}

let name = "Steve".to_string();
let mut x: Option<Person> = Some(Person { name: Some(name) });
match x {
    Some(Person { name: ref a @ Some(_), .. }) => println!("{:?}", a),
    _ => {}
}
```

This prints `Some("Steve")`: we've bound the inner `name` to `a`.

If you use `@` with `|`, you need to make sure the name is bound in each part of the pattern:

```
let x = 5;

match x {
    e @ 1 ... 5 | e @ 8 ... 10 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

Guards

You can introduce 'match guards' with `if`:

```
enum OptionalInt {
    Value(i32),
    Missing,
}

let x = OptionalInt::Value(5);

match x {
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than
↪   five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing => println!("No such luck."),
}
```

This prints `Got an int!`.

If you're using `if` with multiple patterns, the `if` applies to both sides:

```
let x = 4;
let y = false;

match x {
    4 | 5 if y => println!("yes"),
    _ => println!("no"),
}
```

This prints `no`, because the `if` applies to the whole of `4 | 5`, and not to only the `5`. In other words, the precedence of `if` behaves like this:

```
(4 | 5) if y => ...
```

not this:

```
4 | (5 if y) => ...
```

Mix and Match

Whew! That's a lot of different ways to match things, and they can all be mixed and matched, depending on what you're doing:

```
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

Patterns are very powerful. Make good use of them.

4.15 Method Syntax

Functions are great, but if you want to call a bunch of them on some data, it can be awkward. Consider this code:

```
baz(bar(foo));
```

We would read this left-to-right, and so we see 'baz bar foo'. But this isn't the order that the functions would get called in, that's inside-out: 'foo bar baz'. Wouldn't it be nice if we could do this instead?

```
foo.bar().baz();
```

Luckily, as you may have guessed with the leading question, you can! Rust provides the ability to use this 'method call syntax' via the `impl` keyword.

Method calls

Here's how it works:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());
}
```

This will print `12.566371`.

We've made a `struct` that represents a circle. We then write an `impl` block, and inside it, define a method, `area`.

Methods take a special first parameter, of which there are three variants: `self`, `&self`, and `&mut self`. You can think of this first parameter as being the `foo` in `foo.bar()`. The three variants correspond to the three kinds of things `foo` could be: `self` if it's a value on the stack, `&self` if it's a reference, and `&mut self` if it's a mutable reference. Because we took the `&self` parameter to `area`, we can use it like any other parameter. Because we know it's a `Circle`, we can access the `radius` like we would with any other `struct`.

We should default to using `&self`, as you should prefer borrowing over taking ownership, as well as taking immutable references over mutable ones. Here's an example of all three variants:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!("taking self by reference!");
    }

    fn mutable_reference(&mut self) {
       println!("taking self by mutable reference!");
    }

    fn takes_ownership(self) {
       println!("taking ownership of self!");
    }
}
```

You can use as many `impl` blocks as you'd like. The previous example could have also been written like this:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn reference(&self) {
       println!("taking self by reference!");
    }
}

impl Circle {
    fn mutable_reference(&mut self) {
       println!("taking self by mutable reference!");
    }
}

impl Circle {
    fn takes_ownership(self) {
       println!("taking ownership of self!");
    }
}
```

Chaining method calls

So, now we know how to call a method, such as `foo.bar()`. But what about our original example, `foo.bar().baz()`? This is called 'method chaining'. Let's look at an example:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }

    fn grow(&self, increment: f64) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius + increment }
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());

    let d = c.grow(2.0).area();
    println!("{}", d);
}
```

Check the return type:

```
fn grow(&self, increment: f64) -> Circle {
```

We say we're returning a `Circle`. With this method, we can grow a new `Circle` to any arbitrary size.

Associated functions

You can also define associated functions that do not take a `self` parameter. Here's a pattern that's very common in Rust code:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
}
```

This 'associated function' builds a new `Circle` for us. Note that associated functions are called with the `Struct::function()` syntax, rather than the `ref.method()` syntax. Some other languages call associated functions 'static methods'.

Builder Pattern

Let's say that we want our users to be able to create `Circles`, but we will allow them to only set the properties they care about. Otherwise, the `x` and `y` attributes will be `0.0`, and the `radius` will be `1.0`. Rust doesn't have method overloading, named arguments, or variable arguments. We employ the builder pattern instead. It looks like this:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct CircleBuilder {
    x: f64,
    y: f64,
    radius: f64,
}

impl CircleBuilder {
    fn new() -> CircleBuilder {
        CircleBuilder { x: 0.0, y: 0.0, radius: 1.0, }
    }

    fn x(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.x = coordinate;
        self
    }

    fn y(&mut self, coordinate: f64) -> &mut CircleBuilder {
        self.y = coordinate;
        self
    }

    fn radius(&mut self, radius: f64) -> &mut CircleBuilder {
        self.radius = radius;
        self
    }

    fn finalize(&self) -> Circle {
        Circle { x: self.x, y: self.y, radius: self.radius }
    }
}

fn main() {
    let c = CircleBuilder::new()
                .x(1.0)
                .y(2.0)
                .radius(2.0)
                .finalize();

    println!("area: {}", c.area());
    println!("x: {}", c.x);
    println!("y: {}", c.y);
}
```

What we've done here is make another `struct`, `CircleBuilder`. We've defined our builder methods on it. We've also defined our `area()` method on `Circle`. We also made one more method on `CircleBuilder`: `finalize()`. This method creates our final `Circle` from the builder. Now, we've used the type system to enforce our concerns: we can use the methods on `CircleBuilder` to constrain making `Circles` in any way we choose.

4.16 Vectors

A 'vector' is a dynamic or 'growable' array, implemented as the standard library type Vec<T>. The `T` means that we can have vectors of any type (see the chapter on generics for more). Vectors always allocate their data on the heap. You can create them with the `vec!` macro:

```
let v = vec![1, 2, 3, 4, 5]; // v: Vec<i32>
```

(Notice that unlike the `println!` macro we've used in the past, we use square brackets `[]` with `vec!` macro. Rust allows you to use either in either situation, this is just convention.)

There's an alternate form of `vec!` for repeating an initial value:

```
let v = vec![0; 10]; // ten zeroes
```

Accessing elements

To get the value at a particular index in the vector, we use `[]`s:

```
let v = vec![1, 2, 3, 4, 5];

println!("The third element of v is {}", v[2]);
```

The indices count from `0`, so the third element is `v[2]`.

It's also important to note that you must index with the `usize` type:

```
let v = vec![1, 2, 3, 4, 5];

let i: usize = 0;
let j: i32 = 0;

// works
v[i];

// doesn't
v[j];
```

Indexing with a non-`usize` type gives an error that looks like this:

```
error: the trait `core::ops::Index<i32>` is not implemented for the type
`collections::vec::Vec<_>` [E0277]
v[j];
^~~~
note: the type `collections::vec::Vec<_>` cannot be indexed by `i32`
error: aborting due to previous error
```

There's a lot of punctuation in that message, but the core of it makes sense: you cannot index with an `i32`.

Out-of-bounds Access

If you try to access an index that doesn't exist:

```
let v = vec![1, 2, 3];
println!("Item 7 is {}", v[7]);
```

then the current thread will panic with a message like this:

```
thread '<main>' panicked at 'index out of bounds: the len is 3 but the index is 7'
```

If you want to handle out-of-bounds errors without panicking, you can use methods like get or get_mut that return `None` when given an invalid index:

```
let v = vec![1, 2, 3];
match v.get(7) {
    Some(x) => println!("Item 7 is {}", x),
    None => println!("Sorry, this vector is too short.")
}
```

Iterating

Once you have a vector, you can iterate through its elements with `for`. There are three versions:

```
let mut v = vec![1, 2, 3, 4, 5];

for i in &v {
    println!("A reference to {}", i);
}

for i in &mut v {
    println!("A mutable reference to {}", i);
}

for i in v {
    println!("Take ownership of the vector and its element {}", i);
}
```

Vectors have many more useful methods, which you can read about in their API documentation.

4.17 Strings

Strings are an important concept for any programmer to master. Rust's string handling system is a bit different from other languages, due to its systems focus. Any time you have a data structure of variable size, things can get tricky, and strings are a re-sizable data structure. That being said, Rust's strings also work differently than in some other systems languages, such as C.

Let's dig into the details. A 'string' is a sequence of Unicode scalar values encoded as a stream of UTF-8 bytes. All strings are guaranteed to be a valid encoding of UTF-8 sequences. Additionally, unlike some systems languages, strings are not null-terminated and can contain null bytes.

Rust has two main types of strings: `&str` and `String`. Let's talk about `&str` first. These are called 'string slices'. A string slice has a fixed size, and cannot be mutated. It is a reference to a sequence of UTF-8 bytes.

```
let greeting = "Hello there."; // greeting: &'static str
```

`"Hello there."` is a string literal and its type is `&'static str`. A string literal is a string slice that is statically allocated, meaning that it's saved inside our compiled program, and exists for the entire duration it runs. The `greeting` binding is a reference to this statically allocated string. Any function expecting a string slice will also accept a string literal.

String literals can span multiple lines. There are two forms. The first will include the newline and the leading spaces:

```
let s = "foo
    bar";

assert_eq!("foo\n        bar", s);
```

The second, with a
, trims the spaces and the newline:

```
let s = "foo\
    bar";

assert_eq!("foobar", s);
```

Rust has more than only `&str`s though. A `String`, is a heap-allocated string. This string is growable, and is also guaranteed to be UTF-8. `String`s are commonly created by converting from a string slice using the `to_string` method.

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

`String`s will coerce into `&str` with an &:

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(&s);
}
```

This coercion does not happen for functions that accept one of `&str`'s traits instead of `&str`. For example, TcpStream::connect has a parameter of type `ToSocketAddrs`. A `&str` is okay but a `String` must be explicitly converted using `&*`.

```
use std::net::TcpStream;

TcpStream::connect("192.168.0.1:3000"); // &str parameter

let addr_string = "192.168.0.1:3000".to_string();
TcpStream::connect(&*addr_string); // convert addr_string to &str
```

Viewing a `String` as a `&str` is cheap, but converting the `&str` to a `String` involves allocating memory. No reason to do that unless you have to!

Indexing

Because strings are valid UTF-8, strings do not support indexing:

```
let s = "hello";

println!("The first letter of s is {}", s[0]); // ERROR!!!
```

Usually, access to a vector with `[]` is very fast. But, because each character in a UTF-8 encoded string can be multiple bytes, you have to walk over the string to find the n_{th} letter of a string. This is a significantly more expensive operation, and we don't want to be misleading. Furthermore, 'letter' isn't something defined in Unicode, exactly. We can choose to look at a string as individual bytes, or as codepoints:

```
let hachiko = " 忠犬ハチ公";

for b in hachiko.as_bytes() {
    print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
    print!("{}, ", c);
}

println!("");
```

```
let hachiko = " 忠犬ハチ公";

for b in hachiko.as_bytes() {
    print!("{}, ", b);
}

println!("");

for c in hachiko.chars() {
    print!("{}, ", c);
}

println!("");
```

This prints:

```
229, 191, 160, 231, 138, 172, 227, 131, 143, 227, 131, 129, 229, 133, 172,
忠, 犬, ハ, チ, 公,
```

As you can see, there are more bytes than `char`s.

You can get something similar to an index like this:

```
let dog = hachiko.chars().nth(1); // kinda like hachiko[1]
```

This emphasizes that we have to walk from the beginning of the list of `char`s.

Slicing

You can get a slice of a string with slicing syntax:

```
let dog = "hachiko";
let hachi = &dog[0..5];
```

But note that these are *byte offsets*, not *character offsets*. So this will fail at runtime:

```
let dog = " 忠犬ハチ公";
let hachi = &dog[0..2];
```

with this error:

```
thread '<main>' panicked at 'index 0 and/or 2 in `忠犬ハチ公` do not lie on
character boundary'
```

Concatenation

If you have a `String`, you can concatenate a `&str` to the end of it:

```
let hello = "Hello ".to_string();
let world = "world!";

let hello_world = hello + world;
```

But if you have two `String`s, you need an `&`:

```
let hello = "Hello ".to_string();
let world = "world!".to_string();

let hello_world = hello + &world;
```

This is because `&String` can automatically coerce to a `&str`. This is a feature called 'Deref coercions'.

4.18 Generics

Sometimes, when writing a function or data type, we may want it to work for multiple types of arguments. In Rust, we can do this with generics. Generics are called 'parametric polymorphism' in type theory, which means that they are types or functions that have multiple forms ('poly' is multiple, 'morph' is form) over a given parameter ('parametric').

Anyway, enough type theory, let's check out some generic code. Rust's standard library provides a type, `Option<T>`, that's generic:

```
enum Option<T> {
    Some(T),
    None,
}
```

The `<T>` part, which you've seen a few times before, indicates that this is a generic data type. Inside the declaration of our `enum`, wherever we see a `T`, we substitute that type for the same type used in the generic. Here's an example of using `Option<T>`, with some extra type annotations:

```
let x: Option<i32> = Some(5);
```

In the type declaration, we say `Option<i32>`. Note how similar this looks to `Option<T>`. So, in this particular Option, `T` has the value of `i32`. On the right-hand side of the binding, we make a `Some(T)`, where `T` is `5`. Since that's an `i32`, the two sides match, and Rust is happy. If they didn't match, we'd get an error:

```
let x: Option<f64> = Some(5);
// error: mismatched types: expected `core::option::Option<f64>`,
// found `core::option::Option<_>` (expected f64 but found integral
↪  variable)
```

That doesn't mean we can't make `Option<T>`s that hold an `f64`! They have to match up:

```
let x: Option<i32> = Some(5);
let y: Option<f64> = Some(5.0f64);
```

This is just fine. One definition, multiple uses.

Generics don't have to only be generic over one type. Consider another type from Rust's standard library that's similar, `Result<T, E>`:

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

This type is generic over two types: `T` and `E`. By the way, the capital letters can be any letter you'd like. We could define `Result<T, E>` as:

```
enum Result<A, Z> {
    Ok(A),
    Err(Z),
}
```

if we wanted to. Convention says that the first generic parameter should be `T`, for 'type', and that we use `E` for 'error'. Rust doesn't care, however.

The `Result<T, E>` type is intended to be used to return the result of a computation, and to have the ability to return an error if it didn't work out.

Generic functions

We can write functions that take generic types with a similar syntax:

```
fn takes_anything<T>(x: T) {
    // do something with x
}
```

The syntax has two parts: the `<T>` says "this function is generic over one type, `T`", and the `x: T` says "`x` has the type `T`."

Multiple arguments can have the same generic type:

```
fn takes_two_of_the_same_things<T>(x: T, y: T) {
    // ...
}
```

We could write a version that takes multiple types:

```
fn takes_two_things<T, U>(x: T, y: U) {
    // ...
}
```

Generic structs

You can store a generic type in a `struct` as well:

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

let int_origin = Point { x: 0, y: 0 };
let float_origin = Point { x: 0.0, y: 0.0 };
```

Similar to functions, the `<T>` is where we declare the generic parameters, and we then use `x: T` in the type declaration, too.

When you want to add an implementation for the generic `struct`, you declare the type parameter after the `impl`:

```
impl<T> Point<T> {
    fn swap(&mut self) {
        std::mem::swap(&mut self.x, &mut self.y);
    }
}
```

So far you've seen generics that take absolutely any type. These are useful in many cases: you've already seen `Option<T>`, and later you'll meet universal container types like Vec<T>. On the other hand, often you want to trade that flexibility for increased expressive power. Read about trait bounds to see why and how.

4.19 Traits

A trait is a language feature that tells the Rust compiler about functionality a type must provide.

Recall the `impl` keyword, used to call a function with method syntax:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Traits are similar, except that we first define a trait with a method signature, then implement the trait for a type. In this example, we implement the trait `HasArea` for `Circle`:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

As you can see, the `trait` block looks very similar to the `impl` block, but we don't define a body, only a type signature. When we `impl` a trait, we use `impl Trait for Item`, rather than only `impl Item`.

Trait bounds on generic functions

Traits are useful because they allow a type to make certain promises about its behavior. Generic functions can exploit this to constrain, or Bounds, the types they accept. Consider this function, which does not compile:

```
fn print_area<T>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Rust complains:

```
error: no method named `area` found for type `T` in the current scope
```

Because `T` can be any type, we can't be sure that it implements the `area` method. But we can add a trait bound to our generic `T`, ensuring that it does:

```
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

The syntax `<T: HasArea>` means “any type that implements the `HasArea` trait.” Because traits define function type signatures, we can be sure that any type which implements `HasArea` will have an `.area()` method.

Here’s an extended example of how this works:

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 1.0f64,
    };

    print_area(c);
    print_area(s);
}
```

This program outputs:

```
This shape has an area of 3.141593
This shape has an area of 1
```

As you can see, `print_area` is now generic, but also ensures that we have passed in the correct types. If we pass in an incorrect type:

```
print_area(5);
```

We get a compile-time error:

```
error: the trait `HasArea` is not implemented for the type `_` [E0277]
```

Trait bounds on generic structs

Your generic structs can also benefit from trait bounds. All you need to do is append the bound when you declare type parameters. Here is a new type `Rectangle<T>` and its operation `is_square()`:

```
struct Rectangle<T> {
    x: T,
    y: T,
    width: T,
    height: T,
}

impl<T: PartialEq> Rectangle<T> {
    fn is_square(&self) -> bool {
        self.width == self.height
    }
}

fn main() {
    let mut r = Rectangle {
        x: 0,
        y: 0,
        width: 47,
        height: 47,
    };

    assert!(r.is_square());

    r.height = 42;
    assert!(!r.is_square());
}
```

`is_square()` needs to check that the sides are equal, so the sides must be of a type that implements the core::cmp::PartialEq trait:

```
impl<T: PartialEq> Rectangle<T> { ... }
```

Now, a rectangle can be defined in terms of any type that can be compared for equality.

Here we defined a new struct `Rectangle` that accepts numbers of any precision—really, objects of pretty much any type—as long as they can be compared for equality. Could we do the same for our HasArea structs, Square and Circle? Yes, but they need multiplication, and to work with that we need to know more about operator traits.

