Introduction to C++

Ryan Baker

January 3, 2025

Contents

1 Course Introduction

1.1 Lecture Series Overview

Week 1. Introduction to C++
Week 2. How C++ Works
Week 3. C++ Control Flow
Week 4. Introduction to Object-Oriented Programming
Week 5. Advanced Object-Oriented Programming
Week 6. The Standard Library
Week 7. Safety in C++
Week 8. Templates
Week 9. Compile-Time Programming

1.2 Why Learn C++

C++ is a powerful, versatile, and foundational programming language that remains relevant in modern software development.

Performance and Efficiency: C++ is known for its high performance and low-level memory control, making it ideal for performance-critical applications such as game engines, real-time systems, and high-frequency trading.

Foundational Language: Many languages are inspired by or built on C++ concepts. Learning C++ provides a solid foundation for understanding these languages.

Extensive Ecosystem: C++ has a rich ecosystem of libraries and frameworks that make it easier to build powerful and complex systems efficiently.

2 Features of C++

2.1 Evolution of C++

C++ has evolved from a simple extension of C into a modern multi-paradigm programming language:

1979: C++ was created by Bjarne Stroustrup as a simple extension of C.

1998: C++98 standard formalized the language, solidifying its features.

2011: C++11 brings the largest update the language has seen, modernizing it.

2020: C++20 brings features that make the language more powerful and useable.

2026: C++26 is in development, and will likely bring with it reflection and more.

2.2 The C++ Philosophy

C++ is a sharp tool. It offers great control over system resources and performance, but requires careful and skilled use. It combines low-level features, like pointers, with high-level abstractions, enabling efficient, optimized code. While powerful, C++ comes with complexity, demanding expertise to avoid issues like memory leaks or crashes. Its flexibility makes it ideal for performance-critical applications, but it requires precision and understanding to use effectively.

2.3 C++ vs. Other Languages

Compiled vs. Interpreted: C++ is a *compiled* language, meaning code is directly translated into machine code, which typically leads to faster execution compared to interpreted languages like Python or JavaScript. This makes C++ ideal for performance-critical applications. *Interpreted* languages are read and executed line-by-line and often have a simpler developer experience.

Strongly Typed: C++ uses *strong typing*, meaning types are strictly checked at compile time. This reduces runtime errors and leads to better performance and reliability. Dynamically typed languages don't have the ability to detect type errors before run time.

Multi-Paridigm: C++ supports multiple programming paradigms, including procedural, object-oriented, and generic programming. This flexibility allows developers to use the best paradigm for the task at hand.

3 Environment Setup

3.1 Tools Required

To develop C++, you need two basic tools: a **text editor** and a **compiler**.

3.1.1 Text Editor

- What is a text editor? A tool at edits text. `// duh`
- **Text Editor vs. IDE:** A text editor is a basic tool for writing plain text, while an IDE (Integrated Development Environment) is a more comprehensive tool that includes a code editor, debugging tools, code completion, an build automation.
- Some popular text editors include:
  - **Visual Studio Code:** A free, open-source IDE with support for C++ through extensions.
  - **CLion:** An IDE specifically built for C++ with advanced features.

- **Vim/Neovim:** My personal choice. Has a steep learning curve, but absolutely worth it.
  - If you decide to go with Vim or Neovim, I recommend spending some time configuring your setup. Feel free to ask me for help.

3.1.2 Compiler

- **Definition:** The compiler converts your C++ source code into machine-readable instructions.
- Common C++ compilers include:
  - **gcc:** The GNU Compiler Collection, a popular open-source compiler. Best for Windows OS and Linux.
  - **clang:** My personal favorite, known for its performance and diagnostics. Best for Mac OS.
  - **MSVC:** An increasingly irrelevant piece of garbage. Second best for Windows.
- Throughout the lecture series, I will be using `clang`. If a certain `clang` flag or directive does not work for your compiler, simply look up its equivalent.

