C++ for Rustaceans

Tianyi Shi

2020-11-14

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 C++

1.1 Hello world

This is a hello world program in C++:

```
#include <iostream>
int main()
{
    std::cout << "Hello C++!" << std::endl;
    return 0;
}
```

Write the above code 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 things to note here:

Every C++ executable (as opposed to library) must have a `main()` function that returns `int`. Returning `0` signifies that the program terminates without errors. The final `return 0;` statement can be omitted in the `main()` function.

`#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");
```

Now you might begin to wonder, why isn't `std::cout` called `std::iostream::cout`, and why `printf` can be called without any prefix. This is because in C++ filenames have no relationships to `namespace`s by default. `namespace` is similar to Rust's `mod`, but more flexible. In this case, the `iostream` file contains something conceptually like this:

```
void printf(...);

namespace std {
    class cout {}
    class endl {}
}
```

`printf` isn't placed inside `std` because it is a heritage from C. Many other C functions are also available in C++, and they can be distinguished by the absence of the `std::` prefix.

1.2 Hello Data Types and Variables

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

Rust	C++	C & C++
i8	int8_t	char
i16	int16_t	short
i32	int32_t	int
i64	int64_t	long
i128		
u8	uint8_t	unsigned char
u16	uint16_t	unsigned short
u32	uint32_t	unsigned int
u64	uint64_t	unsigned long
u128		

Rust	C++	C & C++
`isize`		
`usize`	`size_t`	

While the equivalence between the first column and the second column always holds true, the third column depends on the platform and here I'm assuming you're on a modern, 64-bit system.

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++ artifacts 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).

Like in Rust, creating a variable requires two steps, declaration and initialization. In C++, there are usually more than one syntax to do any task, and these two basic operations are no exception.

The traditional syntax for declaration and initialization is `<type> <var name> = <value>;`, and these two steps can be separated:

```
int a = 5;
char b;
b = 'A';
```

The above syntax is compatible with C, which has a problem: if you initialize an `int` with a float:

```
int a = 5.5;
assert(a == 5);
```

The value will be implicitly converted. Instead of giving a warning or simply disallowing this (recent compilers DO give a warning; see Figure 1.1), the designers of C++ decided to invent another syntax for declaration and initialization, using curly braces:

```
int a{5};
```

If you try `int a{5.5};` with this syntax, the compiler will give an error and abort (Figure 1.2). In addition, you can't separate the two parts:

Figure 1.1: When implicit conversion occurs, a decent modern C++ compiler will give a warning.

```
// not allowed
int a;
a{5}
```

Figure 1.2: When implicit conversion occurs, a decent modern C++ compiler will give a warning.

1.3 Pointers and References

Syntaxes related to pointers and references in C++ are…interesing.

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 demonstrati
```

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.

But wait, how did people learn this weird convention in the first place? On cppreference, and in Bjarne Stroustrup's *A Tour of C++*, the `int* p_a = &5` style is always used, so just…why? Why are people sticking to the anti-standard?

Since pointers are a heritage from C, it is not surprising that the *creative* C++ designers invented yet another similar syntax for achieving almost the same task: references. You create and read a reference like this:

```
int l = 5;
int &m = l;
assert(m == 5); // not `*m` !
```

Note that you do not (and cannot) use the deference operator (`*`) on a reference. In addition, a reference cannot be re-assigned to point to another value. Apart from these two rules, references are effectively the same as pointers. Since we are Rustaceans, we are sensitive to mutability. Are there any difference between pointers and references in terms of mutability? The answer is no. Both can be use to mutate the referent.

```
#include <iostream>
int main()
{
    int a{5};
    int &r_a = a;  // create a reference
    int *p_a = &a; // create a pointer
    printf("| a|r_a|*p_a|     p_a      |\n");
    a = 10; // mutate the value
    printf("|%d| %d|  %d|%p|\n", a, r_a, *p_a, p_a);
    r_a = 15; // mutate via reference; no asterisk!
    printf("|%d| %d|  %d|%p|\n", a, r_a, *p_a, p_a);
    *p_a = 20; // mutate via pointer
```

```
    printf("|%d| %d|  %d|%p|\n", a, r_a, *p_a, p_a);
}
```

```
| a|r_a|*p_a|     p_a      |
|10| 10|  10|0x7ffe09e40404|
|15| 15|  15|0x7ffe09e40404|
|20| 20|  20|0x7ffe09e40404|
```

1.4 Control Flow

C++ offers 5 types of control flow statements. `if...else`, `for` loop and `while` loop are pretty much the same as in Rust, but the `switch` statement is much less powerful than Rust's `match`. Additionally there is a `goto` statement which performs unconditional jump and is notorious for leading to "unmaintainable spaghetti code" (however, people are still using them, and you'll see why in a minute).

If you have experience in Javascript or Java, most of C++'s control flow syntax will be familiar to you. The conditional test associated with `if`, `for`, `while` and `switch` must be surrounded by parentheses.

