CSC 180 lab 5: Printf

Thurs Oct 11 or Mon Oct 15, 2001

Since Monday October 8th is a holiday, the Monday lab section is now pushed back by a week. This makes up for the fact that the Thursday 6 September lab session was cancelled (because it was before most of its enrolees' first CSC 180 lecture). Both groups, thus, have twelve lab sessions.

This lab contains some material I hope you will find interesting, but the previous lab about loops is more important. Some of you seemed to be working on your assignment last week instead of doing the lab; this is quite understandable, but if you did not do most of lab 4 on loops, I strongly recommend you complete that *first*, and do this lab afterwards, if you have time.

Input and output occur in just about every computer program. C has a wealth of different data types, so input and output are made more complex than in some other programming languages. Also, printf() is a library function, and C does not have ''polymorphism'' for function calls. To explain polymorphism, contrast the printf situation with that of operators: ''x+y'' automatically performs floating-point addition if x and y are of float or double type, but integer addition if x and y are both of type int. Thus we say that ''+'' in C is ''polymorphic'' (to this small extent). But since output is performed by function calls in C, it is not possible for there to be a library function print(x) which automatically works differently depending upon the type of x.

Thus we have to tell printf the type of the data we're passing in for printing, e.g. whether it is int, double, or ''string'' (''string'' being an array of chars with a terminating zero). Furthermore, we often are concerned with the details of how the output is formatted, and there are many ways to control this in the first parameter to printf.

1. Write a short program to print a value in dollars and cents. Your main function should begin with

```
int dollars = 2;
int cents = 3;
```

and then print these two values with a format of "$%d.%d". Try this out and observe the problem, which we will fix in the next few steps.

2. Additional instructions can be placed between the '%' and the printf key letter (here 'd') in the printf format. A number indicates a ''field width''. Try using 2 for the cents value's format's field width (but leave the dollar value's format at simply ''%d''). This is not good enough yet, but it is better. Try it before moving on to the next step (in which we fix it).

3. Other options can go between the '%' and the key letter. When the cents value does not take up as much space as the field width, it is padded on the left, so that the padded output does take the required width. By default, the padding is done by inserting spaces. However, for our dollars-and-cents output, we want the cents field to be padded by zeroes instead of spaces. There is an option '0' which tells printf to use zeroes instead of spaces. (There is no general mechanism; your choices are zeroes or spaces.)

This option is required to be placed prior to the number indicating the field width; ''02'' means a field width of two and also specifies this zeroes option; ''20'' would mean a field width of twenty and would not specify this zeroes option.

Use ''%02d'' to format the cents correctly.

(continued)

4. Write a program which prints the squares of the numbers from one to ten, looking like this, by using field width specifiers to make them line up.

```
 n   n**2
---  ----
 1     1
 2     4
 3     9
 4    16
 5    25
 6    36
 7    49
 8    64
 9    81
10   100
```

We usually want numbers to be ‘‘right-justified’’ in this way.

5. What happens when the integer is too large (too many digits) for the field width? What happens when the integer is too small? Write little test programs to answer these questions, if the answer is not already apparent.

6. How many ways can you think of formatting a floating-point number, say 1.034?

```
1.034
1.034000
  1.034   (with a width of three for the "1" part)
```

To produce such formattings, we need to add a *precision* specifier. We do this by putting a decimal point and then the precision, after the field width if any, before the key letter. The field width value includes the entire output of that number, including all the decimal places and the decimal point itself. So the above formats would be produced by "%5.3f", "%8.6f", and "%7.3f", respectively.

7. Experiment with width and precisions for %f. Also keep in mind the availability of %g (usually used without precision values, although if you use a field width, you usually want a precision value so that the decimal points line up in a column, as in the table of squares of integers above).

What happens when the format is too small for the number of digits required to the right? What happens when the field width is too small for the number of digits required to the left, for a given precision?

8. What effect do the field width and precision values have on the printing of strings (with "%s")? (For strings, we often provide only a field width or only a precision; you can omit the field width and provide a precision by including the ‘.’ separator.) And again, what happens when the string is too long for the field width?

9. You should have noticed in investigating the previous question that the strings are ‘‘left-justified’’ in their fields. This is usually desired (look at tables of names and numbers, such as student record listings with student numbers, or course transcripts with course names and numeric grades, or a phone book with names and phone numbers). But this is not *always* what we want. The option character ‘−’, which appears before any field width, requests the opposite justification. Experiment with this.

(continued)

10. When we initialize an array, we specify a complete list of values for all of its elements, not just one data value. In the case of an array of char, we can initialize it with a string. When we initialize an array, we can omit the array dimension, which means that we are asking the compiler to count the number of data values in the initializer, and dimension the array appropriately. Hence the common string variable declaration style of:

```
char hello[] = "Hello, world";
```

This declares hello to be of type array-of-char, of dimension 13, where hello[0] is 'H', hello[1] is 'e', and so on, and hello[11] is 'd', and hello[12] is 0.

In the 1989 version of C, aggregate initializations can only be made to global variables, so the above declaration would have to appear *outside* any function. As we did in lab 2.

If we would like to centre this string in an 80-column screen width, we need to print it right-justified in a field width of (80+strlen(hello))/2. That is, there are 80−strlen(hello) blank columns, half of which should proceed the string; so the *end* of the string will occur at (80−strlen(hello))/2+strlen(hello) characters, which equals (80+strlen(hello))/2.

We could construct an appropriate printf format string of the form "%−46s". Better yet, though, would be if that value of 46 could be another parameter to printf.

Indeed printf does have such a feature, as follows. Anywhere that there is supposed to be a field width *or* a precision value, you can put an asterisk, and then it takes a parameter which specifies what number should go there. A suitable format in this case, therefore, would look like "%−*s". The size parameter (e.g. 46) appears before the argument parameter (e.g. the ‘‘hello’’).

Write a program which contains

```
char s[] = "something";
```

and whose main() function outputs that ‘‘something’’ centred on the screen, with the formula involving strlen(), so that you can change the string initialization to any string and have it output centred.

11. Write a program which takes two input numbers and then prints the second number in a field width specified by the first, padded with zeroes. For example, if the input is ‘‘10 123’’, the output should be ‘‘0000000123’’.

12. If you have more time, redo our triangle.c (http://www.dgp.toronto.edu/~ajr/180/example/3/triangle.c) by using "%*c" to print an asterisk with a calculated number of spaces preceding it.