3.2 "Hello, World!" Example

With your text editor of choice, write the following C++ program:

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

Compile and run the program with your compiler of choice. You should see "Hello, World!" printed to the console.

4 Basic Syntax and Structure

4.1 Basic Structure of a C++ Program

4.1.1 `int main()`

- **Entry Point:** The `main()` function is where every C++ program starts executing. It serves as the "entry point" for a program.
- **Return Value:** `main()` returns an integer to indicate the program's exit status. By convention, returning `0` means successful execution.

- In modern C++, `return 0` is implicit

```
int main() {} // the shortest complete C++ program
```

4.1.2 Statements and Expressions

The code inside the `main()` function is made up of statements (commands).

4.2 Foundational Concepts

4.2.1 Semicolons, `/* comments */`, and Whitespace

- **Semicolons:** Every statement in C++ ends with a semicolon:

  ```
  std::cout << "Hello, World!" << std::endl;
  ```

  - This lets the compiler know that the line is finished

- **Comments:** Comments are a way of "taking notes" within your code. They are ignored by the compiler entirely, but help you and other developers understand the code.

- There are two ways of writing comments:

  - **Single Line:** Single line comments are prefixed with `//`:

    ```
    int main() // main function serves as the entry point
    ```

  - **Block:** Block comments are written with `/* */` notation:

    ```
    int /* why is there a comment here? */ main()
    ```

  - *Ryan's Advice:* Lightly prefer `//` to `/* */` because the parsing of `/* */` is more complicated and can lead to errors if you aren't careful.

- **Whitespace:** C++ could not care less about whitespace. Whitespace includes spaces, tabs, and newlines.

  - This means that is its possible, although not generally recommended, to write a C++ program in one line of code:

  ```
  int main() { std::cout << "Hello, World!" << std::endl; }
  ```

4.2.2 Line-by-Line Execution

In C++, the program executes statement sequentially, starting from the top of `main()` and moving downward.

```
int main()
{
    std::cout << "First"  << std::endl;
    std::cout << "Second" << std::endl;
}
```

4.3 Input and Output with `iostream`

- `std::cout` (character out) is used for printing output to the console.
- The **insertion operator** `<<` is used to send data to the output stream:

```
std::cout << "Hello, World!" << std::endl;
```

- `std::cin` (character in) is used to get input from the user via the console.
- The **extraction operator** `>>` is used to read from the input stream:

```
std::cin >> data; // reads into a variable called "data"
```

- `std::endl` is used to inset a newline character and flush the output buffer.

```
#include <iostream>

int main()
{
    int age;
    std::cout << "Enter your age: ";
    std::cin >> age;
}
```

5 Datatypes and Variables

All of programming is manipulating and interpreting data. Data comes in different shapes and sizes, forming a *datatype*. An instance of a datatype is stored in a *variable*.

5.1 `sizeof` Operator

The `sizeof` operator in C++ is used to determine the size, in bytes, of a data type or object:

```
std::cout << sizeof(int) << std::endl; // usually 4
```

Arguably, the only difference between any two datatype is their size. At the end of the day, all datatypes are made up of bits, and it is only how we interpret their bits that gives them any meaning.

5.2 Primitive Types

C++ provides several built-in datatypes, called *primitive* types:

`int`: Represents integer values (e.g., 5, -10, 42)

- Integers can be *signed* (±) or *unsigned* (only +).
- There are several integer modifiers for both the signed-ness and size:

`short`: Short integer, `sizeof(short)` = 2 (usually)
`long`: Long integer, `sizeof(long)` = 4 (sometimes 8)
`long long`: Extra long integer, `sizeof(long long) = 8 (usually)`
`unsigned`: Modifies any integer to be unsigned

