Rust

For C++ Programmers

Published
with GitBook

Table of Contents

Rust for C++ Programmers

This gitbook is a collection of "Rust for C++ programmers" posts by Nick Cameron.

I found it a bit hard to read the posts on his blog, hence I started copy-pasting them into a git repo to make it easier to read. Of course, all the credit for content goes to Nick Cameron.

After a while, I decided to turn them into a gitbook, so here it is!

Update (June 2015): I've realized that Nick has created a repo for his posts on GitHub (nrc/r4cppp); I've merged new content and will look into possibility of merging this repo with Nick's.

Introduction - hello world!

This is the first in a series of blog posts (none written yet) which aim to help experienced C++ programmers learn Rust. Expect updates to be sporadic at best. In this first blog post we'll just get setup and do a few super basic things. Much better resources are at the tutorial and reference manual.

First you need to install Rust. You can download a nightly build from http://www.rust-lang.org/install.html (I recommend the nightlies rather than 'stable' versions - the nightlies are stable in that they won't crash too much (no more than the stable versions) and you're going to have to get used to Rust evolving under you sooner or later anyway). Assuming you manage to install things properly, you should then have a `rustc` command available to you. Test it with `rustc -v`.

Now for our first program. Create a file, copy and paste the following into it and save it as `hello.rs` or something equally imaginative.

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

Compile this using `rustc hello.rs`, and then run `./hello`. It should display the expected greeting \o/

Two compiler options you should know are `-o ex_name` to specify the name of the executable and `-g` to output debug info; you can then debug as expected using gdb or lldb, etc. Use `-h` to show other options.

OK, back to the code. A few interesting points - we use `fn` to define a function or method. `main()` is the default entry point for our programs (we'll leave program args for later). There are no separate declarations or header files as with C++. `println!` is Rust's equivalent of printf. The `!` means that it is a macro, for now you can just treat it like a regular function. A subset of the standard library is available without needing to be explicitly imported/included (we'll talk about that later). The `println!` macros is included as part of that subset.

Lets change our example a little bit:

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

`let` is used to introduce a variable, world is the variable name and it is a string (technically the type is `&'static str`, but more on that in a later post). We don't need to specify the type, it will be inferred for us.

Using `{}` in the `println!` statement is like using `%s` in printf. In fact, it is a bit more general than that because Rust will try to convert the variable to a string if it is not one already[1]. You can easily play around with this sort of thing - try multiple strings and using numbers (integer and float literals will work).

If you like, you can explicitly give the type of `world`:

```
let world: &'static str = "world";
```

In C++ we write `T x` to declare a variable `x` with type `T`. In Rust we write `x: T`, whether in `let` statements or function signatures, etc. Mostly we omit explicit types in `let` statements, but they are required for function arguments. Lets add another function to see it work:

```
fn foo(_x: &'static str) -> &'static str {
    "world"
}

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

The function `foo` has a single argument `_x` which is a string literal (we pass it "bar" from `main`). We don't actually use that argument in `foo`. Usually, Rust will warn us about this. By prefixing the argument name with `_` we avoid these warnings. In fact, we don't need to name the argument at all, we could just use `_`.

The return type for a function is given after `->`. If the function doesn't return anything (a void function in C++), we don't need to give a return type at all (as in `main`). If you want to be super-explicit, you can write `-> ()`, `()` is the void type in Rust. `foo` returns a string literal.

You don't need the `return` keyword in Rust, if the last expression in a function body (or any other body, we'll see more of this later) is not finished with a semicolon, then it is the return value. So `foo` will always return "world". The `return` keyword still exists so we can do early returns. You can replace `"world"` with `return "world";` and it will have the same effect.

1