Gorgo C for C++ Programmers

April 11, 2026

1. Introduction

In the beginning, knowing C++ automatically meant you knew C. The original C++ compiler, `cfront`, literally translated C++ code into C before compiling it. But modern C++ has diverged dramatically from C. You have `std::string`, `std::vector`, smart pointers, RAII, templates, `iostream`, and exceptions — none of which exist in C. If someone hands you a C codebase today, your modern C++ instincts will not get you very far.

So why learn C? Because C is everywhere. Operating systems, embedded firmware, database engines, language runtimes — the foundational software that the world runs on is written in C. Even if you spend most of your career in C++, you will encounter C code, C libraries, and C APIs. Understanding C makes you a better programmer, period.

My go-to C textbook is *The C Programming Language* (*Second Edition*) by Brian Kernighan and Dennis Ritchie — often called "K&R" [1]. It is one of the most influential programming books ever written, and it opens like this:

> ### Getting Started
>
> The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages: Print the words hello, world. This is the big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy.
>
> In C, the program to print "hello, world" is

```
#include <stdio.h>
main()
```

```c
{
    printf("hello, world\n");
}
```

Just how to run this program depends on the system you are using. As a specific example, on the UNIX operating system you must create the program in a file whose name ends in ".c", such as hello.c, then compile it with the command `cc hello.c`. If you haven't botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called a.out. If you run a.out by typing the command `a.out`, it will print `hello, world`. On other systems, the rules will be different; check with a local expert.

Differences Summary

Notice the differences from C++. There is no `#include <iostream>`, no `std::println`, no `std::cout`. In C, you use `printf` from `<stdio.h>` for output. The file ends in `.c`, not `.cpp`. You compile with `cc` (the C compiler) rather than `c++`.

Here is a quick summary of the biggest differences you will encounter:

C++	C
`std::string`	`char` arrays with `'\0'`
`std::vector`	raw arrays or `malloc`
`std::cout` / `std::println`	`printf`
`std::cin`	`scanf`
`new` / `delete`	`malloc` / `free`
Smart pointers	Raw pointers (only option)
Classes and objects	Structs and functions
References (`&`)	Pointers (`*`)
`bool` (built-in)	`_Bool` or `#include <stdbool.h>`
`// comments`	`/* comments */` (C89); `//` allowed since C99

Tip: C source files use the `.c` extension and are compiled with `cc`. If you accidentally compile a `.c` file with `c++`, it will be treated as C++ and may accept syntax that real C compilers reject. Always use `cc` when writing C.

Printing with `printf`

```c
int printf(const char *format, ...);
```

In C++, you use `std::cout` or `std::println` for output. In C, you use `printf` from `<stdio.h>`. Unlike `std::println`, `printf` does not automatically add a newline at the end of the output. If you want each call to end on its own line, you must include `\n` in the format string yourself.

The first argument to `printf` is a **format string** containing literal text and **format specifiers** that start with `%`. Each specifier is replaced by the corresponding argument that follows:

```c
int year = 1984;
printf("Year: %d\n", year);   // Year: 1984
```

Here are the format specifiers you will use most:

Specifier	Type	Example	Output
`%d`	`int` (decimal)	`printf("%d", 42)`	`42`
`%x`	`int` (hex, lowercase)	`printf("%x", 255)`	`ff`
`%X`	`int` (hex, uppercase)	`printf("%X", 255)`	`FF`

Specifier	Type	Example	Output
`%f`	`double`	`printf("%f", 3.14)`	`3.140000`
`%e`	`double` (scientific)	`printf("%e", 3.14)`	`3.140000e+00`
`%c`	`char`	`printf("%c", 'A')`	`A`
`%s`	`char *` (string)	`printf("%s", "hola")`	`hola`
`%p`	pointer	`printf("%p", (void *)ptr)`	`0x7ffd...`
`%zu`	`size_t`	`printf("%zu", sizeof(int))`	`4`

Trap: The first argument to `printf` should always be a string literal. Never pass a variable as the first argument. It may work, but it is a potential security vulnerability (format string attack). If you only want to print a string variable, don't do ~~`printf(str)`~~, do `printf("%s", str)`.

You can control the width and precision of output by placing numbers between the `%` and the specifier letter. A number before the specifier sets the minimum field width, and a `.` followed by a number sets the precision (decimal places for floats, max characters for strings):

```c
double score = 98.6;
printf("Score: %f\n", score);      // 98.600000  (default: 6 decimal places)
printf("Score: %.2f\n", score);    // 98.60      (2 decimal places)
printf("Score: %e\n", score);      // 9.860000e+01 (scientific notation)
```

Zero-filled output is useful for track numbers, timestamps, and hex addresses. Place a `0` before the width to pad with zeros instead of spaces:

```c
for (int i = 1; i <= 5; i++) {
    printf("Track %02d\n", i);
}
// Track 01
// Track 02
// Track 03
// Track 04
// Track 05

int color = 0xFF8800;
printf("Color: 0x%06X\n", color);   // Color: 0xFF8800

int score = 95;
printf("Score: %d%%\n", score);      // Score: 95%
```

Since `%` introduces a format specifier, you must write `%%` to print a literal percent sign.

