Formatting Numbers with C++ Output Streams

David Kieras, EECS Dept., Univ. of Michigan
Revised for EECS 381, Winter 2004.

Using the output operator with C++ streams is generally easy as pie, with the only hard part being controlling the format of the output. As you will see, this is relatively clumsy compared to C stdio's printf function, but printf is not type-safe. An output stream has a bunch of member variables that control the details of what output looks like, and these variables can be controlled by a set of member functions and special objects called **manipulators**. Most of these variables retain their values until they are changed, but a few only act temporarily - for the next output operation only.

Throughout this document, the output stream `cout` is used in the examples. However, everything works for any text output stream, such as an output file stream.

The prototypical output statement in C++ is:

```
cout << "Hello, world!" << endl;
```

This contains a **manipulator**, `endl`. This is an object, which when supplied to `operator<<,` causes a newline character to be put into the output stream, followed by a call of cout's flush function, which causes the internal buffer to be immediately emptied. This makes sure all of the output is displayed before the program goes on to the next statement. So manipulators are objects that cause the output stream object to do something, either to its output, or to its member variables.

The manipulators with no arguments, like `endl`, are included in `<iostream>`. If you use manipulators that take arguments (like `setprecision` and `setw`) you need to `#include <iomanip>`.

Like the input operator, `operator<<` has overloaded definitions for all of the built-in types. By default, characters and strings are simply output as is, and this is usually satisfactory. So the main complexity is controlling the format of numbers. There are a zillion variables that control numeric and other formats. This document focuses on how to control the most important and common aspects of numeric output.

You can control the format by using member functions or manipulators to change a variety of member variables. This document summarizes the most important and common things you might want to do; check a complete reference to do more. Quite a few things can be done with the stream formatting controls, and not just with numbers; if you find yourself tempted to write complicated code to make the output look neat, stop and consult a good reference first - you might be about to reinvent the wheel.

Default output format for integers and doubles

All of the digits of an integer will be printed using decimal (base 10), with no leading zeros and a leading minus if it is negative, using as many characters as needed, but no more.

A double value will be printed using a **general** format that usually works fine to represent a number in a small amount of space. The basic amount of space used is determined by the **precision**. The default precision is 6, which means up to 6 significant digits are used to represent the number. This counts digits both to the left and the right of the decimal point. Fewer digits are printed if the result is accurate; e.g. trailing zeros at the right of the decimal point are not printed. As shown in the example below, as the number increases in size, places to the right of the decimal point will be dropped, and the result rounded off as needed to stay within 6 digits. If the number cannot be represented with six digits to the left of the decimal point after all to the right have been dropped, the output will flip into exponential (scientific) notation showing 6 significant digits and the exponent of 10. You can increase or decrease the precision, and this will change the number of significant digits shown in the output.

The problem with the general format is that if you want to make numbers line up in neat columns, as in a table of data, the general format works very poorly because the total number of characters, and the number of digits to the right of the decimal point, will vary depending on the size of the number. The rest of this handout descri bes the basic techniques for controlling the format to produce neat-looking output.

Saving and restoring stream settings

First, if you are going to change the format settings of the stream, you normally will want to restore them to what they were before you changed them, so that different parts of your program will not interfere with each other. You can get the current settings (called "flags") from the output stream with a member function named `flags`. The settings can be stored in a type of object called a `fmtflags` object, defined in a class called `ios`, which is included with `<iostream>`. You can declare one of these objects, but you have to declare it using the scope resolution operator. To make a long story short, the following statement will save certain aspects of the format state in the variable `old_settings`:

```
ios::fmtflags old_settings = cout.flags();
```

Ugly! Then, after doing the output using the new setting, you can restore the old setting by calling the same function with the old settings as an argument:

```
cout.flags(old_settings);
```

Other settings can be obtained and restored with member functions. For example,

```
int old_precision = cout.precision();
```

will save the current precision specification. Then

```
cout.precision(old_precision);
```

will restore the precision to the original value.

Controlling minimum field width

You can control the minimum number of characters used to print the number by specifying the field width. Leading blanks will be included to fill up the field, but if the number doesn't fit, the extra digits will be printed regardless of the field width. The field width also affects output of strings.

Important: Field width changes are only temporary; they affect only the next relevant output. Changing the width affects only the immediately following output of a number, whereupon the width setting automatically reverts to the standard behavior of "as many characters as needed" (specified by a field width of zero).

For example:
```
cout << "*" << setw(4) << 12 << "*" << endl;  // manipulator
```
will produce:
```
*  12*
```
There are two spaces between the '*' and the '1' for a total of 4 characters between the '*'s.

We can do the same thing with a member function, but we have to call the member function right before the relevant item is output. If we called it before outputting the first "*", the new width would control the output of the "*", and then would reset, and so not affect the output of the number.

