Gorgo Starting C++

April 11, 2026

Contents

1. Introduction **2**
Hello, World! . . . 2
Semicolons and Curly Braces . . . 3
Compiling and Running . . . 3
Namespaces . . . 3
Output with std::cout . . . 4
Escape Sequences . . . 4
Input with std::cin . . . 6
Command-Line Arguments . . . 7
Try It . . . 7
Key Points . . . 8
Exercises . . . 8

1. Introduction

Welcome to C++. A program that cannot communicate is a black box — it runs, but you have no way to see what it did or tell it what to do. Input and output are the first skills every programmer needs. In this chapter you will write your first program, learn how to compile and run it, and get comfortable with basic I/O using `std::cout` and `std::cin`. By the end, you will be able to write programs that talk to the user and respond to command-line arguments.

Hello, World!

Every programming journey begins with the same tradition: printing "Hello, World!" to the screen. Here is what that looks like in C++:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

That is not a lot of code, but there is a lot going on. Let's break it down.

`#include <iostream>` tells the compiler to include the **iostream** library, which gives you tools for input and output. Think of it as importing functionality that someone else already wrote for you.

`int main()` is the entry point of every C++ program. When you run your program, execution starts here. The `int` before `main` means the function returns an integer value. By convention, returning `0` means the program finished successfully.

`std::cout << "Hello, World!" << std::endl;` is the line that actually prints text to the screen. We will explain what `std::` means shortly. The `<<` operator sends data to the output stream.

`std::cout << "Hello, World!"` writes `Hello, World!` to the screen, with two caveats. First, `Hello World!` is not on its own line. If something else is output, it will appear on the same line right after `!`. Second, `cout` uses **buffer**ing — a technique to improve performance by collecting data to output, so that it can do fewer writes to the terminal. When writing to an actual terminal, starting a new line will also cause **buffered** – collected – data to be written to the screen. `<< std::endl` addresses both of these issues. It starts a new line so that subsequent text will be written to the next line of output, and it **flush**es the buffer by immediately writing any previously buffered output.

The `<<` operator has this signature – this may not mean much to you now, but for consistency we are showing you here:

```cpp
std::ostream& operator<<(std::ostream& os, const T& value);
```

It takes an output stream and a value, writes the value to the stream, and returns the stream so you can chain multiple `<<` together.

`std::endl` ends the line and flushes the output buffer. It is actually a function with this signature:

```cpp
std::ostream& endl(std::ostream& os);
```

It writes a newline character and then flushes the stream.

Tip: You can also use `"\n"` instead of `std::endl` to end a line. The difference is that `std::endl` flushes the output buffer, while `"\n"` does not. For most programs, `"\n"` is fine and slightly faster.

Semicolons and Curly Braces

Every statement in C++ ends with a semicolon (`;`). A statement is a single instruction — printing text, declaring a variable, returning a value. Forgetting a semicolon is one of the most common mistakes beginners make, and the compiler error message can be confusing because it often points to the *next* line rather than the line with the missing semicolon.

Curly braces (`{` and `}`) define a **block** of code. In the Hello World program, the braces after `int main()` mark the beginning and end of the `main` function's body. Everything between `{` and `}` belongs to that function.

```
int main()
{                   // start of block
    // statements go here
    return 0;
}                   // end of block
```

You will see curly braces everywhere in C++ — around functions, loops, and if statements. They always come in pairs: every `{` needs a matching `}`.

Tip: Indent the code inside curly braces to make it easier to read. Most C++ programmers use 4 spaces per level of indentation.

Compiling and Running

C++ is a **compiled** language. You write your source code in a text file, then use a compiler to translate it into a program your computer can run.

Save the Hello World program to a file called `hello.cpp`. C++ source files typically end in `.cpp`.

To compile it, open a terminal and run:

```
c++ -o hello hello.cpp
```

This tells the compiler to take `hello.cpp` and produce an executable called `hello`. The `-o hello` part names the output file.

Now run it:

```
./hello
Hello, World!
```

Congratulations, you just compiled and ran your first C++ program!

Tip: Always compile with warnings enabled. Use `c++ -Wall -Wextra -pedantic -o hello hello.cpp` to catch potential problems early. The compiler is your friend — listen to its warnings.

If you get an error, read the error message carefully. The compiler will tell you the file name and line number where the problem is. Common mistakes include forgetting a semicolon at the end of a line or misspelling `iostream`.

Namespaces

You have been typing `std::cout` and `std::endl`, and you might be wondering what the `std::` part is about.

C++ uses **namespaces** to organize code and avoid naming conflicts. The standard library puts all of its names inside a namespace called `std`. When you write `std::cout`, you are saying "I want the `cout` that lives inside the `std` namespace."

The `::` is called the **scope resolution operator**. You will see it a lot in C++.