Rules for implementing traits

So far, we've only added trait implementations to structs, but you can implement a trait for any type. So technically, we *could* implement HasArea for i32:

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for i32 {
    fn area(&self) -> f64 {
        println!("this is silly");

        *self as f64
    }
}

5.area();
```

It is considered poor style to implement methods on such primitive types, even though it is possible.

This may seem like the Wild West, but there are two restrictions around implementing traits that prevent this from getting out of hand. The first is that if the trait isn't defined in your scope, it doesn't apply. Here's an example: the standard library provides a Write trait which adds extra functionality to Files, for doing file I/O. By default, a File won't have its methods:

```
let mut f = std::fs::File::open("foo.txt").expect("Couldn't open foo.txt");
let buf = b"whatever"; // byte string literal. buf: &[u8; 8]
let result = f.write(buf);
```

Here's the error:

```
error: type `std::fs::File` does not implement any method in scope named `write`
let result = f.write(buf);
               ^~~~~~~~~~
```

We need to use the Write trait first:

```
use std::io::Write;

let mut f = std::fs::File::open("foo.txt").expect("Couldn't open foo.txt");
let buf = b"whatever";
let result = f.write(buf);
```

This will compile without error.

This means that even if someone does something bad like add methods to `i32`, it won't affect you, unless you `use` that trait.

There's one more restriction on implementing traits: either the trait, or the type you're writing the `impl` for, must be defined by you. So, we could implement the `HasArea` type for `i32`, because `HasArea` is in our code. But if we tried to implement `ToString`, a trait provided by Rust, for `i32`, we could not, because neither the trait nor the type are in our code.

One last thing about traits: generic functions with a trait bound use 'monomorphization' (mono: one, morph: form), so they are statically dispatched. What's that mean? Check out the chapter on trait objects for more details.

Multiple trait bounds

You've seen that you can bound a generic type parameter with a trait:

```
fn foo<T: Clone>(x: T) {
    x.clone();
}
```

If you need more than one bound, you can use `+`:

```
use std::fmt::Debug;

fn foo<T: Clone + Debug>(x: T) {
    x.clone();
    println!("{:?}", x);
}
```

`T` now needs to be both `Clone` as well as `Debug`.

Where clause

Writing functions with only a few generic types and a small number of trait bounds isn't too bad, but as the number increases, the syntax gets increasingly awkward:

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}
```

The name of the function is on the far left, and the parameter list is on the far right. The bounds are getting in the way.

Rust has a solution, and it's called a '`where` clause':

```
use std::fmt::Debug;

fn foo<T: Clone, K: Clone + Debug>(x: T, y: K) {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn bar<T, K>(x: T, y: K) where T: Clone, K: Clone + Debug {
    x.clone();
    y.clone();
    println!("{:?}", y);
}

fn main() {
    foo("Hello", "world");
    bar("Hello", "world");
}
```

`foo()` uses the syntax we showed earlier, and `bar()` uses a `where` clause. All you need to do is leave off the bounds when defining your type parameters, and then add `where` after the parameter list. For longer lists, whitespace can be added:

```
use std::fmt::Debug;

fn bar<T, K>(x: T, y: K)
    where T: Clone,
          K: Clone + Debug {

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

This flexibility can add clarity in complex situations.

`where` is also more powerful than the simpler syntax. For example:

```
trait ConvertTo<Output> {
    fn convert(&self) -> Output;
}

impl ConvertTo<i64> for i32 {
    fn convert(&self) -> i64 { *self as i64 }
}

// can be called with T == i32
fn normal<T: ConvertTo<i64>>(x: &T) -> i64 {
    x.convert()
}

// can be called with T == i64
fn inverse<T>() -> T
        // this is using ConvertTo as if it were "ConvertTo<i64>"
        where i32: ConvertTo<T> {
    42.convert()
}
```

This shows off the additional feature of `where` clauses: they allow bounds on the left-hand side not only of type parameters `T`, but also of types (`i32` in this case). In this example, `i32` must implement `ConvertTo<T>`. Rather than defining what `i32` is (since that's obvious), the `where` clause here constrains `T`.

Default methods

A default method can be added to a trait definition if it is already known how a typical implementor will define a method. For example, `is_invalid()` is defined as the opposite of `is_valid()`:

```
trait Foo {
    fn is_valid(&self) -> bool;

    fn is_invalid(&self) -> bool { !self.is_valid() }
}
```

Implementors of the `Foo` trait need to implement `is_valid()` but not `is_invalid()` due to the added default behavior. This default behavior can still be overridden as in:

```
struct UseDefault;

impl Foo for UseDefault {
    fn is_valid(&self) -> bool {
        println!("Called UseDefault.is_valid.");
        true
    }
}

struct OverrideDefault;

impl Foo for OverrideDefault {
    fn is_valid(&self) -> bool {
        println!("Called OverrideDefault.is_valid.");
        true
    }

    fn is_invalid(&self) -> bool {
        println!("Called OverrideDefault.is_invalid!");
        true // overrides the expected value of is_invalid()
    }
}

let default = UseDefault;
assert!(!default.is_invalid()); // prints "Called UseDefault.is_valid."

let over = OverrideDefault;
assert!(over.is_invalid()); // prints "Called OverrideDefault.is_invalid!"
```

Inheritance

Sometimes, implementing a trait requires implementing another trait:

```
trait Foo {
    fn foo(&self);
}

trait FooBar : Foo {
    fn foobar(&self);
}
```

Implementors of `FooBar` must also implement `Foo`, like this:

```
struct Baz;

impl Foo for Baz {
    fn foo(&self) { println!("foo"); }
}

impl FooBar for Baz {
    fn foobar(&self) { println!("foobar"); }
}
```

If we forget to implement `Foo`, Rust will tell us:

```
error: the trait `main::Foo` is not implemented for the type `main::Baz` [E0277]
```

Deriving

Implementing traits like `Debug` and `Default` repeatedly can become quite tedious. For that reason, Rust provides an attribute that allows you to let Rust automatically implement traits for you:

```rust
#[derive(Debug)]
struct Foo;

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

However, deriving is limited to a certain set of traits:

- Clone
- Copy
- Debug
- Default
- Eq
- Hash
- Ord
- PartialEq
- PartialOrd

4.20 Drop

Now that we've discussed traits, let's talk about a particular trait provided by the Rust standard library, Drop. The `Drop` trait provides a way to run some code when a value goes out of scope. For example:

```rust
struct HasDrop;

impl Drop for HasDrop {
    fn drop(&mut self) {
        println!("Dropping!");
    }
}

fn main() {
    let x = HasDrop;

    // do stuff

} // x goes out of scope here
```

When `x` goes out of scope at the end of `main()`, the code for `Drop` will run. `Drop` has one method, which is also called `drop()`. It takes a mutable reference to `self`.

That's it! The mechanics of `Drop` are very simple, but there are some subtleties. For example, values are dropped in the opposite order they are declared. Here's another example:

```rust
struct Firework {
    strength: i32,
}

impl Drop for Firework {
    fn drop(&mut self) {
        println!("BOOM times {}!!!", self.strength);
    }
}

fn main() {
    let firecracker = Firework { strength: 1 };
    let tnt = Firework { strength: 100 };
}
```

This will output:

```
BOOM times 100!!!
BOOM times 1!!!
```

The TNT goes off before the firecracker does, because it was declared afterwards. Last in, first out.

So what is `Drop` good for? Generally, `Drop` is used to clean up any resources associated with a `struct`. For example, the Arc<T> type is a reference-counted type. When `Drop` is called, it will decrement the reference count, and if the total number of references is zero, will clean up the underlying value.

4.21 if let

`if let` allows you to combine `if` and `let` together to reduce the overhead of certain kinds of pattern matches.

For example, let's say we have some sort of `Option<T>`. We want to call a function on it if it's `Some<T>`, but do nothing if it's `None`. That looks like this:

```
match option {
    Some(x) => { foo(x) },
    None => {},
}
```

We don't have to use `match` here, for example, we could use `if`:

```
if option.is_some() {
    let x = option.unwrap();
    foo(x);
}
```

Neither of these options is particularly appealing. We can use `if let` to do the same thing in a nicer way:

```
if let Some(x) = option {
    foo(x);
}
```

If a pattern (see Patterns) matches successfully, it binds any appropriate parts of the value to the identifiers in the pattern, then evaluates the expression. If the pattern doesn't match, nothing happens.

If you want to do something else when the pattern does not match, you can use `else`:

```
if let Some(x) = option {
    foo(x);
} else {
    bar();
}
```

`while let`

In a similar fashion, `while let` can be used when you want to conditionally loop as long as a value matches a certain pattern. It turns code like this:

```
let mut v = vec![1, 3, 5, 7, 11];
loop {
    match v.pop() {
        Some(x) =>  println!("{}", x),
        None => break,
    }
}
```

Into code like this:

```
let mut v = vec![1, 3, 5, 7, 11];
while let Some(x) = v.pop() {
    println!("{}", x);
}
```

4.22 Trait Objects

When code involves polymorphism, there needs to be a mechanism to determine which specific version is actually run. This is called 'dispatch'. There are two major forms of dispatch: static dispatch and dynamic dispatch. While Rust favors static dispatch, it also supports dynamic dispatch through a mechanism called 'trait objects'.

Background

For the rest of this chapter, we'll need a trait and some implementations. Let's make a simple one, `Foo`. It has one method that is expected to return a `String`.

```
trait Foo {
    fn method(&self) -> String;
}
```

We'll also implement this trait for `u8` and `String`:

```
impl Foo for u8 {
    fn method(&self) -> String { format!("u8: {}", *self) }
}

impl Foo for String {
    fn method(&self) -> String { format!("string: {}", *self) }
}
```

Static dispatch

We can use this trait to perform static dispatch with trait bounds:

```rust
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

Rust uses 'monomorphization' to perform static dispatch here. This means that Rust will create a special version of `do_something()` for both `u8` and `String`, and then replace the call sites with calls to these specialized functions. In other words, Rust generates something like this:

```rust
fn do_something_u8(x: u8) {
    x.method();
}

fn do_something_string(x: String) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something_u8(x);
    do_something_string(y);
}
```

This has a great upside: static dispatch allows function calls to be inlined because the callee is known at compile time, and inlining is the key to good optimization. Static dispatch is fast, but it comes at a tradeoff: 'code bloat', due to many copies of the same function existing in the binary, one for each type.

Furthermore, compilers aren't perfect and may "optimize" code to become slower. For example, functions inlined too eagerly will bloat the instruction cache (cache rules everything around us). This is part of the reason that `#[inline]` and `#[inline(always)]` should be used carefully, and one reason why using a dynamic dispatch is sometimes more efficient.

However, the common case is that it is more efficient to use static dispatch, and one can always have a thin statically-dispatched wrapper function that does a dynamic dispatch, but not vice versa, meaning static calls are more flexible. The standard library tries to be statically dispatched where possible for this reason.

Dynamic dispatch

Rust provides dynamic dispatch through a feature called 'trait objects'. Trait objects, like `&Foo` or `Box<Foo>`, are normal values that store a value of *any type* that implements the given trait, where the precise type can only be known at runtime.

A trait object can be obtained from a pointer to a concrete type that implements the trait by casting it (e.g. `&x` as `&Foo`) or coercing it (e.g. using `&x` as an argument to a function that takes `&Foo`).

These trait object coercions and casts also work for pointers like `&mut T` to `&mut Foo` and `Box<T>` to `Box<Foo>`, but that's all at the moment. Coercions and casts are identical.

This operation can be seen as 'erasing' the compiler's knowledge about the specific type of the pointer, and hence trait objects are sometimes referred to as 'type erasure'.

Coming back to the example above, we can use the same trait to perform dynamic dispatch with trait objects by casting:

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = 5u8;
    do_something(&x as &Foo);
}
```

or by coercing:

```
fn do_something(x: &Foo) {
    x.method();
}

fn main() {
    let x = "Hello".to_string();
    do_something(&x);
}
```

A function that takes a trait object is not specialized to each of the types that implements `Foo`: only one copy is generated, often (but not always) resulting in less code bloat. However, this comes at the cost of requiring slower virtual function calls, and effectively inhibiting any chance of inlining and related optimizations from occurring.

Why pointers?

Rust does not put things behind a pointer by default, unlike many managed languages, so types can have different sizes. Knowing the size of the value at compile time is important for things like passing it as an argument to a function, moving it about on the stack and allocating (and deallocating) space on the heap to store it.

For `Foo`, we would need to have a value that could be at least either a `String` (24 bytes) or a `u8` (1 byte), as well as any other type for which dependent crates may implement `Foo` (any number of bytes at all). There's no way to guarantee that this last point can work if the values are stored without a pointer, because those other types can be arbitrarily large.

Putting the value behind a pointer means the size of the value is not relevant when we are tossing a trait object around, only the size of the pointer itself.

Representation

The methods of the trait can be called on a trait object via a special record of function pointers traditionally called a 'vtable' (created and managed by the compiler).

Trait objects are both simple and complicated: their core representation and layout is quite straight-forward, but there are some curly error messages and surprising behaviors to discover.

Let's start simple, with the runtime representation of a trait object. The `std::raw` module contains structs with layouts that are the same as the complicated built-in types, including trait objects:

```
pub struct TraitObject {
    pub data: *mut (),
    pub vtable: *mut (),
}
```

That is, a trait object like `&Foo` consists of a 'data' pointer and a 'vtable' pointer.

The data pointer addresses the data (of some unknown type `T`) that the trait object is storing, and the vtable pointer points to the vtable ('virtual method table') corresponding to the implementation of `Foo` for `T`.

A vtable is essentially a struct of function pointers, pointing to the concrete piece of machine code for each method in the implementation. A method call like `trait_object.method()` will retrieve the correct pointer out of the vtable and then do a dynamic call of it. For example:

```
struct FooVtable {
    destructor: fn(*mut ()),
    size: usize,
    align: usize,
    method: fn(*const ()) -> String,
}

// u8:

fn call_method_on_u8(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a u8
    let byte: &u8 = unsafe { &*(x as *const u8) };

    byte.method()
}

static Foo_for_u8_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    size: 1,
    align: 1,

    // cast to a function pointer
    method: call_method_on_u8 as fn(*const ()) -> String,
};


// String:

fn call_method_on_String(x: *const ()) -> String {
    // the compiler guarantees that this function is only called
    // with `x` pointing to a String
    let string: &String = unsafe { &*(x as *const String) };

    string.method()
}

static Foo_for_String_vtable: FooVtable = FooVtable {
    destructor: /* compiler magic */,
    // values for a 64-bit computer, halve them for 32-bit ones
    size: 24,
    align: 8,

    method: call_method_on_String as fn(*const ()) -> String,
};
```

The `destructor` field in each vtable points to a function that will clean up any resources of the vtable's type: for `u8` it is trivial, but for `String` it will free the memory. This is necessary for owning trait objects like `Box<Foo>`, which need to clean-up both the `Box` allocation as well as the internal type when they go out of scope. The `size` and `align` fields store the size of the erased type, and its alignment requirements; these are essentially unused at the moment since the information is embedded in the destructor, but will be used in the future, as trait objects are progressively made more flexible.

Suppose we've got some values that implement `Foo`. The explicit form of construction and use of `Foo` trait objects might look a bit like (ignoring the type mismatches: they're all pointers anyway):

```
let a: String = "foo".to_string();
let x: u8 = 1;

// let b: &Foo = &a;
let b = TraitObject {
    // store the data
    data: &a,
    // store the methods
    vtable: &Foo_for_String_vtable
};

// let y: &Foo = x;
let y = TraitObject {
    // store the data
    data: &x,
    // store the methods
    vtable: &Foo_for_u8_vtable
};

// b.method();
(b.vtable.method)(b.data);

// y.method();
(y.vtable.method)(y.data);
```

Object Safety

Not every trait can be used to make a trait object. For example, vectors implement `Clone`, but if we try to make a trait object:

```
let v = vec![1, 2, 3];
let o = &v as &Clone;
```

We get an error:

```
error: cannot convert to a trait object because trait `core::clone::Clone` is not object-saf
let o = &v as &Clone;
        ^~
note: the trait cannot require that `Self : Sized`
let o = &v as &Clone;
        ^~
```

The error says that `Clone` is not 'object-safe'. Only traits that are object-safe can be made into trait objects. A trait is object-safe if both of these are true:

- the trait does not require that `Self: Sized`
- all of its methods are object-safe

So what makes a method object-safe? Each method must require that `Self: Sized` or all of the following:

- must not have any type parameters
- must not use `Self`

Whew! As we can see, almost all of these rules talk about `Self`. A good intuition is "except in special circumstances, if your trait's method uses `Self`, it is not object-safe."

4.23 Closures

Sometimes it is useful to wrap up a function and *free variables* for better clarity and reuse. The free variables that can be used come from the enclosing scope and are 'closed over' when used in the function. From this, we get the name 'closures' and Rust provides a really great implementation of them, as we'll see.

Syntax

Closures look like this:

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

We create a binding, `plus_one`, and assign it to a closure. The closure's arguments go between the pipes (`|`), and the body is an expression, in this case, `x + 1`. Remember that `{ }` is an expression, so we can have multi-line closures too:

```
let plus_two = |x| {
    let mut result: i32 = x;

    result += 1;
    result += 1;

    result
};

assert_eq!(4, plus_two(2));
```

You'll notice a few things about closures that are a bit different from regular named functions defined with `fn`. The first is that we did not need to annotate the types of arguments the closure takes or the values it returns. We can:

```
let plus_one = |x: i32| -> i32 { x + 1 };

assert_eq!(2, plus_one(1));
```

But we don't have to. Why is this? Basically, it was chosen for ergonomic reasons. While specifying the full type for named functions is helpful with things like documentation and type inference, the full type signatures of closures are rarely documented since they're anonymous, and they don't cause the kinds of error-at-a-distance problems that inferring named function types can.

The second is that the syntax is similar, but a bit different. I've added spaces here for easier comparison:

```
fn  plus_one_v1   (x: i32) -> i32 { x + 1 }
let plus_one_v2 = |x: i32| -> i32 { x + 1 };
let plus_one_v3 = |x: i32|          x + 1  ;
```

Small differences, but they're similar.

