The Rust Programming Language

Darth-Revan

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

March 22, 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.

3 Tutorial: Guessing Game

4 Syntax and Semantics

5 Effective Rust

6 Nightly Rust

7 Glossary

8 Syntax Index

9 Bibliography