Ryan's Advice: I find these keywords to be rather confusing, especially because their sizes are not defined by C++. I tend to use the `cstdint` header which defines all the same types:

```
#include <cstdint>

int main()
{
    int32_t x = 0;  // guaranteed 32 bits
    int64_t y = 0;  // guaranteed 64 bits
    uint16_t z = 0; // 'u' means unsigned
}
```

- Because a computer's memory is finite, so too is the range of an `int`.
  - The maximum value of an `unsigned int` can be calculated as 2^w where w is the width of the `int` in bits. The minimum is 0.
    - The range of a `uint32_t` is $[0, 2^{32} - 1] = [0, 4294967295]$
  - For signed integers, the maximum value is $2^{w-1} - 1$ and the minimum value is -2^{w-1} where w is the width in bits.
    - The range of an `int32_t` is $[-2147483648, 2147483647]$

`char`: Used to store single characters (e.g., 'a', '2')

- Characters are just integers in disguise. Every character has a corresponding integer value according to the **ASCII Table**.

```
std::cout << int('a') << std::endl; // 97

std::cout << char(97) << std::endl; // 'a'
```

- `sizeof(char)` = 1 (usually)

`bool`: Represents boolean values (`true` or `false`)

- Booleans are just integers in disguise.

```
std::cout << int(true) << std::endl; // 1
```

```
std::cout << int(false) << std::endl; // 0
```

- `sizeof(bool)` = 1 (usually)

`float`: Represents fractional values (e.g., 5.5, -5.5, 3.14159)

- `sizeof(float)` = 4 (usually), `sizeof(double)` = 8 (usually)
  - `double` (precision) is a larger, more precise version of a float.
- Casting a `float` to an `int` truncates the decimal portion:

```
std::cout << int(5.5) << std::endl; // 5
```

`void`: Represents "no type"

- `sizeof(void)` is a senseless operation and throws errors.

5.3 Declaration and Definition

Creating variables can be broken into two steps: **declaration**, and **definition**.

- **Declaration:** "I declare that this variable exists!":

```
int x; // x exists! Who cares about its value?
```

- **Definition:** "I define this variable to be __!":

```
x = 42; // x is defined to be 42!
```

- Often times it makes sense to combine declaration and definition:

```
int x = 42; // declared and defined
```

5.3.1 Assignment Operator =

The **assignment operator** = is used to assign a value to a variable:

```
variable = value;
```

```
int x = 5; // assign 5 to x
int y;
y = x; // assign the value of x to y
```

- **Chaining:** The assignment operator can be chained:

```
int a, b, c, d; a = b = c = d = 5;
```

- **Reassignment:** Can be used to reassign a new value to a variable:

```
int a = 5; a = 6; a = 7;
```

5.3.2 Brace Initialization {}

Brace initialization is a feature introduced in C++11 that allows variables to be initialized using curly braces {}. This provides a more uniform and safer way to initialize variables.

- **Syntax:** `int x{5}; // initializes x to 5`
- Brace initialization prevents narrowing conversions which occur when a larger data type is assigned to a smaller one:

```
int x = 5.5; // 5.5 gets truncated, but no error
int y{5.5};  // an error gets thrown
```

- Default Initialization: For primitive types, if no value is provided, the variable will be value-initialized:

```
int x{}; // x is initialized to 0
```

- *Ryan's Advice:* Prefer brace initialization to initialization with =.

5.4 Arithmetic Operators

Operator	Description	Example
+	Adds two operands	x + y
-	Subtracts two operands	x - y subtracts y from x
*	Multiplies two operands	x * y multiplies x and y
/	Divides two operands	x / y divides x by y
%	Remainder of a division	x % y = remainder of x / y
++	Increments the operand	++x or x++ increases x by 1
--	Decrements the operand	--x or x-- decreases x by 1

Table 1: Basic Arithmetic Operators in C++

- Basic arithmetic operators can be combined with the assignment operator to perform both operations at once:

```
int x = 10;
x += 10; // same as x = x + 10
x /= 4;  // same as x = x / 4
```