Closures and their environment

The environment for a closure can include bindings from its enclosing scope in addition to parameters and local bindings. It looks like this:

```
let num = 5;
let plus_num = |x: i32| x + num;

assert_eq!(10, plus_num(5));
```

This closure, `plus_num`, refers to a `let` binding in its scope: `num`. More specifically, it borrows the binding. If we do something that would conflict with that binding, we get an error. Like this one:

```
let mut num = 5;
let plus_num = |x: i32| x + num;

let y = &mut num;
```

Which errors with:

```
error: cannot borrow `num` as mutable because it is also borrowed as immutable
    let y = &mut num;
                 ^~~
note: previous borrow of `num` occurs here due to use in closure; the immutable
  borrow prevents subsequent moves or mutable borrows of `num` until the borrow
  ends
    let plus_num = |x| x + num;
                   ^~~~~~~~~~~
note: previous borrow ends here
```

```
fn main() {
    let mut num = 5;
    let plus_num = |x| x + num;

    let y = &mut num;
}
^
```

A verbose yet helpful error message! As it says, we can't take a mutable borrow on `num` because the closure is already borrowing it. If we let the closure go out of scope, we can:

```
let mut num = 5;
{
    let plus_num = |x: i32| x + num;

} // plus_num goes out of scope, borrow of num ends

let y = &mut num;
```

If your closure requires it, however, Rust will take ownership and move the environment instead. This doesn't work:

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

let takes_nums = || nums;

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

We get this error:

```
note: `nums` moved into closure environment here because it has type
  `[closure(()) -> collections::vec::Vec<i32>]`, which is non-copyable
let takes_nums = || nums;
                    ^~~~~~~
```

`Vec<T>` has ownership over its contents, and therefore, when we refer to it in our closure, we have to take ownership of `nums`. It's the same as if we'd passed `nums` to a function that took ownership of it.

`move` closures

We can force our closure to take ownership of its environment with the `move` keyword:

```
let num = 5;

let owns_num = move |x: i32| x + num;
```

Now, even though the keyword is `move`, the variables follow normal move semantics. In this case, `5` implements `Copy`, and so `owns_num` takes ownership of a copy of `num`. So what's the difference?

```
let mut num = 5;

{
    let mut add_num = |x: i32| num += x;

    add_num(5);
}

assert_eq!(10, num);
```

So in this case, our closure took a mutable reference to `num`, and then when we called `add_num`, it mutated the underlying value, as we'd expect. We also needed to declare `add_num` as `mut` too, because we're mutating its environment.

If we change to a `move` closure, it's different:

```
let mut num = 5;

{
    let mut add_num = move |x: i32| num += x;

    add_num(5);
}

assert_eq!(5, num);
```

We only get `5`. Rather than taking a mutable borrow out on our `num`, we took ownership of a copy.

Another way to think about `move` closures: they give a closure its own stack frame. Without `move`, a closure may be tied to the stack frame that created it, while a `move` closure is self-contained. This means that you cannot generally return a non-`move` closure from a function, for example.

But before we talk about taking and returning closures, we should talk some more about the way that closures are implemented. As a systems language, Rust gives you tons of control over what your code does, and closures are no different.

Closure implementation

Rust's implementation of closures is a bit different than other languages. They are effectively syntax sugar for traits. You'll want to make sure to have read the traits section (see Traits) before this one, as well as the section on trait objects (see Trait Objects).

Got all that? Good.

The key to understanding how closures work under the hood is something a bit strange: Using `()` to call a function, like `foo()`, is an overloadable operator. From this, everything else clicks into place. In Rust, we use the trait system to overload operators. Calling functions is no different. We have three separate traits to overload with:

```
pub trait Fn<Args> : FnMut<Args> {
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}

pub trait FnMut<Args> : FnOnce<Args> {
    extern "rust-call" fn call_mut(&mut self, args: Args) -> Self::Output;
}

pub trait FnOnce<Args> {
    type Output;

    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

You'll notice a few differences between these traits, but a big one is `self`: `Fn` takes `&self`, `FnMut` takes `&mut self`, and `FnOnce` takes `self`. This covers all three kinds of self via the usual method call syntax. But we've split them up into three traits, rather than having a single one. This gives us a large amount of control over what kind of closures we can take.

The `|| {}` syntax for closures is sugar for these three traits. Rust will generate a `struct` for the environment, `impl` the appropriate trait, and then use it.

Taking closures as arguments

Now that we know that closures are traits, we already know how to accept and return closures: the same as any other trait!

This also means that we can choose static vs dynamic dispatch as well. First, let's write a function which takes something callable, calls it, and returns the result:

```
fn call_with_one<F>(some_closure: F) -> i32
    where F : Fn(i32) -> i32 {

    some_closure(1)
}

let answer = call_with_one(|x| x + 2);

assert_eq!(3, answer);
```

We pass our closure, `|x| x + 2`, to `call_with_one`. It does what it suggests: it calls the closure, giving it `1` as an argument.

Let's examine the signature of `call_with_one` in more depth:

```
fn call_with_one<F>(some_closure: F) -> i32
```

We take one parameter, and it has the type `F`. We also return a `i32`. This part isn't interesting. The next part is:

```
    where F : Fn(i32) -> i32 {
```

Because `Fn` is a trait, we can bound our generic with it. In this case, our closure takes a `i32` as an argument and returns an `i32`, and so the generic bound we use is `Fn(i32) -> i32`.

There's one other key point here: because we're bounding a generic with a trait, this will get monomorphized, and therefore, we'll be doing static dispatch into the closure. That's pretty neat. In many languages, closures are inherently heap allocated, and will always involve dynamic dispatch. In Rust, we can stack allocate our closure environment, and statically dispatch the call. This happens quite often with iterators and their adapters, which often take closures as arguments.

Of course, if we want dynamic dispatch, we can get that too. A trait object handles this case, as usual:

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

let answer = call_with_one(&|x| x + 2);

assert_eq!(3, answer);
```

Now we take a trait object, a `&Fn`. And we have to make a reference to our closure when we pass it to `call_with_one`, so we use `&||`.

Function pointers and closures

A function pointer is kind of like a closure that has no environment. As such, you can pass a function pointer to any function expecting a closure argument, and it will work:

```
fn call_with_one(some_closure: &Fn(i32) -> i32) -> i32 {
    some_closure(1)
}

fn add_one(i: i32) -> i32 {
    i + 1
}

let f = add_one;

let answer = call_with_one(&f);

assert_eq!(2, answer);
```

In this example, we don't strictly need the intermediate variable `f`, the name of the function works just fine too:

```
let answer = call_with_one(&add_one);
```

Returning closures

It's very common for functional-style code to return closures in various situations. If you try to return a closure, you may run into an error. At first, it may seem strange, but we'll figure it out. Here's how you'd probably try to return a closure from a function:

```
fn factory() -> (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

This gives us these long, related errors:

```
error: the trait `core::marker::Sized` is not implemented for the type
`core::ops::Fn(i32) -> i32` [E0277]
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
fn factory() -> (Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~
error: the trait `core::marker::Sized` is not implemented for the type `core::ops::Fn(i32) -
let f = factory();
    ^
note: `core::ops::Fn(i32) -> i32` does not have a constant size known at compile-time
let f = factory();
    ^
```

In order to return something from a function, Rust needs to know what size the return type is. But since `Fn` is a trait, it could be various things of various sizes: many different types can implement `Fn`. An easy way to give something a size is to take a reference to it, as references have a known size. So we'd write this:

```
fn factory() -> &(Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

But we get another error:

```
error: missing lifetime specifier [E0106]
fn factory() -> &(Fn(i32) -> i32) {
                ^~~~~~~~~~~~~~~~~
```

Right. Because we have a reference, we need to give it a lifetime. But our `factory()` function takes no arguments, so elision (see Lifetime Elision) doesn't kick in here. Then what choices do we have? Try `'static`:

```
fn factory() -> &'static (Fn(i32) -> i32) {
    let num = 5;

    |x| x + num
}

let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

But we get another error:

```
error: mismatched types:
 expected `&'static core::ops::Fn(i32) -> i32`,
    found `[closure@<anon>:7:9: 7:20]`
(expected &-ptr,
    found closure) [E0308]
         |x| x + num
         ^~~~~~~~~~~
```

This error is letting us know that we don't have a `&'static Fn(i32) -> i32`, we have a `[closure@<anon>:7:9: 7:20]`. Wait, what?

Because each closure generates its own environment `struct` and implementation of `Fn` and friends, these types are anonymous. They exist solely for this closure. So Rust shows them as `closure@<anon>`, rather than some autogenerated name.

The error also points out that the return type is expected to be a reference, but what we are trying to return is not. Further, we cannot directly assign a `'static` lifetime to an object. So we'll take a different approach and return a 'trait object' by `Box`ing up the `Fn`. This almost works:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(|x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

There's just one last problem:

```
error: closure may outlive the current function, but it borrows `num`,
which is owned by the current function [E0373]
Box::new(|x| x + num)
         ^~~~~~~~~~~
```

Well, as we discussed before, closures borrow their environment. And in this case, our environment is based on a stack-allocated `5`, the `num` variable binding. So the borrow has a lifetime of the stack frame. So if we returned this closure, the function call would be over, the stack frame would go away, and our closure is capturing an environment of garbage memory! With one last fix, we can make this work:

```
fn factory() -> Box<Fn(i32) -> i32> {
    let num = 5;

    Box::new(move |x| x + num)
}
let f = factory();

let answer = f(1);
assert_eq!(6, answer);
```

By making the inner closure a `move Fn`, we create a new stack frame for our closure. By `Box`ing it up, we've given it a known size, and allowing it to escape our stack frame.

4.24 Universal Function Call Syntax

Sometimes, functions can have the same names. Consider this code:

```
trait Foo {
    fn f(&self);
}

trait Bar {
    fn f(&self);
}

struct Baz;

impl Foo for Baz {
    fn f(&self) { println!("Baz's impl of Foo"); }
}

impl Bar for Baz {
    fn f(&self) { println!("Baz's impl of Bar"); }
}

let b = Baz;
```

If we were to try to call `b.f()`, we'd get an error:

```
error: multiple applicable methods in scope [E0034]
b.f();
  ^~~
note: candidate #1 is defined in an impl of the trait `main::Foo` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Foo"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
note: candidate #2 is defined in an impl of the trait `main::Bar` for the type
`main::Baz`
    fn f(&self) { println!("Baz's impl of Bar"); }
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

We need a way to disambiguate which method we need. This feature is called 'universal function call syntax', and it looks like this:

```
Foo::f(&b);
Bar::f(&b);
```

Let's break it down.

```
Foo::
Bar::
```

These halves of the invocation are the types of the two traits: `Foo` and `Bar`. This is what ends up actually doing the disambiguation between the two: Rust calls the one from the trait name you use.

```
f(&b)
```

When we call a method like `b.f()` using method syntax (see Method Syntax), Rust will automatically borrow `b` if `f()` takes `&self`. In this case, Rust will not, and so we need to pass an explicit `&b`.

Angle-bracket Form

The form of UFCS we just talked about:

```
Trait::method(args);
```

Is a short-hand. There's an expanded form of this that's needed in some situations:

```
<Type as Trait>::method(args);
```

The `<>::` syntax is a means of providing a type hint. The type goes inside the `<>`s. In this case, the type is `Type as Trait`, indicating that we want `Trait`'s version of `method` to be called here. The `as Trait` part is optional if it's not ambiguous. Same with the angle brackets, hence the shorter form.

Here's an example of using the longer form.

```
trait Foo {
    fn foo() -> i32;
}

struct Bar;

impl Bar {
    fn foo() -> i32 {
        20
    }
}

impl Foo for Bar {
    fn foo() -> i32 {
        10
    }
}

fn main() {
    assert_eq!(10, <Bar as Foo>::foo());
    assert_eq!(20, Bar::foo());
}
```

Using the angle bracket syntax lets you call the trait method instead of the inherent one.

4.25 Crates and Modules

When a project starts getting large, it's considered good software engineering practice to split it up into a bunch of smaller pieces, and then fit them together. It is also important to have a well-defined interface, so that some of your functionality is private, and some is public. To facilitate these kinds of things, Rust has a module system.

Basic terminology: Crates and Modules

Rust has two distinct terms that relate to the module system: 'crate' and 'module'. A crate is synonymous with a 'library' or 'package' in other languages. Hence "Cargo" as the name of Rust's package management tool: you ship your crates to others with Cargo. Crates can produce an executable or a library, depending on the project.

Each crate has an implicit root module that contains the code for that crate. You can then define a tree of sub-modules under that root module. Modules allow you to partition your code within the crate itself.

As an example, let's make a phrases crate, which will give us various phrases in different languages. To keep things simple, we'll stick to 'greetings' and 'farewells' as two kinds of phrases, and use English and Japanese (日本語) as two languages for those phrases to be in. We'll use this module layout:

```
                                    +-----------+
                                +---| greetings |
                                |   +-----------+
                  +---------+   |
              +---| english |---+
              |   +---------+   |   +-----------+
              |                 +---| farewells |
+---------+   |                     +-----------+
| phrases |---+
+---------+   |                     +-----------+
              |                 +---| greetings |
              |   +----------+  |   +-----------+
              +---| japanese |--+
                  +----------+  |
                                |   +-----------+
                                +---| farewells |
                                    +-----------+
```

In this example, `phrases` is the name of our crate. All of the rest are modules. You can see that they form a tree, branching out from the crate *root*, which is the root of the tree: `phrases` itself.

Now that we have a plan, let's define these modules in code. To start, generate a new crate with Cargo:

```
$ cargo new phrases
$ cd phrases
```

If you remember, this generates a simple project for us:

```
$ tree .
.
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs` is our crate root, corresponding to the `phrases` in our diagram above.

Defining Modules

To define each of our modules, we use the `mod` keyword. Let's make our `src/lib.rs` look like this:

```
mod english {
    mod greetings {
    }

    mod farewells {
    }
}

mod japanese {
    mod greetings {
    }

    mod farewells {
    }
}
```

After the `mod` keyword, you give the name of the module. Module names follow the conventions for other Rust identifiers: `lower_snake_case`. The contents of each module are within curly braces (`{}`).

Within a given mod, you can declare sub-`mod`s. We can refer to sub-modules with double-colon (`::`) notation: our four nested modules are `english::greetings`, `english::farewells`, `japanese::greetings`, and `japanese::farewells`. Because these sub-modules are namespaced under their parent module, the names don't conflict: `english::greetings` and `japanese::greetings` are distinct, even though their names are both `greetings`.

Because this crate does not have a `main()` function, and is called `lib.rs`, Cargo will build this crate as a library:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target/debug
build  deps  examples  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrases-hash.rlib` is the compiled crate. Before we see how to use this crate from another crate, let's break it up into multiple files.

Multiple file crates

If each crate were just one file, these files would get very large. It's often easier to split up crates into multiple files, and Rust supports this in two ways.

Instead of declaring a module like this:

```rust
mod english {
    // contents of our module go here
}
```

We can instead declare our module like this:

```rust
mod english;
```

If we do that, Rust will expect to find either a `english.rs` file, or a `english/mod.rs` file with the contents of our module.

Note that in these files, you don't need to re-declare the module: that's already been done with the initial `mod` declaration.

Using these two techniques, we can break up our crate into two directories and seven files:

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
```

```
    └── debug
        ├── build
        ├── deps
        ├── examples
        ├── libphrases-a7448e02a0468eaa.rlib
        └── native
```

`src/lib.rs` is our crate root, and looks like this:

```
mod english;
mod japanese;
```

These two declarations tell Rust to look for either `src/english.rs` and `src/japanese.rs`, or `src/english/mod.rs` and `src/japanese/mod.rs`, depending on our preference. In this case, because our modules have sub-modules, we've chosen the second. Both `src/english/mod.rs` and `src/japanese/mod.rs` look like this:

```
mod greetings;
mod farewells;
```

Again, these declarations tell Rust to look for either `src/english/greetings.rs` and `src/japanese/greetings.rs` or `src/english/farewells/mod.rs` and `src/japanese/farewells/mod.rs`. Because these sub-modules don't have their own sub-modules, we've chosen to make them `src/english/greetings.rs` and `src/japanese/farewells.rs`. Whew!

The contents of `src/english/greetings.rs` and `src/japanese/farewells.rs` are both empty at the moment. Let's add some functions.

Put this in `src/english/greetings.rs`:

```
fn hello() -> String {
    "Hello!".to_string()
}
```

Put this in `src/english/farewells.rs`:

```
fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Put this in `src/japanese/greetings.rs`:

```
fn hello() -> String {
    "こんにちは".to_string()
}
```

Of course, you can copy and paste this from this web page, or type something else. It's not important that you actually put 'konnichiwa' to learn about the module system.

Put this in `src/japanese/farewells.rs`:

```
fn goodbye() -> String {
    "さようなら".to_string()
}
```

(This is 'Sayōnara', if you're curious.)

Now that we have some functionality in our crate, let's try to use it from another crate.

Importing External Crates

We have a library crate. Let's make an executable crate that imports and uses our library.

Make a `src/main.rs` and put this in it (it won't quite compile yet):

```
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}",
↪   phrases::english::farewells::goodbye());

    println!("Hello in Japanese: {}",
↪   phrases::japanese::greetings::hello());
    println!("Goodbye in Japanese: {}",
↪   phrases::japanese::farewells::goodbye());
}
```

The `extern crate` declaration tells Rust that we need to compile and link to the `phrases` crate. We can then use `phrases`' modules in this one. As we mentioned earlier, you can use double colons to refer to sub-modules and the functions inside of them.

(Note: when importing a crate that has dashes in its name "like-this", which is not a valid Rust identifier, it will be converted by changing the dashes to underscores, so you would write `extern crate like_this;`.)

Also, Cargo assumes that `src/main.rs` is the crate root of a binary crate, rather than a library crate. Our package now has two crates: `src/lib.rs` and `src/main.rs`. This pattern is quite common for executable crates: most functionality is in a library crate, and the executable crate uses that library. This way, other programs can also use the library crate, and it's also a nice separation of concerns.

This doesn't quite work yet, though. We get four errors that look similar to this:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:38: 4:72 error: function `hello` is private
```

```
src/main.rs:4     println!("Hello in English: {}", phrases::english::greetings::hello());
                                                   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

note: in expansion of format_args!
<std macros>:2:25: 2:58 note: expansion site
<std macros>:1:1: 2:62 note: in expansion of print!
<std macros>:3:1: 3:54 note: expansion site
<std macros>:1:1: 3:58 note: in expansion of println!
phrases/src/main.rs:4:5: 4:76 note: expansion site
```

By default, everything is private in Rust. Let's talk about this in some more depth.

