C++ for Rustaceans

Tianyi Shi

2020-11-13

Contents

Disclaimer

I am neither a pro in Rust nor in C++. It is possible that some of my conceptual understandings are wrong, and it is very likely that some examples, especially C++ ones, are not the best practice. I could only promise that all programs should compile and run without safety issues. If you spot anything that could be improved, please submit a PR!

About this Book

I'm a biochemistry student wishing to specialize in computational biology, and I need a fast (specifically, no-GC) language for implementing algorithms. Since the decision was made in April 2020, I naturally chose Rust. Soon I fell in love with it. Cargo, rustdoc, crates.io, clippy etc. just makes Rust so nice–even better than Python. However, I have to face the reality: the majority of bioinformatics algorithms to date are written in C or C++ (either as pure C or C++ libraries or as extensions to Python or R), and most labs are still developing on them. It turns out that some C and C++ literacy is necessary for me.

While there is a project called r4cppp that introduces Rust to C++ programmers, I haven't found any cpp4r, so I started this one. I'm not an expert in Rust and C++ and I'm writing this book while learning them, so it'll be more like a personal notebook than a perfessional guide. I'll try to make it readable, though.

Prerequisites

You'll need a C++ compiler. I recommend using `clang++` on Linux & MacOS because it generally gives better error message than `g++`. On Windows you should use `msvc`.

Chapter 1

Hello

1.1 Hello world

This is a hello world program in C++:

```cpp
#include <iostream>
int main()
{
    std::cout << "Hello C++!" << std::endl;
}
```

Write that in `hello.cpp`, then you can compile it with `g++ hello.cpp -o hello` and run `./hello` (you can replace `g++` with `clang++` or any other compiler).

A couple of thing to note here:

`#include` is a preprocessor. We'll meet more preprocessors in the future, for now just accept that they are "naive macros" that are "expanded" before the actual compilation. Here `#include` copies the content of file called `iostream`, which has tens of thousands lines, and pastes it here. Yes, it literally does so, and you can check this by running `g++ -E main.c`, which "expands" all preprocessor statements.

`iostream` contains definitions of functions and objects such as `std::cout` and `std::endl`, which are used for IO manipulations. `cout` stands for "character output", and `endl` stands for "endline" (it appends `\n` and flushes the buffer). `<<` is the bitwise left shift operator, and the designers of C++ decided that overloading bitwise shift operators for `cout` and `cin` can make C++ look fancy from the beginning. That's why we need to learn yet another special syntax.

Fortunately (or unfortunately), there's another way to do exactly the same thing:

```
printf("Hello from printf\n");
```

1.2 Hello Data Types

The following table summarises the relationship between Rust's and C++'s integer data types:

Rust	C++
`i8`	`int8_t`
`i16`	`int16_t`
`i32`	`int32_t`
`i64`	`int64_t`
`i128`	
`u8`	`uint8_t`
`u16`	`uint16_t`
`u32`	`uint32_t`
`u64`	`uint64_t`
`u128`	
`isize`	
`usize`	`size_t`

While the relationships described in the table are always true, C++'s integer types are much more complex. The types above are fixed width integer types, and there are additional integer types whose width is dependent on the implementation. These include C-compatible ones (i.e. `char`, `short`, `int`, `long`, `long long`), and other C++ artifects such as `int_fast16_t` and `int_least32_t`. You can learn about them at cppreference.

For floating numbers, `f32` and `f64` correspond to `float` and `double`, respectively (stand for single-precision and double-precision floating point numbers).

1.3 Pointers

Syntaxes related to pointers in C are…interesing. (There are no 'references' in C; that's a C++ thing and has even more interesting syntaxes)

In rust, when you take a reference to a type `T`, the type of the reference is `&T`. The syntax can't be more natual: you add `&` to both LHS and RHS:

```
let   a:  i32 =  5;
let p_a: &i32 = &5; // type annotations are not required; this is just for demonstration
```

and when you deference, you use `*`. Just remembering that `*` is the reverse of `&`, everything is natual as well. Every `*` just removes one `&` from both LHS and RHS:

```
let p_a: &i32 = &5;
let   a:  i32 = *p_a;
let   a:  i32 = *&*&*&a
```

In C, this is what you would do to make a pointer:

```
int    a =  5;
int *p_a = &5;
// or
int * p_a = &5;
// or
int*  p_a = &5;
```

and to dereference a pointer:

```
int b = *p_a;
```

You add `&` to RHS, but you add `*` to LHS. Weird. What's worse, despite the fact that the variable name really is `p_a`, not `*p_a`, and the type really is `int*`, most people and formatters prepend the asterisk before the variable name, which looks like you're declaring an `int` which is clearly not true. So why are people doing that? There are some discussion on StackOverflow.

To be consistent with this weird convention, tools will also name pointer types as `` `T *` `` (with a space in between), not `` `T*` ``.