Gorgo Starting C++

April 11, 2026

Contents

9. I/O Streams

In Chapter 1 you met `std::cout` and `std::cin` for printing to the screen and reading from the keyboard. Every program you have written since then reads from the keyboard and writes to the console. But console I/O is ephemeral — once your program ends, everything it printed is gone. You cannot save results for later, you cannot process data from an existing file, and you cannot build a complex string in memory before outputting it. C++ solves this with a uniform streaming interface: the `<<` and `>>` operators you already know work the same way with files and in-memory strings as they do with the console. In this chapter you will learn about string streams for building and parsing strings in memory, and file streams for reading and writing files.

A Quick Review

You already know the basics from Chapter 1:

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

int main()
{
    std::string song;

    std::cout << "Favorite 90s song? ";
    std::getline(std::cin, song);
    std::cout << "Good choice: " << song << std::endl;

    return 0;
}
```

`std::cout` is an **output stream** that sends data to the screen. `std::cin` is an **input stream** that reads data from the keyboard. The `<<` operator inserts data into an output stream, and `>>` extracts data from an input stream.

What makes C++ streams powerful is that this same interface — `<<` and `>>` — works with string streams and file streams too. Once you learn one, you know them all.

Stream Manipulators

Before `std::format` arrived in C++20 (Chapter 10), C++ formatted output using **stream manipulators** — special values you insert into a stream with `<<` to change how subsequent output is formatted. They live in the `<iomanip>` and `<iostream>` headers. You will encounter them in older code, so it is worth knowing what they do.

Output manipulators:

On	Off	Description
`std::boolalpha`	`std::noboolalpha`	print `bool` as `true`/`false` instead of `1`/`0`
`std::fixed`	`std::defaultfloat`	fixed-point notation for floating-point numbers
`std::scientific`	`std::defaultfloat`	scientific notation (e.g., `3.14e+00`)
`std::showpoint`	`std::noshowpoint`	always show the decimal point
`std::showpos`	`std::noshowpos`	show `+` sign for positive numbers
`std::left`	—	left-align within field width
`std::right`	—	right-align within field width (default)
`std::setw(n)`	—	pad the next value to at least `n` characters; only applies to the next `<<`, then resets

On	Off	Description
`std::setprecision(n)`	—	set number of digits (significant, or decimal places with `std::fixed`)
`std::setfill(c)`	—	set the fill character (default is space)

`std::setw`, `std::setprecision`, and `std::setfill` require the `<iomanip>` header. The rest are in `<iostream>`.

Some manipulators also affect input streams. `std::boolalpha` appears in both tables because it changes how bools are both printed and read.

Input manipulators:

On	Off	Description
`std::boolalpha`	`std::noboolalpha`	read `true`/`false` as `bool` instead of `1`/`0`
`std::hex`	`std::dec`	read integers as hexadecimal
`std::oct`	`std::dec`	read integers as octal
`std::skipws`	`std::noskipws`	skip leading whitespace before `>>` (default on)
`std::ws`	—	consume all whitespace right now (one-shot)

`std::hex` and `std::oct` also work on output — `std::cout << std::hex << 255` prints `ff`.

```cpp
#include <iomanip>
#include <iostream>

int main()
{
    bool on_tour = true;
    std::cout << on_tour << std::endl;                 // 1
    std::cout << std::boolalpha << on_tour << std::endl; // true

    double score = 9.87654;
    std::cout << std::fixed << std::setprecision(2);
    std::cout << std::setw(10) << score << std::endl;  //       9.88

    return 0;
}
```

Tip: Prefer `std::format` (Chapter 10) for new code. Stream manipulators are **sticky** — once set, they stay in effect for every subsequent output operation on that stream, which can cause surprising formatting changes later in your program.

Wut: `std::setw` is the exception to the sticky rule — it resets after each `<<` operation. Every other manipulator stays in effect until you explicitly change it.

String Streams

Sometimes you want to build a string piece by piece, or parse values out of a string. String streams let you treat a `std::string` like a stream. They live in the `<sstream>` header.

There are three flavors:

- `std::ostringstream` — output only (writing into a string)
- `std::istringstream` — input only (reading from a string)
- `std::stringstream` — both input and output

Their constructors:

```
std::ostringstream();                    // default
std::istringstream(const std::string& s); // from string
```

`std::stringstream` has both constructors.

Building Strings with std::ostringstream

An `std::ostringstream` lets you use `<<` to build a string the same way you use `std::cout` to print to the screen.

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

int main()
{
    std::ostringstream oss;

    oss << "Man, it's a hot one" << " — " << 1999;
    std::string result = oss.str();

    std::cout << result << std::endl;

    return 0;
}
```

Output:

```
Man, it's a hot one — 1999
```

You stream data into `oss` just like you would into `std::cout`. When you are done, call `.str()` to get the built string:

```
std::string str() const;         // get the string
void str(const std::string& s);  // replace the string
```

This is useful when you need to construct a string from mixed types — integers, floats, other strings — without worrying about manual conversion.

Parsing Strings with std::istringstream

An `std::istringstream` lets you use `>>` to read values out of a string, just like reading from `std::cin`.