Exporting a Public Interface

Rust allows you to precisely control which aspects of your interface are public, and so private is the default. To make things public, you use the `pub` keyword. Let's focus on the english module first, so let's reduce our `src/main.rs` to only this:

```rust
extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}",
↪   phrases::english::farewells::goodbye());
}
```

In our `src/lib.rs`, let's add `pub` to the `english` module declaration:

```rust
pub mod english;
mod japanese;
```

And in our `src/english/mod.rs`, let's make both `pub`:

```rust
pub mod greetings;
pub mod farewells;
```

In our `src/english/greetings.rs`, let's add `pub` to our `fn` declaration:

```rust
pub fn hello() -> String {
    "Hello!".to_string()
}
```

And also in `src/english/farewells.rs`:

```rust
pub fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Now, our crate compiles, albeit with warnings about not using the `japanese` functions:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/japanese/greetings.rs:1:1: 3:2 warning: function is never used: `hello`, #[warn(dead_code
src/japanese/greetings.rs:1 fn hello() -> String {
src/japanese/greetings.rs:2     " こんにちは".to_string()
src/japanese/greetings.rs:3 }
src/japanese/farewells.rs:1:1: 3:2 warning: function is never used: `goodbye`, #[warn(dead_c
src/japanese/farewells.rs:1 fn goodbye() -> String {
src/japanese/farewells.rs:2     " さようなら".to_string()
src/japanese/farewells.rs:3 }
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

`pub` also applies to `struct`s and their member fields. In keeping with Rust's tendency toward safety, simply making a `struct` public won't automatically make its members public: you must mark the fields individually with `pub`.

Now that our functions are public, we can use them. Great! However, typing out `phrases::english::greeting` is very long and repetitive. Rust has another keyword for importing names into the current scope, so that you can refer to them with shorter names. Let's talk about `use`.

Importing Modules with `use`

Rust has a `use` keyword, which allows us to import names into our local scope. Let's change our `src/main.rs` to look like this:

```rust
extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

The two `use` lines import each module into the local scope, so we can refer to the functions by a much shorter name. By convention, when importing functions, it's considered best practice to import the module, rather than the function directly. In other words, you can do this:

```rust
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

But it is not idiomatic. This is significantly more likely to introduce a naming conflict. In our short program, it's not a big deal, but as it grows, it becomes a problem. If we have conflicting names, Rust will give a compilation error. For example, if we made the `japanese` functions public, and tried to do this:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust will give us a compile-time error:

```
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
src/main.rs:4:5: 4:40 error: a value named `hello` has already been imported in this module
src/main.rs:4 use phrases::japanese::greetings::hello;
                  ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `phrases`.
```

If we're importing multiple names from the same module, we don't have to type it out twice. Instead of this:

```
use phrases::english::greetings;
use phrases::english::farewells;
```

We can use this shortcut:

```
use phrases::english::{greetings, farewells};
```

Re-exporting with `pub use`

You don't only use `use` to shorten identifiers. You can also use it inside of your crate to re-export a function inside another module. This allows you to present an external interface that may not directly map to your internal code organization.

Let's look at an example. Modify your `src/main.rs` to read like this:

```
extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());

    println!("Hello in Japanese: {}", japanese::hello());
    println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

Then, modify your `src/lib.rs` to make the `japanese` mod public:

```
pub mod english;
pub mod japanese;
```

Next, make the two functions public, first in `src/japanese/greetings.rs`:

```
pub fn hello() -> String {
    " こんにちは".to_string()
}
```

And then in `src/japanese/farewells.rs`:

```
pub fn goodbye() -> String {
    " さようなら".to_string()
}
```

Finally, modify your `src/japanese/mod.rs` to read like this:

```
pub use self::greetings::hello;
pub use self::farewells::goodbye;

mod greetings;
mod farewells;
```

The `pub use` declaration brings the function into scope at this part of our module hierarchy. Because we've `pub use`d this inside of our `japanese` module, we now have a `phrases::japanese::hello()` function and a `phrases::japanese::goodbye()` function, even though the code for them lives in `phrases::japanese::greetings::hello()` and `phrases::japanese::farewells::goodbye()`. Our internal organization doesn't define our external interface.

Here we have a `pub use` for each function we want to bring into the `japanese` scope. We could alternatively use the wildcard syntax to include everything from `greetings` into the current scope: `pub use self::greetings::*`.

What about the `self`? Well, by default, use declarations are absolute paths, starting from your crate root. `self` makes that path relative to your current place in the hierarchy instead. There's one more special form of `use`: you can `use super::` to reach one level up the tree from your current location. Some people like to think of `self` as `.` and `super` as `..`, from many shells' display for the current directory and the parent directory.

Outside of `use`, paths are relative: `foo::bar()` refers to a function inside of `foo` relative to where we are. If that's prefixed with `::`, as in `::foo::bar()`, it refers to a different `foo`, an absolute path from your crate root.

This will build and run:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
     Running `target/debug/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese: こんにちは
Goodbye in Japanese: さようなら
```

Complex imports

Rust offers several advanced options that can add compactness and convenience to your `extern crate` and `use` statements. Here is an example:

```
extern crate phrases as sayings;

use sayings::japanese::greetings as ja_greetings;
use sayings::japanese::farewells::*;
use sayings::english::{self, greetings as en_greetings, farewells as
↪  en_farewells};

fn main() {
    println!("Hello in English; {}", en_greetings::hello());
    println!("And in Japanese: {}", ja_greetings::hello());
    println!("Goodbye in English: {}", english::farewells::goodbye());
    println!("Again: {}", en_farewells::goodbye());
    println!("And in Japanese: {}", goodbye());
}
```

What's going on here?

First, both `extern crate` and `use` allow renaming the thing that is being imported. So the crate is still called "phrases", but here we will refer to it as "sayings". Similarly, the first `use` statement pulls in the `japanese::greetings` module from the crate, but makes it available as `ja_greetings` as opposed to simply `greetings`. This can help to avoid ambiguity when importing similarly-named items from different places.

The second `use` statement uses a star glob to bring in *all* symbols from the `sayings::japanese::farewells` module. As you can see we can later refer to the Japanese `goodbye` function with no module qualifiers. This kind of glob should be used sparingly.

The third `use` statement bears more explanation. It's using "brace expansion" globbing to compress three `use` statements into one (this sort of syntax may be familiar if you've written Linux shell scripts before). The uncompressed form of this statement would be:

```
use sayings::english;
use sayings::english::greetings as en_greetings;
use sayings::english::farewells as en_farewells;
```

As you can see, the curly brackets compress `use` statements for several items under the same path, and in this context `self` refers back to that path. Note: The curly brackets cannot be nested or mixed with star globbing.

4.26 'const' and 'static'

Rust has a way of defining constants with the `const` keyword:

```
const N: i32 = 5;
```

Unlike `let` bindings (see Variable Bindings), you must annotate the type of a `const`.

Constants live for the entire lifetime of a program. More specifically, constants in Rust have no fixed address in memory. This is because they're effectively inlined to each place that they're used. References to the same constant are not necessarily guaranteed to refer to the same memory address for this reason.

`static`

Rust provides a 'global variable' sort of facility in static items. They're similar to constants, but static items aren't inlined upon use. This means that there is only one instance for each value, and it's at a fixed location in memory.

Here's an example:

```
static N: i32 = 5;
```

Unlike `let` bindings, you must annotate the type of a `static`.

Statics live for the entire lifetime of a program, and therefore any reference stored in a constant has a `'static` lifetime (see Lifetimes):

```
static NAME: &'static str = "Steve";
```

Mutability

You can introduce mutability with the `mut` keyword:

```
static mut N: i32 = 5;
```

Because this is mutable, one thread could be updating `N` while another is reading it, causing memory unsafety. As such both accessing and mutating a `static mut` is unsafe, and so must be done in an `unsafe` block:

```
unsafe {
    N += 1;

    println!("N: {}", N);
}
```

Furthermore, any type stored in a `static` must be `Sync`, and may not have a `Drop` implementation (see Drop).

Initializing

Both `const` and `static` have requirements for giving them a value. They may only be given a value that's a constant expression. In other words, you cannot use the result of a function call or anything similarly complex or at runtime.

Which construct should I use?

Almost always, if you can choose between the two, choose `const`. It's pretty rare that you actually want a memory location associated with your constant, and using a const allows for optimizations like constant propagation not only in your crate but downstream crates.

4.27 Attributes

Declarations can be annotated with 'attributes' in Rust. They look like this:

```
#[test]
```

or like this:

```
#![test]
```

The difference between the two is the `!`, which changes what the attribute applies to:

```
#[foo]
struct Foo;

mod bar {
    #![bar]
}
```

The `#[foo]` attribute applies to the next item, which is the `struct` declaration. The `#![bar]` attribute applies to the item enclosing it, which is the `mod` declaration. Otherwise, they're the same. Both change the meaning of the item they're attached to somehow.

For example, consider a function like this:

```
#[test]
fn check() {
    assert_eq!(2, 1 + 1);
}
```

It is marked with `#[test]`. This means it's special: when you run tests, this function will execute. When you compile as usual, it won't even be included. This function is now a test function.

Attributes may also have additional data:

```
#[inline(always)]
fn super_fast_fn() {
```

Or even keys and values:

```
#[cfg(target_os = "macos")]
mod macos_only {
```

Rust attributes are used for a number of different things. There is a full list of attributes in the reference. Currently, you are not allowed to create your own attributes, the Rust compiler defines them.

4.28 'type' Aliases

The `type` keyword lets you declare an alias of another type:

```
type Name = String;
```

You can then use this type as if it were a real type:

```
type Name = String;

let x: Name = "Hello".to_string();
```

Note, however, that this is an *alias*, not a new type entirely. In other words, because Rust is strongly typed, you'd expect a comparison between two different types to fail:

```
let x: i32 = 5;
let y: i64 = 5;

if x == y {
   // ...
}
```

this gives

```
error: mismatched types:
 expected `i32`,
    found `i64`
(expected i32,
    found i64) [E0308]
     if x == y {
              ^
```

But, if we had an alias:

```
type Num = i32;

let x: i32 = 5;
let y: Num = 5;

if x == y {
   // ...
}
```

This compiles without error. Values of a `Num` type are the same as a value of type `i32`, in every way. You can use tuple struct (see Tuple structs) to really get a new type.

You can also use type aliases with generics:

```
use std::result;

enum ConcreteError {
    Foo,
    Bar,
}

type Result<T> = result::Result<T, ConcreteError>;
```

This creates a specialized version of the `Result` type, which always has a `ConcreteError` for the `E` part of `Result<T, E>`. This is commonly used in the standard library to create custom errors for each subsection. For example, io::Result.

4.29 Casting Between Types

Rust, with its focus on safety, provides two different ways of casting different types between each other. The first, `as`, is for safe casts. In contrast, `transmute` allows for arbitrary casting, and is one of the most dangerous features of Rust!

Coercion

Coercion between types is implicit and has no syntax of its own, but can be spelled out with `as`.

Coercion occurs in `let`, `const`, and `static` statements; in function call arguments; in field values in struct initialization; and in a function result.

The most common case of coercion is removing mutability from a reference:

- `&mut T` to `&T`

An analogous conversion is to remove mutability from a raw pointer:

- `*mut T` to `*const T`

References can also be coerced to raw pointers:

- `&T` to `*const T`
- `&mut T` to `*mut T`

Custom coercions may be defined using Deref.

Coercion is transitive.

as

The `as` keyword does safe casting:

```
let x: i32 = 5;

let y = x as i64;
```

There are three major categories of safe cast: explicit coercions, casts between numeric types, and pointer casts.

Casting is not transitive: even if `e as U1 as U2` is a valid expression, `e as U2` is not necessarily so (in fact it will only be valid if `U1` coerces to `U2`).

Explicit coercions

A cast `e as U` is valid if `e` has type `T` and `T` *coerces* to `U`.

Numeric casts

A cast `e as U` is also valid in any of the following cases:

- `e` has type `T` and `T` and `U` are any numeric types; *numeric-cast*
- `e` is a C-like enum (with no data attached to the variants), and `U` is an integer type; *enum-cast*
- `e` has type `bool` or `char` and `U` is an integer type; *prim-int-cast*
- `e` has type `u8` and `U` is `char`; *u8-char-cast*

For example

```
let one = true as u8;
let at_sign = 64 as char;
let two_hundred = -56i8 as u8;
```

The semantics of numeric casts are:

- Casting between two integers of the same size (e.g. i32 -> u32) is a no-op
- Casting from a larger integer to a smaller integer (e.g. u32 -> u8) will truncate
- Casting from a smaller integer to a larger integer (e.g. u8 -> u32) will
  - zero-extend if the source is unsigned
  - sign-extend if the source is signed
- Casting from a float to an integer will round the float towards zero
  - NOTE: currently this will cause Undefined Behavior if the rounded value cannot be represented by the target integer type. This includes Inf and NaN. This is a bug and will be fixed.
- Casting from an integer to float will produce the floating point representation of the integer, rounded if necessary (rounding strategy unspecified)
- Casting from an f32 to an f64 is perfect and lossless
- Casting from an f64 to an f32 will produce the closest possible value (rounding strategy unspecified)
  - NOTE: currently this will cause Undefined Behavior if the value is finite but larger or smaller than the largest or smallest finite value representable by f32. This is a bug and will be fixed.

Pointer casts

Perhaps surprisingly, it is safe to cast raw pointers to and from integers, and to cast between pointers to different types subject to some constraints. It is only unsafe to dereference the pointer:

```
let a = 300 as *const char; // a pointer to location 300
let b = a as u32;
```

`e as U` is a valid pointer cast in any of the following cases:

- `e` has type `*T`, `U` has type `*U_0`, and either `U_0: Sized` or `unsize_kind(T) == unsize_kind(U_0)`; *a ptr-ptr-cast*
- `e` has type `*T` and `U` is a numeric type, while `T: Sized`; *ptr-addr-cast*
- `e` is an integer and `U` is `*U_0`, while `U_0: Sized`; *addr-ptr-cast*
- `e` has type `&[T; n]` and `U` is `*const T`; *array-ptr-cast*
- `e` is a function pointer type and `U` has type `*T`, while `T: Sized`; *fptr-ptr-cast*
- `e` is a function pointer type and `U` is an integer; *fptr-addr-cast*

`transmute`

`as` only allows safe casting, and will for example reject an attempt to cast four bytes into a `u32`:

```
let a = [0u8, 0u8, 0u8, 0u8];

let b = a as u32; // four eights makes 32
```

This errors with:

```
error: non-scalar cast: `[u8; 4]` as `u32`
let b = a as u32; // four eights makes 32
        ^~~~~~~~
```

This is a 'non-scalar cast' because we have multiple values here: the four elements of the array. These kinds of casts are very dangerous, because they make assumptions about the way that multiple underlying structures are implemented. For this, we need something more dangerous.

The `transmute` function is provided by a compiler intrinsic, and what it does is very simple, but very scary. It tells Rust to treat a value of one type as though it were another type. It does this regardless of the typechecking system, and completely trusts you.

In our previous example, we know that an array of four `u8`s represents a `u32` properly, and so we want to do the cast. Using `transmute` instead of `as`, Rust lets us:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u32>(a);
}
```

We have to wrap the operation in an `unsafe` block for this to compile successfully. Technically, only the `mem::transmute` call itself needs to be in the block, but it's nice in this case to enclose everything related, so you know where to look. In this case, the details about `a` are also important, and so they're in the block. You'll see code in either style, sometimes the context is too far away, and wrapping all of the code in `unsafe` isn't a great idea.

While `transmute` does very little checking, it will at least make sure that the types are the same size. This errors:

```
use std::mem;

unsafe {
    let a = [0u8, 0u8, 0u8, 0u8];

    let b = mem::transmute::<[u8; 4], u64>(a);
}
```

with:

```
error: transmute called with differently sized types: [u8; 4] (32 bits) to u64
(64 bits)
```

Other than that, you're on your own!

4.30 Associated Types

Associated types are a powerful part of Rust's type system. They're related to the idea of a 'type family', in other words, grouping multiple types together. That description is a bit abstract, so let's dive right into an example. If you want to write a `Graph` trait, you have two types to be generic over: the node type and the edge type. So you might write a trait, `Graph<N, E>`, that looks like this:

```
trait Graph<N, E> {
    fn has_edge(&self, &N, &N) -> bool;
    fn edges(&self, &N) -> Vec<E>;
    // etc
}
```

While this sort of works, it ends up being awkward. For example, any function that wants to take a `Graph` as a parameter now also needs to be generic over the `N`ode and `E`dge types too:

```
fn distance<N, E, G: Graph<N, E>>(graph: &G, start: &N, end: &N) -> u32 {
↪  ... }
```

Our distance calculation works regardless of our `Edge` type, so the `E` stuff in this signature is a distraction.

What we really want to say is that a certain `E`dge and `N`ode type come together to form each kind of `Graph`. We can do that with associated types:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
    // etc
}
```

Now, our clients can be abstract over a given `Graph`:

```
fn distance<G: Graph>(graph: &G, start: &G::N, end: &G::N) -> u32 { ... }
```

No need to deal with the `E`dge type here!

Let's go over all this in more detail.

Defining associated types

Let's build that `Graph` trait. Here's the definition:

```
trait Graph {
    type N;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Simple enough. Associated types use the `type` keyword, and go inside the body of the trait, with the functions.

These `type` declarations can have all the same thing as functions do. For example, if we wanted our `N` type to implement `Display`, so we can print the nodes out, we could do this:

```
use std::fmt;

trait Graph {
    type N: fmt::Display;
    type E;

    fn has_edge(&self, &Self::N, &Self::N) -> bool;
    fn edges(&self, &Self::N) -> Vec<Self::E>;
}
```

Implementing associated types

Just like any trait, traits that use associated types use the `impl` keyword to provide implementations. Here's a simple implementation of Graph:

```
struct Node;

struct Edge;

struct MyGraph;

impl Graph for MyGraph {
    type N = Node;
    type E = Edge;

    fn has_edge(&self, n1: &Node, n2: &Node) -> bool {
        true
    }

    fn edges(&self, n: &Node) -> Vec<Edge> {
        Vec::new()
    }
}
```

This silly implementation always returns `true` and an empty `Vec<Edge>`, but it gives you an idea of how to implement this kind of thing. We first need three `struct`s, one for the graph, one for the node, and one for the edge. If it made more sense to use a different type, that would work as well, we're going to use `struct`s for all three here.

