

# Output Formatting

Formatting program output is an essential part of any serious application. Surprisingly, most C++ textbooks don't give a full treatment of output formatting. The purpose of this section is to describe the full range of formatting abilities available in C++.

Formatting in the standard C++ libraries is done through the use of *manipulators*, special variables or objects that are placed on the output stream. Most of the standard manipulators are found in `<iostream>` and so are included automatically. The standard C++ manipulators are not keywords in the language, just like `cin` and `cout`, but it is often convenient to think of them as a permanent part of the language.

The standard C++ output manipulators are:

## **endl**

- places a new line character on the output stream. This is identical to placing `'\n'` on the output stream.

```
#include<iostream>
using namespace std;

int main()
{
    cout << "Hello world 1" << endl;
    cout << "Hello world 2\n";

    return 0;
}
```

## **produces**

```
Hello world 1
Hello world 2
```

## **flush**

- flushes the output buffer to its final destination.

Some explanation is in order here. For efficiency reasons, most output is not sent directly to the output device at the time a program uses a `cout` statement. Rather it is collected in an output buffer in the system memory. Periodically, the collected contents of the buffer are flushed to the output device in one large block. This is much faster than sending the output one character at a time. This flushing happens automatically when the buffer contents get too large.

The output buffer is also flushed before an input statement is executed. This is because the output may be a prompt, and a prompt should appear on the screen in full before any input is accepted.

The output buffer is also flushed when the stream is closed at the end of the program.

In short, the output buffer and its flushing mechanism is largely transparent to the user. The `flush` manipulator is rarely needed.

One exception to this is in using printing statements for debugging. When a program crashes, the output buffer may or may not make it to the output device when the program terminates. Consequently, if print statements are used to locate where the crash occurs, misleading results may be obtained. Using the `flush` operator after every output will help prevent this problem (but is not guaranteed to work every time).

A better solution to this problem is to use the output stream `cerr` for error output. This stream is also defined in `<iostream>` and is an output stream of type `ostream`. It works just like `cout` except that all output is automatically flushed.

## Field width

A very useful thing to do in a program is output numbers and strings in fields of fixed width. This is crucial for reports.

`setw()`

- Adjusts the field width for the item about to be printed.

## Important Point

The `setw()` manipulator only affects the *next* value to be printed.

The `setw()` manipulator takes an integer argument which is the minimum field width for the value to be printed.

## Important Point

All manipulators that take arguments are defined in the header file `iomanip`. This header file must be included to use such manipulators.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    // A test of setw()

    cout << "*" << -17 << "*" << endl;
```

```

cout << "*" << setw(6) << -17 << "*" << endl << endl;

cout << "*" << "Hi there!" << "*" << endl;
cout << "*" << setw(20) << "Hi there!" << "*" << endl;
cout << "*" << setw(3) << "Hi there!" << "*" << endl;

return 0;
}

```

produces

```

*-17*
*   -17*

*Hi there!*
*                Hi there!*
*Hi there!*

```

Note that the values are right justified in their fields. This can be changed. Note also what happens if the value is too big to fit in the field.

### Important Point

The argument given to `setw()` is a *minimum* width. If the value needs more space, the output routines will use as much as is needed.

This field overflow strategy is different than that used in other programming languages. Some languages will fill a small field with \*'s or #'s. Others will truncate the value. The philosophy for the C++ standard output is that it's better to have a correct value formatted poorly than to have a nicely formatted error.

### Important Point

The default field width is 0.

### Justification

Values can be justified in their fields. There are three manipulators for adjusting the justification: `left`, `right`, and `internal`.

### Important Point

The default justification is right justification.

## Important Point

All manipulators except `setw()` are persistent. Their effect continues until explicitly changed.

**left**

- left justify all values in their fields.

**right**

- right justify all values in their fields. This is the default justification value.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
    cout << "*" << -17 << "*" << endl;
    cout << "*" << setw(6) << -17 << "*" << endl;
    cout << left;
    cout << "*" << setw(6) << -17 << "*" << endl << endl;

    cout << "*" << "Hi there!" << "*" << endl;
    cout << "*" << setw(20) << "Hi there!" << "*" << endl;
    cout << right;
    cout << "*" << setw(20) << "Hi there!" << "*" << endl;

    return 0;
}
```