So the following does the same thing as the previous example:

```
cout << "*";
cout.width(4);  // member function
cout << 12 << "*" << endl;
```

Precision and the general floating-point format

You can change the maximum number of significant digits used to express a floating point number by using the `precision` member function or manipulator. For example,

```
cout.precision(4); // member function
cout << 1234.56789 << " " << 245.678 << " " << 12345.0 << endl;
```

or

```
cout << setprecision(4)    // manipulator
  << 1234.56789 << " " << 245.678 << " " << 12345.0 << endl;
```

will produce:

```
1235 245.7 1.234e+04
```

Notice that when places to the right of the decimal point are dropped, the result is rounded off. If the number is too large to be represented in normal notation in that precision, as in the last value, the precision specification is ignored and the value is output in a more general form. The precision stays changed until you change it again.

Setting the precision to zero with:

```
cout.precision(0);
```

or

```
cout << setprecision(0);
```

restores the 6-digit default.

Precision and the fixed floating-point format

For neat output of doubles, the **fixed** format is most useful. (You can also select a scientific notation format.) In fixed format, the precision specifies the number of digits to the right of the decimal point, and a precision of zero means zero places to the right of the decimal point (i.e. round to the nearest unit).

Using fixed format together with setting the minimum field width allows one to arrange numbers that take up a uniform number of characters, providing that they all fit in the minimum field width.

Using a member function to set fixed floating-point format is incredibly ugly. The `setf` member function is used to set a certain bit in the format flags, using constants defined in `ios`:

```
cout.setf(ios::fixed, ios::floatfield);
```

Fortunately, you can select the fixed format with a simple manipulator:

```
cout << fixed;
```

You can reset the floating-point format flags to the default with

```
cout.setf(0, ios::floatfield);
```

but usually you will want to restore the previous settings.

