C++ for Rustaceans

Tianyi Shi

2020-11-15

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

I'm assuming you're an intermediate-level Rustacean. You should understand the majority of the concepts in The Book and also the basics of raw pointers.

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++

In this chapter, we'll learn the very basics of C++.

1.1 Hello world

This is a hello world program in C++:

```cpp
#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.

`iostream` also introduces another function into scope, the C-compatible `printf` into scope, which can also be used to print "Hello world".

```
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 Data Types

1.2.1 Integer Types

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

Rust	C++	C & C++
`u16`	`uint16_t`	`unsigned short`
`u32`	`uint32_t`	`unsigned int`
`u64`	`uint64_t`	`unsigned long`
`u128`		
`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.

1.2.2 Floating Point Numbers

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

1.3 Variables

Like in Rust, creating a variable requires two steps, declaration and initialization.

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 (recent compilers will give a warning when this happens; see Figure 1.1). C++11 introduced the “uniform initialization” syntax, which forbids this implicit conversion. In its simplest form:

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

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

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:

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

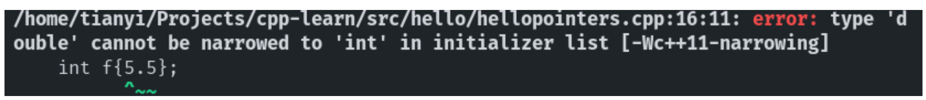

Figure 1.2: The uniform initialization syntax.

Of course, the uniform initialization syntax isn’t invented just to prevent implicit conversion. As you’ll see later, it can become handy when initializing complicated, non-primitive data types.

1.3.1 Mutability

Variables are mutable by default. If you want to create an immutable variable, use the `const` keyword.

```
// Rust
let a = 5;
let mut b = 10;
```

is equivalent to

```cpp
//C++
const auto a = 5;
auto b = 10;
```

If `const` is used to create an immutable variable, how to create a real "constant" that's evaluated at compile time? The answer is `consexpr`.

```rust
// Rust
const LIGHT_SPEED: f64 = 2.99792458e8;
```

is equivalent to

```cpp
constexpr double LIGHT_SPEED = 2.99792458e8;
```

What about constant strings? Well, like Rust, you can't use the dynamically allocated `std::string`.

```rust
// Rust
const NAME: String = "Hideyo".to_string(); // not allowed!
const NAME: &'static str = "Hideyo";        // good
```

```cpp
//C++
constexpr std::string NAME = "Hideyo"; // not allowed!
constexpr char NAME[] = "Hideyo";      // good
```

So you have to use an array of characters. Well, it's technically closer to `const NAME: &[u8] = b"Hideyo";`. Then, if you need to use the `std::string`, you need explicit conversion:

```cpp
#include <iostream>

constexpr char NAME[] = "Hideyo";
constexpr char NAME_UTF8[] = " 英世";

int main()
{
    const std::string NAME_STRING(NAME);
    const std::string NAME_UTF8_STRING(NAME_UTF8);
    // thank god UTF8 works! If you were in C you would have a hard time.
    std::cout << NAME_UTF8_STRING << " " << NAME_STRING << std::endl;
}
```

1.4 Functions

Functions are declared and defined in the following way:

```
<return_type> <function_name>(<params>) {
    // do something
    return something;
}
```

For example:

```cpp
#include <iostream>

float square(float a)
{
    return a * a;
}

int main()
{
    float x = 2.5;
    std::cout << "square of " << x << "is" << square(x) << std::endl;
}
```

Note that a function must be *declared* before it can be used. This means the following won't work:

```cpp
#include <iostream>

// float square(float);

int main()
{
    float x = 2.5;
    std::cout << "square of " << x << "is" << square(x) << std::endl;
}


float square(float a)
{
    return a * a;
}
```

However, by uncommenting the third line, it works. When you use a function, the compiler must know the signature, but not necessarily the *definition* of the function. This is why in C++ (and in C) people split their funtions into signatures which go into header files, and definitions which go into `.cpp` files. I'll get back to header files later. For now we'll be working with single-file programs without splitting declaration and definitions.

