Gorgo Starting C++

April 11, 2026

Contents

10. std::format and std::print

In Chapter 9, you learned how to read and write data using streams. Formatting that output — aligning columns, controlling decimal places, padding with characters — used to require verbose stream manipulators. C++20 introduced `std::format`, and C++23 added `std::print` and `std::println`, giving you clean, readable formatting in one step. In this chapter, you will learn how to use these modern formatting tools.

std::format

Before C++20, formatting output with `std::cout` could be awkward. Mixing text and values with lots of `<<` operators gets hard to read quickly.

C++20 introduced `std::format` in the `<format>` header. It uses format strings with `{}` placeholders, similar to Python's f-strings or C's `printf`.

```
std::string format(format_string fmt, Args... args);
```

`std::format` takes a format string and one or more arguments, and returns a `std::string`.

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

int main()
{
    std::string artist = "Santana";
    int year = 1999;

    std::string msg = std::format("{} — Smooth ({})", artist, year);
    std::cout << msg << std::endl;

    return 0;
}
```

Output:

```
Santana — Smooth (1999)
```

`std::format` returns a `std::string`. Each `{}` is replaced by the next argument, in order. This is called **implicit** argument numbering.

Implicit vs. Indexed Arguments

You can also use **indexed** arguments by putting a number inside the braces. The number refers to the zero-based position of the argument:

```
std::string msg = std::format("{1} — {0} ({2})", "Santana", "Smooth", 1999);
// "Smooth — Santana (1999)"
```

Indexed arguments let you reorder or reuse arguments without changing the argument list.

```
std::format("{0}, {0}, {0}!", "yeah");  // "yeah, yeah, yeah!"
```

Trap: You cannot mix implicit `{}` and indexed `{0}` in the same format string. `std::format("{},{1}", 1, "hi")` is an error. Pick one style and use it consistently within each format string.

Format Specifiers

You can control how values are formatted by adding specifiers inside the braces.

Width and alignment:

```
// Right-align in a field of 10 characters
std::format("{:>10}", "hola");    // "      hola"

// Left-align in a field of 10 characters
std::format("{:<10}", "hola");    // "hola      "

// Center in a field of 10 characters
std::format("{:^10}", "hola");    // "   hola   "
```

Fill characters:

```
std::format("{:*>10}", "hola");   // "******hola"
std::format("{:-^20}", "Smooth"); // "-------Smooth-------"
```

Sign:

The sign specifier controls how positive numbers are displayed:

```
std::format("{:+}", 42);          // "+42" (always show sign)
std::format("{:-}", 42);          // "42"  (negatives only)
std::format("{: }", 42);          // " 42" (space where + would go)
```

Alternate form (`#`) and zero-padding (`0`):

The `#` flag shows a prefix that identifies the number base. The `0` flag pads with zeros instead of spaces:

```
std::format("{:#x}", 255);        // "0xff"  (hex with 0x prefix)
std::format("{:#b}", 10);         // "0b1010" (binary with 0b prefix)
std::format("{:05}", 42);         // "00042" (zero-padded to width 5)
std::format("{:#010x}", 255);     // "0x000000ff" (combined)
```

Number formatting:

```
std::format("{:d}", 42);          // "42" (decimal integer)
std::format("{:x}", 255);         // "ff" (hexadecimal)
std::format("{:o}", 8);           // "10" (octal)
std::format("{:b}", 10);          // "1010" (binary)
```

Floating-point precision:

```
std::format("{:.2f}", 3.14159);   // "3.14"
std::format("{:.4f}", 2.5);       // "2.5000"
std::format("{:10.2f}", 3.14);    // "      3.14"
```

Here is a more complete example:

```
#include <format>
#include <iostream>

int main()
{
    std::cout << std::format("{:<20} {:>5} {:>8}", "Song", "Year", "Score")
              << std::endl;
    std::cout << std::format("{:<20} {:>5} {:>8.1f}",
        "Wonderwall", 1995, 9.5) << std::endl;
    std::cout << std::format("{:<20} {:>5} {:>8.1f}", "Jumper", 1997, 9.8)
```

```
              << std::endl;
    std::cout << std::format("{:<20} {:>5} {:>8.1f}",
        "Say My Name", 1999, 8.7) << std::endl;

    return 0;
}
```

Output:

```
Song                 Year    Score
Wonderwall           1995      9.5
Jumper               1997      9.8
Say My Name          1999      8.7
```

Tip: `std::format` is much easier to read than chaining `<<` operators with `std::setw` and `std::setprecision`. If your compiler supports C++20 or later, prefer `std::format` for any non-trivial formatting.

std::print and std::println

C++23 took things one step further with `std::print` and `std::println` in the `<print>` header. These combine `std::format` and `std::cout` into a single call.

