28.3 — Output with ostream and ios

ALEX[1] MARCH 16, 2024

In this section, we will look at various aspects of the iostream output class (ostream).

The insertion operator

The insertion operator (<<) is used to put information into an output stream. C++ has predefined insertion operations for all of the built-in data types, and you've already seen how you can overload the insertion operator (https://www.learncpp.com/cpp-tutorial/93-overloading-the-io-operators/)[2] for your own classes.

In the lesson on streams (https://www.learncpp.com/cpp-tutorial/131-input-and-output-io-streams/)[3], you saw that both istream and ostream were derived from a class called ios. One of the jobs of ios (and ios_base) is to control the formatting options for output.

Formatting

There are two ways to change the formatting options: flags, and manipulators. You can think of **flags** as boolean variables that can be turned on and off. **Manipulators** are objects placed in a stream that affect the way things are input and output.

To switch a flag on, use the **setf()** function, with the appropriate flag as a parameter. For example, by default, C++ does not print a + sign in front of positive numbers. However, by using the std::ios::showpos flag, we can change this behavior:

```
std::cout.setf(std::ios::showpos); // turn on the std::ios::showpos flag
std::cout << 27 << '\n';
```

This results in the following output:

```
+27
```

It is possible to turn on multiple ios flags at once using the Bitwise OR (|) operator:

```
std::cout.setf(std::ios::showpos | std::ios::uppercase); // turn on the std::ios::showpos and std::ios::uppercase flag
std::cout << 1234567.89f << '\n';
```

This outputs:

```
+1.23457E+06
```

To turn a flag off, use the **unsetf()** function:

```
std::cout.setf(std::ios::showpos); // turn on the std::ios::showpos flag
std::cout << 27 << '\n';
std::cout.unsetf(std::ios::showpos); // turn off the std::ios::showpos flag
std::cout << 28 << '\n';
```

This results in the following output:

```
+27
28
```

There's one other bit of trickiness when using setf() that needs to be mentioned. Many flags belong to groups, called format groups. A **format group** is a group of flags that perform similar (sometimes mutually exclusive) formatting options. For example, a format group named "basefield" contains the flags "oct", "dec", and "hex", which controls the base of integral values. By default, the "dec" flag is set. Consequently, if we do this:

```
std::cout.setf(std::ios::hex); // try to turn on hex output
std::cout << 27 << '\n';
```

We get the following output:

```
27
```

It didn't work! The reason why is because setf() only turns flags on -- it isn't smart enough to turn mutually exclusive flags off. Consequently, when we turned std::hex on, std::ios::dec was still on, and std::ios::dec apparently takes precedence. There are two ways to get around this problem.

First, we can turn off std::ios::dec so that only std::hex is set:

```
std::cout.unsetf(std::ios::dec); // turn off decimal output
std::cout.setf(std::ios::hex); // turn on hexadecimal output
std::cout << 27 << '\n';
```

Now we get output as expected:

```
1b
```

The second way is to use a different form of setf() that takes two parameters: the first parameter is the flag to set, and the second is the formatting group it belongs to. When using this form of setf(), all of the flags belonging to the group are turned off, and only the flag passed in is turned on. For example:

```
// Turn on std::ios::hex as the only std::ios::basefield flag
std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << 27 << '\n';
```

This also produces the expected output:

```
1b
```

Using setf() and unsetf() tends to be awkward, so C++ provides a second way to change the formatting options: manipulators. The nice thing about manipulators is that they are smart enough to turn on and off the appropriate flags. Here is an example of using some manipulators to change the base:

```
std::cout << std::hex << 27 << '\n'; // print 27 in hex
std::cout << 28 << '\n'; // we're still in hex
std::cout << std::dec << 29 << '\n'; // back to decimal
```

This program produces the output:

```
1b
1c
29
```

In general, using manipulators is much easier than setting and unsetting flags. Many options are available via both flags and manipulators (such as changing the base), however, other options are only available via flags or via manipulators, so it's important to know how to use both.

Useful formatters

Here is a list of some of the more useful flags, manipulators, and member functions. Flags live in the std::ios class, manipulators live in the std namespace, and the member functions live in the std::ostream class.

