I/O Manipulators

Before there were format strings, there were I/O manipulators -- a natural way to format output for those already used to stream I/O. In case you're dealing with legacy code and need to understand it...here it is.

Suppose you want to print tables in columns. Your best option (short of format strings) is the **manipulator** setw, found in the iomanip include file.

cout << setw(10) << Thing1 << Thing2 << "\n";

This prints Thing1 right-justified in 10 spaces. (Thing2 is unaffected.) If Thing1 is too big, well, it goes over. If you want it left-justified, do this:

cout << left;

cout << setw(10) << Thing1 << Thing2 << "\n";

left is a message we're sending to cout saying left justification is now on. Set it back to default with cout << right;.

This will help with printing floating-point numbers neatly:

cout << setprecision (2);

Like left, setprecision continues to have effect until you change it to another value. The default is 6.

Example 1 uses these manipulators to neatly print statistics on three familiar planets. It's in source code, in ch25's ioManipulators project/folder.

Example 1. Program to neatly print a table of astronomical data using iomanip.

//Program to print temp, pressure for Venus and Earth

// -- from \_C++ for Lazy Programmers\_

#include <iostream>

#include <iomanip> // for setw and setprecision

using namespace std;

int main ()

{

// constexprants related to spacing on the page

constexpr int PLANET\_SPACE = 7;

constexpr int TEMP\_SPACE = 12;

constexpr int PRESSURE\_SPACE = 13;

// planetary temperature and pressure

constexpr double VENUS\_TEMP = 464;

constexpr double VENUS\_PRESSURE = 90000;

constexpr double EARTH\_TEMP = 15;

constexpr double EARTH\_PRESSURE = 1000;

constexpr double MARS\_TEMP = -62;

constexpr double MARS\_PRESSURE = 1;

// Use fixed format for floats -- no scientific

cout << fixed;

//Print the headers

// First column is justified left, others right

cout << left

<< setw (PLANET\_SPACE) << "Planet"

<< right

<< setw (TEMP\_SPACE ) << "Temperature"

<< setw (PRESSURE\_SPACE)<< "Pressure" << endl;

cout << left

<< setw (PLANET\_SPACE) << " "

<< right

<< setw (TEMP\_SPACE ) << "(celsius)"

<< setw (PRESSURE\_SPACE)<< "(millibars)" << endl;

cout << endl;

// Print the data

// Column 1 has 1 decimal place precision; Col 2 has none

cout << left

<< setw (PLANET\_SPACE) << "Venus"

<< right << setprecision (1)

<< setw (TEMP\_SPACE ) << VENUS\_TEMP

<< setprecision (0)

<< setw (PRESSURE\_SPACE)<< VENUS\_PRESSURE << endl;

cout << left

<< setw (PLANET\_SPACE) << "Earth"

<< right << setprecision (1)

<< setw (TEMP\_SPACE ) << EARTH\_TEMP

<< setprecision (0)

<< setw (PRESSURE\_SPACE)<< EARTH\_PRESSURE << endl;

cout << left

<< setw (PLANET\_SPACE) << "Mars"

<< right << setprecision (1)

<< setw (TEMP\_SPACE) << MARS\_TEMP

<< setprecision (0)

<< setw (PRESSURE\_SPACE)<< MARS\_PRESSURE << endl;

cout << "\n...I think I'll just stay home.\n\n";

return 0;

}

That was a lot of typing! Here's the output:

Planet Temperature Pressure

(celsius) (millibars)

Venus 464.0 90000

Earth 15.0 1000

Mars -62.0 1

...I think I'll just stay home.

Other iostream manipulators are in Table 1. To use manipulators that take arguments, like setw and setprecision, you'll need to #include <iomanip>. More detail on how to use these follows the table, but you'll rarely need it; setw and setprecision usually do all I need.

