18.3 — Output with ostream and ios

BY ALEX ON MARCH 12TH, 2008 | LAST MODIFIED BY ALEX ON AUGUST 21ST, 2017

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

Note: All of the I/O functionality in this lesson lives in the std namespace. That means all I/O objects and functions either have to be prefixed with "std::", or the "using namespace std;" statement has to be used.

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** for your own classes.

In the lesson on **streams**, 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::showpos flag, we can change this behavior:

```
std::cout.setf(std::showpos); // turn on the std::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 OR (|) operator:

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

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

```
std::cout.setf(std::showpos); // turn on the std::showpos flag
std::cout << 27 << '\n';
std::cout.unsetf(std::showpos); // turn off the std::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::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::dec was still on, and std::dec apparently takes precedence. There are two ways to get around this problem.

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

```
std::cout.unsetf(std::dec); // turn off decimal output
std::cout.setf(std::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::hex as the only std::basefield flag
std::cout.setf(std::hex, std::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 ios class, manipulators live in the std namespace, and the member functions live in the ostream class.

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

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

Example:

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

```

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

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

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

Result:

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

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

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

Example:

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

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

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

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

Result:

5
+5
5
+5

Group	Flag	Meaning
	uppercase	If set, uses upper case letters

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

Example:

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

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

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

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

Result:

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

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

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

Example:

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

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

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

std::cout.setf(std::hex, std::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
floatfield	fixed	Uses decimal notation for floating-point numbers
floatfield	scientific	Uses scientific notation for floating-point numbers
floatfield	(none)	Uses fixed for numbers with few digits, scientific otherwise

floatfield	showpoint	Always show a decimal point and trailing 0’s for floating-point values

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

Member function	Meaning
precision()	Returns the current precision of floating-point numbers
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.

```cpp
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
adjustfield	internal	Left-justifies the sign of the number, and right-justifies the value
adjustfield	left	Left-justifies the sign and value

adjustfield	right	Right-justifies the sign and value (default)

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

Member function	Meaning
fill()	Returns the current fill character
fill(char)	Sets the fill character and returns the old fill character
width()	Returns the current field width
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) << left << -12345 << '\n'; // print left justified
std::cout << std::setw(10) << right << -12345 << '\n'; // print right justified
std::cout << std::setw(10) << 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) << left << -12345 << '\n'; // print left justified
std::cout << std::setw(10) << right << -12345 << '\n'; // print right justified
std::cout << std::setw(10) << 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