May 25, 2023 • 6 min read

A guide to cross-compilation in Rust

Greg Stoll
Greg is a software engineer with over 20 years of experience in the industry. He enjoys working on projects in his spare time and enjoys writing about them!

Table of contents

Cross-compiling is a very handy capability to have in multiple scenarios. Let's take a look at why you might want to do it and how to get set up in Rust for cross-compilation.

What we'll cover:

- Understanding cross-compiling and its Rust benefits
- Setting up an example Rust cross-compilation project
- How Rust represents platforms
- Cross-compiling our demo Rust project from Linux to Windows
- How to write platform-specific code

To follow along, see the GitHub repo for this project.

Understanding cross-compiling and its Rust benefits

Cross-compiling means compiling a program on a platform for a different platform. For example, if you are on a Windows machine, you can compile a program that can run on Linux.

There are a few reasons cross-compiling can be helpful. One is that if you have a product that you want to ship on multiple platforms, it can be convenient to be able to build all versions from a single machine instead of having one Windows machine, one Mac machine, etc.

Cross-compilation can be helpful in cloud-based build scenarios as well. Rust even supports running tests across multiple target platforms on the same host platform.

Another reason you may want to cross-compile is that it might be necessary, as the Rust compiler and host tools are not supported on every platform they can build for. For example, the Rust compiler supports building an app for iOS, but the Rust compiler itself doesn't run on iOS.

Setting up an example Rust cross-compilation project

There is some built-in support in `rustc` for cross-compiling, but getting the build to actually work can be tricky due to the need for an appropriate linker. Instead, we're going to use the Cross crate, which used to be maintained by the Rust Embedded Working Group Tools group.

First, let's set up a simple project that will show which platform it's running on. To do this, we're going to use the `current_platform` crate, which is an easy way to see what platform your code is running on, as well as what platform it was compiled on.

Let's make a new crate with `cargo new` and add the crate with `cargo add current_platform`. Then we can add the following to the `src/main.rs` file:

```
use current_platform::{COMPILED_ON, CURRENT_PLATFORM};

fn main() {
    println!("Hello, world from {}! I was compiled on {}.", CURREN
}
```

On my Linux machine, running this with `cargo run` leads to this output:

```
Hello, world from x86_64-unknown-linux-gnu! I was compiled on x86_
```

This agrees with what `rustc` thinks the platform is; running `rustc -vV` gives this output:

```
rustc 1.68.2 (9eb3afe9e 2023-03-27)
binary: rustc
commit-hash: 9eb3afe9ebe9c7d2b84b71002d44f4a0edac95e0
commit-date: 2023-03-27
host: x86_64-unknown-linux-gnu
release: 1.68.2
LLVM version: 15.0.6
```

How Rust represents platforms

To cross-compile, you need to know the "target triple" for the platform you're building for. Rust uses the same format that LLVM does. The format is `<arch><sub>-<vendor>-<sys>-<env>`, although figuring out these values for a given platform is not obvious.

As we saw above, `x86_64-unknown-linux-gnu` represents a 64-bit Linux machine. Running `rustc --print target-list` will print all targets that Rust supports, but the list is long, and it's hard to find the one you want.

The two best ways to find the target triple for a platform you care about are:

1. Run `rustc -vV` on the platform and look for the line that starts with `host:` — the rest of that line will be the target triple
2. Look it up in the list provided on the Rust Platform Support page

For quick reference, here are a few common values:

Target triple name	Description
`x86_64-unknown-linux-gnu`	**64-bit Linux (kernel 3.2+, glibc 2.17+)**
`x86_64-pc-windows-msvc`	**64-bit MSVC (Windows 7+)**

`x86_64-apple-darwin`	**64-bit macOS (10.7+, Lion+)**
`aarch64-unknown-linux-gnu`	**ARM64 Linux (kernel 4.1, glibc 2.17+)**
`aarch64-apple-darwin`	**ARM64 macOS (11.0+, Big Sur+)**
`aarch64-apple-ios`	**ARM64 iOS**
`aarch64-apple-ios-sim`	**Apple iOS Simulator on ARM64**
`armv7-linux-androideabi`	**ARMv7a Android**

Cross-compiling our Rust project from Linux to Windows

Now that we know that the target triple for Windows is `x86_64-pc-windows-msvc`, let's get to cross-compiling!

To install the `cross` crate, the first step is to run `cargo install cross`. This will install Cross to `$HOME/.cargo/bin`. You can add this to your `$PATH` if you'd like, or just run it from there when we're ready to do so.

Cross works by using a container engine with images that have the appropriate toolchain for cross-compiling. All of this is transparent to the user, as we'll see below, but you do need a container engine installed.

If your machine is running Windows, the official Getting Started guide from Cross recommends using Docker as your container engine. However, for Linux, it recommends using Podman, a popular Docker alternative. On my Ubuntu system, installing this was as easy as `sudo apt-get install podman`.

That's all the setup we need! Now we can cross-compile to Windows and run the executable with the following command:

```
cross run --target x86_64-pc-windows-gnu
```

Remember, the Cross executable is in `$HOME/.cargo/bin`.

