Secrets of "printf"

Professor Don Colton

Brigham Young University Hawaii

`printf` is the C language function to do formatted printing. The same function is also available in PERL. This paper explains how `printf` works, and how to design the proper formatting specification for any occasion.

1 Background

In the early days, computer programmers would write their own subroutines to read in and print out numbers. It is not terribly difficult, actually. Just allocate a character array to hold the result, divide the number by ten, keep the remainder, add x30 to it, and store it at the end of the array. Repeat the process until all the digits are found. Then print it. Too easy, right?

But even though it was easy (for Einstein), it still took some effort. And what about error checking, and negative numbers? So the computer programmers brought forth libraries of prerecorded functions. And it was good. Eventually the most popular of these functions were canonized into membership in the "standard" libraries. Number printing was popular enough to gain this hallowed honor.

This meant that programmers did not have to reinvent the number-printing subroutine again and again. It also meant that everybody's favorite options tried to make it into the standard.

Thus was `printf` born.

2 Simple Printing

In the most simple case, `printf` takes one argument: a string of characters to be printed. This string is composed of characters, each of which is printed exactly as it appears. So `printf("xyz");` would simply print an x, then a y, and finally a z. This is not exactly "formatted" printing, but it is still the basis of what `printf` does.

2.1 Naturally Special Characters

To identify the start of the string, we put a double-quote (`"`) at the front. To identify the end of the string we put another double-quote at the end. But what if we want to actually print a double-quote? We can't exactly put a double-quote in the middle of the string because it would be mistaken for the end-of-string marker. Double-quote is a special character. The normal print-what-you-see rules do not apply.

Different languages take different approaches to this problem. Some require the special character to be entered twice. C uses backslash (virgule, \) as an escape character to change the meaning of the next character after it. Thus, to print a double-quote you type in backslash double-quote. To print a backslash, you must escape it by typing another backslash in front of it. The first backslash means "give the next character its alternate meaning." The second backslash has an alternate meaning of "print a backslash."

Without a backslash, special characters have a natural special meaning. With a backslash they print as they appear. Here is a partial list.

`\`	escape the next character
`\\`	print a backslash
`"`	start or end of string
`\"`	print a double quote
`'`	start or end a character constant
`\'`	print a single quote
`%`	start a format specification
`\%`	print a percent sign

2.2 Alternately Special Characters

On the other hand we have characters that normally print as you would expect, but when you add a backslash, then they become special. An example is the newline character. To print an `n`, we simply type in an `n`. To print a newline, we type in a `\n`, thus in-

voking the alternate meaning of `n`, which is newline. Here is a partial list.

`\a`	audible alert (bell)
`\b`	backspace
`\f`	form feed
`\n`	newline (linefeed)
`\r`	carriage return
`\t`	tab
`\v`	vertical tab

3 Format Specifications

The real power of `printf` is when we are printing the contents of variables. Let's take the format specifier `%d` for example. This prints a number. So, a number must be provided for printing. This is done by adding another argument to the `printf` statement, as shown here.

```
int age;
age = 25;
printf ( "I am %d years old\n", age );
```

In this example, `printf` has two arguments. The first is a string: `"I am %d years old\n"`. The second is an integer, `age`.

3.1 The Argument List

When `printf` processes its arguments, it starts printing the characters it finds in the first argument, one by one. When it finds a percent it knows it has a format specification. It goes to the next argument and uses its value, printing it according to that format specification. It then returns to printing a character at a time (from the first argument).

It is okay to include more than one format specification in the `printf` string. In that case, the first format specification goes with the first additional argument, second goes with second, and so forth. Here is an example:

```
int x = 5, y = 10;
printf ( "x is %d and y is %d\n", x, y );
```

3.2 Percent

Every format specification starts with a percent sign and ends with a letter. The letters are chosen to have some mnemonic meaning. Here is a partial list:

`%c`	print a single **c**haracter
`%d`	print a **d**ecimal (base 10) number
`%e`	print an **e**xponential floating-point number
`%f`	print a **f**loating-point number
`%g`	print a **g**eneral-format floating-point number
`%i`	print an **i**nteger in base 10
`%o`	print a number in **o**ctal (base 8)
`%s`	print a **s**tring of characters
`%u`	print an **u**nsigned decimal (base 10) number
`%x`	print a number in he**x**idecimal (base 16)
`%%`	print a percent sign (`\%` also works)

To print a number in the simple way, the format specifier is simply `%d`. Here are some sample cases and results.

