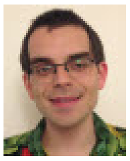
Printf Format Strings



By Alex Allain

By default, C provides a great deal of power for formatting output. The standard display function, printf, takes a "format string" that allows you to specify lots of information about how a program is formatted.

Note: if you are looking for information on formatting output in C++, take a look at formatting C++ output using iomanip.

Let's look at the anatomy of a format string followed by some short example programs to show the different settings in action. I won't include every single possible option--instead, my goal is to make it easy to understand the mini-language that you can use for creating format strings and teach you how to use the common formatting you're most likely to need.

Anatomy of a Format String

When you make a call to printf, the basic idea is that you are going to provide a string of characters that has some literal characters and some elements that are to be replaced. For example, a string like:

```
"a b c"
```

Will be printed literally as it appears. While it is sometimes enough to literally write into your code exactly what you want to print, you usually want to do something fancier--either introducing special characters using **escape sequences** or introducing variable values using **format specifiers**.

Escape Sequences

There are some characters that you cannot directly enter into a string. These are characters like a newline, which must be represented using some special syntax. These are called escape sequences and look like this:

```
"a\nb\nc"
```

Here, I've entered the newlines between each letter, a, b and c. Each escape sequence starts with a backslash ('\') character. The main escape sequences that you'll use are: \n, to put a newline, and \t, to put in a tab. Since a backslash normally indicates the start of an escape sequence, if you want to put in an escape sequence you need to use \\ to display a backslash:

```
"C:\\Program Files\\World of
Warcraft"
```

is how you'd write a Windows path in C++.

There's one other advanced trick, which is that you can write \<num> to display the ASCII character represented by the value num. This is useful if you want to display a character that you can't easily type on your keyboard, suc as accented letters. For example, \130 will print out an é character (in some cases, depending on what your machine is set up to do with extended ASCII characters.)

Format Specifiers

If you want to introduce some variance into the output, you do so by indicating that external data is needed:

```
"We have %d
cats"
```

In this string, the %d indicates that the value to be displayed at that point in the string needs to be taken from a variable. The % sign indicates that we are splicing some data into the string, and the d character indicates that we are splicing in a decimal number. The part of the string that begins with % is called the format specifier. In order to actually get that number, we need to provide that value to printf:

```
printf( "We have %d cats", 3
);
```

which will display:

```
"We have 3
cats"
```

All of the interesting formatting that you can do involves changing the values you put after the % sign, which is the actual format.

The format for what appears about a % sign is:

```
%[flag][min width][precision][length modifier][conversion specifier]
```

Most of these fields are optional, other than providing a conversion specifier, which you've already seen (for example, using %d to print out a decimal number).

Understanding this formatting is best done by working backward, starting with the conversion specifier and working outward. So let's begin at the end!

Conversion Specifier

The conversion specifier is the part of the format specifier that determines the basic formatting of the value that is to be printed.

Conversion specifiers for integers

If you want to print a decimal integer number in base 0, you'd use either **d** or **i**: %d or %i. If you want to print an integer in octal or hexadecimal you'd use **o** for octal, or **x** for hexadecimal. If you want capital letters (A instead of a when printing out decimal 10) then you can use **X**.

Conversion specifiers for floating point numbers

Displaying floating point numbers has a ton of different options, best shown in a table:

Specifier	Description	Example
f	Display the floating point number using decimal representation	3.1415

e	Display the floating point number using scientific notation with e	1.86e6 (same as 1,860,000)
E	Like e, but with a capital E in the output	1.86E6
g	Use shorter of the two representations: f or e	3.1 or 1.86e6
G	Like g, except uses the shorter of f or E	3.1 or 1.86E6

Okay, that wasn't too bad was it? But that chart is kind of complicated. My recommendation: just use %g, and it wil usually do what you want:

```
printf( "%g", 3.1415926
);
```

which displays:

```
3.1515926
```

or

```
printf( "%g", 93000000.0
);
```

which displays

```
9.3e+07
```

Where scientific notation is most appropriate.

Displaying a Percent Sign

Since the percent sign is used to define format specifiers, there's a special format specifier that means "print the percent sign":

```
%%
```

to simply print out a percent sign.

Now, let's walk through each of the different components of a format specifier.

Length Modifier

The length modifier is perhaps oddly-named; it does not modify the length of the output. Instead, it's what you use to specify the length of the input. Huh? Say you have:

```
long double d =
3.1415926535;
printf( "%g", d );
```

Here, d is the input to printf; and what you're saying is that you want to print d as an double; but d is not a double, it is a long double. A long double is likely to be 16 bytes (compared to 8 for a double), so the difference matters. Try running that small snippet and you'll find that you get garbage output that looks something like this:

```
4.94066e-324
```

Remember, the bytes that are given to printf are being treated like a double--but they aren't a double, they're a long double. The length is wrong, and the results are ugly!

The length modifier is all about helping printf deal with cases where you're using unusually big (or unusually small) variables.