Group	Flag	Meaning
	std::ios::boolalpha	If set, booleans print "true" or "false". If not set, booleans print 0 or 1

Manipulator	Meaning
std::boolalpha	Booleans print "true" or "false"
std::noboolalpha	Booleans print 0 or 1 (default)

Example:

```
std::cout << true << ' ' << false << '\n';

std::cout.setf(std::ios::boolalpha);
std::cout << true << ' ' << false << '\n';

std::cout << std::noboolalpha << true << ' ' << false << '\n';

std::cout << std::boolalpha << true << ' ' << false << '\n';
```

Result:

```
1 0
true false
1 0
true false
```

Group	Flag	Meaning
	std::ios::showpos	If set, prefix positive numbers with a +

Manipulator	Meaning
std::showpos	Prefixes positive numbers with a +
std::noshowpos	Doesn't prefix positive numbers with a +

Example:

```
std::cout << 5 << '\n';

std::cout.setf(std::ios::showpos);
std::cout << 5 << '\n';

std::cout << std::noshowpos << 5 << '\n';

std::cout << std::showpos << 5 << '\n';
```

Result:

```
5
+5
5
+5
```

Group	Flag	Meaning
	std::ios::uppercase	If set, uses upper case letters

Manipulator	Meaning
std::uppercase	Uses upper case letters
std::nouppercase	Uses lower case letters

Example:

```cpp
std::cout << 12345678.9 << '\n';

std::cout.setf(std::ios::uppercase);
std::cout << 12345678.9 << '\n';

std::cout << std::nouppercase << 12345678.9 << '\n';

std::cout << std::uppercase << 12345678.9 << '\n';
```

Result:

```
1.23457e+007
1.23457E+007
1.23457e+007
1.23457E+007
```

Group	Flag	Meaning
std::ios::basefield	std::ios::dec	Prints values in decimal (default)
std::ios::basefield	std::ios::hex	Prints values in hexadecimal
std::ios::basefield	std::ios::oct	Prints values in octal
std::ios::basefield	(none)	Prints values according to leading characters of value

Manipulator	Meaning
std::dec	Prints values in decimal
std::hex	Prints values in hexadecimal
std::oct	Prints values in octal

Example:

```cpp
std::cout << 27 << '\n';

std::cout.setf(std::ios::dec, std::ios::basefield);
std::cout << 27 << '\n';

std::cout.setf(std::ios::oct, std::ios::basefield);
std::cout << 27 << '\n';

std::cout.setf(std::ios::hex, std::ios::basefield);
std::cout << 27 << '\n';

std::cout << std::dec << 27 << '\n';
std::cout << std::oct << 27 << '\n';
std::cout << std::hex << 27 << '\n';
```

Result:

```
27
27
33
1b
27
33
1b
```

By now, you should be able to see the relationship between setting formatting via flag and via manipulators. In future examples, we will use manipulators unless they are not available.

Precision, notation, and decimal points

Using manipulators (or flags), it is possible to change the precision and format with which floating point numbers are displayed. There are several formatting options that combine in somewhat complex ways, so we will take a closer look at this.

Group	Flag	Meaning
std::ios::floatfield	std::ios::fixed	Uses decimal notation for floating-point numbers
std::ios::floatfield	std::ios::scientific	Uses scientific notation for floating-point numbers
std::ios::floatfield	(none)	Uses fixed for numbers with few digits, scientific otherwise
std::ios::floatfield	std::ios::showpoint	Always show a decimal point and trailing 0's for floating-point values

Manipulator	Meaning
std::fixed	Use decimal notation for values
std::scientific	Use scientific notation for values
std::showpoint	Show a decimal point and trailing 0's for floating-point values
std::noshowpoint	Don't show a decimal point and trailing 0's for floating-point values
std::setprecision(int)	Sets the precision of floating-point numbers (defined in the iomanip header)

Member function	Meaning
std::ios_base::precision()	Returns the current precision of floating-point numbers
std::ios_base::precision(int)	Sets the precision of floating-point numbers and returns old precision

If fixed or scientific notation is used, precision determines how many decimal places in the fraction is displayed. Note that if the precision is less than the number of significant digits, the number will be rounded.

```
std::cout << std::fixed << '\n';
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';

std::cout << std::scientific << '\n';
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';
```

Produces the result:

```
123.456
123.4560
123.45600
123.456000
123.4560000

1.235e+002
1.2346e+002
1.23456e+002
1.234560e+002
1.2345600e+002
```

If neither fixed nor scientific are being used, precision determines how many significant digits should be displayed. Again, if the precision is less than the number of significant digits, the number will be rounded.

```
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';
```

Produces the following result:

```
123
123.5
123.46
123.456
123.456
```

Using the showpoint manipulator or flag, you can make the stream write a decimal point and trailing zeros.

```
std::cout << std::showpoint << '\n';
std::cout << std::setprecision(3) << 123.456 << '\n';
std::cout << std::setprecision(4) << 123.456 << '\n';
std::cout << std::setprecision(5) << 123.456 << '\n';
std::cout << std::setprecision(6) << 123.456 << '\n';
std::cout << std::setprecision(7) << 123.456 << '\n';
```