`printf`	produces
`("%d",0)`	`0`
`("%d",-7)`	`-7`
`("%d",1560133635)`	`1560133635`
`("%d",-2035065302)`	`-2035065302`

Notice that in the simple, `%d` way, there is no predetermined size for the result. `printf` simply takes as much space as it needs.

3.3 The Width Option

As I mentioned above, simply printing numbers was not enough. There were special options that were desired. The most important was probably the width option. By saying `%5d` the number was guaranteed to take up five spaces (more if necessary, never less). This was very useful in printing tables because small and large numbers both took the same amount of space. Nearly all printing was monospaced in those days, which means that a `w` and an `i` both took the same amount of space. This is still common in text editors used by programmers.

To print a number with a certain (minimum) width, say 5 spaces wide, the format specifier is `%5d`. Here are some sample cases and results. (We will use the ␣ symbol to explicitly indicate a space.)

`printf`	produces
`("%5d",0)`	␣␣␣␣`0`
`("%5d",-7)`	␣␣␣`-7`
`("%5d",1560133635)`	`1560133635`
`("%5d",-2035065302)`	`-2035065302`

Notice that for shorter numbers, the result is padded out with leading spaces. For excessively long

numbers there is no padding, and the full number is printed.

In normal usage, one would make the field wide enough for the biggest number one would ever expect. If your numbers are usually one, two, or three digits long, then `%3d` is probably adequate. In abnormal usage, one could end up printing a number that is too big for the field. `printf` makes the decision to print such numbers fully, even though they take too much space. This is because it is better to print the right answer and look ugly than to print the wrong answer and look pretty.

3.4 Filling the Extra Space

When printing a small number like 27 in a `%5d` field, the question then became where to put the 27 and what to put in the other three slots. It could be printed in the first two spaces, the last two spaces, or maybe the middle two spaces (if that can be determined). The empty spaces could be filled with the blank character, or perhaps stars (`***27` or `27***` or `**27*`), or dollar signs (`$$$27`), or equal signs (`===27`), or leading zeros (like `00027`).

These extra characters are often called "check protection" characters because the are intended to prevent bad guys from changing the dollar amount on a printed check. It is relatively easy to change a space into something else. It is harder to change a star, a dollar sign, or an equal sign.

`printf` provides space fill (left or right) and zero fill (left only). If you want check protection or centering you need to make other arrangements. But even without check protection or centering `printf` still has an impressive (and bewildering) collection of options.

3.5 The Justify Option

Using `printf` numbers can be left-justified (printed in the left side of the field) or right-justified (printed in the right side of the field). The most natural way to print numbers seems to be right-justified with leading spaces. That is what `%5d` means: print a base-10 number in a field of width 5, with the number right-aligned and front-filled with spaces.

To make the number left-aligned, a minus sign is added to the format specifier. To print a number 5 spaces wide and left-justified (left-aligned) the format specifier is `%-5d`. Here are some sample cases and results.

printf	produces
`("%-5d",0)`	`0␣␣␣␣`
`("%-5d",-7)`	`-7␣␣␣`
`("%-5d",1560133635)`	`1560133635`
`("%-5d",-2035065302)`	`-2035065302`

As before, for shorter numbers, the result is padded out with spaces. For longer numbers there is no padding, and the number is not shortened.

3.6 The Zero-Fill Option

To make things line up nice and pretty, it is common to write a date using leading zeros. We can write May 5, 2003 in the US as 05/05/2003. We could also write it as 2003.05.05. Notice that in both cases, the leading zeros do not change the meaning. They just make it line up nicely in lists.

When a number is zero-filled, the zeros always go in front, and the resulting number is both left- and right-justified. In this case the minus sign has no effect. To print a zero-filled number 5 spaces wide the format specifier is `%05d`. Here are some sample cases and results.

printf	produces
`("%05d",0)`	`00000`
`("%05d",-7)`	`-0007`
`("%05d",1560133635)`	`1560133635`
`("%05d",-2035065302)`	`-2035065302`

Shorter numbers are padded out with leading zeros. Longer numbers are unchanged.

3.7 Fun With Plus Signs

Negative numbers always print with a minus sign. Positive numbers and zero usually do not print with a sign, but you can request one. A plus (`+`) in the format specifier makes that request.

