

C++: From Code to Execution

Ryan Baker

December 31, 2024

Contents

1	Introduction to C++	2
2	C++ Developer Tools	2
2.1	Text Editor	2
2.2	Compiler	3
2.3	Hello, World!	3
3	The C++ Build Process	4
3.1	Source Code	4
3.2	Preprocessor	4
3.3	Compiler	5
3.4	Linker	5
4	C++ Programming Basics	6
4.1	Types and Variables	6
4.2	Input and Output with <code>iostream</code>	7

Lecture Objectives

By the end of this lecture, you should:

- Understand the developer tools needed to write C++
- Understand the build process: preprocessing, compilation, linking
- Be able to utilize types, arithmetic, and I/O to write a basic C++ program

1 Introduction to C++

- What is C++?
 - General-purpose programming language
 - **Brief history:** Development, standardization, updates
 - **Use cases:** embedded systems, HFT, OS, etc.
- Why learn C++?
 - There has been a concerted effort to push C++ to the side
 - * Some valid concerns (some invalid)
 - * We will investigate throughout the course
 - Performance, flexibility, power (sharpest tool in the toolbox)
 - C++ has a very strong “knowledge passport”
 - * Becoming proficient at C++ is not easy
 - * Proficiency in C++ translates very well to other languages

2 C++ Developer Tools

Two basic tools are needed to develop C++: a **text editor** and a **compiler**. You may also use an **Integrated Development Environment (IDE)** which is a single tool that bundles both.

2.1 Text Editor

- What is a text editor? A tool to edit text.
 - Any tool that lets you edit text is technically workable
- Which text editor should I use?
 - **Visual Studio Code IDE:** a beginner-friendly option, very popular
 - **CLion IDE:** Premium IDE made by JetBrains, widely appreciated
 - **Vim/Nvim:** My choice, steep learning curve, more rewarding

- * Not an IDE, download instructions depend on your OS
- * Look up “vim install” or “neovim install”
- * I recommend spending some time configuring your setup, feel free to talk to me for recommendations

2.2 Compiler

- What is a compiler?
 - A compiler is a tool that turns your code into an executable program
 - Performs lexical analysis, syntax checking, and optimization
- Which compiler should I use?
 - **gcc/g++**: One of the best. Download depends on OS
 - **clang/llvm**: My personal choice. Download depends on OS
 - * Best for Mac
 - **MSVC**: Microsoft visual compiler, a tier below **clang** and **gcc** (IMO)
 - * An easy option if you’re developing on windows

2.3 Hello, World!

With your text editor of choice, write the following code:

```
1 // helloworld.cpp
2 #include <iostream>
3
4 int main()
5 {
6     std::cout << "Hello, World!" << std::endl;
7     return 0;
8 }
```

Compile your code using your compiler of choice:

```
>> g++ helloworld.cpp -o exe
>> clang++ helloworld.cpp -o exe
```

These commands will compile `helloworld.cpp` and output an executable `exe`. You can run the executable file:

```
>> ./exe
Hello, World!
```

The remainder of this lecture will be spent on understanding the process that transforms `helloworld.cpp` into an executable that prints “Hello, World!”.

3 The C++ Build Process

How do we get from the .cpp file to an executable program? Answering this question will allow us to understand and debug our programs more efficiently.

3.1 Source Code

- **Source code:** Human-readable code written in .cpp files
- `main.cpp` → `Preprocessor` → `Compiler` → `Linker` → `main.exe`
- **Key point:** Source code is just a text file:
 - >> `cp helloworld.cpp ryan.baker`
 - >> `clang++ -x c++ ryan.baker -o exe && ./exe`
 - * There is nothing special about .cpp files. The compiler just needs text written in a language called C++.
 - * `-x` is a flag that clang takes that says: “The language is C++, even if the file extensions are weird”

3.2 Preprocessor

- What does the preprocessor do?
 - Text substitution: `#define MAX 500` replaces “MAX” with “500”
 - File inclusion: `#include <iostream>` copies and pastes `iostream`
 - * `#include "filename"`: searches current folder, use for local files
 - * `#include <filename>`: searches standard include directories
 - Conditional compilation: `#if` and others to select code that compiles

```
1 // preprocessor.cpp
2 #include <iostream> // pastes 'iostream' file
3
4 #define MAX 500 // replaces "MAX" with "500"
5
6 // conditional compilation with #if, #ifdef, #ifndef
7 #if (MAX > 1000) // evaluates to false
8     #define STATUS 20 // does not compile
9 #else
10     #define STATUS 10 // compiles
11 #endif
12
13 int main()
14 {
15     std::cout << "Hello, World!" << std::endl;
16     std::cout << MAX << std::endl;
17     std::cout << STATUS << std::endl;
18     return 0;
19 }
```

- *Probably won't discuss #pragma*
 - Less relevant in C++ than in C, usually bad practice
- Use of -E to view preprocessor output


```
>> clang++ -E preprocessor.cpp > preprocessor_output.cpp
      * iostream file has been copied and pasted in
      * macros MAX and STATUS have been expanded to their values
```
- **Key point:** `preprocessor_output.cpp` is the same as `preprocessor.cpp`

```
>> clang++ preprocessor_output.cpp -o exe && ./exe
      * This command produces the same thing as the first program
```