An Example of Controlling Numerical Output Format

In the following examples, we compute the valaue of pi raised to various negative and positive powers, and output the exponent and the result. The example shows the effects of setting precision, then what we get when we set the fixed format and then use different precisions and output widths. By using a big enough field width to cover the range of values with a fixed format, we can get a neat tabular output result. The example saves and restores the default settings as an example. Each batch of output has been pasted into the code as a comment with some discussion to make the example easier to follow. This code has also been posted in the examples section of the course web site.

```
#include <iostream>
#include <iomanip>  // needed to use manipulators with parameters (precision, width)
#include <cmath>    // needed for pow function
using namespace std;

int main ()
{
	const int beginvalues = -10;
	const int endvalues = 16;
	const int nvalues = endvalues - beginvalues;

	int ipow[nvalues];
	double ary[nvalues];       // an array for demo values

	int ipowindex = 0;
	// fill array with interesting range of values
	for (int i = beginvalues; i < endvalues; i++) {
		ipow[ipowindex] = i;
		ary[ipowindex] = pow(3.14159265, i);
		ipowindex++;
		}

	// output index and array[index] using default settings
	cout << "Output using default settings" << endl;
	for (int i = 0; i < nvalues; i++)
		cout << ipow[i] << ' ' << ary[i] << endl;
/*
Output using default settings
-10 1.06783e-05
-9 3.35468e-05
-8 0.00010539
-7 0.000331094
-6 0.00104016
-5 0.00326776
-4 0.010266
```

```
-3 0.0322515
-2 0.101321
-1 0.31831
0 1
1 3.14159
2 9.8696
3 31.0063
4 97.4091
5 306.02
6 961.389
7 3020.29
8 9488.53
9 29809.1
10 93648
11 294204
12 924269
13 2.90368e+06
14 9.12217e+06
15 2.86581e+07

Each output double value has either six significant digits or fewer if the value can be
expressed just as accurately. For example, for ipow = 0, the value of one shows with no
decimal places and not even a decimal point. These are not printed in the default format
unless there are non-zero places printed to the right of the decimal
point. See also ipow = 10 through 12, where the decimal places have all been rounded off.
At i = 13 and beyond, 6 digits are not enough, so the output flips into scientific notation,
still showing six significant digits, but with an exponent of ten. A different rule, not
so easy to state, governs when small values flip into scientific notation.
 */

	// save the current settings
	ios::fmtflags old_settings = cout.flags(); //save previous format flags
	int old_precision = cout.precision();   // save previous precision setting

	// don't need to save width setting, because it automatically resets to default value
	// after each numeric output item.

	// just change the precision
	cout << setprecision(8);
	cout << "\nOutput using integers with default output" << endl;
	cout << "doubles in general format, precision 8" << endl;
	for (int i = 0; i < nvalues; i++)
		cout << ipow[i] << ' ' << ary[i] << endl;

/*
Output using integers with default output
doubles in general format, precision 8
-10 1.0678279e-05
-9 3.3546804e-05
-8 0.00010539039
-7 0.00033109368
-6 0.0010401615
-5 0.0032677637
-4 0.010265982
-3 0.032251535
-2 0.10132118
-1 0.31830989
0 1
1 3.1415927
2 9.8696044
3 31.006277
```

```
4 97.409091
5 306.01968
6 961.38919
7 3020.2932
8 9488.5309
9 29809.099
10 93648.046
11 294204.01
12 924269.17
13 2903677.2
14 9122171
15 28658145

Here up to 8 significant digits are printed, which is enough to avoid the scientific notation
at ipow = 15. The result is still a mess because the values take up different numbers of
spaces.

*/

	// change output format settings with member functions
	cout.setf(ios::fixed, ios::floatfield);  // set fixed floating format
	cout.precision(2);  // for fixed format, two decimal places
	// cout << fixed << setprecision(2);  // same effects, but using manipulators
	cout << "\nOutput using integers with width 2," << endl;
	cout << "doubles in fixed format, precision 2, width 8" << endl;
	for (int i = 0; i < nvalues; i++)
		cout << setw(2) << ipow[i] << ' ' << setw(8) << ary[i] << endl;
/*
Output using integers with width 2,
doubles in fixed format, precision 2, width 8
-10     0.00
-9     0.00
-8     0.00
-7     0.00
-6     0.00
-5     0.00
-4     0.01
-3     0.03
-2     0.10
-1     0.32
 0     1.00
 1     3.14
 2     9.87
 3    31.01
 4    97.41
 5   306.02
 6   961.39
 7  3020.29
 8  9488.53
 9 29809.10
10 93648.05
11 294204.01
12 924269.17
13 2903677.23
14 9122171.04
15 28658145.48

All the double values show two places to the right of the decimal point, with the results
rounded to hundredths (possibly to zero). This output would be quite neat except for two
problems: (1) Since the minus sign counts in the width of the integers, the -10 value won't
fit into two spaces, and this messes up the first line. (2) Starting at ipow = 11, the output
```

```
is messed up because the results will not fit into the total space of 8 characters
(the decimal point counts as one space). Note that setting fixed format prevents the flipping
into scientific notation, and forces the value of exactly one to show with a decimal point
and the specified number of places to the right of the decimal point.
*/

        cout << "\nOutput using integers with width 3 integers, " << endl;
        cout << "doubles in fixed format, precision 0, width 5" << endl;
        // can use manipulators to change precision and others inside an output statement
        // in fixed format, precision of zero means no decimal places
        for (int i = 0; i < nvalues; i++)
          cout << setw(3) << ipow[i] << ' ' << setprecision(0) << setw(5) << ary[i] << endl;

/*
Output using integers with width 3 integers,
doubles in fixed format, precision 0, width 5
-10     0
 -9     0
 -8     0
 -7     0
 -6     0
 -5     0
 -4     0
 -3     0
 -2     0
 -1     0
  0     1
  1     3
  2    10
  3    31
  4    97
  5   306
  6   961
  7  3020
  8  9489
  9 29809
 10 93648
 11 294204
 12 924269
 13 2903677
 14 9122171
 15 28658145

This gives room for the negative integer values, and so it produces a neat output until
ipow = 11, whereupon the output takes additional digits just as in the previous example.
Because the fixed precision is zero, everything is rounded to the nearest integer value,
and thus neither a decimal point nor places to the right of the decimal point appear.
For values less the one, of course, the result rounds off to zero.
*/

        cout << "\nOutput using integers with width 3 integers, " << endl;
        cout << "doubles in fixed format, precision 8, width 18" << endl;
        cout << setprecision(8);
        for (int i = 0; i < nvalues; i++)
                cout << setw(3) << ipow[i] << ' ' << setw(18) << ary[i] << endl;

/*
Output using integers with width 3 integers,
doubles in fixed format, precision 8, width 18
-10         0.00001068
 -9         0.00003355
 -8         0.00010539
 -7         0.00033109
 -6         0.00104016
 -5         0.00326776
 -4         0.01026598
 -3         0.03225153
 -2         0.10132118
 -1         0.31830989
  0         1.00000000
  1         3.14159265
  2         9.86960438
  3        31.00627657
  4        97.40909059
  5       306.01968304
  6       961.38918698
  7      3020.29320362
  8      9488.53092933
  9     29809.09902689
 10     93648.04640600
 11    294204.01427594
 12    924269.16884980
 13   2903677.22748013
 14   9122171.03582395
 15  28658145.47818740

 This output is the first that is completely neat over the whole range of values.
 The width for the doubles leaves enough room for the additional digits to the left
 of the decimal point. Of course, the precision of 8 produces a lots of decimal places
 which we may not need.
*/


      // restore output format flags and precision
      cout.flags(old_settings);
      cout.precision(old_precision);

      cout << "\nOutput using original settings" << endl;
      for (int i = 0; i < nvalues; i++)
            cout << ipow[i] << ' ' << ary[i] << endl;
/*
Output using original settings
-10 1.06783e-05
-9 3.35468e-05
-8 0.00010539
-7 0.000331094
-6 0.00104016
-5 0.00326776
-4 0.010266
-3 0.0322515
-2 0.101321
-1 0.31831
0 1
1 3.14159
2 9.8696
3 31.0063
4 97.4091
5 306.02
6 961.389
7 3020.29
8 9488.53
9 29809.1
```

```
10 93648
11 294204
12 924269
13 2.90368e+06
14 9.12217e+06
15 2.86581e+07
*/
}
```