If you get tired of typing `std::` everywhere, you can add a **using directive** at the top of your file:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

Now you can write `cout` instead of `std::cout`.

Trap: Using `using namespace std;` in large programs or header files can cause naming conflicts. It is fine for small programs while you are learning, but be aware that many professional C++ programmers avoid it.

For the rest of this book, we will use `std::` explicitly so you always know where names come from.

Output with std::cout

`std::cout` is the **standard output stream**. You send data to it using the `<<` operator, and that data appears on the screen.

You can chain multiple `<<` operators together to print several things on one line:

```
#include <iostream>

int main()
{
    std::cout << "Come as you are" << ", " << "as you were" << std::endl;
    return 0;
}
```

Output:

```
Come as you are, as you were
```

You can also print numbers directly:

```
std::cout << "The year is " << 1991 << std::endl;
```

Output:

```
The year is 1991
```

Escape Sequences

You have already seen `std::endl` end a line. There is another way to do the same thing: put `\n` *inside* the string.

```
std::cout << "Smells Like\nTeen Spirit\n";
```

Output:

```
Smells Like
Teen Spirit
```

The `\n` is not two characters — it is a single character called **newline**. The backslash tells the compiler "the next character is special, do not take it literally." This is called an **escape sequence**.

Why do you need escape sequences at all? Because some characters cannot be written literally inside a string. A double quote (`"`) marks the *end* of a string, so this does not work:

```
std::cout << "She said "hello" and walked away" << std::endl;  // ERROR
```

The compiler sees `"She said "`, thinks the string ends there, and gets confused by the leftover `hello" and walked away"`. You have to **escape** the inner quotes with `\"` to tell the compiler they are part of the string:

```
std::cout << "She said \"hello\" and walked away" << std::endl;
```

Output:

```
She said "hello" and walked away
```

The same problem happens with single quotes inside a **character literal**. A character literal is one character wrapped in single quotes, like `'A'` or `'?'`. Writing `'''` is ambiguous — is the second `'` ending the literal, or is it the character? You escape it with `\'`:

```
char apostrophe = '\'';   // a single-quote character
// no escape needed in a char literal
char quote      = '"';
```

Inside a string literal, the rule flips: `"` needs escaping but `'` does not. Inside a char literal, `'` needs escaping but `"` does not. You only have to escape the delimiter that would otherwise end the literal.

And of course, since `\` itself is the escape character, you need `\\` to write a literal backslash:

```
std::cout << "C:\\Users\\Kurt" << std::endl;
```

Output:

```
C:\Users\Kurt
```

Here is the full list of escape sequences you will see in C++:

Escape	Meaning
`\n`	Newline
`\t`	Horizontal tab
`\r`	Carriage return
`\\`	Backslash
`\"`	Double quote
`\'`	Single quote
`\0`	Null character
`\a`	Alert (bell)
`\b`	Backspace
`\f`	Form feed
`\v`	Vertical tab
`\?`	Question mark

Tip: You will use `\n`, `\t`, `\\`, and `\"` constantly. The others are rare in modern code — `\a` rings the terminal bell, `\b` and `\f` come from the days of teletype printers, and `\?` exists only to defeat an obscure C feature called **trigraphs** that you will probably never see.

Input with std::cin

`std::cin` is the **standard input stream**. It reads data from the keyboard using the `>>` operator, whose signature follows this pattern:

```
std::istream& operator>>(std::istream& is, T& value);
```

It takes an input stream and a variable, reads a value from the stream into the variable, and returns the stream.

```
#include <iostream>
#include <string>

int main()
{
    std::string name;

    std::cout << "What is your name? ";
    std::cin >> name;
    std::cout << "Hola, " << name << "!" << std::endl;

    return 0;
}
```

When you run this program, it waits for you to type something and press Enter:

```
What is your name? Nirvana
Hola, Nirvana!
```

`std::cin >> name` reads one word from the keyboard and stores it in the variable `name`. We are using `std::string` here to hold text — we will cover strings in detail in a later chapter. For now, just know that you need `#include <string>` to use them.

Trap: `std::cin >>` reads one word at a time, stopping at whitespace. If you type "Los Del Rio", only "Los" would be stored in `name`. To read an entire line, use `std::getline(std::cin, name)` instead.

You can read numbers too:

```
#include <iostream>

int main()
{
    int year;

    std::cout << "What year? ";
    std::cin >> year;
    std::cout << "Dale a tu cuerpo alegria, " << year << "!" << std::endl;

    return 0;
}
```

```
What year? 1996
Dale a tu cuerpo alegria, 1996!
```

Command-Line Arguments

So far, your programs have asked the user for input interactively. But programs can also receive input when they are launched, through **command-line arguments**.