Produces the following result:

```
123.
123.5
123.46
123.456
123.4560
```

Here's a summary table with some more examples:

Option	Precision	12345.0	0.12345
Normal	3	1.23e+004	0.123
	4	1.235e+004	0.1235
	5	12345	0.12345
	6	12345	0.12345
Showpoint	3	1.23e+004	0.123
	4	1.235e+004	0.1235
	5	12345.	0.12345
	6	12345.0	0.123450
Fixed	3	12345.000	0.123
	4	12345.0000	0.1235
	5	12345.00000	0.12345
	6	12345.000000	0.123450
Scientific	3	1.235e+004	1.235e-001
	4	1.2345e+004	1.2345e-001
	5	1.23450e+004	1.23450e-001
	6	1.234500e+004	1.234500e-001

Width, fill characters, and justification

Typically when you print numbers, the numbers are printed without any regard to the space around them. However, it is possible to left or right justify the printing of numbers. In order to do this, we have to first define a field width, which defines the number of output spaces a value will have. If the actual number printed is smaller than the field width, it will be left or right justified (as specified). If the actual number is larger than the field width, it will not be truncated -- it will overflow the field.

Group	Flag	Meaning
std::ios::adjustfield	std::ios::internal	Left-justifies the sign of the number, and right-justifies the value
std::ios::adjustfield	std::ios::left	Left-justifies the sign and value
std::ios::adjustfield	std::ios::right	Right-justifies the sign and value (default)

Manipulator	Meaning
std::internal	Left-justifies the sign of the number, and right-justifies the value
std::left	Left-justifies the sign and value
std::right	Right-justifies the sign and value
std::setfill(char)	Sets the parameter as the fill character (defined in the iomanip header)
std::setw(int)	Sets the field width for input and output to the parameter (defined in the iomanip header)

Member function	Meaning
std::basic_ostream::fill()	Returns the current fill character
std::basic_ostream::fill(char)	Sets the fill character and returns the old fill character
std::ios_base::width()	Returns the current field width
std::ios_base::width(int)	Sets the current field width and returns old field width

In order to use any of these formatters, we first have to set a field width. This can be done via the width(int) member function, or the setw() manipulator. Note that right justification is the default.

```
std::cout << -12345 << '\n'; // print default value with no field width
std::cout << std::setw(10) << -12345 << '\n'; // print default with field width
std::cout << std::setw(10) << std::left << -12345 << '\n'; // print left justified
std::cout << std::setw(10) << std::right << -12345 << '\n'; // print right justified
std::cout << std::setw(10) << std::internal << -12345 << '\n'; // print internally justified
```

This produces the result:

```
-12345
    -12345
-12345
    -12345
-    12345
```

One thing to note is that setw() and width() only affect the next output statement. They are not persistent like some other flags/manipulators.

Now, let's set a fill character and do the same example:

```
std::cout.fill('*');
std::cout << -12345 << '\n'; // print default value with no field width
std::cout << std::setw(10) << -12345 << '\n'; // print default with field width
std::cout << std::setw(10) << std::left << -12345 << '\n'; // print left justified
std::cout << std::setw(10) << std::right << -12345 << '\n'; // print right justified
std::cout << std::setw(10) << std::internal << -12345 << '\n'; // print internally justified
```

This produces the output:

```
-12345
****-12345
-12345****
****-12345
-****12345
```

Note that all the blank spaces in the field have been filled up with the fill character.

The ostream class and iostream library contain other output functions, flags, and manipulators that may be useful, depending on what you need to do. As with the istream class, those topics are really more suited for a tutorial or book focusing on the standard library.

Next lesson
28.4 Stream classes for strings

4

Back to table of contents

5

Previous lesson
28.2 Input with istream

6

7

AJAY.CHALLA
June 21, 2025 9:12 pm PDT

I have a doubt like what chapters not to learn like they are not used for Competitive programming or DSA

0 Reply

xcyxiner
May 7, 2025 7:15 pm PDT

```
std::cout<<std::oct<<27<<'\n';
std::cout.fill('*');
std::cout<<std::setw(10)<<-12345<<'\n';
std::cout<<std::setw(10)<<std::left<<-12345<<std::endl;
```

print

```
33
37777747707
37777747707
```

```
std::cout<<std::oct<<27<<'\n';
std::cout << std::resetiosflags(std::ios_base::basefield) << 27 << '\n';
std::cout.fill('*');
std::cout<<std::setw(10)<<-12345<<'\n';
std::cout<<std::setw(10)<<std::left<<-12345<<std::endl;
```

print

```
33
27
****-12345
-12345****
```

Last edited 2 months ago by xcyxiner

0 Reply

sandersan
April 26, 2025 6:23 am PDT

There are too many rules, flags, and functions here. I feel like it's almost impossible for me to remember them all. I should use this content as a manual and look it up when I need to use it, rather than expecting to memorize everything, right?

