

Controlling Output Format within *iostreams*

The header files `iostream.h` and `iomanip.h` provide the C++ programmer with useful output format capabilities, allowing more complete control over the appearance of a program's output. This handout summarizes the most commonly used of these features.

NOTE: The original C language uses a different Input/Output system (i. e., the routines `printf` and `scanf`). While the older-style routines are still available in C++, depending upon the computer and software being used the two systems are not necessarily compatible with one another and mixing the two causes unusual results!

Default Output Format Settings

If no formatting is specified, the I/O system uses default values for each of the settings:

- Values are printed in minimum *fields* (no spaces surrounding the values)
- Floating point values are printed with 6 significant digits, with the least significant digit that is printed rounded (the internal value 17.4 prints as 17.4000, 1435.87956 prints as 1435.88).
- For very large or very small floating point numbers, *scientific* format is used, which is a *mantissa-plus-exponent* form (the value 1234567. is printed as 1.23457e+06, representing the value 1.23457×10^6).

If more precise control over the output is needed, then specific format commands can be included in the output.

Function vs. Manipulator Forms

Most of the formatting commands can be expressed in two distinct ways:

Function form - the formatting command appears on its own separate line, and is of the form `cout.funcname(param);` (for example, `cout.precision(2);`). The “cout.” is necessary to indicate that this function is part of the output system (In C++ lingo, the function is a *member* of the `cout` class.)

Manipulator form - the formatting command appears within the actual `cout` statement that it modifies, as if it were one of the items to be output (for example, `cout << setprecision(2);`).

One form is usually easier to use than the other form in a given situation. However, the two forms are completely equivalent and interchangeable. Typical uses of each form are shown in the examples.

The **function** forms are defined in the header file `<iostream.h>`. The **manipulator** forms are defined in `<iomanip.h>`. Usually, both header files are `#include`'d in a program – that way both forms of each of the formatting commands are available, without having to remember which command is included where.)

Modal vs. Next-Field-Only Commands

For some of the format commands, the changes remain in effect until the program ends or until the values are changed by another command. These are called *modal* commands (they change some output “mode”). For other commands, the changes only affect the next output field – they do NOT change subsequent outputs. These are called the *next-field-only* commands. The type of each command is listed in the Formatting Commands Summary Table below.

Specifying Field Widths and Precision

The *field width* that will be used to output a number is specified using either `width(n)` (function form) or `setw(n)` (manipulator form). This command is *next-field-only*, affecting only the next value that is output. The next value will be output in *n* spaces. The examples below (one using function forms, the other manipulator forms) both output the value of *x* in a 6 column field:

```
int x = 123;           int x = 123;
cout.width(6);        cout << setw(6) << x << endl;
cout << x << endl;
```

`precision(n)` (function) or `setprecision(n)` (manipulator) set the number of places to output behind the decimal for floating point numbers. Its effect is *modal*. The value *n* specifies the number of decimal places that are output. The least significant decimal digit output is rounded.

Setting the Format *flags*

Many of the output format settings are *flags* (bits), which are explicitly set with either:

```
cout.setf(ios::flagname) // function form
cout << setiosflags(ios::flagname) // manipulator form
```

More than one flag can be set in a command by listing the flag names (using the form `ios::flagname`) separated by a single vertical bar (|) (i. e., `cout.setf(ios::fixed | ios::showpoint)`). All of the flags are *modal*. A list of the flags and their meanings is shown in the Formatting Commands Summary Table.

To *unset* a flag or group of flags, one of the following commands is used:

```
cout.unsetf(ios::flagname) // function form
resetiosflags(ios::flagname) // manipulator form
```

Formatting Commands Summary Table

Function Form	Command		Type	Description
		Manipulator Form		
<code>width(<i>n</i>)</code>		<code>setw(<i>n</i>)</code>	NxtFld	Output next value in field of <i>n</i> cols
<code>precision(<i>n</i>)</code>		<code>setprecision(<i>n</i>)</code>	Modal	Output floats with <i>n</i> decimal positions
<code>setf(ios::<i>flag</i>)</code>		<code>setiosflags(ios::<i>flag</i>)</code>	Modal	Set (turn on) value of <i>flag</i>
<code>unsetf(ios::<i>flag</i>)</code>		<code>resetiosflags(ios::<i>flag</i>)</code>	Modal	Unset (turn off) value of <i>flag</i>
		Flag values: <code>right</code> <code>left</code> <code>fixed (Floating pt)</code> <code>scientific (Floating pt)</code> <code>showpoint (Floating pt)</code>		Place value right justified in field Setting <code>right</code> unsets <code>left</code> ¹ Place value left justified in field Setting <code>left</code> unsets <code>right</code> ¹ Use "regular" form for output Setting <code>fixed</code> unsets <code>scientific</code> Use scientific form for output Setting <code>scientific</code> unsets <code>fixed</code> Always print decimal pt.

Putting it all together - An Output Example

In the following output sample, spaces in the output have been marked with a `␣`. The actual computer output does not show these boxes, but they are used here to clarify where the fields are in the output example. The C++ program which produced this output is shown on the next page.

Examples␣with␣no␣output␣formatting␣-␣pretty␣ugly!

1121231234

1.23457e+06123.4571.23456e-05

The␣same␣outputs␣with␣formatting

␣␣␣␣1␣␣␣␣12

␣␣123␣␣1234

␣␣1234567.00␣␣␣␣123.46␣␣0.000012

␣␣␣␣String␣example␣-␣Right␣justified

Here␣is␣some␣left␣justification

1␣␣␣␣12␣␣␣␣

123␣␣1234␣

String␣example␣␣␣␣␣␣␣␣Left␣justified

␣␣␣␣String␣example␣-␣Right␣justified␣again␣!

␣␣␣It␣is␣usually␣easier␣to␣justify␣strings␣just␣using␣spaces!

The Example C++ Program

```
#include <iostream.h>
#include <iomanip.h>

void main()
{
// Define some values to play with

    int a = 1, b = 12, c = 123, d = 1234;
    float big = 1234567.0, medium = 123.4567, little = 0.0000123456;

// First, show some output with no formatting. Note that there are
// no extra spaces output around values, and default settings are used

    cout << "Examples with no output formatting - pretty ugly!" << endl;
    cout << a << b << c << d << endl;
    cout << big << medium << little << endl;

// Now, set up some field widths and some reasonable floating pt parameters
// Note the use of both the "function" and "manipulator" forms

    cout << endl << "The the same values with formatting" << endl;
    cout.width(5); // function form for specifying field widths ...
                // ... most people prefer the manipulator form below!
    cout << a << setw(5) << b << endl << setw(5) << c << setw(5) << d << endl;
    cout.setf(ios::fixed | ios::showpoint);
    cout << setprecision(2) << setw(12) << big << setw(10) << medium;
    cout.precision(6);
    cout << setw(9) << little << endl;

// Now show some string output. Note carefully that strings (like other
// values) are right-justified in the field, unless specified otherwise.

    cout << setw(19) << "String example" << " - Right justified" << endl;

// Finally, look at left justified. Several of the following statements are
// identical to the ones above, but compare the output between the two.

    cout << setiosflags(ios::left);
    cout << endl << "Here is some left justification" << endl;

    cout.width(5);
    cout << a << setw(5) << b << endl << setw(5) << c << setw(5) << d << endl;
    cout << setw(19) << "String example" << " - Left justified" << endl;

    cout.unsetf(ios::left);
    cout << setw(19) << "String example" << " - Right justified again!"
        << endl << endl;
    cout << " It is usually easier to justify strings just using spaces!"
        << endl;
}

```