1.5 Pointers and References

Let's revisit how we make references and pointers in Rust:

In Rust, when you take a reference to a type `T`, the type of the reference is `&T`. Basically, you add `&` to both LHS and RHS:

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

and when you deference, you use `*`. Just remember that `*` is the reverse of `&`, and every `*` removes one `&` from both LHS and RHS:

```
let r_a: &i32 = &5;
let   a:  i32 = *r_a; // a == 5
let   a:  i32 = *&*&*&a
```

You can coerce a reference into a raw pointer, using either of the two syntaxes:

```
let a = 5;
let p_a: *const i32 = &a;
// or
let p_a = &a as *const i32;
```

You need to be explicit about mutability:

```
let a = 5;
let p_a: *mut i32 = &mut a;
// or
let p_a = &mut a as *mut i32;
```

In C++, the difference between a reference and pointer is smaller. Pointers are compatible with C, and references are a C++-only thing.

In C++ (and C), this is how you 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. 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 (`int *p_a = &a;`). There are some discussion at StackOverflow on why people are preferring this style. I personally prefer using the `int* p_a = &a` style, which is also being used consistently on cppreference, and in Bjarne Stroustrup's *A Tour of C++*.

Anyway, you can choose whichever form you like when you write your code, but you need to able to read all forms so that you can read others' code.

References have similar capabilities as pointers, with the only big difference being their semantics. This is how you create a reference and read its referent's value:

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

Note that you can directly create a pointer, but not a reference to a literal, that is to say:

```
int* i = &5; // is valid
int& j = 5;  // is not allowed
```

You do not need to (and cannot) use the deference operator (`*`) on a reference to access the value of the referent. In addition, a reference cannot be re-assigned to refer to another value. Apart from these two rules, references are effectively the same as pointers. It can be helpful to see a reference as an *alias* to a *named variable*. Indeed, a reference shares the same memory address as its referent.

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 used 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|
```

I think it would be helpful to make a line-by-line comparison of some common tasks in Rust and in C++:

1.5.0.1 Reading The Value Without Mutation with A Pointer or Reference

step	C++ (pointer)	C++ (reference)	Rust	Rust (raw pointer)
init referent	`const int a = 5`	`const int a = 5`	`let a = 5`	`let  a = 5`
make ptr/ref	`const int* p = &a`	`const int& r = a`	`let r: &i32 = &a`	`let p: *const i32 = &a;`
read referent value	`*p`	`r`	`*r`	`*p`

1.5.0.2 Mutating the Value of the Referent with A Pointer or Reference

step	C++ (pointer)	C++ (reference)	Rust	Rust (raw pointer)
init referent	`int a = 5`	`int a = 5`	`let mut a = 5`	`let mut a = 5`
make ptr/ref	`int* p = &a`	`int* r = &a`	`let r: &mut i32 = &mut a`	`let p: *mut i32 = &mut a;`
mutate	`*p = 10`	`r = 10`	`*r = 10`	`*p = 10`

1.6 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.

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.

1.6.1 `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;
```

The braces around the statement after the condition of `if` can be omitted if the statement can be written in a single line. For example:

```
if (i > 5) {
    std::cout << "i is greater than 5" << std::endl;
}
```

can be reduced to:

```
if (i > 5)
    std::cout << "i is greater than 5" << std::endl;
```

1.6.2 `while` Loop

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

```
#include <iostream>
int i = 10;
while (i > 0)
{
    i--;
    if (i == 8)
    {
        continue;
    }
    if (i == 5)
    {
        break;
    }
    std::cout << i << std::endl;
}
```

1.6.3 `for` Loop

A traditional C-style `for` loop looks like this:

```
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|\n", 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++.

1.6.3.1 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.

1.6.3.2 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!
}
```

1.6.3.3 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`.

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

1.6.4 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);
        }
    }
}
```

C++ doesn't support this natively, so it's common to use the `goto` trick to achive this[2].

```
#include <iostream>
printf("Break outer loop using goto:\n");
std::cout << "ijk " << std::endl;
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 for_j_end;
            }
            std::cout << i << j << k << std::endl;
        }
    }
    for_j_end:
    {
    }
}
```

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

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

```cpp
printf("Break outer loop using flag:\n");
std::cout << "ijk " << std::endl;
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)
        {
            if (i == 1)
            {
                i_is_1 = true;
                break;
            }
            else
            {
                i_is_1 = false;
            }
            std::cout << i << j << k << std::endl;
        }
        if (i_is_1)
        {
            break;
        }
    }
}
```

There is another cleaner way, using the lambda trick (less common than the `goto` approach):

```cpp
printf("Break outer loop using lambda:\n");
std::cout << "ijk " << std::endl;
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 << j << k << std::endl;
```