3.3 Compiler

- What does the compiler do?
 - Reads the output of the preprocessor and turns it to machine code
 - Responsible for alerting the user about various types of errors
 - Outputs object files (.o or .obj)
 - * Mostly machine code, has some directives for the linker to use
- Use of -c to view the object file (completely unreadable)
 - -c stands for “just compile” (don't link)


```
>> clang++ -c helloworld.cpp produces helloworld.o
>> clang++ helloworld.o -o exe && ./exe runs the program
```
- Use of -S to view the assembly output (readable)
 - Contains traces of our original program


```
>> clang++ -S helloworld.cpp produces helloworld.s
>> clang++ helloworld.s -o exe && ./exe runs the program
```

3.4 Linker

- What does the linker do?
 - Resolves symbols, matching declarations to definitions
 - Combines multiple translation units into an executable
 - This allows us to write code across files, enhancing modularity

```
1 // file2.hpp
2 void greet();
```

```

1 // file1.cpp
2 #include <iostream>
3 #include "file2.hpp"
4
5 int main()
6 {
7     std::cout << "Hello, World!" << std::endl;
8     greet();
9     return 0;
10 }

1 // file2.cpp
2 #include <iostream>
3 #include "file2.hpp"
4
5 void greet()
6 {
7     std::cout << "Greetings from file2.cpp!" << std::endl;
8 }

```

- **Key point:** `clang++ file1.cpp -o exe` produces a linker error
 - The symbol “`greet()`” is not defined
 - Recognize the difference between compilation and linking errors
 - * Linker errors are often more convoluted
 - * Often denoted by “`ld:`” or “`linker error:`”

```
>> clang++ file1.cpp file2.cpp -o exe
```

- We need to pass in `file2.cpp` to the linker

4 C++ Programming Basics

4.1 Types and Variables

All programming involves storing and manipulating data, typically in variables. A variable’s *datatype* defines the set of values it can hold. For example, a `character` datatype represents letters like ‘a’ through ‘z’, while a `boolean` datatype represents true or false.

C++ has some built-in datatypes, called *primitive* or *integral* datatypes. Each datatype is designed to serve a different purpose. However, with a low-level programming language such as C++, **the only real difference** between any of these datatypes is **the amount of memory** they occupy.

- `int`: represents an integer
 - >> `int x = 42;` assigns the value 42 to a variable ‘x’
 - >> `sizeof(x); sizeof(int)` returns the # of bytes ‘x’ occupies

- Because ‘x’ occupies a finite number of bytes, its range is limited
 - * We can calculate its total range as 2^w where w represents the width of ‘x’ in bits
 - * Note that the maximum value may be halved if ‘x’ is signed
- If we want a smaller int, we use `short`. If we want a longer int, we can use `long` or `long long`
 - * I find these names very confusing
 - * I recommend `#include <cstdint>`
- Signedness: we can prepend a ‘u’ or `unsigned` to the type to make the number unsigned. This expands its positive range
- `char`: represents a character


```
>> char letter = 'a'; assigns 'a' to 'letter'
```

 - * ‘a’ is really just a number

```
>> int x = 'a'; std::cout << x << std::endl;
```
- `float`, `double`: represents a floating-point (fractional) type
 - `double` is (usually) twice as large as a `float`
 - `sizeof(double) = 8`, `sizeof(float) = 4` (usually)
- `bool`: represents a boolean value (True or False)
 - `sizeof(bool) = 1` (usually)
- `void`: represents “no type”
 - `sizeof(void)` is a senseless operation, produces an error

4.2 Input and Output with `iostream`

Our programs are useless unless we can communicate with them. C++ provides various methods of passing data into and out-of our programs. The `iostream` library is the most widely used library for input and printing data in C++.

- “`iostream`” stands for input and output (IO) stream


```
>> #include <iostream>
```
- Output with `iostream`
 - `std::cout` is used to print data to the console
 - * `std::` is a namespace access, says search namespace “`std`” (standard) for function called “`cout`”
 - * We will discuss namespaces later in the course
 - `std::endl` is used to output a newline and flush the buffer

- The ‘<<’ operator is the output stream operator
- Input with `istream`
 - `std::cin` is used to fetch data at runtime from the user
 - The ‘>>’ operator is the stream input operator

We will discuss the workings of `istream` more in depth when we discuss **streams**. Presently, just get familiar with the syntax of `cin` and `cout`.

```
1 // iostreamio.cpp
2 #include <iostream>
3
4 int main()
5 {
6     int number; // declare an integer variable called "number"
7
8     std::cout << "Enter a number: "; // print a prompt message
9     std::cin >> number; // accept a number as input
10
11     std::cout << "You entered: " << number; // print the number
12     std::cout << std::endl;                // print a newline
13
14     return 0;
15 }
```