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

int main()
{
    std::string data = "I get knocked down 7 times";
    std::istringstream iss(data);

    std::string word;
```

```
    while (iss >> word) {
        std::cout << "[" << word << "]" << std::endl;
    }

    return 0;
}
```

Output:

```
[I]
[get]
[knocked]
[down]
[7]
[times]
```

The `>>` operator reads one whitespace-delimited token at a time, just like `std::cin >>`. When there is nothing left to read, the stream evaluates to `false` and the loop ends.

You can also extract typed values:

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

int main()
{
    std::string data = "42 3.14 hola";
    std::istringstream iss(data);

    int n;
    double d;
    std::string s;

    iss >> n >> d >> s;
    std::cout << "int: " << n << ", double: " << d
              << ", string: " << s << std::endl;

    return 0;
}
```

Output:

```
int: 42, double: 3.14, string: hola
```

Tip: String streams are great for converting between strings and numbers. To turn an `int` into a `std::string`, stream it into an `std::ostringstream` and call `.str()`. To turn a `std::string` into an `int`, put it in an `std::istringstream` and extract with `>>`.

File Streams

File streams let you read from and write to files on disk. They live in the `<fstream>` header and work exactly like `std::cin` and `std::cout`, but connected to files instead of the keyboard and screen.

- `std::ifstream` — input file stream (reading)
- `std::ofstream` — output file stream (writing)
- `std::fstream` — both reading and writing

Writing to a File

`std::ofstream` opens a file for writing. Its constructor takes the filename:

```
std::ofstream(const std::string& filename);
```

```
#include <fstream>
#include <iostream>

int main()
{
    std::ofstream outfile("setlist.txt");

    if (!outfile) {
        // std::cerr is the standard error stream --- like std::cout, but
        // intended for error messages.  It is unbuffered, so messages
        // appear immediately.
        std::cerr << "Could not open file for writing" << std::endl;
        return 1;
    }

    outfile << "Closing Time" << std::endl;
    outfile << "Smooth" << std::endl;
    outfile << "Tubthumping" << std::endl;

    outfile.close();
    std::cout << "Setlist saved!" << std::endl;

    return 0;
}
```

You create an `std::ofstream` by passing the filename to its constructor. Then you use `<<` exactly like you would with `std::cout`. When you are done, call `.close()`:

```
void close();
```

Tip: Always check if a file opened successfully before using it. If the file could not be opened, the stream evaluates to `false`. Using a stream that failed to open will silently do nothing — no errors, no data, just silence.

Reading from a File

To read a file, use `std::ifstream`. Its constructor mirrors `std::ofstream`:

```
std::ifstream(const std::string& filename);
```

The most common pattern is reading line by line with `std::getline`:

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

int main()
{
    std::ifstream infile("setlist.txt");

    if (!infile) {
```

```
        std::cerr << "Could not open setlist.txt" << std::endl;
        return 1;
    }

    std::string line;
    int count = 0;

    while (std::getline(infile, line)) {
        ++count;
        std::cout << count << ": " << line << std::endl;
    }

    infile.close();

    return 0;
}
```

Output (assuming the file from the previous example exists):

```
1: Closing Time
2: Smooth
3: Tubthumping
```

`std::getline(infile, line)` reads one full line from the file into `line`. When the end of the file is reached, `std::getline` returns a value that evaluates to `false`, ending the loop.

You can also read word by word using `>>`:

```
std::ifstream infile("setlist.txt");
std::string word;

while (infile >> word) {
    std::cout << word << std::endl;
}
```

This would print each word on its own line, splitting on whitespace.

Trap: Do not forget to check if the file opened before reading. A common beginner mistake is to skip the check and then wonder why the program produces no output. The stream just silently fails.

Closing Files

You should call `.close()` when you are done with a file. However, file streams automatically close when they go out of scope (when the variable is destroyed at the end of a block).

```
void write_log()
{
    std::ofstream log("event.log");
    log << "It's closing time" << std::endl;
    // log.close() happens automatically here
}
```

Tip: While files close automatically when the stream goes out of scope, calling `.close()` explicitly makes your intent clear and ensures data is flushed immediately.

File Modes

By default, `std::ofstream` truncates the file — it erases any existing content when it opens. You can change this behavior with **file mode flags** passed as a second argument to the constructor:

```
std::ofstream(const std::string& filename, std::ios::openmode mode);
```

Flag	Meaning
`std::ios::out`	open for writing (default for `ofstream`)
`std::ios::app`	append to the end of the file
`std::ios::in`	open for reading (default for `ifstream`)
`std::ios::binary`	open in binary mode (no text translations)
`std::ios::trunc`	truncate file on open (default with `out`)

You combine multiple flags with the `|` operator — the same bitwise OR you learned in Chapter 4:

```
#include <fstream>
#include <iostream>

int main()
{
    // Append to a log file instead of overwriting it
    std::ofstream log("setlist.log", std::ios::out | std::ios::app);

    if (!log) {
        std::cerr << "Could not open log" << std::endl;
        return 1;
    }

    log << "Closing Time" << std::endl;
    log.close();

    return 0;
}
```

Each time you run this program, it adds a line to `setlist.log` instead of replacing the file.

Tip: The `|` operator here is the same bitwise OR from Chapter 4. File mode flags are implemented as bitmasks — each flag sets a different bit, and combining them with `|` turns on multiple bits at once.

Putting It All Together

Here is a program that writes data to a file, reads it back, and uses string streams to parse each line:

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

int main()
{
    std::ofstream outfile("setlist.txt");
    if (!outfile) {
```

```
        std::cerr << "Could not open file" << std::endl;
        return 1;
    }

    outfile << "ClosingTime 1998" << std::endl;
    outfile << "Tubthumping 1997" << std::endl;
    outfile << "Smooth 1999" << std::endl;
    outfile.close();

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

    std::string line;
    std::ostringstream summary;
    int count = 0;

    while (std::getline(infile, line)) {
        std::istringstream iss(line);
        std::string song;
        int year;
        iss >> song >> year;
        ++count;
        summary << count << ". " << song << " (" << year << ")" << std::endl;
    }

    infile.close();

    std::cout << "Setlist:" << std::endl;
    std::cout << summary.str();

    return 0;
}
```

Output:

```
Setlist:
1. ClosingTime (1998)
2. Tubthumping (1997)
3. Smooth (1999)
```

Key Points

- All C++ streams share the same `<<` and `>>` interface — once you learn one, you know them all.
- `std::ostringstream` builds strings from mixed types; `std::istringstream` parses values out of strings.
- `std::ofstream` writes to files; `std::ifstream` reads from files.
- Always check if a file stream opened successfully before using it.
- File streams close automatically when they go out of scope, but explicit `.close()` makes intent clear.
- Stream manipulators (`std::setw`, `std::setprecision`, `std::fixed`, `std::boolalpha`) control formatting but are largely superseded by `std::format` (Chapter 10).
- File mode flags (`std::ios::app`, `std::ios::binary`, etc.) control how files are opened; combine them with `|`.

Exercises

1. What does the following program print?

```cpp
#include <sstream>
#include <iostream>

int main()
{
    std::ostringstream oss;
    oss << 10 << " + " << 20 << " = " << 10 + 20;
    std::cout << oss.str() << std::endl;
    return 0;
}
```

2. What does this program print?

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

int main()
{
    std::istringstream iss("100 hola 3.14");
    int n;
    std::string s;
    double d;

    iss >> n >> s >> d;
    std::cout << d << " " << n << " " << s << std::endl;
    return 0;
}
```

3. What is wrong with this code?

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

int main()
{
    std::ifstream infile("data.txt");
    std::string line;

    while (std::getline(infile, line)) {
        std::cout << line << std::endl;
    }

    return 0;
}
```

4. What is wrong with this file-writing code?

```cpp
#include <fstream>

int main()
{
    std::ofstream out;
```

```
    out << "Yo me la paso bien" << std::endl;
    out.close();
    return 0;
}
```

5. Why is it useful that string streams, file streams, and `std::cout`/`std::cin` all share the same `<<` and `>>` interface?

6. Write a program that reads three song names from the user using `std::getline`, builds a single string containing all three songs separated by `/` using an `std::ostringstream`, writes that string to a file called `favorites.txt`, then reads the file back and prints its contents.

7. What does this program print?

```
#include <iomanip>
#include <iostream>

int main()
{
    std::cout << std::boolalpha << (5 > 3) << std::endl;
    std::cout << std::fixed << std::setprecision(1);
    std::cout << 3.14159 << std::endl;
    return 0;
}
```

8. What happens if you open an `std::ofstream` with `std::ios::app` and write to it? How does this differ from the default behavior?

9. Given the input string `"Closing Time 1998 Smooth 1999"`, how many times will this loop iterate?

```
std::istringstream iss("Closing Time 1998 Smooth 1999");
std::string word;
int count = 0;
while (iss >> word) {
    count++;
}
```

What is the final value of `count`?

10. **What does this print?**

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

int main()
{
    std::stringstream ss;
    ss << "year " << 1999;
    std::string word;
    int year{};
    ss >> word >> year;
    std::cout << "[" << word << "] [" << year << "]\n";
    return 0;
}
```

`std::stringstream` is bidirectional — you can write into it with `<<` and then read out of it with `>>`. Walk through what is in the stream after the `<<` line and what each `>>` extracts.

11. **Calculation:** What is the value of each of these `std::ios_base::openmode` expressions, and what does each combination do?

    ```
    std::ios::out
    std::ios::out | std::ios::app
    std::ios::out | std::ios::trunc
    std::ios::in  | std::ios::out | std::ios::binary
    ```

    Why do you OR the flags together with `|` instead of using `+` or `,`?

12. **Write a program** that opens a file called `oldies.txt`, reads each line into a `std::vector<std::string>`, and prints them. If the file cannot be opened, write a clear error message to `std::cerr` (not `std::cout`) and return a non-zero exit code. Why does sending the error to `std::cerr` matter even though both streams print to the same terminal by default?