To print a signed number 5 spaces wide the format specifier is `%+5d`. Here are some sample cases and results.

printf	produces
`("%+5d",0)`	`␣␣␣+0`
`("%+5d",-7)`	`␣␣␣-7`
`("%+5d",1560133635)`	`+1560133635`
`("%+5d",-2035065302)`	`-2035065302`

Notice that zero is treated as a positive number. Shorter numbers are padded. Longer numbers are unchanged.

Plus and minus are not related. Both can appear in a format specifier.

3.8 The Invisible Plus Sign

This one is a little bizarre. It is an invisible plus sign. Instead of printing a plus on positive numbers (and zero), we print a space where the sign would go. This can be useful in printing left-justified numbers where you want the minus signs to really stand out. Notice these two alternatives.

printf	produces
("%+-5d",0)	+0␣␣␣␣
("%+-5d",-7)	-7␣␣␣␣
("%+-5d",1560133635)	+1560133635
("%+-5d",-2035065302)	-2035065302

printf	produces
("%␣-5d",0)	␣0␣␣␣␣
("%␣-5d",-7)	-7␣␣␣␣
("%␣-5d",1560133635)	␣1560133635
("%␣-5d",-2035065302)	-2035065302

Remember from above that the format specifier `%-5d` we get the following results (shown again for easier comparison).

printf	produces
("%-5d",0)	0␣␣␣␣
("%-5d",-7)	-7␣␣␣
("%-5d",1560133635)	1560133635
("%-5d",-2035065302)	-2035065302

Notice that the plus signs disappear, but the sign still takes up space at the front of the number.

Notice also that we can combine several options in the same format specifier. In this case, we have combined the options plus, minus, and five, or space, minus, and five, or just minus and five.

3.9 Plus, Space, and Zero

Here is another example of combining several options at the same time.

Using the format specifier `%␣05d` or `%0␣5d` we get the following results.

printf	produces
("%␣05d",0)	␣0000
("%␣05d",-7)	-0007
("%␣05d",1560133635)	␣1560133635
("%␣05d",-2035065302)	-2035065302

Using the format specifier `%+05d` or `%0+5d` we get the following results.

printf	produces
("%+05d",0)	+0000
("%+05d",-7)	-0007
("%+05d",1560133635)	+1560133635
("%+05d",-2035065302)	-2035065302

When we combine plus and space at the same time, the space arranges for room for a sign and the plus uses it. It is the same as if the space was not even specified. The plus takes priority over the space.

3.10 Summary

The options are also called "flags" and among themselves they can appear in any order. Here is a partial list.

flag	effect
none	print normally (right justify, space fill)
-	left justify
0	leading zero fill
+	print plus on positive numbers
␣	invisible plus sign

After the options, if any, the minimum field width can be specified.

4 Printing Strings

The `%s` option allows us to print a string inside a string. Here is an example.

```
char * grade;
if ( year == 11 ) grade = "junior";
printf ( "%s is a %s\n", "Fred", grade );
```

The left-justify flag applies to strings, but of course the zero fill, plus sign, and invisible plus sign are meaningless.

printf	produces
("%5s","")	␣␣␣␣␣
("%5s","a")	␣␣␣␣a
("%5s","ab")	␣␣␣ab
("%5s","abcdefg")	abcdefg

printf	produces
("%-5s","")	␣␣␣␣␣
("%-5s","a")	a␣␣␣␣
("%-5s","ab")	ab␣␣␣
("%-5s","abcdefg")	abcdefg

5 Floating Point

Floating point numbers are those like `3.1415` that have a decimal point someplace inside. This is in contrast to ordinary integers like `27` that have no decimal point.

All the same flags and rules apply for floating point numbers as do for integers, but we have a few new options. The most important is a way to specify how many digits appear after the decimal point. This number is called the **precision** of the number.

For ordinary commerce, prices are often mentioned as whole dollars or as dollars and cents (zero or two digits of precision). For gasoline, prices are mentioned as dollars, cents, and tenths of a cent (three digits of precision). Here are some examples of how to print these kinds of numbers. Let `e=2.718281828`.

printf	produces
`("%.0f",e)`	`3`
`("%.0f.",e)`	`3.`
`("%.1f",e)`	`2.7`
`("%.2f",e)`	`2.72`
`("%.6f",e)`	`2.718282`
`("%f",e)`	`2.718282`
`("%.7f",e)`	`2.7182818`

Notice that if a dot and a number are specified, the number (the precision) indicates how many places should be shown after the decimal point.