You have already seen `int main()`. There is another form that accepts arguments:

```
#include <iostream>

int main(int argc, char *argv[])
{
    if (argc < 2) {
        std::cout << "USAGE: " << argv[0] << " <name>" << std::endl;
        return 1;
    }

    std::cout << "Hello, " << argv[1] << "!" << std::endl;
    return 0;
}
```

`argc` is the **argument count** — the number of command-line arguments, including the program name itself. `argv` is the **argument vector** — an array of strings containing each argument.

- `argv[0]` is always the program name
- `argv[1]` is the first argument the user provides
- `argv[2]` would be the second, and so on

If you compile this as `greet` and run it:

```
./greet Kurt
Hello, Kurt!
```

If you run it without an argument:

```
./greet
USAGE: ./greet <name>
```

The program checks if `argc < 2` — meaning no argument was provided — and prints a **USAGE message** to tell the user how to run the program correctly. Returning `1` instead of `0` signals that something went wrong.

Tip: Always validate command-line arguments before using them. Accessing `argv[1]` when no argument was provided leads to undefined behavior — your program might crash, or worse, silently do the wrong thing.

Try It

Here is a starter program that combines what you have learned. Type it in, compile it, and experiment with it. Try changing the output, adding more `cin` reads, or using command-line arguments.

```
#include <iostream>
#include <string>

int main()
{
    std::string song;
    int year;

    std::cout << "Name a 90s song: ";
```

```
    std::getline(std::cin, song);

    std::cout << "What year? ";
    std::cin >> year;

    std::cout << song << " (" << year
              << ") es una cancion increible!"
              << std::endl;

    return 0;
}
```

```
Name a 90s song: Smells Like Teen Spirit
What year? 1991
Smells Like Teen Spirit (1991) es una cancion increible!
```

Key Points

- Every C++ program starts at `main()`.
- Every statement ends with a semicolon (`;`).
- Curly braces (`{}`) define blocks of code — they group statements together.
- `#include <iostream>` gives you access to `std::cout` and `std::cin`.
- `std::cout << value` prints to the screen; `std::cin >> variable` reads from the keyboard.
- **Escape sequences** like `\n`, `\t`, `\"`, and `\\` let you put special characters inside string and character literals.
- C++ source files end in `.cpp` and are compiled with `c++`.
- Namespaces like `std::` organize code and prevent naming conflicts.
- Command-line arguments are accessed through `argc` and `argv`.
- Always validate command-line arguments before using them.
- Always compile with warnings enabled.

Exercises

1. What does the following program print?

   ```
   #include <iostream>

   int main()
   {
       std::cout << "A" << "B" << std::endl;
       std::cout << "C" << std::endl;
       return 0;
   }
   ```

2. What is wrong with the following program?

   ```
   #include <iostream>

   int main()
   {
       std::cout << "Here we are now" << std::endl
       return 0;
   }
   ```

3. Why does `std::cout` have `std::` in front of it? What would happen if you removed the `std::` without adding a `using namespace std;` directive?

4. When you compile a program with `c++ -o hello hello.cpp`, what does the `-o hello` part do? What would happen if you left it out?

5. Consider the following program:

```cpp
#include <iostream>

int main(int argc, char *argv[])
{
    std::cout << argv[2] << std::endl;
    return 0;
}
```

   What happens if you run it with `./program alpha beta gamma`? What happens if you run it with `./program alpha`?

6. If `argc` is 4, how many arguments did the user provide on the command line (not counting the program name)?

7. Write a program that asks for the user's name and favorite number, then prints a message using both. For example: "Hola, Carlos! Your favorite number is 7."

8. **Think about it:** What is the difference between writing `std::endl` and writing `"\n"` at the end of a line? When does it actually matter, and when is it the same?

9. **Where is the bug?** A program does this:

```cpp
#include <iostream>

int main()
{
    std::cout << "Loading...";
    // ... pretend a long computation happens here ...
    std::cout << "Done!\n";
    return 0;
}
```

   The user reports that "Loading..." does not appear on the screen until the computation finishes and the program exits. Why does that happen, and how would you fix it so "Loading..." shows up immediately?

10. What does this program print if the user types `Como estas` and presses Enter?

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string greeting;
    std::cout << "Greeting: ";
    std::cin >> greeting;
    std::cout << "[" << greeting << "]\n";
    return 0;
}
```

    Now change `std::cin >> greeting;` to `std::getline(std::cin, greeting);` and answer the same question. What is the difference, and why?

11. **Where is the bug?** The author wants to print `He said "wassup" and left.` but the compiler refuses to build this:

```cpp
#include <iostream>

int main()
{
    std::cout << "He said "wassup" and left." << std::endl;
    return 0;
}
```

Explain what the compiler sees and rewrite the line so it prints the intended text.

12. What does the following program print?

```cpp
#include <iostream>

int main()
{
    std::cout << "a\\b\tc\nd" << std::endl;
    return 0;
}
```