Next is the `impl` line, which is an implementation like any other trait.

From here, we use `=` to define our associated types. The name the trait uses goes on the left of the `=`, and the concrete type we're `impl`ementing this for goes on the right. Finally, we use the concrete types in our function declarations.

Trait objects with associated types

There's one more bit of syntax we should talk about: trait objects. If you try to create a trait object from an associated type, like this:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph>;
```

You'll get two errors:

```
error: the value of the associated type `E` (from the trait `main::Graph`) must
be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
24:44 error: the value of the associated type `N` (from the trait
`main::Graph`) must be specified [E0191]
let obj = Box::new(graph) as Box<Graph>;
          ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

We can't create a trait object like this, because we don't know the associated types. Instead, we can write this:

```
let graph = MyGraph;
let obj = Box::new(graph) as Box<Graph<N=Node, E=Edge>>;
```

The `N=Node` syntax allows us to provide a concrete type, `Node`, for the `N` type parameter. Same with `E=Edge`. If we didn't provide this constraint, we couldn't be sure which `impl` to match this trait object to.

4.31 Unsized Types

Most types have a particular size, in bytes, that is knowable at compile time. For example, an `i32` is thirty-two bits big, or four bytes. However, there are some types which are useful to express, but do not have a defined size. These are called 'unsized' or 'dynamically sized' types. One example is `[T]`. This type represents a certain number of `T` in sequence. But we don't know how many there are, so the size is not known.

Rust understands a few of these types, but they have some restrictions. There are three:

1. We can only manipulate an instance of an unsized type via a pointer. An `&[T]` works fine, but a `[T]` does not.
2. Variables and arguments cannot have dynamically sized types.
3. Only the last field in a `struct` may have a dynamically sized type; the other fields must not. Enum variants must not have dynamically sized types as data.

So why bother? Well, because `[T]` can only be used behind a pointer, if we didn't have language support for unsized types, it would be impossible to write this:

```
impl Foo for str {
```

or

```
impl<T> Foo for [T] {
```

Instead, you would have to write:

```
impl Foo for &str {
```

Meaning, this implementation would only work for references (see References and Borrowing), and not other types of pointers. With the `impl for str`, all pointers, including (at some point, there are some bugs to fix first) user-defined custom smart pointers, can use this `impl`.

?Sized

If you want to write a function that accepts a dynamically sized type, you can use the special bound, `?Sized`:

```
struct Foo<T: ?Sized> {
    f: T,
}
```

This `?`, read as "T may be `Sized`", means that this bound is special: it lets us match more kinds, not less. It's almost like every `T` implicitly has `T: Sized`, and the `?` undoes this default.

4.32 Operators and Overloading

Rust allows for a limited form of operator overloading. There are certain operators that are able to be overloaded. To support a particular operator between types, there's a specific trait that you can implement, which then overloads the operator.

For example, the `+` operator can be overloaded with the `Add` trait:

```rust
use std::ops::Add;

#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point { x: self.x + other.x, y: self.y + other.y }
    }
}

fn main() {
    let p1 = Point { x: 1, y: 0 };
    let p2 = Point { x: 2, y: 3 };

    let p3 = p1 + p2;

    println!("{:?}", p3);
}
```

In `main`, we can use `+` on our two `Point`s, since we've implemented `Add<Output=Point>` for `Point`.

There are a number of operators that can be overloaded this way, and all of their associated traits live in the std::ops module. Check out its documentation for the full list.

Implementing these traits follows a pattern. Let's look at Add in more detail:

```rust
pub trait Add<RHS = Self> {
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

There's three types in total involved here: the type you `impl Add` for, `RHS`, which defaults to `Self`, and `Output`. For an expression `let z = x + y`, `x` is the `Self` type, `y` is the `RHS`, and `z` is the `Self::Output` type.

```rust
impl Add<i32> for Point {
    type Output = f64;

    fn add(self, rhs: i32) -> f64 {
        // add an i32 to a Point and get an f64
    }
}
```

will let you do this:

```
let p: Point = // ...
let x: f64 = p + 2i32;
```

Using operator traits in generic structs

Now that we know how operator traits are defined, we can define our `HasArea` trait and `Square` struct from the traits chapter (see Traits) more generically:

```
use std::ops::Mul;

trait HasArea<T> {
    fn area(&self) -> T;
}

struct Square<T> {
    x: T,
    y: T,
    side: T,
}

impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy {
    fn area(&self) -> T {
        self.side * self.side
    }
}

fn main() {
    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 12.0f64,
    };

    println!("Area of s: {}", s.area());
}
```

For `HasArea` and `Square`, we declare a type parameter `T` and replace `f64` with it. The `impl` needs more involved modifications:

```
impl<T> HasArea<T> for Square<T>
        where T: Mul<Output=T> + Copy { ... }
```

The `area` method requires that we can multiply the sides, so we declare that type `T` must implement `std::ops::Mul`. Like `Add`, mentioned above, `Mul` itself takes an `Output` parameter: since we know that numbers don't change type when multiplied, we also set it to `T`. `T` must also support copying, so Rust doesn't try to move `self.side` into the return value.

4.33 'Deref' coercions

The standard library provides a special trait, Deref. It's normally used to overload `*`, the dereference operator:

```
use std::ops::Deref;

struct DerefExample<T> {
    value: T,
}

impl<T> Deref for DerefExample<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.value
    }
}

fn main() {
    let x = DerefExample { value: 'a' };
    assert_eq!('a', *x);
}
```

This is useful for writing custom pointer types. However, there's a language feature related to `Deref`: 'deref coercions'. Here's the rule: If you have a type `U`, and it implements `Deref<Target=T>`, values of `&U` will automatically coerce to a `&T`. Here's an example:

```
fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();

// therefore, this works:
foo(&owned);
```

Using an ampersand in front of a value takes a reference to it. So `owned` is a `String`, `&owned` is an `&String`, and since `impl Deref<Target=str> for String`, `&String` will deref to `&str`, which `foo()` takes.

That's it. This rule is one of the only places in which Rust does an automatic conversion for you, but it adds a lot of flexibility. For example, the `Rc<T>` type implements `Deref<Target=T>`, so this works:

```
use std::rc::Rc;

fn foo(s: &str) {
    // borrow a string for a second
}

// String implements Deref<Target=str>
let owned = "Hello".to_string();
let counted = Rc::new(owned);

// therefore, this works:
foo(&counted);
```

All we've done is wrap our `String` in an `Rc<T>`. But we can now pass the `Rc<String>` around anywhere we'd have a `String`. The signature of `foo` didn't change, but works just as well with either type. This example has two conversions: `Rc<String>` to `String` and then `String` to `&str`. Rust will do this as many times as possible until the types match.

Another very common implementation provided by the standard library is:

```
fn foo(s: &[i32]) {
    // borrow a slice for a second
}

// Vec<T> implements Deref<Target=[T]>
let owned = vec![1, 2, 3];

foo(&owned);
```

Vectors can `Deref` to a slice.

Deref and method calls

`Deref` will also kick in when calling a method. Consider the following example.

```
struct Foo;

impl Foo {
    fn foo(&self) { println!("Foo"); }
}

let f = &&Foo;

f.foo();
```

Even though `f` is a `&&Foo` and `foo` takes `&self`, this works. That's because these things are the same:

```
f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

A value of type `&&&&&&&&&&&&&&&&Foo` can still have methods defined on `Foo` called, because the compiler will insert as many `*` operations as necessary to get it right. And since it's inserting `*`s, that uses `Deref`.

4.34 Macros

By now you've learned about many of the tools Rust provides for abstracting and reusing code. These units of code reuse have a rich semantic structure. For example, functions have a type signature, type parameters have trait bounds, and overloaded functions must belong to a particular trait.

This structure means that Rust's core abstractions have powerful compile-time correctness checking. But this comes at the price of flexibility. If you visually identify a pattern of repeated code, you may find it's difficult or cumbersome to express that pattern as a generic function, a trait, or anything else within Rust's semantics.

Macros allow us to abstract at a syntactic level. A macro invocation is shorthand for an "expanded" syntactic form. This expansion happens early in compilation, before any static checking. As a result, macros can capture many patterns of code reuse that Rust's core abstractions cannot.

The drawback is that macro-based code can be harder to understand, because fewer of the built-in rules apply. Like an ordinary function, a well-behaved macro can be used without understanding its implementation. However, it can be difficult to design a well-behaved macro! Additionally, compiler errors in macro code are harder to interpret, because they describe problems in the expanded code, not the source-level form that developers use.

These drawbacks make macros something of a "feature of last resort". That's not to say that macros are bad; they are part of Rust because sometimes they're needed for truly concise, well-abstracted code. Just keep this tradeoff in mind.

Defining a macro

You may have seen the `vec!` macro, used to initialize a vector with any number of elements.

```
let x: Vec<u32> = vec![1, 2, 3];
```

This can't be an ordinary function, because it takes any number of arguments. But we can imagine it as syntactic shorthand for

```
let x: Vec<u32> = {
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
};
```

We can implement this shorthand, using a macro:[1]

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Whoa, that's a lot of new syntax! Let's break it down.

```
macro_rules! vec { ... }
```

This says we're defining a macro named `vec`, much as `fn vec` would define a function named `vec`. In prose, we informally write a macro's name with an exclamation point, e.g. `vec!`. The exclamation point is part of the invocation syntax and serves to distinguish a macro from an ordinary function.

Matching

The macro is defined through a series of rules, which are pattern-matching cases. Above, we had

```
( $( $x:expr ),* ) => { ... };
```

This is like a `match` expression arm, but the matching happens on Rust syntax trees, at compile time. The semicolon is optional on the last (here, only) case. The "pattern" on the left-hand side of `=>` is known as a 'matcher'. These have their own little grammar within the language.

The matcher `$x:expr` will match any Rust expression, binding that syntax tree to the 'metavariable' `$x`. The identifier `expr` is a 'fragment specifier'; the full possibilities are enumerated later

[1] The actual definition of `vec!` in libcollections differs from the one presented here, for reasons of efficiency and reusability.

in this chapter. Surrounding the matcher with `$(...),*` will match zero or more expressions, separated by commas.

Aside from the special matcher syntax, any Rust tokens that appear in a matcher must match exactly. For example,

```
macro_rules! foo {
    (x => $e:expr) => (println!("mode X: {}", $e));
    (y => $e:expr) => (println!("mode Y: {}", $e));
}

fn main() {
    foo!(y => 3);
}
```

will print

```
mode Y: 3
```

With

```
foo!(z => 3);
```

we get the compiler error

```
error: no rules expected the token `z`
```

Expansion

The right-hand side of a macro rule is ordinary Rust syntax, for the most part. But we can splice in bits of syntax captured by the matcher. From the original example:

```
$(
    temp_vec.push($x);
)*
```

Each matched expression `$x` will produce a single `push` statement in the macro expansion. The repetition in the expansion proceeds in “lockstep” with repetition in the matcher (more on this in a moment).

Because `$x` was already declared as matching an expression, we don’t repeat `:expr` on the right-hand side. Also, we don’t include a separating comma as part of the repetition operator. Instead, we have a terminating semicolon within the repeated block.

Another detail: the `vec!` macro has *two* pairs of braces on the right-hand side. They are often combined like so:

```
macro_rules! foo {
    () => {{
        ...
    }}
}
```

The outer braces are part of the syntax of `macro_rules!`. In fact, you can use `()` or `[]` instead. They simply delimit the right-hand side as a whole.

The inner braces are part of the expanded syntax. Remember, the `vec!` macro is used in an expression context. To write an expression with multiple statements, including `let`-bindings, we use a block. If your macro expands to a single expression, you don't need this extra layer of braces.

Note that we never *declared* that the macro produces an expression. In fact, this is not determined until we use the macro as an expression. With care, you can write a macro whose expansion works in several contexts. For example, shorthand for a data type could be valid as either an expression or a pattern.

Repetition

The repetition operator follows two principal rules:

1. `$(...)*` walks through one "layer" of repetitions, for all of the `$names` it contains, in lockstep, and
2. each `$name` must be under at least as many `$(...)*`s as it was matched against. If it is under more, it'll be duplicated, as appropriate.

This baroque macro illustrates the duplication of variables from outer repetition levels.

```
macro_rules! o_O {
    (
        $(
            $x:expr; [ $( $y:expr ),* ]
        );*
    ) => {
        &[ $($( $x + $y ),*),* ]
    }
}

fn main() {
    let a: &[i32]
        = o_O!(10; [1, 2, 3];
               20; [4, 5, 6]);

    assert_eq!(a, [11, 12, 13, 24, 25, 26]);
}
```

That's most of the matcher syntax. These examples use `$(...)*`, which is a "zero or more" match. Alternatively you can write `$(...)+` for a "one or more" match. Both forms optionally include a separator, which can be any token except `+` or `*`.

This system is based on Macro-by-Example (PDF link).

Hygiene

Some languages implement macros using simple text substitution, which leads to various problems. For example, this C program prints `13` instead of the expected `25`.

```
#define FIVE_TIMES(x) 5 * x

int main() {
    printf("%d\n", FIVE_TIMES(2 + 3));
    return 0;
}
```

After expansion we have `5 * 2 + 3`, and multiplication has greater precedence than addition. If you've used C macros a lot, you probably know the standard idioms for avoiding this problem, as well as five or six others. In Rust, we don't have to worry about it.

```
macro_rules! five_times {
    ($x:expr) => (5 * $x);
}

fn main() {
    assert_eq!(25, five_times!(2 + 3));
}
```

The metavariable `$x` is parsed as a single expression node, and keeps its place in the syntax tree even after substitution.

Another common problem in macro systems is 'variable capture'. Here's a C macro, using a GNU C extension to emulate Rust's expression blocks.

```
#define LOG(msg) ({ \
    int state = get_log_state(); \
    if (state > 0) { \
        printf("log(%d): %s\n", state, msg); \
    } \
})
```

Here's a simple use case that goes terribly wrong:

```
const char *state = "reticulating splines";
LOG(state)
```

This expands to

```
const char *state = "reticulating splines";
{
    int state = get_log_state();
    if (state > 0) {
        printf("log(%d): %s\n", state, state);
    }
}
```

The second variable named `state` shadows the first one. This is a problem because the print statement should refer to both of them.

The equivalent Rust macro has the desired behavior.

```
macro_rules! log {
    ($msg:expr) => {{
        let state: i32 = get_log_state();
        if state > 0 {
            println!("log({}): {}", state, $msg);
        }
    }};
}

fn main() {
    let state: &str = "reticulating splines";
    log!(state);
}
```

This works because Rust has a hygienic macro system. Each macro expansion happens in a distinct 'syntax context', and each variable is tagged with the syntax context where it was introduced. It's as though the variable `state` inside `main` is painted a different "color" from the variable state inside the macro, and therefore they don't conflict.

This also restricts the ability of macros to introduce new bindings at the invocation site. Code such as the following will not work:

```
macro_rules! foo {
    () => (let x = 3);
}

fn main() {
    foo!();
    println!("{}", x);
}
```

Instead you need to pass the variable name into the invocation, so it's tagged with the right syntax context.

```
macro_rules! foo {
    ($v:ident) => (let $v = 3);
}

fn main() {
    foo!(x);
    println!("{}", x);
}
```

This holds for `let` bindings and loop labels, but not for items. So the following code does compile:

```
macro_rules! foo {
    () => (fn x() { });
}

fn main() {
    foo!();
    x();
}
```

Recursive macros

A macro's expansion can include more macro invocations, including invocations of the very same macro being expanded. These recursive macros are useful for processing tree-structured input, as illustrated by this (simplistic) HTML shorthand:

```
macro_rules! write_html {
    ($w:expr, ) => (());

    ($w:expr, $e:tt) => (write!($w, "{}", $e));

    ($w:expr, $tag:ident [ $($inner:tt)* ] $($rest:tt)*) => {{
        write!($w, "<{}>", stringify!($tag));
        write_html!($w, $($inner)*);
        write!($w, "</{}>", stringify!($tag));
        write_html!($w, $($rest)*);
    }};
}

fn main() {
    use std::fmt::Write;
    let mut out = String::new();

    write_html!(&mut out,
        html[
            head[title["Macros guide"]]
            body[h1["Macros are the best!"]]
        ]);

    assert_eq!(out,
        "<html><head><title>Macros guide</title></head>\
         <body><h1>Macros are the best!</h1></body></html>");
}
```

Debugging macro code

To see the results of expanding macros, run `rustc --pretty expanded`. The output represents a whole crate, so you can also feed it back in to `rustc`, which will sometimes produce better error messages than the original compilation. Note that the `--pretty expanded` output may have a different meaning if multiple variables of the same name (but different syntax contexts) are in play in the same scope. In this case `--pretty expanded,hygiene` will tell you about the syntax contexts.

`rustc` provides two syntax extensions that help with macro debugging. For now, they are unstable and require feature gates.

- `log_syntax!(...)` will print its arguments to standard output, at compile time, and "expand" to nothing.
- `trace_macros!(true)` will enable a compiler message every time a macro is expanded. Use `trace_macros!(false)` later in expansion to turn it off.

Syntactic requirements

Even when Rust code contains un-expanded macros, it can be parsed as a full syntax tree (see Abstract Syntax Tree). This property can be very useful for editors and other tools that process code. It also has a few consequences for the design of Rust's macro system.

One consequence is that Rust must determine, when it parses a macro invocation, whether the macro stands in for

- zero or more items,
- zero or more methods,
- an expression,
- a statement, or
- a pattern.

A macro invocation within a block could stand for some items, or for an expression / statement. Rust uses a simple rule to resolve this ambiguity. A macro invocation that stands for items must be either

- delimited by curly braces, e.g. `foo! { ... }`, or
- terminated by a semicolon, e.g. `foo!(...);`

Another consequence of pre-expansion parsing is that the macro invocation must consist of valid Rust tokens. Furthermore, parentheses, brackets, and braces must be balanced within a macro invocation. For example, `foo!([)` is forbidden. This allows Rust to know where the macro invocation ends.

More formally, the macro invocation body must be a sequence of 'token trees'. A token tree is defined recursively as either

- a sequence of token trees surrounded by matching `()`, `[]`, or `{}`, or
- any other single token.

Within a matcher, each metavariable has a 'fragment specifier', identifying which syntactic form it matches.