Table 1. Partial list of iostream manipulators. Defaults are in **bold**.

|  |  |  |
| --- | --- | --- |
| manipulator | meaning | persistence |
| columns and justification | | |
| left/right/internal | when filling with the fill character after setw, add your padding on the left/**right**/inside the value (see below). | until changed |
| setfill   (char fillchar) | when filling after setw, use character fillchar. Default is ' '. | until changed |
| setw (int width) | print the next thing using width characters, filling in with the fill character. If the next thing requires more room, give it what it needs. Default width is **0**. | next thing printed only |
| flushing output |  |  |
| flush | go ahead and print anything in the print buffer (see explanation below) | immediate |
| unitbuf/nounitbuf | send print buffer to output immediately/**not immediately** after a << operation (see explanation below) | until changed |
| numeric representation | |  |
| defaultfloat | uses default format for floating-point numbers (see below) | until changed |
| fixed | use fixed format for floating-points: exactly as many digits right of the decimal point as setprecision specified, and no exponent | until changed |
| hex/oct/dec | read and print integral values in hexadecimal/octal/**decimal** | until changed |
| scientific | use scientific format for floating-point: exactly one digit left of the decimal point; exactly as many digits to the right as setprecision specified; and an exponent part, such as e+003 | until changed |
| setbase (int base) | set base for printing integers to 8, **10**, or 16 | until changed |
| setprecision (int p) | set precision of floating-point printing to p. Default is **6** | until changed |
| showbase/noshowbase | print integral values with/**without** a preceding h if they're in hexadecimal format | until changed |
| showpoint/noshowpoint | always show/**don't always show** decimal point when printing floating-points (see below) | until changed |
| showpos/noshowpos | print positive numbers with/**without** initial "+" | until changed |
| uppercase/nouppercase | print the e in scientific notation and x in hexadecimal base, in upper/**lower** case | until changed |
| set/reset flags |  |  |
| setiosflags   (int flags) | set formatting flags. This function duplicates, by setting those flags, the effects of other manipulators in this table[[1]](#footnote-1) | until changed |
| resetiosflags   (int flags) | unset (clear) formatting flags | until changed |
| whitespace in input |  |  |
| skipws/noskipws | **always skip**/don't skip whitespace in upcoming input, stopping at first non-whitespace character | until changed |
| ws | skip whitespace in upcoming input, stopping at the first non-whitespace character. Not needed if skipws is already on | immediate |
| other |  |  |
| boolalpha/noboolalpha | print/read bool values as "true" or "false"/**as 1 or 0** | until changed |
| endl | print end-of-line ('\n') character and flush | immediate |
| ends | print null ('\0') character | immediate |

left**,** right**,** internal**.** left and right say, put fill characters so as to left- or right-justify the value printed. With internal, if the value printed is a number with a preceding + or - sign, the sign is printed on the left, the number on the right, and fill characters are added between. If the value printed is anything else, internal justification works like right justification.

showpoint. If you're using fixed format for floating-point -- or default -- and it's showing nothing right of the decimal place, it won't show the decimal place either, unless you cout << showpoint. For example, by default, 350.0 shows up as   
350  
But if you cout << showpoint, it'll have a . at the end, as in  
350.

scientific**,** fixed**, and** default\_float. scientificformat has one digit left of the decimal point, exactly as many digits to the right as specified by setprecision, and an exponent: for example, 6.023e+023, which means 6.023×1023, or 3.14159e+000, which means 3.14159×100, or 3.14159, or π.

fixedformat has no exponent, and, like scientific, as many digits right of the decimal point as was specified by setprecision.

defaultfloat considers precision to be the maximum number of digits in the number, right or left of the decimal point -- a maximum that may be overridden for large numbers. (If precision is 4, the number 12345.2 will be printed as 12345 -- overriding the maximum of 4 so you can read the number.) It may omit trailing 0's; 6.1500 may be printed as 6.15, even if the precision is more than 3.

(Best not to think too much about defaultfloat; it's for when you really don't care.)

flush**,** unitbuf. When you print something, it may not immediately appear on the screen. cout << flush makes whatever's waiting to be printed, show up now. cout << unitbuf says to do that every time something is printed. (These are useful even if you are using format strings.) endl flushes the line too, in addition to printing the end-of-line character.

So how does this all stack up to format strings? Why did the community make the change?

This method certainly does require more typing! I got weary in Example 1 of swapping precision and justification back and forth. And sometimes it's hard to remember the commands (what's the difference in ws and skipws?). At the Fluent {C++} blog,[[2]](#footnote-2) guest writer Victor Zverovich, lead creator of the {fmt} library, identifies worse problems including unexpected output.

Exercises

Do the same exercises as in Chapter 25, the section on format strings, only with I/O manipulators.

1. Search for fmtflags on www.cplusplus.com for a complete list of formatting flags. [↑](#footnote-ref-1)
2. At time of writing, https://www.fluentcpp.com/2018/12/04/an-extraterrestrial-guide-to-c-formatting/. [↑](#footnote-ref-2)