2 Reply

AJAY.CHALLA

Reply to sandersan [9] June 21, 2025 9:10 pm PDT

That can be done based on your needs like if you want to learn

```
C++
```

for `Competative programming` or `creating games` or for `school` or some other uses.

1 Reply

Nidhi Gupta

February 23, 2025 9:47 pm PST

The insertion operator (<<) of C++ is used to print data in an output stream and is declared for the default types but also overloadable for user-defined types. Input streams and output streams are both inherited from ios, which manages the possibilities of formatting by flags and manipulators. Flags, which work much like on/off switches (set by setf() and unset by unsetf()), enable you to enable output features such as printing a plus sign for positive values or choosing among octal, decimal, and hexadecimal bases —though care should be taken when using flags from the same group of formats lest they conflict. Manipulators offer a simpler-to-use mechanism to alter formatting by automatically setting and clearing the proper flags to switch between formats. Both of these techniques are also usable for controlling precision, notation (scientific or fixed), and the display of decimal points and trailing zeros, and therefore form a complete system for specifying how data will be presented in C++ output streams.

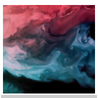
0 Reply

Something

Reply to Nidhi Gupta [10] June 23, 2025 1:30 am PDT

good work ChatGPT!

0 Reply



EmtyC

January 22, 2025 2:39 pm PST

Although this lesson isn't related to optimizations, I prefer to post it here:
Repeated calls to std::cout output functions are inefficient, for example:

```cpp
#include <iostream>

#include "Timer.h" // see timing your code lesson

constexpr int g_max{ 100'000 };

int main() {

    Timer t{};

    for (int i{ 1 }; i <= g_max; ++i)
    {
        std::cout << i << '\n';
    }

    std::cout << "Elapsed: " << t.elapsed() << " s\n";

    return 0;
}
```

My laptop's output:

Debug:

`Elapsed: 13.5442 s` (about same if repeated)

Release: (with special optimization flags for speed, used same configuration for this whole comment)

`Elapsed: 11.8448 s` (about the same if repeated)

So it took ~12s to print 100'000 numbers to the screen, we can do better:

>>> C's printf

```cpp
#include <cstdio> // new
#include <iostream>

#include "Timer.h"

constexpr int g_max{ 100'000 };

int main() {

    Timer t{};

    for (int i{ 1 }; i <= g_max; ++i)
    {
        printf("%d\n", i); // new
    }

    std::cout << "Elapsed: " << t.elapsed() << " s\n"; // still the same

    return 0;
}
```

Debug:

`Elapsed: 3.24116 s`

Release:

`Elapsed: 3.01804 s` (not much gained lel)

about 1/4 of the original time, type safety is expensive :>

Guess what, we can do even better, with the STL

>>> std::ostringstream as a buffer:

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

#include "Timer.h"

constexpr int g_max{ 100'000 };

int main() {

    Timer t{};

    std::ostringstream buf{};

    for (int i{ 1 }; i <= g_max; ++i)
    {
        buf << i << '\n'; // new
    }

    std::cout << buf.str(); // only one call

    std::cout << "Elapsed: " << t.elapsed() << " s\n"; // still the same

    return 0;
}
```

Debug:

`Elapsed: 0.778403 s`

Release:

`Elapsed: 0.562955 s`

from 12s to 0.5s, same behavior, different speeds.
The only optimization I know more that this is to implement your own buffer with printf, but not much difference (0.57 -> 0.51)

Well, I don't really know why I shared this now :D, maybe you could find some use for it Alex :>

Have a good day reader, or Sir Alex the great <3

Edit: wait a sec, std::cout is buffered, so, shouldn't it already have the speed of the last example ? Or does it periodically flush every few outputs, or when it internal buffer is filled ?

Last edited 5 months ago by EmtyC

6 Reply

Delici0us_

Reply to EmtyC 11 February 13, 2025 4:49 am PST

This is actually quite interesting. And i think the reason for that is simply the amount that the "printing" is actually called. Here is another simple example:

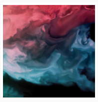
```cpp
#include "src/Timer.h"
#include <chrono>
#include <iostream>
#include <string>

constexpr int g_max{100'000};
int main() {
  Timer t{};