- `ident`: an identifier. Examples: `x`; `foo`.
- `path`: a qualified name. Example: `T::SpecialA`.
- `expr`: an expression. Examples: `2 + 2`; `if true { 1 } else { 2 }`; `f(42)`.
- `ty`: a type. Examples: `i32`; `Vec<(char, String)>`; `&T`.

- `pat`: a pattern. Examples: `Some(t)`; `(17, 'a')`; `_`.
- `stmt`: a single statement. Example: `let x = 3`.
- `block`: a brace-delimited sequence of statements. Example: `{ log(error, "hi"); return 12; }`.
- `item`: an item. Examples: `fn foo() { }`; `struct Bar;`.
- `meta`: a "meta item", as found in attributes. Example: `cfg(target_os = "windows")`.
- `tt`: a single token tree.

There are additional rules regarding the next token after a metavariable:

- `expr` and `stmt` variables may only be followed by one of: `=> , ;`
- `ty` and `path` variables may only be followed by one of: `=> , = | ; : > [ { as where`
- `pat` variables may only be followed by one of: `=> , = | if in`
- Other variables may be followed by any token.

These rules provide some flexibility for Rust's syntax to evolve without breaking existing macros.

The macro system does not deal with parse ambiguity at all. For example, the grammar `$($i:ident)* $e:expr` will always fail to parse, because the parser would be forced to choose between parsing `$i` and parsing `$e`. Changing the invocation syntax to put a distinctive token in front can solve the problem. In this case, you can write `$(I $i:ident)* E $e:expr`.

Scoping and macro import/export

Macros are expanded at an early stage in compilation, before name resolution. One downside is that scoping works differently for macros, compared to other constructs in the language.

Definition and expansion of macros both happen in a single depth-first, lexical-order traversal of a crate's source. So a macro defined at module scope is visible to any subsequent code in the same module, which includes the body of any subsequent child `mod` items.

A macro defined within the body of a single `fn`, or anywhere else not at module scope, is visible only within that item.

If a module has the `macro_use` attribute, its macros are also visible in its parent module after the child's `mod` item. If the parent also has `macro_use` then the macros will be visible in the grandparent after the parent's `mod` item, and so forth.

The macro_use attribute can also appear on extern crate. In this context it controls which macros are loaded from the external crate, e.g.

```
#[macro_use(foo, bar)]
extern crate baz;
```

If the attribute is given simply as #[macro_use], all macros are loaded. If there is no #[macro_use] attribute then no macros are loaded. Only macros defined with the #[macro_export] attribute may be loaded.

To load a crate's macros without linking it into the output, use #[no_link] as well.

An example:

```
macro_rules! m1 { () => (()) }

// visible here: m1

mod foo {
    // visible here: m1

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // visible here: m1, m2
}

// visible here: m1

macro_rules! m3 { () => (()) }

// visible here: m1, m3

#[macro_use]
mod bar {
    // visible here: m1, m3

    macro_rules! m4 { () => (()) }

    // visible here: m1, m3, m4
}

// visible here: m1, m3, m4
```

When this library is loaded with #[macro_use] extern crate, only m2 will be imported.

The Rust Reference has a listing of macro-related attributes.

The variable $crate

A further difficulty occurs when a macro is used in multiple crates. Say that mylib defines

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc_a {
    ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
    ($x:expr) => ( ::mylib::increment($x) )
}
```

`inc_a` only works within `mylib`, while `inc_b` only works outside the library. Furthermore, `inc_b` will break if the user imports `mylib` under another name.

Rust does not (yet) have a hygiene system for crate references, but it does provide a simple workaround for this problem. Within a macro imported from a crate named foo, the special macro variable `$crate` will expand to `::foo`. By contrast, when a macro is defined and then used in the same crate, `$crate` will expand to nothing. This means we can write

```
#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
```

to define a single macro that works both inside and outside our library. The function name will expand to either `::increment` or `::mylib::increment`.

To keep this system simple and correct, `#[macro_use] extern crate ...` may only appear at the root of your crate, not inside `mod`.

The deep end

The introductory chapter mentioned recursive macros, but it did not give the full story. Recursive macros are useful for another reason: Each recursive invocation gives you another opportunity to pattern-match the macro's arguments.

As an extreme example, it is possible, though hardly advisable, to implement the Bitwise Cyclic Tag automaton within Rust's macro system.

```
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));

    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));

    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));

    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
```

Exercise: use macros to reduce duplication in the above definition of the `bct!` macro.

Common macros

Here are some common macros you'll see in Rust code.

panic!

This macro causes the current thread to panic. You can give it a message to panic with:

```
panic!("oh no!");
```

vec!

The `vec!` macro is used throughout the book, so you've probably seen it already. It creates `Vec<T>`s with ease:

```
let v = vec![1, 2, 3, 4, 5];
```

It also lets you make vectors with repeating values. For example, a hundred zeroes:

```
let v = vec![0; 100];
```

assert! and assert_eq!

These two macros are used in tests. `assert!` takes a boolean. `assert_eq!` takes two values and checks them for equality. `true` passes, `false` `panic!`s. Like this:

```
// A-ok!

assert!(true);
assert_eq!(5, 3 + 2);

// nope :(

assert!(5 < 3);
assert_eq!(5, 3);
```

try!

`try!` is used for error handling. It takes something that can return a `Result<T, E>`, and gives `T` if it's a `Ok<T>`, and returns with the `Err(E)` if it's that. Like this:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = try!(File::create("foo.txt"));

    Ok(())
}
```

This is cleaner than doing this:

```
use std::fs::File;

fn foo() -> std::io::Result<()> {
    let f = File::create("foo.txt");

    let f = match f {
        Ok(t) => t,
        Err(e) => return Err(e),
    };

    Ok(())
}
```

unreachable!

This macro is used when you think some code should never execute:

```
if false {
    unreachable!();
}
```

Sometimes, the compiler may make you have a different branch that you know will never, ever run. In these cases, use this macro, so that if you end up wrong, you'll get a `panic!` about it.

```
let x: Option<i32> = None;

match x {
    Some(_) => unreachable!(),
    None => println!("I know x is None!"),
}
```

unimplemented!

The `unimplemented!` macro can be used when you're trying to get your functions to typecheck, and don't want to worry about writing out the body of the function. One example of this situation is implementing a trait with multiple required methods, where you want to tackle one at a time. Define the others as `unimplemented!` until you're ready to write them.

Procedural macros

If Rust's macro system can't do what you need, you may want to write a compiler plugin instead. Compared to `macro_rules!` macros, this is significantly more work, the interfaces are much less stable, and bugs can be much harder to track down. In exchange you get the flexibility of running arbitrary Rust code within the compiler. Syntax extension plugins are sometimes called 'procedural macros' for this reason.

4.35 Raw Pointers

Rust has a number of different smart pointer types in its standard library, but there are two types that are extra-special. Much of Rust's safety comes from compile-time checks, but raw pointers don't have such guarantees, and are unsafe to use.

`*const T` and `*mut T` are called 'raw pointers' in Rust. Sometimes, when writing certain kinds of libraries, you'll need to get around Rust's safety guarantees for some reason. In this case, you can use raw pointers to implement your library, while exposing a safe interface for your users. For example, `*` pointers are allowed to alias, allowing them to be used to write shared-ownership types, and even thread-safe shared memory types (`the Rc<T>` and `Arc<T>` types are both implemented entirely in Rust).

Here are some things to remember about raw pointers that are different than other pointer types. They:

- are not guaranteed to point to valid memory and are not even guaranteed to be non-null (unlike both `Box` and `&`);
- do not have any automatic clean-up, unlike `Box`, and so require manual resource management;
- are plain-old-data, that is, they don't move ownership, again unlike `Box`, hence the Rust compiler cannot protect against bugs like use-after-free;
- lack any form of lifetimes, unlike `&`, and so the compiler cannot reason about dangling pointers; and
- have no guarantees about aliasing or mutability other than mutation not being allowed directly through a `*const T`.

Basics

Creating a raw pointer is perfectly safe:

```
let x = 5;
let raw = &x as *const i32;

let mut y = 10;
let raw_mut = &mut y as *mut i32;
```

However, dereferencing one is not. This won't work:

```
let x = 5;
let raw = &x as *const i32;

println!("raw points at {}", *raw);
```

It gives this error:

```
error: dereference of raw pointer requires unsafe function or block [E0133]
     println!("raw points at {}", *raw);
                                  ^~~~
```

When you dereference a raw pointer, you're taking responsibility that it's not pointing somewhere that would be incorrect. As such, you need `unsafe`:

```
let x = 5;
let raw = &x as *const i32;

let points_at = unsafe { *raw };

println!("raw points at {}", points_at);
```

For more operations on raw pointers, see their API documentation.

FFI

Raw pointers are useful for FFI: Rust's `*const T` and `*mut T` are similar to C's `const T*` and `T*`, respectively. For more about this use, consult the FFI chapter.

References and raw pointers

At runtime, a raw pointer `*` and a reference pointing to the same piece of data have an identical representation. In fact, an `&T` reference will implicitly coerce to an `*const T` raw pointer in safe code and similarly for the `mut` variants (both coercions can be performed explicitly with, respectively, `value as *const T` and `value as *mut T`).

Going the opposite direction, from `*const` to a reference `&`, is not safe. A `&T` is always valid, and so, at a minimum, the raw pointer `*const T` has to point to a valid instance of type `T`. Furthermore, the resulting pointer must satisfy the aliasing and mutability laws of references. The compiler assumes these properties are true for any references, no matter how they are created, and so any conversion from raw pointers is asserting that they hold. The programmer *must* guarantee this.

The recommended method for the conversion is:

```
// explicit cast
let i: u32 = 1;
let p_imm: *const u32 = &i as *const u32;

// implicit coercion
let mut m: u32 = 2;
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

The `&*x` dereferencing style is preferred to using a `transmute`. The latter is far more powerful than necessary, and the more restricted operation is harder to use incorrectly; for example, it requires that `x` is a pointer (unlike `transmute`).

4.36 Unsafe

Rust's main draw is its powerful static guarantees about behavior. But safety checks are conservative by nature: there are some programs that are actually safe, but the compiler is not able to verify this is true. To write these kinds of programs, we need to tell the compiler to relax its restrictions a bit. For this, Rust has a keyword, `unsafe`. Code using `unsafe` has less restrictions than normal code does.

Let's go over the syntax, and then we'll talk semantics. `unsafe` is used in four contexts. The first one is to mark a function as unsafe:

```
unsafe fn danger_will_robinson() {
    // scary stuff
}
```

All functions called from FFI must be marked as `unsafe`, for example. The second use of `unsafe` is an unsafe block:

```
unsafe {
    // scary stuff
}
```

The third is for unsafe traits:

```
unsafe trait Scary { }
```

And the fourth is for `impl`ementing one of those traits:

```
unsafe impl Scary for i32 {}
```

It's important to be able to explicitly delineate code that may have bugs that cause big problems. If a Rust program segfaults, you can be sure the cause is related to something marked `unsafe`.

What does 'safe' mean?

Safe, in the context of Rust, means 'doesn't do anything unsafe'. It's also important to know that there are certain behaviors that are probably not desirable in your code, but are expressly *not* unsafe:

- Deadlocks
- Leaks of memory or other resources
- Exiting without calling destructors
- Integer overflow

Rust cannot prevent all kinds of software problems. Buggy code can and will be written in Rust. These things aren't great, but they don't qualify as `unsafe` specifically.

In addition, the following are all undefined behaviors in Rust, and must be avoided, even when writing `unsafe` code:

- Data races
- Dereferencing a null/dangling raw pointer
- Reads of undef (uninitialized) memory

- Breaking the pointer aliasing rules with raw pointers.
- `&mut T` and `&T` follow LLVM's scoped noalias model, except if the `&T` contains an `UnsafeCell<U>`. Unsafe code must not violate these aliasing guarantees.
- Mutating an immutable value/reference without `UnsafeCell<U>`
- Invoking undefined behavior via compiler intrinsics:
  - Indexing outside of the bounds of an object with `std::ptr::offset` (`offset` intrinsic), with the exception of one byte past the end which is permitted.
  - Using `std::ptr::copy_nonoverlapping_memory` (`memcpy32/memcpy64` intrinsics) on overlapping buffers
- Invalid values in primitive types, even in private fields/locals:
  - Null/dangling references or boxes
  - A value other than `false` (0) or `true` (1) in a `bool`
  - A discriminant in an `enum` not included in its type definition
  - A value in a `char` which is a surrogate or above `char::MAX`
  - Non-UTF-8 byte sequences in a `str`
- Unwinding into Rust from foreign code or unwinding from Rust into foreign code.

Unsafe Superpowers

In both unsafe functions and unsafe blocks, Rust will let you do three things that you normally can not do. Just three. Here they are:

- Access or update a static mutable variable (see `static`.
- Dereference a raw pointer.
- Call unsafe functions. This is the most powerful ability.

That's it. It's important that `unsafe` does not, for example, 'turn off the borrow checker'. Adding `unsafe` to some random Rust code doesn't change its semantics, it won't start accepting anything. But it will let you write things that do break some of the rules.

You will also encounter the `unsafe` keyword when writing bindings to foreign (non-Rust) interfaces. You're encouraged to write a safe, native Rust interface around the methods provided by the library.

Let's go over the basic three abilities listed, in order.

Access or update a `static mut`

Rust has a feature called '`static mut`' which allows for mutable global state. Doing so can cause a data race, and as such is inherently not safe. For more details, see the static section of the book (see `static`).

Dereference a raw pointer

Raw pointers let you do arbitrary pointer arithmetic, and can cause a number of different memory safety and security issues. In some senses, the ability to dereference an arbitrary pointer is one of the most dangerous things you can do. For more on raw pointers, see their section of the book (Raw Pointers).

Call unsafe functions

This last ability works with both aspects of `unsafe`: you can only call functions marked `unsafe` from inside an unsafe block.

This ability is powerful and varied. Rust exposes some compiler intrinsics as unsafe functions, and some unsafe functions bypass safety checks, trading safety for speed.

I'll repeat again: even though you *can do* arbitrary things in unsafe blocks and functions doesn't mean you should. The compiler will act as though you're upholding its invariants, so be careful!

5 Effective Rust

So you've learned how to write some Rust code. But there's a difference between writing any Rust code and writing good Rust code.

This chapter consists of relatively independent tutorials which show you how to take your Rust to the next level. Common patterns and standard library features will be introduced. Read these sections in any order of your choosing.

5.1 The Stack and the Heap

As a systems language, Rust operates at a low level. If you're coming from a high-level language, there are some aspects of systems programming that you may not be familiar with. The most important one is how memory works, with a stack and a heap. If you're familiar with how C-like languages use stack allocation, this chapter will be a refresher. If you're not, you'll learn about this more general concept, but with a Rust-y focus.

As with most things, when learning about them, we'll use a simplified model to start. This lets you get a handle on the basics, without getting bogged down with details which are, for now, irrelevant. The examples we'll use aren't 100% accurate, but are representative for the level we're trying to learn at right now. Once you have the basics down, learning more about how allocators are implemented, virtual memory, and other advanced topics will reveal the leaks in this particular abstraction.

Memory management

These two terms are about memory management. The stack and the heap are abstractions that help you determine when to allocate and deallocate memory.

Here's a high-level comparison:

The stack is very fast, and is where memory is allocated in Rust by default. But the allocation is local to a function call, and is limited in size. The heap, on the other hand, is slower, and is explicitly allocated by your program. But it's effectively unlimited in size, and is globally accessible.

The Stack

Let's talk about this Rust program:

```
fn main() {
    let x = 42;
}
```

This program has one variable binding, `x`. This memory needs to be allocated from somewhere. Rust 'stack allocates' by default, which means that basic values 'go on the stack'. What does that mean?

Well, when a function gets called, some memory gets allocated for all of its local variables and some other information. This is called a 'stack frame', and for the purpose of this tutorial, we're going to ignore the extra information and only consider the local variables we're allocating. So in this case, when `main()` is run, we'll allocate a single 32-bit integer for our stack frame. This is automatically handled for you, as you can see; we didn't have to write any special Rust code or anything.

When the function exits, its stack frame gets deallocated. This happens automatically as well.

That's all there is for this simple program. The key thing to understand here is that stack allocation is very, very fast. Since we know all the local variables we have ahead of time, we can grab the memory all at once. And since we'll throw them all away at the same time as well, we can get rid of it very fast too.

The downside is that we can't keep values around if we need them for longer than a single function. We also haven't talked about what the word, 'stack', means. To do that, we need a slightly more complicated example:

```
fn foo() {
    let y = 5;
    let z = 100;
}

fn main() {
    let x = 42;

    foo();
}
```

This program has three variables total: two in `foo()`, one in `main()`. Just as before, when `main()` is called, a single integer is allocated for its stack frame. But before we can show what happens when `foo()` is called, we need to visualize what's going on with memory. Your operating system presents a view of memory to your program that's pretty simple: a huge list of addresses, from 0 to a large number, representing how much RAM your computer has. For example, if you have a gigabyte of RAM, your addresses go from `0` to `1,073,741,823`. That number comes from 2^{30}, the number of bytes in a gigabyte.[2]

This memory is kind of like a giant array: addresses start at zero and go up to the final number. So here's a diagram of our first stack frame:

[2]'Gigabyte' can mean two things: 10^9, or 2^{30}. The SI standard resolved this by stating that 'gigabyte' is 10^9, and 'gibibyte' is 2^{30}. However, very few people use this terminology, and rely on context to differentiate. We follow in that tradition here.

Address	Name	Value
0	x	42

We've got `x` located at address `0`, with the value `42`.
When `foo()` is called, a new stack frame is allocated:

Address	Name	Value
2	z	100
1	y	5
0	x	42

Because `0` was taken by the first frame, `1` and `2` are used for `foo()`'s stack frame. It grows upward, the more functions we call.

There are some important things we have to take note of here. The numbers 0, 1, and 2 are all solely for illustrative purposes, and bear no relationship to the address values the computer will use in reality. In particular, the series of addresses are in reality going to be separated by some number of bytes that separate each address, and that separation may even exceed the size of the value being stored.

After `foo()` is over, its frame is deallocated:

Address	Name	Value
0	x	42

And then, after `main()`, even this last value goes away. Easy!

It's called a 'stack' because it works like a stack of dinner plates: the first plate you put down is the last plate to pick back up. Stacks are sometimes called 'last in, first out queues' for this reason, as the last value you put on the stack is the first one you retrieve from it.

Let's try a three-deep example:

