C++

- A concise C++ guide for a web developer

A concise C++ guide for a web developer ?

Since you already know **HTML, CSS, and JavaScript**, the fastest way to learn C++ is to map new ideas to things you already understand—while also noticing where C++ is *very different*.

1. What C++ is, in web-dev terms

C++ is a **compiled**, **statically typed**, **high-performance** language.

Compared with JavaScript:

1. **Compilation**
   - JavaScript is usually interpreted or JIT-compiled by the engine.
   - C++ is typically compiled ahead of time into a native executable.
2. **Types**
   - JavaScript is dynamically typed.
   - C++ requires you to care much more about the exact type of your data.
3. **Memory**
   - JavaScript has automatic garbage collection.
   - In C++, memory management matters a lot more, though modern C++ gives you safe tools to avoid manual cleanup in many cases.
4. **Use cases**
   - JavaScript often powers web apps and servers.
   - C++ is common in game engines, embedded systems, graphics, browsers, databases, and performance-critical applications.

---

2. Your learning path in the right order

Follow this sequence:

1. **Set up a compiler and editor**
2. **Write your first program**
3. **Learn basic syntax and types**
4. **Understand control flow**
5. **Learn functions**
6. **Learn strings, arrays, and vectors**

7. **Understand references and pointers**
8. **Learn classes and object-oriented basics**
9. **Learn memory management and modern C++ habits**
10. **Use the standard library**
11. **Split code into multiple files**
12. **Practice with small projects**

That order matters because C++ builds on itself more than JavaScript does.

---

3. Step 1: set up your environment

You need:

1. **A compiler**
   - Windows: MSVC via Visual Studio, or MinGW
   - macOS: Clang via Xcode Command Line Tools
   - Linux: `g++` or `clang++`
2. **An editor or IDE**
   - Good beginner choices:
     1. Visual Studio
     2. VS Code
     3. CLion
3. **A way to compile**
   - Example with `g++`:

     ```
     g++ -std=c++17 main.cpp -o app
     ./app
     ```

If you want the smoothest beginner experience on Windows, **Visual Studio** is often easier than manually configuring VS Code.

---

4. Step 2: your first C++ program

```
#include <iostream>

int main() {
    std::cout << "Hello, world!\n";
    return 0;
}
```

What this means

1. `#include <iostream>`
   - Imports input/output tools from the standard library.
2. `int main()`
   - The main entry point of the program.
   - Similar to “this is where execution starts.”
3. `{}`
   - Defines a block of code.
4. `std::cout`
   - Prints output to the console.
5. `return 0;`
   - Ends the program successfully.

First syntax differences from JavaScript

1. Statements usually end with `;`
2. Types are declared explicitly
3. Code structure is stricter
4. There is no DOM, browser API, or built-in `console.log`

---

5. Step 3: variables and basic types

Example:

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

int main() {
    int age = 28;
    double price = 19.99;
    bool isOnline = true;
    char grade = 'A';
    std::string name = "Sam";

    std::cout << name << " is " << age << " years old.\n";
}
```

Core types to know first

1. `int`
   - Whole numbers
2. `double`
   - Decimal numbers
3. `bool`
   - `true` or `false`
4. `char`
   - A single character
5. `std::string`
   - Text

Compared with JavaScript

- JS:

```
let age = 28;
let name = "Sam";
```

- C++:

```
int age = 28;
std::string name = "Sam";
```

In C++, you usually choose the type up front.

---

6. Step 4: operators and control flow

Conditionals

```
if (age >= 18) {
    std::cout << "Adult\n";
} else {
    std::cout << "Minor\n";
}
```

Loops

```
for (int i = 0; i < 5; i++) {
    std::cout << i << "\n";
```

```
}
```

```
int count = 0;
while (count < 3) {
    std::cout << count << "\n";
    count++;
}
```

This will feel familiar if you know JavaScript.

One important difference

Variables declared in a block usually only exist in that block:

```
if (true) {
    int x = 10;
}
// x no longer exists here
```

That's **scope**, and it matters a lot in C++.

---

7. Step 5: functions

Functions work similarly to JavaScript, but types are explicit.

```
#include <iostream>

int add(int a, int b) {
    return a + b;
}

int main() {
    std::cout << add(2, 3) << "\n";
}
```

Structure

1. Return type: `int`

2. Function name: `add`
3. Parameters: `int a, int b`

JavaScript equivalent

```
function add(a, b) {
  return a + b;
}
```

Important C++ habit

Think about:

1. **What type goes in**
2. **What type comes out**
3. **Who owns the data**

That third question becomes very important later.

---

8. Step 6: strings, arrays, and vectors

Strings

```
std::string message = "Hello";
message += " world";
```

Arrays

```
int numbers[3] = {10, 20, 30};
```

Arrays are fixed-size and less flexible.

Vectors

Use `std::vector` early—it's usually a better default.

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {10, 20, 30};

    numbers.push_back(40);

    for (int n : numbers) {
        std::cout << n << "\n";
    }
}
```

Why vectors matter ?

They are like arrays that can grow, similar in spirit to JS arrays, though not identical.

Learn these methods early:

1. `push_back`
2. `size`
3. indexing with `[]`

---

9. Step 7: references and pointers

This is one of the biggest jumps from JavaScript.

