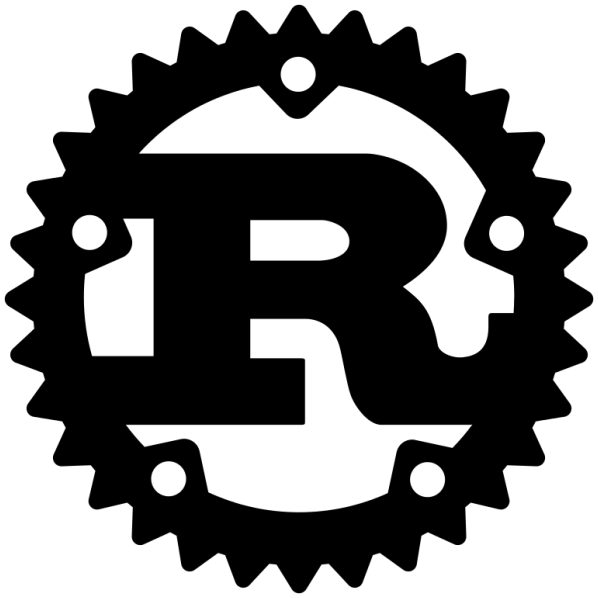

The Rust Programming Language

Darth-Revan

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

March 23, 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 `^0.3.0`, 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

3.5 Looping

3.6 Complete

4 Syntax and Semantics

5 Effective Rust

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.