  std::string a{""};
  for (int i{1}; i <= g_max; ++i) {
    a.append(std::to_string(i));
    a.append("\n");
  }
  std::cout << a;

  std::cout << "Elapsed: " << t.elapsed<std::chrono::milliseconds>() << " s\n";

  return 0;
}
```

on my machine this Elapsed in 749 milliseconds. And this program is quite unoptimized. I feel like this answers your question of the 'bufferedness'. I think std::cout wasn't designed to print mass amounts of statements in a short time. But im just a beginner in c++ too, so take everything i say with a grain of salt.

btw the std::cout << i << '\n'; elapsed in 14681 milliseconds. compilation flags i used: -Og -g -Wall -Wextra -pedantic -O2

*Last edited 4 months ago by Delici0us_*

3 Reply



EmtyC

Reply to Delici0us_ [12] February 16, 2025 4:56 am PST

There is now a better alternative to std::cout, C++23's std::print

```cpp
#include <print>
#include "Timer.h"

int main()
{
    Timer t{};

    for (int i{1}; i <= 100'000; ++i)
    {
        std::print("{:d}\n", i);
    }

    std::print("\n\nELAPSED: {:f}s\n", t.elapsed());
}
```

Output: (Debug)

```
ELAPSED: 5.310333s
```

Still not as good as manual buffering, but at least only 2 second slower than a unsafe printf (std::print is surprisingly type safe!)

Edit: Release output: ELAPSED: 4.576441s

Possible downside, code bloat because of templates

Last edited 4 months ago by EmtyC

1 Reply

EmtyC

Reply to Delici0us_ [12] February 13, 2025 11:16 am PST

Possibly ! Thanks for responding

0 Reply

HardyHoneybadger

October 15, 2024 7:30 am PDT

cant believe im almost finished :(

5 Reply

EmtyC

Reply to HardyHoneybadger [13] January 3, 2025 2:45 pm PST

Ya, me too D:

1 Reply

Cells

March 14, 2024 7:51 am PDT

Hi, Alex. The last paragraph recommend a book “The C++ Standard Template Library” by Nicolai M. Josuttis， but I can only find “The C++ Standard Library, A Tutorial and Reference” by Nicolai M. Josuttis. Is this book that I need to read?

0 Reply

Alex Author

Reply to Cells [14] March 16, 2024 4:10 pm PDT

That's the most recent version of the book I was recommending. However, that book only covers up to C++11. At this point a more modern book would probably be more suitable.

I've removed the recommendation accordingly.

1 Reply

D D

D D
February 24, 2024 7:50 am PST

Hello, add the table with `std::basic_ostream::fill()` after the example:

```
std::cout << -12345 << '\n'; // print default value with no field width
std::cout << std::setw(10) << -12345 << '\n'; // print default with field width
std::cout << std::setw(10) << std::left << -12345 << '\n'; // print left justified
std::cout << std::setw(10) << std::right << -12345 << '\n'; // print right justified
std::cout << std::setw(10) << std::internal << -12345 << '\n'; // print internally justified
```

Because it is inconvenient to read. You've splitted the table with the example

0 Reply

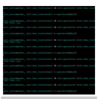
Jestin PJ
September 11, 2023 1:05 am PDT

why this is not printing TRUE FALSE instead of true false ?

```
std::cout.setf(std::ios::uppercase);
    std::cout <<std::boolalpha << std::uppercase<<  true << ' ' << false << '\n';
```

0 Reply

> 
>
> **Alex** Author
> Reply to Jestin PJ [15] September 14, 2023 9:58 am PDT
>
> Per https://en.cppreference.com/w/cpp/io/manip/uppercase, it only uppercases floating-point and hexadecimal integer output.
>
> If you want to print something else, https://raymii.org/s/articles/Print_booleans_as_True_or_False_in_C++.html has a method.
>
> 1 Reply



learnccp lesson reviewer
August 2, 2023 5:22 pm PDT

awesome!!! W lesson

2 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/93-overloading-the-io-operators/
3. https://www.learncpp.com/cpp-tutorial/131-input-and-output-io-streams/
4. https://www.learncpp.com/cpp-tutorial/stream-classes-for-strings/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/input-with-istream/
7. https://www.learncpp.com/output-with-ostream-and-ios/
8. https://gravatar.com/
9. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/#comment-609569
10. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/#comment-608009
11. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/#comment-606867
12. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/#comment-607690
13. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/#comment-603148
14. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/#comment-594681
15. https://www.learncpp.com/cpp-tutorial/output-with-ostream-and-ios/#comment-586957
16. https://g.ezoic.net/privacy/learncpp.com