Notice that if no dot and precision are specified for `%f`, the default is `%.6f` (six digits after the decimal point).

Notice that if a precision of zero is specified, the decimal point also disappears. If you want it back, you must list it separately (after the `%f` format specifier).

We can specify both a width and a precision at the same time. Notice especially that `5.2` means a total width of five, with two digits after the decimal. **It is very common and natural to think it means five digits in front of the decimal and two digits after, but that is not correct.** Be careful.

printf	produces
`("%5.0f",e)`	`␣␣␣␣3`
`("%5.0f.",e)`	`␣␣␣␣3.`
`("%5.1f",e)`	`␣␣2.7`
`("%5.2f",e)`	`␣2.72`
`("%5.7f",e)`	`2.7182818`

We can also combine precision with the flags we learned before, to specify left justification, leading zeros, plus signs, etc.

printf	produces
`("%5.1f",e)`	`␣␣2.7`
`("%-5.1f",e)`	`2.7␣␣`
`("%+5.1f",e)`	`␣+2.7`
`("%+-5.1f",e)`	`+2.7␣`
`("%05.1f",e)`	`002.7`
`("%+05.1f",e)`	`+02.7`
`("%␣05.1f",e)`	`␣02.7`
`("%-␣5.1f",e)`	`␣2.7␣`

6 Designing The Perfect Spec

If you are designing a `printf` format specifier, the first step is to decide what kind of a thing you are printing. If it is an integer, a float, a string, or a character, you will make different choices about which basic format to use.

The second big question is how wide your field should be. Usually this will be the size of the biggest number you ever expect to print under normal circumstances. Sometimes this is controlled by the amount of space that is provided on a pre-printed form (such as a check or an invoice).

Decide what you would like printed under a variety of circumstances. In this paper we have often illustrated the results by using a small positive number, a small negative number, an oversized positive number, and an oversized negative number. You should include this options as well as large (but not oversized) numbers. Design your format for the biggest number that you would normally expect to see.

7 Hints for the Test

The `printf` test includes a variety of matching problems. They are designed to be tricky, and student feedback indicates that if anything, they are more tricky that expected.

You can use the process of elimination to make this test very fast and accurate. As you look at a common feature in the answer line, you can rule out all those `printf` statements that do not have that feature. Very quickly you can narrow your options to one or two.

7.1 Easy Features

It is easy to see if the short numbers have leading zeros. If so, there must be a zero in the formatting specification.

It is easy to see if the positive numbers have plus signs. If so, there must be a plus in the formatting specification.

7.2 Before, Between, and Behind

The next thing to watch for is the before, between, and behind of the number that is printed. In a formatting specification like `x%5dz` there is an `x` before the number and a `z` behind the number. The `x` and `z` are not part of the format specification, but they are part of the printed result. Everything else that prints is "between."

To decide whether there is anything before or behind a number, look at the oversized negative number. Any spaces before it are surely before the format specification. Any spaces behind it are surely behind the format specification. Here is an example.

If -2035065302 prints as ␣␣`-2035065302`␣, you can be sure that the `printf` string is ␣␣`%...`␣, with two spaces before the formatting specification and one space behind. That is because all the print positions between (the `%` and whatever goes with it) are used up by the oversized number.

Once you have determined what is before and behind, you can use that information to match against the matching choices. Often this will give you the answer directly.

7.3 The Invisible Plus Sign

Compare the oversized negative number to the oversized positive number. If the positive number has an extra space in front, it is an invisible plus sign. If there is no extra space, there is no invisible plus sign.

7.4 Left Justification

Subtract the before and behind. Look at what is left. Look at the small negative number. Where are the extra spaces printed? If the are in front, the number is right justified. If they are in back, the number is left justified. If they are in front and in back, you did something wrong.

8 Conclusion

The `printf` function is a powerful device for printing numbers and other things stored in variables. With this power there is a certain amount of complexity. Taken all at once, the complexity makes `printf` seem almost impossible to understand. But the complexity can be easily unfolded into simple features, including width, precision, signage, justification, and fill. By recognizing and understanding these features, `printf` will become a useful and friendly servant in your printing endeavors.