Trap: `printf` does not check that your format specifiers match the types of your arguments. If you write `printf("%d", 3.14)`, the compiler may warn you, but it will not stop you. The result is garbage. Always match specifiers to types: `%d` for `int`, `%f` for `double`, `%s` for `char *`, and so on.

Reading Input with scanf

```c
int scanf(const char *format, ...);
```

In C++, you read input with `std::cin >>`. In C, you use `scanf` from `<stdio.h>`. Like `printf`, `scanf` uses a format string with specifiers — but instead of printing values, it reads them from standard input and stores them in variables.

The critical difference from `printf` is that `scanf` needs to fill in a variable rather than read a variable. C doesn't have references like C++ has. Instead, we give `scanf` the **address** (location in memory) of each variable, not the value. You obtain the address of a variable using the `&` operator. (It's kind of confusing that `&` is also used for references, right?)

```c
int age;
scanf("%d", &age);    // read an integer into age
```

Trap: Without the `&`, `scanf` would receive the current (uninitialized) value of `age` instead of its address. The compiler will warn you that it is wrong, but will still compile the code and you can try to run it even though it will not work.

Strings in C are an array of characters. A variable representing an array takes on the address of the first element of the array when used in expressions or passed to functions. (This is called **array decay**.) For this reason, you don't need to use the `&` operator when reading strings — the array variable already represents an address.

```c
char name[50];        // make sure you have enough space
scanf("%s", name);    // no & needed — name is already an address
```

`%s` reads only until the first whitespace character. If the user types `Rick James`, `scanf("%s", name)` stores only `"Rick"`. Chapter 10 covers how to read full lines and handle input more robustly.

You can also read multiple items in a single `scanf`. `scanf` returns the number of items read. If there is no more input to be read, also known as **end of file**, `scanf` will return `EOF`.

```c
#include <stdio.h>

int main(void) {
    int track;
    char name[50];

    if (scanf("%d %s", &track, name) != 2) {
        printf("I needed a number and a name\n");
    } else {
        printf("Got %s for track %d\n", name, track);
    }
}
```

If `scanf` returns `2`, we know that both variables have been read.

Trap: `scanf` does no bounds checking. If the user types more characters than your buffer can hold, it writes past the end of the array. You can limit how many characters `%s` reads by specifying a width: `scanf("%49s", name)` reads at most 49 characters (leaving room for the `'\0'`).

`scanf` will be covered more extensively in Chapter 10 (Standard I/O). For now, `%d` for integers and `%s` for strings are enough to write interactive programs.

Try It

```c
#include <stdio.h>

int main(void) {
    int num;

    printf("Enter a number: ");
    scanf("%d", &num);

    // Basic format specifiers
    printf("Integer: %d\n", num);
    printf("Hex:     %x (lowercase)  %X (uppercase)\n", num, num);
    printf("Char:    %c\n", num);
    printf("Float:   %f\n", 3.14);
    printf("Sci:     %e\n", 3.14);
    printf("String:  %s\n", "Hola");

    // Width and precision
    printf("\nFormatted output:\n");
    for (int i = 1; i <= 3; i++)
        printf("  Track %02d\n", i);

    printf("  Pi: %.2f\n", 3.14159);
    printf("  100%% complete\n");

    return 0;
}
```

Key Points

- C uses `printf` from `<stdio.h>` for output — there is no `std::cout` or `std::println`.
- Format specifiers (`%d`, `%s`, `%f`, etc.) must match the types of the arguments passed to `printf`.
- C uses `scanf` for input — you must pass the **address** of each variable with `&` (except for arrays, which already decay to pointers).
- C source files end in `.c` and are compiled with `cc`, not `c++`.
- C has no classes, no templates, no exceptions, no smart pointers, and no `std::string`.
- `printf` does not add a newline automatically — you must include `\n` yourself.

Exercises

1. **Think about it:** C uses format specifiers in `printf` while C++ uses `operator<<` or `std::format`. What advantage does the format string approach give you when writing output to a log file? What is a disadvantage?
2. **What does this print?**

   ```c
   printf("%05d %x\n", 42, 255);
   ```

3. **Calculation:** How many bytes does the string literal `"hello"` occupy in memory?
4. **Where is the bug?**

   ```c
   double pi = 3.14159;
   printf("Pi is %d\n", pi);
   ```

5. **Where is the bug?**

```c
int count;
scanf("%d", count);
```

6. **Write a program** that reads the user's name (single word) and a year with `scanf`, then prints a greeting like `Hello, Sting! The year is 1983.` using `printf`.
7. **Write a program** that prints a 5x5 multiplication table using `printf` with width formatting so the columns are aligned.

[1] B. W. Kernighan and D. M. Ritchie, *The C programming language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988.