```
fn italic() {
    let i = 6;
}

fn bold() {
    let a = 5;
    let b = 100;
    let c = 1;

    italic();
}

fn main() {
    let x = 42;

    bold();
}
```

We have some kooky function names to make the diagrams clearer.

Okay, first, we call `main()`:

Address	Name	Value
0	x	42

Next up, `main()` calls `bold()`:

Address	Name	Value
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

And then `bold()` calls `italic()`:

Address	Name	Value
4	*i*	*6*
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

Whew! Our stack is growing tall.

After `italic()` is over, its frame is deallocated, leaving only `bold()` and `main()`:

Address	Name	Value
3	**c**	**1**
2	**b**	**100**
1	**a**	**5**
0	x	42

And then `bold()` ends, leaving only `main()`:

Address	Name	Value
0	x	42

And then we're done. Getting the hang of it? It's like piling up dishes: you add to the top, you take away from the top.

The Heap

Now, this works pretty well, but not everything can work like this. Sometimes, you need to pass some memory between different functions, or keep it alive for longer than a single function's execution. For this, we can use the heap.

In Rust, you can allocate memory on the heap with the Box<T> type. Here's an example:

```
fn main() {
    let x = Box::new(5);
    let y = 42;
}
```

Here's what happens in memory when `main()` is called:

Address	Name	Value
1	y	42
0	x	??????

We allocate space for two variables on the stack. `y` is `42`, as it always has been, but what about `x`? Well, `x` is a `Box<i32>`, and boxes allocate memory on the heap. The actual value of the box is a structure which has a pointer to 'the heap'. When we start executing the function, and

`Box::new()` is called, it allocates some memory for the heap, and puts `5` there. The memory now looks like this:

Address	Name	Value
(2^{30}) - 1		5
...	...	...
1	y	42
0	x	42

We have (2^{30}) - 1 addresses in our hypothetical computer with 1GB of RAM. And since our stack grows from zero, the easiest place to allocate memory is from the other end. So our first value is at the highest place in memory. And the value of the struct at `x` has a raw pointer to the place we've allocated on the heap, so the value of `x` is (2^{30}) - 1, the memory location we've asked for.

We haven't really talked too much about what it actually means to allocate and deallocate memory in these contexts. Getting into very deep detail is out of the scope of this tutorial, but what's important to point out here is that the heap isn't a stack that grows from the opposite end. We'll have an example of this later in the book, but because the heap can be allocated and freed in any order, it can end up with 'holes'. Here's a diagram of the memory layout of a program which has been running for a while now:

Address	Name	Value
(2^{30}) - 1		5
(2^{30}) - 2		
(2^{30}) - 3		
(2^{30}) - 4		40
...	...	...
3	y	$\rightarrow (2^{30})$-4
2	y	42
1	y	42
0	x	$\rightarrow (2^{30})$-1

In this case, we've allocated four things on the heap, but deallocated two of them. There's a gap between (2^{30}) - 1 and (2^{30}) - 4 which isn't currently being used. The specific details of how and why this happens depends on what kind of strategy you use to manage the heap. Different programs can use different 'memory allocators', which are libraries that manage this for you. Rust programs use jemalloc for this purpose.

Anyway, back to our example. Since this memory is on the heap, it can stay alive longer than the function which allocates the box. In this case, however, it doesn't.[3] When the function is over, we need to free the stack frame for `main()`. `Box<T>`, though, has a trick up its sleeve: Drop (see Drop). The implementation of `Drop` for `Box` deallocates the memory that was allocated when it was created. Great! So when `x` goes away, it first frees the memory allocated on the heap:

Address	Name	Value
1	y	42
0	x	??????

And then the stack frame goes away, freeing all of our memory.

Arguments and borrowing

We've got some basic examples with the stack and the heap going, but what about function arguments and borrowing? Here's a small Rust program:

```
fn foo(i: &i32) {
    let z = 42;
}

fn main() {
    let x = 5;
    let y = &x;

    foo(y);
}
```

When we enter `main()`, memory looks like this:

Address	Name	Value
1	y	$\rightarrow 0$
0	x	5

`x` is a plain old `5`, and `y` is a reference to `x`. So its value is the memory location that `x` lives at, which in this case is `0`.

What about when we call `foo()`, passing `y` as an argument?

[3]We can make the memory live longer by transferring ownership, sometimes called 'moving out of the box'. More complex examples will be covered later.

Address	Name	Value
3	z	42
2	i	$\rightarrow 0$
1	y	$\rightarrow 0$
0	x	5

Stack frames aren't only for local bindings, they're for arguments too. So in this case, we need to have both `i`, our argument, and `z`, our local variable binding. `i` is a copy of the argument, `y`. Since `y`'s value is `0`, so is `i`'s.

This is one reason why borrowing a variable doesn't deallocate any memory: the value of a reference is a pointer to a memory location. If we got rid of the underlying memory, things wouldn't work very well.

A complex example

Okay, let's go through this complex program step-by-step:

```
fn foo(x: &i32) {
    let y = 10;
    let z = &y;

    baz(z);
    bar(x, z);
}

fn bar(a: &i32, b: &i32) {
    let c = 5;
    let d = Box::new(5);
    let e = &d;

    baz(e);
}

fn baz(f: &i32) {
    let g = 100;
}

fn main() {
    let h = 3;
    let i = Box::new(20);
    let j = &h;

    foo(j);
}
```

First, we call `main()`:

Address	Name	Value
(2^{30}) - 1		20
…	…	…
2	j	$\rightarrow$ 0
1	i	$\rightarrow$ (2^{30})-1
0	h	3

We allocate memory for `j`, `i`, and `h`. `i` is on the heap, and so has a value pointing there.

Next, at the end of `main()`, `foo()` gets called:

Address	Name	Value
(2^{30}) - 1		20
…	…	…
5	z	$\rightarrow$ 4
4	y	10
3	x	$\rightarrow$ 0
2	j	$\rightarrow$ 0
1	i	$\rightarrow$ (2^{30})-1
0	h	3

Space gets allocated for `x`, `y`, and `z`. The argument `x` has the same value as `j`, since that's what we passed it in. It's a pointer to the `0` address, since `j` points at `h`.

Next, `foo()` calls `baz()`, passing `z`:

Address	**Name**	**Value**
(2^{30}) - 1		20
…	…	…
7	g	100
6	f	$\rightarrow$ 4
5	z	$\rightarrow$ 4
4	y	10
3	x	$\rightarrow$ 0
2	j	$\rightarrow$ 0
1	i	$\rightarrow$ (2^{30})-1
0	h	3

We've allocated memory for `f` and `g`. `baz()` is very short, so when it's over, we get rid of its stack frame:

Address	**Name**	**Value**
(2^{30}) - 1		20
…	…	…
5	z	$\rightarrow$ 4
4	y	10
3	x	$\rightarrow$ 0
2	j	$\rightarrow$ 0
1	i	$\rightarrow$ (2^{30})-1
0	h	3

Next, `foo()` calls `bar()` with `x` and `z`:

Address	**Name**	**Value**
(2^{30}) - 1		20
(2^{30}) - 2		5
…	…	…
10	e	$\rightarrow$ 9
9	d	$\rightarrow (2^{30})$-2
8	c	5
7	b	$\rightarrow$ 4
6	a	$\rightarrow$ 0
5	z	$\rightarrow$ 4
4	y	10
3	x	$\rightarrow$ 0
2	j	$\rightarrow$ 0
1	i	$\rightarrow (2^{30})$-1
0	h	3

We end up allocating another value on the heap, and so we have to subtract one from (2^{30}) - 1. It's easier to write that than `1,073,741,822`. In any case, we set up the variables as usual.

At the end of `bar()`, it calls `baz()`:

Address	**Name**	**Value**
(2^{30}) - 1		20
(2^{30}) - 2		5
… 12	… g	… 100
11	f	$\rightarrow (2^{30})$-2
10	e	$\rightarrow$ 9
9	d	$\rightarrow (2^{30})$-2
8	c	5
7	b	$\rightarrow$ 4
6	a	$\rightarrow$ 0
5	z	$\rightarrow$ 4
4	y	10
3	x	$\rightarrow$ 0
2	j	$\rightarrow$ 0
1	i	$\rightarrow (2^{30})$-1
0	h	3

With this, we're at our deepest point! Whew! Congrats for following along this far.

After `baz()` is over, we get rid of `f` and `g`:

Address	Name	Value
(2^{30}) - 1		20
(2^{30}) - 2		5
…	…	…
10	e	$\rightarrow$ 9
9	d	$\rightarrow$ (2^{30})-2
8	c	5
7	b	$\rightarrow$ 4
6	a	$\rightarrow$ 0
5	z	$\rightarrow$ 4
4	y	10
3	x	$\rightarrow$ 0
2	j	$\rightarrow$ 0
1	i	$\rightarrow$ (2^{30})-1
0	h	3

Next, we return from `bar()`. `d` in this case is a `Box<T>`, so it also frees what it points to: (2^{30}) - 2.

Address	Name	Value
(2^{30}) - 1		20
…	…	…
5	z	$\rightarrow$ 4
4	y	10
3	x	$\rightarrow$ 0
2	j	$\rightarrow$ 0
1	i	$\rightarrow$ (2^{30})-1
0	h	3

And after that, `foo()` returns:

Address	Name	Value
(2^{30}) - 1		20
…	…	…
2	j	$\rightarrow$ 0
1	i	$\rightarrow$ (2^{30})-1
0	h	3

And then, finally, `main()`, which cleans the rest up. When `i` is `Drop`ped, it will clean up the last of the heap too.

What do other languages do?

Most languages with a garbage collector heap-allocate by default. This means that every value is boxed. There are a number of reasons why this is done, but they're out of scope for this tutorial. There are some possible optimizations that don't make it true 100% of the time, too. Rather than relying on the stack and `Drop` to clean up memory, the garbage collector deals with the heap instead.

Which to use?

So if the stack is faster and easier to manage, why do we need the heap? A big reason is that Stack-allocation alone means you only have 'Last In First Out (LIFO)' semantics for reclaiming storage. Heap-allocation is strictly more general, allowing storage to be taken from and returned to the pool in arbitrary order, but at a complexity cost.

Generally, you should prefer stack allocation, and so, Rust stack-allocates by default. The LIFO model of the stack is simpler, at a fundamental level. This has two big impacts: runtime efficiency and semantic impact.

Runtime Efficiency

Managing the memory for the stack is trivial: The machine increments or decrements a single value, the so-called "stack pointer". Managing memory for the heap is non-trivial: heap-allocated memory is freed at arbitrary points, and each block of heap-allocated memory can be of arbitrary size, so the memory manager must generally work much harder to identify memory for reuse.

If you'd like to dive into this topic in greater detail, this paper is a great introduction.

Semantic impact

Stack-allocation impacts the Rust language itself, and thus the developer's mental model. The LIFO semantics is what drives how the Rust language handles automatic memory management. Even the deallocation of a uniquely-owned heap-allocated box can be driven by the stack-based LIFO semantics, as discussed throughout this chapter. The flexibility (i.e. expressiveness) of non LIFO-semantics means that in general the compiler cannot automatically infer at compile-time where memory should be freed; it has to rely on dynamic protocols, potentially from outside the language itself, to drive deallocation (reference counting, as used by `Rc<T>` and `Arc<T>`, is one example of this).

When taken to the extreme, the increased expressive power of heap allocation comes at the cost of either significant runtime support (e.g. in the form of a garbage collector) or significant programmer effort (in the form of explicit memory management calls that require verification not provided by the Rust compiler).

5.2 Testing

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
>
> Edsger W. Dijkstra, "The Humble Programmer" (1972)

Let's talk about how to test Rust code. What we will not be talking about is the right way to test Rust code. There are many schools of thought regarding the right and wrong way to write tests. All of these approaches use the same basic tools, and so we'll show you the syntax for using them.

The `test` attribute

At its simplest, a test in Rust is a function that's annotated with the `test` attribute. Let's make a new project with Cargo called `adder`:

```
$ cargo new adder
$ cd adder
```

Cargo will automatically generate a simple test when you make a new project. Here's the contents of `src/lib.rs`:

```
#[test]
fn it_works() {
}
```

Note the `#[test]`. This attribute indicates that this is a test function. It currently has no body. That's good enough to pass! We can run the tests with `cargo test`:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Cargo compiled and ran our tests. There are two sets of output here: one for the test we wrote, and another for documentation tests. We'll talk about those later. For now, see this line:

```
test it_works ... ok
```

Note the `it_works`. This comes from the name of our function:

```
fn it_works() {
```

We also get a summary line:

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

So why does our do-nothing test pass? Any test which doesn't `panic!` passes, and any test that does `panic!` fails. Let's make our test fail:

```
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!` is a macro provided by Rust which takes one argument: if the argument is `true`, nothing happens. If the argument is `false`, it `panic!`s. Let's run our tests again:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
        thread 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder/src/l
```

```
failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

thread '<main>' panicked at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.rs:247
```

Rust indicates that our test failed:

```
test it_works ... FAILED
```

And that's reflected in the summary line:

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

We also get a non-zero status code. We can use `$?` on OS X and Linux:

```
$ echo $?
101
```

On Windows, if you' re using `cmd`:

```
> echo %ERRORLEVEL%
```

And if you' re using PowerShell:

```
> echo $LASTEXITCODE # the code itself
> echo $? # a boolean, fail or succeed
```

This is useful if you want to integrate `cargo test` into other tooling.

We can invert our test's failure with another attribute: `should_panic`:

```
#[test]
#[should_panic]
fn it_works() {
    assert!(false);
}
```

This test will now succeed if we `panic!` and fail if we complete. Let's try it:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests
```

```
test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust provides another macro, `assert_eq!`, that compares two arguments for equality:

```rust
#[test]
#[should_panic]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

Does this test pass or fail? Because of the `should_panic` attribute, it passes:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`should_panic` tests can be fragile, as it's hard to guarantee that the test didn't fail for an unexpected reason. To help with this, an optional `expected` parameter can be added to the `should_panic` attribute. The test harness will make sure that the failure message contains the provided text. A safer version of the example above would be:

```rust
#[test]
#[should_panic(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

That's all there is to the basics! Let's write one 'real' test:

```rust
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

This is a very common use of `assert_eq!`: call some function with some known arguments and compare it to the expected output.

The `ignore` attribute

Sometimes a few specific tests can be very time-consuming to execute. These can be disabled by default by using the `ignore` attribute:

```
#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}

#[test]
#[ignore]
fn expensive_test() {
    // code that takes an hour to run
}
```

Now we run our tests and see that `it_works` is run, but `expensive_test` is not:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 2 tests
test expensive_test ... ignored
test it_works ... ok

test result: ok. 1 passed; 0 failed; 1 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

The expensive tests can be run explicitly using `cargo test -- --ignored`:

```
$ cargo test -- --ignored
     Running target/adder-91b3e234d4ed382a

running 1 test
test expensive_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

The `--ignored` argument is an argument to the test binary, and not to Cargo, which is why the command is `cargo test -- --ignored`.

The `tests` module

There is one way in which our existing example is not idiomatic: it's missing the `tests` module. The idiomatic way of writing our example looks like this:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::add_two;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

There's a few changes here. The first is the introduction of a `mod tests` with a `cfg` attribute. The module allows us to group all of our tests together, and to also define helper functions if needed, that don't become a part of the rest of our crate. The `cfg` attribute only compiles our test code if we're currently trying to run the tests. This can save compile time, and also ensures that our tests are entirely left out of a normal build.

The second change is the `use` declaration. Because we're in an inner module, we need to bring our test function into scope. This can be annoying if you have a large module, and so this is a common use of globs. Let's change our `src/lib.rs` to make use of it:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Note the different `use` line. Now we run our tests:

```
$ cargo test
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
```

```
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

It works!

The current convention is to use the `tests` module to hold your "unit-style" tests. Anything that tests one small bit of functionality makes sense to go here. But what about "integration-style" `tests` instead? For that, we have the tests directory.

The `tests` directory

To write an integration test, let's make a `tests` directory, and put a `tests/lib.rs` file inside, with this as its contents:

```
extern crate adder;

#[test]
fn it_works() {
    assert_eq!(4, adder::add_two(2));
}
```

This looks similar to our previous tests, but slightly different. We now have an `extern crate adder` at the top. This is because the tests in the `tests` directory are an entirely separate crate, and so we need to import our library. This is also why `tests` is a suitable place to write integration-style tests: they use the library like any other consumer of it would.

Let's run them:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

```
   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Now we have three sections: our previous test is also run, as well as our new one.

That's all there is to the `tests` directory. The `tests` module isn't needed here, since the whole thing is focused on tests.

Let's finally check out that third section: documentation tests.

Documentation tests

Nothing is better than documentation with examples. Nothing is worse than examples that don't actually work, because the code has changed since the documentation has been written. To this end, Rust supports automatically running examples in your documentation (**note**: this only works in library crates, not binary crates). Here's a fleshed-out `src/lib.rs` with examples:

```
//! The `adder` crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
//! ```

/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

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

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Note the module-level documentation with `//!` and the function-level documentation with `///`. Rust's documentation supports Markdown in comments, and so triple graves mark code blocks. It is conventional to include the `# Examples` section, exactly like that, with examples following.

Let's run the tests again:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test tests::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 2 tests
test add_two_0 ... ok
test _0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

Now we have all three kinds of tests running! Note the names of the documentation tests: the `_0` is generated for the module test, and `add_two_0` for the function test. These will auto increment with names like `add_two_1` as you add more examples.

We haven' t covered all of the details with writing documentation tests. For more, please see the Documentation chapter.

One final note: documentation tests *cannot* be run on binary crates. To see more on file arrangement see the Crates and Modules section (see Crates and Modules).