References

A reference is an alias for an existing variable.

```
int age = 28;
int& ref = age;

ref = 30;
// age is now 30
```

Pointers

A pointer stores a memory address.

```
int age = 28;
int* ptr = &age;


std::cout << *ptr << "\n";
```

Symbols to know

1. `&age`
   - “address of age”
2. `int* ptr`
   - “pointer to int”
3. `*ptr`
   - “value pointed to by ptr”

What to focus on as a beginner

Do **not** start by obsessing over raw pointers. Instead:

1. Understand the concept
2. Learn references well
3. Prefer modern safe tools like:
   - references
   - `std::vector`
   - `std::string`
   - smart pointers later

---

10. Step 8: classes and objects

If you know JS objects and classes, this will be partly familiar.

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

class User {
public:
    std::string name;
    int age;
```

```
    void greet() {
        std::cout << "Hi, I am " << name << "\n";
    }
};

int main() {
    User user;
    user.name = "Sam";
    user.age = 28;
    user.greet();
}
```

Key ideas

1. A `class` groups data and behavior
2. `public` means accessible from outside
3. Objects are instances of classes

What feels different from JavaScript

C++ classes are often used with much stricter control over:

1. data layout
2. visibility
3. construction
4. destruction

---

11. Step 9: constructors and object lifetime

C++ cares a lot about **when objects are created and destroyed**.

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

class User {
public:
```

```
    std::string name;

    User(std::string userName) {
        name = userName;
    }
};

int main() {
    User user("Sam");
    std::cout << user.name << "\n";
}
```

This is a **constructor**.

Very important concept: lifetime

In C++, many bugs come from using data after it has gone out of scope or been deleted.

So begin learning this mental model early:

1. Where is the object created?
2. Who owns it?
3. When is it destroyed?

That mindset is central to modern C++ □□

---

12. Step 10: memory and modern C++ style

Older C++ tutorials often teach manual memory allocation early:

```
int* p = new int(5);
delete p;
```

You should **avoid relying on this style as a beginner**.

Instead, prefer:

1. Stack variables

2. `std::string`
3. `std::vector`
4. RAII
5. Smart pointers later

RAII in one sentence

Resource Acquisition Is Initialization means resources are tied to object lifetime, so cleanup happens automatically when the object goes out of scope.

This is one of the most important C++ ideas, even if the name sounds intimidating.

---

13. Step 11: the standard library

The C++ standard library is essential. Don’t try to write everything from scratch.

Start with:

1. `iostream`
2. `string`
3. `vector`
4. `array`
5. `algorithm`
6. `map`
7. `unordered_map`

Example:

```
#include <algorithm>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> nums = {4, 1, 3, 2};
    std::sort(nums.begin(), nums.end());

    for (int n : nums) {
        std::cout << n << " ";
    }
}
```

Think of the standard library as the built-in toolbox you should learn early, not avoid.

---

14. Step 12: files and project structure

As programs grow, split code into multiple files.

A very common pattern:

1. **Header file** with declarations
2. **Source file** with definitions
3. **Main file** to run the program

Example layout:

1. `User.h`
2. `User.cpp`
3. `main.cpp`

This is more formal than typical frontend JavaScript modules, but it becomes normal quickly.

---

15. Things that will probably feel hard at first

Expect these topics to be confusing initially:

1. **Pointers**
2. **References**
3. **Const correctness**
4. **Header vs source files**
5. **Compilation and linker errors**
6. **Templates**
7. **Object lifetime**

That’s normal. In C++, many errors come from not yet having the right mental model rather than from syntax alone.

---

16. What to learn first vs later

Learn first

1. Basic syntax
2. Types
3. Conditionals and loops
4. Functions
5. `std::string`
6. `std::vector`
7. Classes
8. References
9. Scope and lifetime
10. Basic standard library use

Learn later

1. Raw manual memory management
2. Advanced templates
3. Move semantics
4. Smart pointers in depth
5. Concurrency
6. Metaprogramming

This order will keep you from drowning too early.

---

17. A simple 4-week path

Week 1

1. Install tools
2. Write and run simple programs
3. Learn variables, types, input/output, conditionals, loops

Week 2

1. Learn functions
2. Practice strings and vectors
3. Solve tiny console exercises

Week 3

1. Learn classes
2. Learn references and basic pointers
3. Learn scope and lifetime

Week 4

1. Learn header/source file structure
2. Use more of the standard library
3. Build a small project

---

18. Good beginner projects

Build console programs first—not GUI apps, not game engines.

Try these in order:

1. **Calculator**
2. **Number guessing game**
3. **To-do list in the terminal**
4. **Simple contact book**
5. **Text-based quiz app**

These are good because they force you to practice:

1. input/output
2. conditions
3. loops
4. functions
5. vectors
6. classes

---

19. Common beginner mistakes

1. **Trying to learn advanced C++ too early**
   - Start with the modern basics.
2. **Using raw pointers everywhere**
   - Most beginner programs do not need that.
3. **Ignoring compiler messages**
   - C++ compiler errors can be long, but they often contain the answer.
4. **Skipping the standard library**

    - You should usually use library containers instead of building your own.
5. **Treating C++ like JavaScript**
    - Some habits transfer, but many do not.

---

20. Your mental model going forward

A useful way to think about C++ is:

1. **What is the type?**
2. **Where does this data live?**
3. **Who owns it?**
4. **How long does it live?**
5. **Can this be expressed with standard library tools?**

If you keep asking those questions, your understanding will grow much faster.

---

21. Recommended next step

If you want the most practical start, do this exact sequence next:

1. Install a compiler and IDE
2. Write `Hello, world`
3. Learn:
    1. variables
    2. `if`
    3. `for`
    4. functions
    5. `std::string`
    6. `std::vector`
4. Build a small calculator
5. Then learn classes, references, and file splitting

That is the cleanest beginner path.