```
void print(format_string fmt, Args... args);
void println(format_string fmt, Args... args);
```

```
#include <print>

int main()
{
    std::println("You get what you give, don't let go");
    std::print("Track {:d}: {}", 1, "You Get What You Give");
    std::println("");

    double score = 9.5;
    std::println("Rating: {:.1f}/10", score);

    return 0;
}
```

Output:

```
You get what you give, don't let go
Track 1: You Get What You Give
Rating: 9.5/10
```

`std::println` prints a formatted string followed by a newline. `std::print` prints without a trailing newline.

These are the modern replacements for `std::cout <<`. They are shorter to write, easier to read, and handle formatting in one step.

Wut: `std::print` and `std::println` require C++23 support. Not all compilers support them yet. If your compiler does not have `<print>`, you can use `std::format` with `std::cout` to achieve the same result.

Putting It All Together

Here is a program that uses the file I/O from Chapter 9 along with the formatting tools from this chapter:

```cpp
#include <fstream>
#include <format>
#include <iostream>
#include <sstream>
#include <string>

int main()
{
    std::ofstream outfile("puntuaciones.txt");
    if (!outfile) {
        std::cerr << "No puedo abrir el archivo" << std::endl;
        return 1;
    }

    outfile << "Wonderwall 9.5" << std::endl;
    outfile << "Jumper 9.8" << std::endl;
    outfile << "SayMyName 8.7" << std::endl;
    outfile.close();

    std::ifstream infile("puntuaciones.txt");
    if (!infile) {
        std::cerr << "Could not open file" << std::endl;
        return 1;
    }

    std::string line;
    std::cout << std::format("{:<15} {:>6}", "Song", "Score")
              << std::endl;
    std::cout << std::string(22, '-') << std::endl;

    while (std::getline(infile, line)) {
        std::istringstream iss(line);
        std::string song;
        double score;

        iss >> song >> score;
        std::cout << std::format("{:<15} {:>6.1f}", song, score)
                  << std::endl;
    }

    infile.close();

    return 0;
}
```

Output:

```
Song             Score
----------------------
Wonderwall         9.5
Jumper             9.8
SayMyName          8.7
```

Key Points

- `std::format` (C++20) uses `{}` placeholders to produce formatted strings.
- Use `{0}`, `{1}`, etc., to reorder or reuse arguments. You cannot mix implicit `{}` and indexed `{0}` in the same format string.
- `std::print` and `std::println` (C++23) combine formatting and output in one step.
- Format specifiers control width, alignment, precision, and base (e.g., `{:>10.2f}`, `{:x}`).
- Fill characters, alignment (`<`, `>`, `^`), and number bases (`d`, `x`, `o`, `b`) give you fine-grained control.
- `std::format` returns a `std::string`; `std::print` and `std::println` write directly to the output.

Exercises

1. What does `std::format("{:>8.2f}", 3.1)` produce? How many characters wide is the result?
2. Why might you prefer `std::format` over chaining `<<` operators with `std::cout`? Give at least two reasons.
3. What is the difference between `std::print` and `std::println`?
4. What does `std::format("{:*^20}", "Hola")` produce?
5. What is wrong with this code?

   ```
   std::string result = std::format("{} scored {1} points", name, score);
   ```

6. Write a program that asks the user for three song names and three scores (as doubles), writes them to a file called `rankings.txt` (one song and score per line), then reads the file back and prints a formatted table with columns for song name and score, right-aligning the scores to one decimal place.
7. **What does this print?**

   ```
   std::println("{1} - {0} ({2})", "Backstreet Boys", "I Want It That Way", 1999);
   ```

   Now change every `{0}`/`{1}`/`{2}` to plain `{}` and predict the output. What is the rule about mixing indexed and implicit placeholders in the same format string?
8. **Calculation:** What does each of these `std::format` calls produce?

   ```
   std::format("{:+d}", 42)
   std::format("{:+d}", -42)
   std::format("{: d}", 42)
   std::format("{:05d}", 42)
   std::format("{:+06d}", -42)
   ```

   For each one, write down the exact characters in the resulting string, including any spaces or zeros.
9. **What does this print?**

   ```
   int n = 255;
   std::println("{:#x}", n);
   std::println("{:#o}", n);
   std::println("{:#b}", n);
   std::println("{:08b}", n);
   ```

   What does the # flag do, and what does the `08` in the last line do?
10. **What does this print?**

    ```
    std::string title = "Smells Like Teen Spirit";
    std::println("[{:.5}]", title);
    std::println("[{:<10.5}]", title);
    std::println("[{:>10.5}]", title);
    ```

For string arguments, what does the precision (`.5`) mean? How is that different from precision on a `double`?

11. **Write a program** that takes three integers, formats them into a single `std::string` using `std::format`, and prints each integer in three different ways:

    - decimal in a 6-character field, right-aligned
    - hexadecimal with the `0x` prefix and zero-padded to 8 hex digits
    - binary with the `0b` prefix, zero-padded to 16 bits

    Use a single `std::format` call per row so you practice combining width, fill, and base specifiers in the same format string.