5.3 Conditional Compilation

Rust has a special attribute, `#[cfg]`, which allows you to compile code based on a flag passed to the compiler. It has two forms:

```
#[cfg(foo)]

#[cfg(bar = "baz")]
```

They also have some helpers:

```
#[cfg(any(unix, windows))]

#[cfg(all(unix, target_pointer_width = "32"))]

#[cfg(not(foo))]
```

These can nest arbitrarily:

```
#[cfg(any(not(unix), all(target_os="macos", target_arch = "powerpc")))]
```

As for how to enable or disable these switches, if you're using Cargo, they get set in the [features] section of your `Cargo.toml`:

```
[features]
# no features by default
default = []

# The "secure-password" feature depends on the bcrypt package.
secure-password = ["bcrypt"]
```

When you do this, Cargo passes along a flag to `rustc`:

```
--cfg feature="${feature_name}"
```

The sum of these `cfg` flags will determine which ones get activated, and therefore, which code gets compiled. Let's take this code:

```
#[cfg(feature = "foo")]
mod foo {
}
```

If we compile it with `cargo build --features "foo"`, it will send the `--cfg feature="foo"` flag to `rustc`, and the output will have the `mod foo` in it. If we compile it with a regular `cargo build`, no extra flags get passed on, and so, no `foo` module will exist.

cfg_attr

You can also set another attribute based on a `cfg` variable with `cfg_attr`:

```
#[cfg_attr(a, b)]
```

Will be the same as `#[b]` if a is set by `cfg` attribute, and nothing otherwise.

cfg!

The `cfg!` syntax extension lets you use these kinds of flags elsewhere in your code, too:

```
if cfg!(target_os = "macos") || cfg!(target_os = "ios") {
    println!("Think Different!");
}
```

These will be replaced by a `true` or `false` at compile-time, depending on the configuration settings.

5.4 Documentation

Documentation is an important part of any software project, and it's first-class in Rust. Let's talk about the tooling Rust gives you to document your project.

About `rustdoc`

The Rust distribution includes a tool, `rustdoc`, that generates documentation. `rustdoc` is also used by Cargo through `cargo doc`.

Documentation can be generated in two ways: from source code, and from standalone Markdown files.

Documenting source code

The primary way of documenting a Rust project is through annotating the source code. You can use documentation comments for this purpose:

````
/// Constructs a new `Rc<T>`.
///
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
pub fn new(value: T) -> Rc<T> {
    // implementation goes here
}
````

This code generates documentation that looks like this. I've left the implementation out, with a regular comment in its place.

The first thing to notice about this annotation is that it uses `///` instead of `//`. The triple slash indicates a documentation comment.

Documentation comments are written in Markdown.

Rust keeps track of these comments, and uses them when generating documentation. This is important when documenting things like enums:

```rust
/// The `Option` type. See [the module level documentation](index.html) for
↪ more.
enum Option<T> {
    /// No value
    None,
    /// Some value `T`
    Some(T),
}
```

The above works, but this does not:

```rust
/// The `Option` type. See [the module level documentation](index.html) for
↪ more.
enum Option<T> {
    None, /// No value
    Some(T), /// Some value `T`
}
```

You'll get an error:

```
hello.rs:4:1: 4:2 error: expected ident, found `}`
hello.rs:4 }
           ^
```

This unfortunate error is correct; documentation comments apply to the thing after them, and there's nothing after that last comment.

Writing documentation comments

Anyway, let's cover each part of this comment in detail:

```rust
/// Constructs a new `Rc<T>`.
```

The first line of a documentation comment should be a short summary of its functionality. One sentence. Just the basics. High level.

```rust
///
/// Other details about constructing `Rc<T>`s, maybe describing complicated
/// semantics, maybe additional options, all kinds of stuff.
///
```

Our original example had just a summary line, but if we had more things to say, we could have added more explanation in a new paragraph.

Special sections

Next, are special sections. These are indicated with a header, `#`. There are four kinds of headers that are commonly used. They aren't special syntax, just convention, for now.

```
/// # Panics
```

Unrecoverable misuses of a function (i.e. programming errors) in Rust are usually indicated by panics, which kill the whole current thread at the very least. If your function has a non-trivial contract like this, that is detected/enforced by panics, documenting it is very important.

```
/// # Failures
```

If your function or method returns a `Result<T, E>`, then describing the conditions under which it returns `Err(E)` is a nice thing to do. This is slightly less important than `Panics`, because failure is encoded into the type system, but it's still a good thing to do.

```
/// # Safety
```

If your function is `unsafe`, you should explain which invariants the caller is responsible for upholding.

```
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

Fourth, `Examples`. Include one or more examples of using your function or method, and your users will love you for it. These examples go inside of code block annotations, which we'll talk about in a moment, and can have more than one section:

```
/// # Examples
///
/// Simple `&str` patterns:
///
/// ```
/// let v: Vec<&str> = "Mary had a little lamb".split(' ').collect();
/// assert_eq!(v, vec!["Mary", "had", "a", "little", "lamb"]);
/// ```
///
/// More complex patterns with a lambda:
///
/// ```
/// let v: Vec<&str> = "abc1def2ghi".split(|c: char|
↪  c.is_numeric()).collect();
/// assert_eq!(v, vec!["abc", "def", "ghi"]);
/// ```
```

Let's discuss the details of these code blocks.

Code block annotations

To write some Rust code in a comment, use the triple graves:

```
/// ```
/// println!("Hello, world");
/// ```
```

If you want something that's not Rust code, you can add an annotation:

```
/// ```c
/// printf("Hello, world\n");
/// ```
```

This will highlight according to whatever language you're showing off. If you're only showing plain text, choose `text`.

It's important to choose the correct annotation here, because `rustdoc` uses it in an interesting way: It can be used to actually test your examples in a library crate, so that they don't get out of date. If you have some C code but `rustdoc` thinks it's Rust because you left off the annotation, `rustdoc` will complain when trying to generate the documentation.

Documentation as tests

Let's discuss our sample example documentation:

```
/// ```
/// println!("Hello, world");
/// ```
```

You'll notice that you don't need a `fn main()` or anything here. `rustdoc` will automatically add a `main()` wrapper around your code, using heuristics to attempt to put it in the right place. For example:

```
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
```

This will end up testing:

```
fn main() {
    use std::rc::Rc;
    let five = Rc::new(5);
}
```

Here's the full algorithm rustdoc uses to preprocess examples:

1. Any leading `#![foo]` attributes are left intact as crate attributes.

2. Some common `allow` attributes are inserted, including `unused_variables`, `unused_assignments`, `unused_mut`, `unused_attributes`, and `dead_code`. Small examples often trigger these lints.
3. If the example does not contain `extern crate`, then `extern crate <mycrate>;` is inserted (note the lack of `#[macro_use]`).
4. Finally, if the example does not contain `fn main`, the remainder of the text is wrapped in `fn main() { your_code }`.

This generated `fn main` can be a problem! If you have `extern crate` or a `mod` statements in the example code that are referred to by `use` statements, they will fail to resolve unless you include at least `fn main() {}` to inhibit step 4. `#[macro_use] extern crate` also does not work except at the crate root, so when testing macros an explicit `main` is always required. It doesn't have to clutter up your docs, though – keep reading!

Sometimes this algorithm isn't enough, though. For example, all of these code samples with `///` we've been talking about? The raw text:

```
/// Some documentation.
# fn foo() {}
```

looks different than the output:

```
/// Some documentation.
```

Yes, that's right: you can add lines that start with `#`, and they will be hidden from the output, but will be used when compiling your code. You can use this to your advantage. In this case, documentation comments need to apply to some kind of function, so if I want to show you just a documentation comment, I need to add a little function definition below it. At the same time, it's only there to satisfy the compiler, so hiding it makes the example more clear. You can use this technique to explain longer examples in detail, while still preserving the testability of your documentation.

For example, imagine that we wanted to document this code:

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

We might want the documentation to end up looking like this:

> First, we set `x` to five:
>
> ```
> let x = 5;
> ```
>
> Next, we set `y` to six:

```rust
let y = 6;
```

Finally, we print the sum of `x` and `y`:

```rust
println!("{}", x + y);
```

To keep each code block testable, we want the whole program in each block, but we don't want the reader to see every line every time. Here's what we put in our source code:

````
    First, we set `x` to five:

    ```text
    let x = 5;
    # let y = 6;
    # println!("{}", x + y);
    ```

    Next, we set `y` to six:

    ```text
    # let x = 5;
    let y = 6;
    # println!("{}", x + y);
    ```

    Finally, we print the sum of `x` and `y`:

    ```text
    # let x = 5;
    # let y = 6;
    println!("{}", x + y);
    ```
````

By repeating all parts of the example, you can ensure that your example still compiles, while only showing the parts that are relevant to that part of your explanation.

Documenting macros

Here' s an example of documenting a macro:

```
/// Panic with a given message unless an expression evaluates to true.
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, “Math is broken.”);
/// # }
/// ```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, “I’m broken.”);
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition {
↪   panic!($($rest),+); } });
}
```

You’ll note three things: we need to add our own `extern crate` line, so that we can add the `#[macro_use]` attribute. Second, we’ll need to add our own `main()` as well (for reasons discussed above). Finally, a judicious use of `#` to comment out those two things, so they don’t show up in the output.

Another case where the use of `#` is handy is when you want to ignore error handling. Lets say you want the following,

```
/// use std::io;
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
```

The problem is that `try!` returns a `Result<T, E>` and test functions don’t return anything so this will give a mismatched types error.

```
/// A doc test using try!
///
/// ```
/// use std::io;
/// # fn foo() -> io::Result<()> {
/// let mut input = String::new();
/// try!(io::stdin().read_line(&mut input));
/// # Ok(())
/// # }
/// ```
```

You can get around this by wrapping the code in a function. This catches and swallows the `Result<T, E>` when running tests on the docs. This pattern appears regularly in the standard library.

Running documentation tests

To run the tests, either:

```
$ rustdoc --test path/to/my/crate/root.rs
# or
$ cargo test
```

That's right, `cargo test` tests embedded documentation too. **However, `cargo test` will not test binary crates, only library ones.** This is due to the way `rustdoc` works: it links against the library to be tested, but with a binary, there' s nothing to link to.

There are a few more annotations that are useful to help `rustdoc` do the right thing when testing your code:

```
/// ```ignore
/// fn foo() {
/// ```
```

The `ignore` directive tells Rust to ignore your code. This is almost never what you want, as it's the most generic. Instead, consider annotating it with `text` if it's not code, or using `#`s to get a working example that only shows the part you care about.

```
/// ```should_panic
/// assert!(false);
/// ```
```

`should_panic` tells `rustdoc` that the code should compile correctly, but not actually pass as a test.

```
/// ```no_run
/// loop {
///     println!("Hello, world");
/// }
/// ```
```

The `no_run` attribute will compile your code, but not run it. This is important for examples such as "Here's how to start up a network service," which you would want to make sure compile, but might run in an infinite loop!

Documenting modules

Rust has another kind of doc comment, `//!`. This comment doesn't document the next item, but the enclosing item. In other words:

```rust
mod foo {
    //! This is documentation for the `foo` module.
    //!
    //! # Examples

    // ...
}
```

This is where you'll see `//!` used most often: for module documentation. If you have a module in `foo.rs`, you'll often open its code and see this:

```rust
//! A module for using `foo`s.
//!
//! The `foo` module contains a lot of useful functionality blah blah blah
```

Documentation comment style

Check out RFC 505 for full conventions around the style and format of documentation.

Other documentation

All of this behavior works in non-Rust source files too. Because comments are written in Markdown, they're often `.md` files.

When you write documentation in Markdown files, you don't need to prefix the documentation with comments. For example:

````rust
/// # Examples
///
/// ```
/// use std::rc::Rc;
///
/// let five = Rc::new(5);
/// ```
````

is:

````
# Examples

```
use std::rc::Rc;

let five = Rc::new(5);
```
````

when it's in a Markdown file. There is one wrinkle though: Markdown files need to have a title like this:

```
% The title

This is the example documentation.
```

This % line needs to be the very first line of the file.

doc attributes

At a deeper level, documentation comments are syntactic sugar for documentation attributes:

```rust
/// this

#[doc="this"]
```

are the same, as are these:

```rust
//! this

#![doc="this"]
```

You won't often see this attribute used for writing documentation, but it can be useful when changing some options, or when writing a macro.

Re-exports

rustdoc will show the documentation for a public re-export in both places:

```rust
extern crate foo;

pub use foo::bar;
```

This will create documentation for bar both inside the documentation for the crate foo, as well as the documentation for your crate. It will use the same documentation in both places.

This behavior can be suppressed with no_inline:

```rust
extern crate foo;

#[doc(no_inline)]
pub use foo::bar;
```

Missing documentation

Sometimes you want to make sure that every single public thing in your project is documented, especially when you are working on a library. Rust allows you to to generate warnings or errors, when an item is missing documentation. To generate warnings you use warn:

```rust
#![warn(missing_docs)]
```

And to generate errors you use deny:

```
#![deny(missing_docs)]
```

There are cases where you want to disable these warnings/errors to explicitly leave something undocumented. This is done by using `allow`:

```
#[allow(missing_docs)]
struct Undocumented;
```

You might even want to hide items from the documentation completely:

```
#[doc(hidden)]
struct Hidden;
```

Controlling HTML

You can control a few aspects of the HTML that `rustdoc` generates through the `#![doc]` version of the attribute:

```
#![doc(html_logo_url =
↪  "https://www.rust-lang.org/logos/rust-logo-128x128-blk-v2.png",
       html_favicon_url = "https://www.rust-lang.org/favicon.ico",
       html_root_url = "https://doc.rust-lang.org/")]
```

This sets a few different options, with a logo, favicon, and a root URL.

Configuring documentation tests

You can also configure the way that `rustdoc` tests your documentation examples through the `#![doc(test(..))]` attribute.

```
#![doc(test(attr(allow(unused_variables), deny(warnings))))]
```

This allows unused variables within the examples, but will fail the test for any other lint warning thrown.

Generation options

`rustdoc` also contains a few other options on the command line, for further customization:

- `--html-in-header FILE`: includes the contents of FILE at the end of the `<head>...</head>` section.
- `--html-before-content FILE`: includes the contents of FILE directly after `<body>`, before the rendered content (including the search bar).

- `--html-after-content FILE`: includes the contents of FILE after all the rendered content.

Security note

The Markdown in documentation comments is placed without processing into the final webpage. Be careful with literal HTML:

```
/// <script>alert(document.cookie)</script>
```

6 Nightly Rust

7 Glossary

Not every Rustacean has a background in systems programming, nor in computer science, so we've added explanations of terms that might be unfamiliar.

Abstract Syntax Tree

When a compiler is compiling your program, it does a number of different things. One of the things that it does is turn the text of your program into an 'abstract syntax tree', or 'AST'. This tree is a representation of the structure of your program. For example, `2 + 3` can be turned into a tree:

```
  +
 / \
2   3
```

And `2 + (3 * 4)` would look like this:

```
  +
 / \
2   *
   / \
  3   4
```

Arity

Arity refers to the number of arguments a function or operation takes.

```
let x = (2, 3);
let y = (4, 6);
let z = (8, 2, 6);
```

In the example above `x` and `y` have arity 2. `z` has arity 3.

Bounds

Bounds are constraints on a type or trait. For example, if a bound is placed on the argument a function takes, types passed to that function must abide by that constraint.

DST (Dynamically Sized Type)

A type without a statically known size or alignment. (more info)

Expression

In computer programming, an expression is a combination of values, constants, variables, operators and functions that evaluate to a single value. For example, `2 + (3 * 4)` is an expression that returns the value 14. It is worth noting that expressions can have side-effects. For example, a function included in an expression might perform actions other than simply returning a value.

Expression-Oriented Language

In early programming languages, Expression and Statement were two separate syntactic categories: expressions had a value and statements did things. However, later languages blurred this distinction, allowing expressions to do things and statements to have a value. In an expression-oriented language, (nearly) every statement is an expression and therefore returns a value. Consequently, these expression statements can themselves form part of larger expressions.

Statement

In computer programming, a statement is the smallest standalone element of a programming language that commands a computer to perform an action.

8 Syntax Index

9 Bibliography

This is a reading list of material relevant to Rust. It includes prior research that has - at one time or another - influenced the design of Rust, as well as publications about Rust.

9.1 Type system

- Region based memory management in Cyclone
- Safe manual memory management in Cyclone
- Typeclasses: making ad-hoc polymorphism less ad hoc
- Macros that work together
- Traits: composable units of behavior
- Alias burying - We tried something similar and abandoned it.
- External uniqueness is unique enough
- Uniqueness and Reference Immutability for Safe Parallelism
- Region Based Memory Management

9.2 Concurrency

- Singularity: rethinking the software stack
- Language support for fast and reliable message passing in singularity OS
- Scheduling multithreaded computations by work stealing
- Thread scheduling for multiprogramming multiprocessors
- The data locality of work stealing
- Dynamic circular work stealing deque - The Chase/Lev deque
- Work-first and help-first scheduling policies for async-finish task parallelism - More general than fully-strict work stealing
- A Java fork/join calamity - critique of Java's fork/join library, particularly its application of work stealing to non-strict computation
- Scheduling techniques for concurrent systems
- Contention aware scheduling
- Balanced work stealing for time-sharing multicores

- Three layer cake for shared-memory programming
- Non-blocking steal-half work queues
- Reagents: expressing and composing fine-grained concurrency
- Algorithms for scalable synchronization of shared-memory multiprocessors
- Epoch-based reclamation.

9.3 Others

- Crash-only software
- Composing High-Performance Memory Allocators
- Reconsidering Custom Memory Allocation

9.4 Papers about Rust

- GPU Programming in Rust: Implementing High Level Abstractions in a Systems Level Language. Early GPU work by Eric Holk.
- Parallel closures: a new twist on an old idea
  - not exactly about Rust, but by nmatsakis
- Patina: A Formalization of the Rust Programming Language. Early formalization of a subset of the type system, by Eric Reed.
- Experience Report: Developing the Servo Web Browser Engine using Rust. By Lars Bergstrom.
- Implementing a Generic Radix Trie in Rust. Undergrad paper by Michael Sproul.
- Reenix: Implementing a Unix-Like Operating System in Rust. Undergrad paper by Alex Light.
- Evaluation of performance and productivity metrics of potential programming languages in the HPC environment. Bachelor's thesis by Florian Wilkens. Compares C, Go and Rust.
- Nom, a byte oriented, streaming, zero copy, parser combinators library in Rust. By Geoffroy Couprie, research for VLC.
- Graph-Based Higher-Order Intermediate Representation. An experimental IR implemented in Impala, a Rust-like language.
- Code Refinement of Stencil Codes. Another paper using Impala.
- Parallelization in Rust with fork-join and friends. Linus Farnstrand's master's thesis.

- Session Types for Rust. Philip Munksgaard's master's thesis. Research for Servo.
- Ownership is Theft: Experiences Building an Embedded OS in Rust - Amit Levy, et. al.