The best way to think about length modifiers is to say: what variable type do I have, and do I need to use a length modifier for it? Here's a table that should help you out:

Variable type	Length Modifier	Example
short int, unsigned short int	h	short int i = 3; printf("%hd", i);
long int or unsigned long int	l	long int i = 3; printf("%ld", i);
wide characters or strings	l	wchar_t* wide_str = L"Wide String"; printf("%ls", wide_str);
long double	L	long double d = 3.1415926535; printf("%Lg", d);

I'd like to make special mention about the wide character handling. If you write

```
wchar_t* wide_str = L"Wide
String";
printf( "%s", wide_str );
```

without the l, the result will be to print a single W to the screen. The reason is that wide characters are two bytes, and for simple ASCII characters like W, the second byte is 0. Therefore, printf thinks the string is done! You must tell printf to look for multibyte characters by adding the l: %ls.

(If you happen to be using wprintf, on the other hand, you can simply use %s and it will natively treat all strings as wide character strings.)

Precision

The "precision" modifier is written ".number", and has slightly different meanings for the different conversion specifiers (like d or g).

For floating point numbers (e.g. %f), it controls the number of digits printed after the decimal point:

```
printf( "%.3f", 1.2
);
```

will print

```
1.200
```

If the number provided has more precision than is given, it will round. For example:

```
printf( "%.3f", 1.2348
);
```

will display as

```
1.235
```

Interestingly, for g and G, it will control the number of significant figures displayed. This will impact not just the value after the decimal place but the whole number.

```
printf( "%.3f\n%.3g\n%.3f\n%.3g\n", 100.2, 100.2, 3.1415926, 3.1415926
);
```

```
100.200 // %.3f, putting 3 decimal places
always
100     // %.3g, putting 3 significant figures
3.142   // %.3f, putting 3 decimal places again
3.14    // %.3g, putting 3 significant figures
```

For integers, on the other hand, the precision it controls the minimum number of digits printed:

```
printf( "%.3d", 10
);
```

Will print the number 10 with three digits:

```
010
```

There's one special case for integers--if you specify '.0', then the number zero will have no output:

```
printf( "%.0d", 0
);
```

has no output!

Finally, for strings, the precision controls the maximum length of the string displayed:

```
printf( "%.5s\n", "abcdefg" );
```

displays only

```
"abcde"
```

This is useful if you need to make sure that your output does not go beyond a fixed number of characters.

Width

The width field is almost the opposite of the precision field. Precision controls the max number of characters to print, width controls the minimum number, and has the same format as precision, except without a decimal point:

```
printf( "%5s\n", "abc"
);
```

prints

```
  abc
```

The blank spaces go at the beginning, by default.

You can combine the precision and width, if you like: <width>.<precision>

```
printf( "%8.5f\n", 1.234
);
```

prints

```
 1.23400
```

(Note the leading space.)

Flag

The flag setting controls 'characters' that are added to a string, such whether to append 0x to a hexadecimal number, or whether to pad numbers with 0s.

The specific flag options are

The Pound Sign:

Adding a **#** will cause a '0' to be prepended to an octal number (when using the **o** conversion specifier), or a **0x** to be prepended to a hexadecimal number (when using a **x** conversion specifier). For most other conversion specifiers, adding a **#** will simply force the inclusion of a decimal point, even if the number has no fractional part.

```
printf( "%#x", 12
);
```

results in

```
0xc
```

being printed. Whereas

```
printf( "%x", 12
);
```

results in simply

```
c
```

being printed.

The Zero Flag: 0

Using **0** will force the number to be padded with 0s. This only really matters if you use the width setting to ask for a minimal width for your number. For example, if you write:

```
printf( "%05d\n", 10
);
```

You would get:

```
00010
```

The Plus Sign Flag: +

The plus sign will include the sign specifier for the number:

```
printf( "%+d\n", 10
);
```

Will print

```
+10
```

The Minus Sign Flag: -

Finally, the minus sign will cause the output to be left-justified. This is important if you are using the width specifier and you want the padding to appear at the end of the output instead of the beginning:

```
printf( "|%-5d|%-5d|\n", 1, 2
);
```

displays:

```
|1    |2
|
```

With the padding at the end of the output.

Combining it all together

For any given format specifier, you can provide must always provide the percent sign and the base specifier. You can then include any, or all, of the flags, width and precision and length that you want. You can even include multiple flags togeher. Here's a particularly complex example demonstrating multiple flags that would be useful for printing memory addresses as hexadecimal values.

```
printf( "%#010x\n", 12
);
```

The easiest way to read this is to first notice the % sign and then read right-to-left--the x indicates that we are printing a hexadecimal value; the 10 indicates we want 10 total characters width; the next 0 is a flag indicating we want to pad with 0s intead of spaces, and finally the # sign indicates we want a leading 0x. Since we start with 0x, this means we'll have 8 digits--exactly the right amount for printing out a 32 bit memory address.

The final result is:

```
0x0000000c
```

Pretty sweet!

Read more similar articles

More on printf format strings in C

Produce nice output in C++ using iomanip