**produces**

```
*-17*
*   -17*
*-17  *

*Hi there!*
*Hi there!          *
*                Hi there!*
```

**showpos and noshowpos**

This manipulator determines how positive numbers are printed. A negative number is traditionally printed with a minus sign in front. The lack of a minus sign means a positive value. However some accounting and scientific applications traditionally place a plus sign in front of positive numbers just to emphasize the fact that the number is positive. Using the `showpos` manipulator makes this happen automatically. The `noshowpos` manipulator returns the output state to placing nothing in front of positive values.

```

#include <iostream>
using namespace std;

int main()
{
    int pos_int = 4, neg_int = -2, zero_int = 0;
    float pos_f = 3.5, neg_f = -31.2, zero_f = 0.0;

    cout << "pos_int: " << pos_int << " neg_int: " << neg_int;
    cout << " zero_int: " << zero_int << endl;
    cout << "pos_f: " << pos_f << " neg_f: " << neg_f;
    cout << " zero_f: " << zero_f << endl << endl;

    cout << showpos;

    cout << "pos_int: " << pos_int << " neg_int: " << neg_int;
    cout << " zero_int: " << zero_int << endl;
    cout << "pos_f: " << pos_f << " neg_f: " << neg_f;
    cout << " zero_f: " << zero_f << endl << endl;

    return 0;
}

```

produces

```

pos_int: 4  neg_int: -2  zero_int: 0
pos_f: 3.5  neg_f: -31.2  zero_f: 0

```

```

pos_int: +4  neg_int: -2  zero_int: +0
pos_f: +3.5  neg_f: -31.2  zero_f: +0

```

Note that zero is considered to be a positive number in this case.

## Integer base

Computer science applications occasionally use number systems other than base 10. The most frequently used number systems (after decimal) are octal (base 8) and hexadecimal (base 16). Octal number systems use the digits 0 through 7. Hexadecimal systems use the digits 0 through 9 and add the digits A through F to represent in one digit the values ten through fifteen.

The manipulators `dec`, `oct`, and `hex` change the base that is used to print out integer values.

## Important Point

The base used for output formatting is completely independent from how a value is stored in machine memory. During output, the bit patterns in memory are converted to character sequences that are meaningful to human beings.

```

#include <iostream>
using namespace std;

int main()
{
    // These sequences of characters are treated as decimal numbers by
    // the compiler and converted to the machine internal format.

    long int pos_value = 12345678;
    long int neg_value = -87654321;
    float value = 2.71828;

    cout << "The decimal value 12345678 is printed out as" << endl;

    cout << "decimal:      " << pos_value << endl;
    cout << "octal:         " << oct << pos_value << endl;
    cout << "hexadecimal: " << hex << pos_value << endl << endl;

    cout << "The decimal value -87654321 is printed out as" << endl;
    cout << "decimal:      " << dec << neg_value << endl;
    cout << "octal:         " << oct << neg_value << endl;
    cout << "hexadecimal: " << hex << neg_value << endl << endl;

    cout << "The decimal value 2.71828 is printed out as" << endl;
    cout << "decimal:      " << dec << value << endl;
    cout << "octal:         " << oct << value << endl;
    cout << "hexadecimal: " << hex << value << endl << endl;

    return 0;
}

```

### produces

```

The decimal value 12345678 is printed out as
decimal:      12345678
octal:        57060516
hexadecimal: bc614e

The decimal value -87654321 is printed out as
decimal:      -87654321
octal:        37261500117
hexadecimal: fac6804f

The decimal value 2.71828 is printed out as
decimal:      2.71828
octal:        2.71828
hexadecimal: 2.71828

```

Note that negative integers are not printed as such in octal or hexadecimal. Rather, the internal bit patterns are interpreted as always being positive values.

Also note that floating point values are always printed out in decimal format.

## Floating Point Output

There are 3 floating point formats: general, fixed, and scientific. Fixed format always has a number, decimal point, and fraction part, no matter how big the number gets, i.e., not scientific notation.  $6.02e+17$  would be displayed as `6020000000000000000` instead of `6.02e+17`.

Scientific format always displays a number in scientific notation. The value of one-fourth would not be displayed as `0.25`, but as `2.5e-01` instead.