```
            }
        }
    }();
}
```

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

1.6.5 `switch`

C++'s `switch` is essentially a shortcut for a series of `if...else if...else if...else` statements when you want to match the value a single expression to a range of constant values. For example:

The following C++ code

```
int d = 5;
if (d == 0) {
    printf("It's Sunday!\n");
} else if (d == 6) {
    printf("It's Saturday!\n");
} else {
    printf("It's weekday.\n");
}
```

is equivalent to:

```
switch (d)
{
case 0:
    printf("It's Sunday!\n");
    break;
case 6:
    printf("It's Saturday!\n");
    break;
default:
    printf("It's weekday.\n");
    break;
}
```

(I'll start to omit headers from now on.)

Note the `break` statement at the end of each `case`, without which the `default` branch will always be triggered, which is clearly not we mean to do[3].

[3]This StackOverflow question might be interesting.

You can group several values into a single branch:

```
int d = 5;
switch (d)
{
case 0:
    printf("It's Sunday!\n");
    break;
case 6:
    printf("It's Saturday!\n");
    break;
case 1:
case 2:
case 3:
case 4:
case 5:
    printf("It's weekday.\n");
    break;
default:
    printf("Not a valid day of week!\n");
    break;
}
```

This is all about `switch` in C++. (You need to unlearn the pattern matching in Rust)

Exercise

1. **Pointers and references.**

```
int x = 5;
int y = 10;
int* px = &x;
int* py = &y;
px = py;
// what are the values of x, y, *px and *py now?

int i = 5;
int j = 10;
int& ri = i;
int& rj = j;
ri = rj;
// what are the values of i, j, ri and rj now?
```

Chapter 2

Enumerations

Like Rust, C++ has `enum`s. Unlike Rust, their `enum`s are much less powerful. C++'s enumeration comes in two flavors: the plain, C-compatible `enum`, and the `enum class`. Generally, you should always use an `enum class`, but you should also learn about the plain `enum` in order to read others' code.

2.1 The Plain `enum`

2.1.1 Defining an Enum

The syntax for defining an `enum` in C++ is similar to Rust. In addition, like in Rust, enum variants are represented as integers at the low level, and by default the value starts from 0.

```
enum LogLevel
{
    Debug,   // 0
    Info,    // 1
    Warning, // 2
    Error    // 3
};
```

You can also manually assign values to the variants (also valid in Rust):

```
enum LogLevel
{
    Debug   = 0x12,
    Info    = 0xd1,
```

```
    Warning = 0x7c,
    Error   = 0x0a,
};
```

Unlike Rust, C++ by default uses the `int` type to represent the variants, even though most of the time the number of variants won't exceed 256. To use a more compact representation, you can do:

```
enum MyEnum : uint8_t
// or `enum Foo : char` (remember that `char` is an integer type)
{
    Foo,
    Bar,
};
```

2.1.2 Using an Enum

Unlike Rust, a C++ `enum` does not create a namespace, which means you cannot write `LogLevel::Info` to refer to the `Info` variant of the `LogLevel` enum defined in the prevous subsection. You should write `Info` directly.

A enum is implicitly converted to an integer. Which means it can be directly compared to an integer (but dont't actually do this) and can be pushed to `std::cout` directly.

```
LogLevel lvl = Info;
assert(Info == 1);
assert(Warning > 1);
assert(Error > Debug);
std::cout << lvl << std::endl;
```

However, an integer cannot be converted implicitly into an enum, but can be done so explicitly:

```
LogLevel lvl = 2;           // won't work
LogLevel lvl = (LogLevel)2; // OK, but don't actually do this
```

You can reassign the identifier `Info` to another value

```
int Info = 999;
```

...but can you set another `LogLevel` to `Info`?

```
LogLevel m = Info; // Error!
```

No you can't. C++ generally forbids re-defining variables, but in the case of enums you are allowed to re-bind a enum variant identifier (`Info`) to another value (`999`), then you just can't use that enum variant.

To avoid this and other kinds of conflicts, it had been a common practice to enum variants with part of the enum name. For example, the `LogLevel` enum may be rewritten as:

```
enum LogLevel
{
    LevelDebug,
    LevelInfo,
    LevelWarning,
    LevelError
};
```

…which violates the DRY rule in a bad way. To solve this problem, `enum class` was introduced in C++11, and we'll learn about it in the next section.