1.4.1 if...else

Like Rust, C++ offers `if` and `else` keywords to work with conditionals. Unlike Rust, C++ is not expression-oriented, so you *cannot* write:

```
int a = if (true) { 5 } else { 10 };
```

But this kind of conditional assignment is a very common pattern, so C++ invented yet another syntax specifically designed for this single task: the ternary operator. So, instead of writing:

```
int a;
if (true) { a = 5; } else { a = 10; };
```

you could write:

```
int a = true ? 5 : 10;
```

1.4.2 `while` Loop

The `while` loop in C++ has nothing different from Rust, just remember to wrap the test expression with parentheses.

```
int i = 10;
while (i > 0)
{
    if (i == 8)
    {
        continue;
    }
    if (i == 5)
    {
        break;
    }
    i--;
}
```

1.4.3 `for` Loop

A traditional C-style `for` loop looks like this (you'll be familiar with this if you code Javascript or Java):

```
for (<initializationStatement>; <testExpression>; <updateStatement>)
{
    // do something
}
```

For example:

```
#include <iostream>
printf("|i|j|\n");
for (int i = 0; i < 2; i++)
{
    for (int j = 0; j < 3; j++)
    {
        printf("|%d|%d|", i, j);
    }
}
```

C++11 introduced the range-based `for` statement, which is also known as a `for...in` loop in most other languages (Rust, Swift, Python, Ruby, …). The syntax itself is easy but knowing the relationship between the element and the iterable can be tricky. Fortuantely, we are Rustaceans, so an easy way for me to illustrate and for you to understand is to write a few equivalent examples in Rust and C++.

1.4.3.1 Scenario 1: Copying (Cloning)

```
// Rust
let v = vec!["a".to_string(), "b".to_string(), "c".to_string()];
for e.clone() in &v {
    println!("{}", e);
}
```

```
// C++
#include <iostream>
#include <vector>
std::vector<std::string> a{"a", "b", "c"};
for (auto s : a) // s has type `std::string`
{
    std::cout << s << std::endl;
}
```

Note how Rust makes it crystal clear that copying during iteration and using `String` for instead of `&str` for static strings are anti-patterns and how C++ makes it easy to write such inefficient code.

1.4.3.2 Scenario 2: As Reference (Borrowing)

```
// Rust
let v = vec![1, 2, 3, 4, 5];
for e in &v {
    println!("{}", *e);
}
```

```
// C++
#include <iostream>
#include <vector>
std::vector<int> a{1, 2, 3, 4, 5};
for (auto& num : a)
{
    printf("%d ", num); // no asterisk!
}
```

1.4.3.3 Scenario 3: Mutation

```
// Rust
let mut v = vec![1, 2, 3, 4, 5];
for e in &mut v {
    *e = e + 1;
}
assert_eq!(v, vec![2, 3, 4, 5, 6]);
```

```
// C++
#include <vector>
#include <cassert>
std::vector<int> a{1, 2, 3, 4, 5};
for (auto &num : a)
{
    num++;
}
assert(a == (std::vector<int>{2, 3, 4, 5, 6}));
```

Note the parentheses surrounding the second argument of `assert`, without which we would get an error. This is because `assert` is a so called `#define` macro, which simply parses its arguments as comma-separated identifiers and does text replacement. Since `std::vector<int>{2, 3, 4, 5, 6}` contains commas, it has to be escaped with parentheses.[1] This macro is actually defined in `assert.h` which is part of C's std, and its more or less copied verbatim into C++'s `cassert`. Theoretically I think it is possible to implement C++'s own `assert` or `#define` that removes the need for adding parentheses here and there. However, the C++ committee never cares about ergonomics.

1.4.4 `goto` Statement and Breaking outer Loops

Rust, like Java and Python, allows you to break an outer loop from an inner loop:

```
for i in 0..3 {
    'for_j: for j in 0..3 {
        for k in 0..3 {
            if i == 1 {
                break 'for_j;
            }
            println!("{} {} {}", i, j, k);
```

[1] Related to this stackoverflow question: https://stackoverflow.com/questions/38030048

```
        }
    }
}
```

C++ doesn't support this natively, so people are using `goto` to achieve this:

```
#include <iostream>
printf("Break outer loop using goto:\n");
for (int i = 0; i < 3; ++i)
{

    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 3; ++k)
        {
            if (i == 1)
            {
                goto the_end;
            }
            std::cout << "i: " << i << " j: " << j << " k " << k << std::endl;
        }
    }
the_end:
{
}
}
```

Using `goto` sounds like I am joking. I'm not. People really suggest doing this[2]. It's a consensus that `goto` statements are error-prone, but there really aren't better ways to do this when you absolutely need to break an outer loop. The C++ committee never solves really problems.

Alternatively you can use a flag, which is more verbose, and I think this is even less readable:

```
printf("Break outer loop using flag:\n");
bool i_is_1{false};
for (int i = 0; i < 3; ++i)
{

    for (int j = 0; j < 3; ++j)
    {
        for (int k = 0; k < 3; ++k)
```

[2]https://stackoverflow.com/questions/1257744/can-i-use-break-to-exit-multiple-nested-for-loops

```
        {
            if (i == 1)
            {
                i_is_1 = true;
                break;
            }
            else
            {
                i_is_1 = false;
            }
            std::cout << "i: " << i << " j: " << j << " k " << k << std::endl;
        }
        if (i_is_1)
        {
            break;
        }
    }
}
```

There is another cleaner way, using the lambda trick (but why the hell must I use a lamba for such a basic task???):

```
printf("Break outer loop using lambda:\n");
for (int i = 0; i < 3; ++i)
{
    [&] {
        for (int j = 0; j < 3; ++j)
        {
            for (int k = 0; k < 3; ++k)
            {
                if (i == 1)
                {
                    return;
                }
                std::cout << "i: " << i << " j: " << j << " k " << k << std::endl;
            }
        }
    }();
}
```

I'll get back to lamdas later. For now just accept that they are roughly equivalent to Rust's closures.