General format is a mix of fixed and scientific formats. If the number is small enough, fixed format is used. If the number gets too large, the output switches over to scientific format. General format is the default format for floating point values.

### **fixed and scientific**

The manipulator `fixed` will set up the output stream for displaying floating point values in fixed format.

The `scientific` manipulator forces all floating point values to be displayed in scientific notation.

Unfortunately, there is no manipulator to place the output stream back into general format. The author of these notes considers this to be a design flaw in the standard C++ libraries. There is a way to place the output stream back into general format, but it's not pretty and requires more explanation than is appropriate here. In short, here's the magic incantation

```
cout.unsetf(ios::fixed | ios::scientific);
```

In order to use this statement, you need a `using` declaration for the `ios` class.

```
#include <iostream>
using namespace std;
```

```
int main()
{
    float small = 3.1415926535897932384626;
    float large = 6.0234567e17;
    float whole = 2.000000000;

    cout << "Some values in general format" << endl;
    cout << "small:  " << small << endl;
    cout << "large:  " << large << endl;
    cout << "whole:  " << whole << endl << endl;

    cout << scientific;
```

```

cout << "The values in scientific format" << endl;
cout << "small:  " << small << endl;
cout << "large:  " << large << endl;
cout << "whole:  " << whole << endl << endl;

cout << fixed;

cout << "The same values in fixed format" << endl;
cout << "small:  " << small << endl;
cout << "large:  " << large << endl;
cout << "whole:  " << whole << endl << endl;

// Doesn't work -- doesn't exist
// cout << general;

cout.unsetf(ios::fixed | ios::scientific);

cout << "Back to general format" << endl;
cout << "small:  " << small << endl;
cout << "large:  " << large << endl;
cout << "whole:  " << whole << endl << endl;

return 0;
}

```

### produces

Some values in general format

```

small:  3.14159
large:  6.02346e+17
whole:  2

```

The values in scientific format

```

small:  3.141593e+00
large:  6.023457e+17
whole:  2.000000e+00

```

The same values in fixed format

```

small:  3.141593
large:  602345661202956288.000000
whole:  2.000000

```

Back to general format

```

small:  3.14159
large:  6.02346e+17
whole:  2

```

### **setprecision()**

An important point about floating point output is the *precision*, which is roughly the number of digits displayed for the number. The exact definition of the precision depends on which output format is currently being used.

In general format, the precision is the maximum number of digits displayed. This includes digits before and after the decimal point, but does not include the decimal point itself. Digits in a scientific exponent are not included.



In fixed and scientific formats, the precision is the number of digits after the decimal point.

## Important Point

The default output precision is 6.

The `setprecision` manipulator allows you to set the precision used for printing out floating point values. The manipulator takes an integer argument. The header file `<iomanip>` must be included to use this manipulator.

### **showpoint and noshowpoint**

There is one aspect of printing numbers in general format that is either very nice or very annoying depending on your point of view. When printing out floating point values, only as many decimal places as needed (up to the precision) are used to print out the values. In other words, trailing zeros are not printed. This is nice and compact, but impossible to get decimal points to line up in tables.

The `showpoint` manipulator forces trailing zeros to be printed, even though they are not needed. By default this option is off. As can be seen from previous examples, this manipulator is not needed in fixed or scientific format, only in general format.

```
#include <iostream>
using namespace std;

int main()
{
    float lots = 3.1415926535;
    float little1 = 2.25;
    float little2 = 1.5;
    float whole = 4.00000;

    cout << "Some values with noshowpoint (the default)" << endl << endl;

    cout << "lots:      " << lots << endl;
    cout << "little1: " << little1 << endl;
    cout << "little2: " << little2 << endl;
    cout << "whole:    " << whole << endl;

    cout << endl << endl;

    cout << "The same values with showpoint" << endl << endl;

    cout << showpoint;

    cout << "lots:      " << lots << endl;
```

```
    cout << "little1: " << little1 << endl;
    cout << "little2: " << little2 << endl;
    cout << "whole:   " << whole << endl;

    return 0;
}
```

### **produces**

Some values with noshowpoint (the default)

```
lots:      3.14159
little1:   2.25
little2:   1.5
whole:     4
```

The same values with showpoint

```
lots:      3.14159
little1:   2.25000
little2:   1.50000
whole:     4.00000
```