Running this the first time will take a while as the appropriate container is downloaded and started. Once it's done, we should see the following output:

```
   Compiling current_platform v0.2.0
   Compiling rustcrosscompile v0.1.0 (/project)
    Finished dev [unoptimized + debuginfo] target(s) in 7.95s
     Running `wine /target/x86_64-pc-windows-gnu/debug/rustcrossc
0054:err:winediag:nodrv_CreateWindow Application tried to create a
0054:err:winediag:nodrv_CreateWindow Make sure that your X server
0054:err:systray:initialize_systray Could not create tray window
```

```
Hello, world from x86_64-pc-windows-gnu! I was compiled on x86_64
```

As expected, we see that `rustcrosscompile.exe` is running on Windows! Actually, through Wine — a compatibility layer — but close enough!

As you can see from the output above, the compiled `.exe` is located in `target/x86_64-pc-windows-gnu/debug`. You can copy it to a Windows machine and run it, which will show the expected output:

```
Hello, world from x86_64-pc-windows-gnu! I was compiled on x86_64
```

Cross even supports running tests on other platforms! Let's add a test to our `main.rs` file:

```
mod tests {
    use current_platform::{COMPILED_ON, CURRENT_PLATFORM};

    #[test]
    fn test_compiled_on_equals_current_platform() {
        assert_eq!(COMPILED_ON, CURRENT_PLATFORM);
    }
}
```

Note that this is a test that we would expect to pass when running on Linux, but fail when we cross-compile to Windows and run it there.

Indeed, if we run `cargo test` on Linux, we get this output:

```
     Running unittests src/main.rs (target/debug/deps/rustcrosscon

running 1 test
test tests::test_compiled_on_equals_current_platform ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filt
```

To run the test on Windows, the syntax is very similar to running the executable:

```
cross test --target x86_64-pc-windows-gnu
```

After a minute or so, we get the output:

```
     Running unittests src/main.rs (/target/x86_64-pc-windows-gnu/
0050:err:winediag:nodrv_CreateWindow Application tried to create a
0050:err:winediag:nodrv_CreateWindow Make sure that your X server
0050:err:systray:initialize_systray Could not create tray window

running 1 test
test tests::test_compiled_on_equals_current_platform ... FAILED

failures:

---- tests::test_compiled_on_equals_current_platform stdout ----
thread 'tests::test_compiled_on_equals_current_platform' panicked
  left: `"x86_64-unknown-linux-gnu"`,
 right: `"x86_64-pc-windows-gnu"`', src/main.rs:22:9
note: run with `RUST_BACKTRACE=1` environment variable to display


failures:
    tests::test_compiled_on_equals_current_platform

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 0
```

```
error: test failed, to rerun pass `--bin rustcrosscompile`
```

As expected, the test fails!

Note that running tests isn't supported on all platforms. Additionally, because of threading issues, tests run sequentially, which can be much slower than running tests natively. See the Cross documentation on supported targets for details.

How to write platform-specific code

Often, you may want to write code that only runs on one platform. Rust makes this easy with the `cfg` attribute. Now that we can cross-compile and run, we can easily try it out.

Let's modify our program to add a message that only gets printed on Windows. In fact, for hypothetical efficiency reasons (😉), we won't even compile this code on non-Windows platforms:

```
use current_platform::{COMPILED_ON, CURRENT_PLATFORM};

#[cfg(target_os="windows")]
fn windows_only() {
    println!("This will only get printed on Windows.");
}

fn main() {
    println!("Hello, world from {}! I was compiled on {}.", CURREN
    #[cfg(target_os="windows")]
    {
        windows_only();
    }
}
```

Here, we applied the `cfg` attribute to the `windows_only()` function so it won't get compiled on non-Windows platforms. But that means we can only call it on Windows, so we apply the same `cfg` attribute to the block of code that calls the function.

You can actually apply the attribute in other places as well, like enum variants, struct fields, and match expression arms!

Running this on Linux with `cargo run` gives this output:

```
Hello, world from x86_64-unknown-linux-gnu! I was compiled on x86_
```

As you can see, the output above does not have the Windows-specific message. But running with `cross run --target x86_64-pc-windows-gnu` gives this output:

```
Hello, world from x86_64-pc-windows-gnu! I was compiled on x86_64
This will only get printed on Windows.
```

Rust also provides an easy way to conditionally apply attributes based on the platform. You can look up the Rust reference guide to the `cfg_attr` attribute for more information on that.

- Explore creating a custom mouse cursor with CSS
- Advisory boards aren't just for executives. Join LogRocket's Content Advisory Board. You'll help inform the type of content we create and get access to exclusive meetups, social accreditation, and swag

---

Conclusion

Cross makes it quite easy to cross-compile, run, and test your Rust library or application. This crate is helpful — and sometimes necessary — if you have a product that you want to ship on multiple platforms.

There are some limitations — notably, performance — since the building and running is done through a virtual machine. So if this is something you're planning on doing with a larger project, definitely try it out first in your build environment to make sure the performance will work for you!

LogRocket: Full visibility into web frontends for Rust apps

Debugging Rust applications can be difficult, especially when users experience issues that are hard to reproduce. If you're interested in monitoring and tracking the performance of your Rust apps, automatically surfacing errors, and tracking slow network requests and load time, try LogRocket.

LogRocket lets you replay user sessions, eliminating guesswork around why bugs happen by showing exactly what users experienced. It captures console logs, errors, network requests, and pixel-perfect DOM recordings — compatible with all frameworks.