Engr 101
Notes on printf, fprintf, and sprint

These notes have been gathered from several different sources as noted.

http://stackoverflow.com/questions/4627330/difference-between-fprintf-printf-and-sprintf

printf, fprintf, sprint Explained

In C, a "stream" is an abstraction; from the program's perspective it is simply a producer (input stream) or consumer (output stream) of bytes. It can correspond to a file on disk, to a pipe, to your terminal, or to some other device such as a printer or tty. The `FILE` type contains information about the stream. Normally, you don't mess with a `FILE` object's contents directly, you just pass a pointer to it to the various I/O routines.

There are three standard streams: `stdin` is a pointer to the standard input stream, `stdout` is a pointer to the standard output stream, and `stderr` is a pointer to the standard error output stream. In an interactive session, the three usually refer to your console, although you can redirect them to point to other files or devices:

```
$ myprog < inputfile.dat > output.txt 2> errors.txt
```

In this example, `stdin` now points to `inputfile.dat`, `stdout` points to `output.txt`, and `stderr`points to `errors.txt`.

`fprintf` writes formatted text to the output stream you specify.

`printf` is equivalent to writing `fprintf(stdout, ...)` and writes formatted text to wherever the standard output stream is currently pointing.

`sprintf` writes formatted text to an array of `char`, as opposed to a stream.

scanf format string

Scanf format string (which stands for "**scan f**ormatted") refers to a control parameter used by a class of functions in the string-processing libraries of various programming languages. The**format string** specifies a method for reading a string into an arbitrary number of varied data type parameter(s). The input string is by default read from the standard input, but variants exist that read the input from other sources.

The term "scanf" is due to the C language, which popularized this type of function, but these functions predate C, and other names are used, such as "readf" in ALGOL 68**.** Scanf format strings, which provide formatted input (parsing), are complementary to printf format strings, which provide formatted output (templating). In both cases these provide simple functionality and fixed format compared to more sophisticated and flexible parsers or template engines, but are sufficient for many purposes.

.[1]

Usage[edit]

The `scanf` function, which is found in C, reads input for numbers and other datatypes from standard input (often a command line interface or similar kind of a text user interface).

The following shows code in C that reads a variable number of unformatted decimal integers from the console and prints out each of them on a separate line:

```
#include <stdio.h>
int main(void)
{
    int  n;
    while (scanf("%d", &n) == 1)
        printf("%d\n", n);
    return 0;
}
```

After being processed by the program above, a messy list of integers such as

```
456 123 789     456 12
456 1
      2378
```

will appear neatly as:

```
456
123
789
```

```
456
12
456
1
2378
```

To print out a word:

```
#include <stdio.h>
int main(void)
{
    char  word[20];
    if (scanf("%19s", word) == 1)
        puts(word);
    return 0;
}
```

No matter what the datatype the programmer wants the program to read, the arguments (such as `&n` above) must be pointers pointing to memory. Otherwise, the function will not perform correctly because it will be attempting to overwrite the wrong sections of memory, rather than pointing to the memory location of the variable you are attempting to get input for.

In the last example an address-of operator (`&`) is *not* used for the argument: as `word` is the name of an array of `char`, as such it is (in all contexts in which it evaluates to an address) equivalent to a pointer to the first element of the array. While the expression `&word` would numerically evaluate to the same value, semantically it has an entirely different meaning in that it stands for the address of the whole array rather than an element of it. This fact needs to be kept in mind when assigning `scanf` output to strings.

As `scanf` is designated to read only from standard input, many programming languages with interfaces, such as PHP, have derivatives such as `sscanf` and `fscanf` but not `scanf` itself.

Format string specifications[edit]

The formatting placeholders in `scanf` are more or less the same as that in `printf`, its reverse function.

There are rarely constants (i.e. characters that are not formatting placeholders) in a format string, mainly because a program is usually not designed to read known data. The exception is one or more whitespace characters, which discards all whitespace characters in the input.

Some of the most commonly used placeholders follow:

- `%d` : Scan an integer as a signed decimal number.
- `%i` : Scan an integer as a signed number. Similar to `%d`, but interprets the number as hexadecimal when preceded by `0x` and octal when preceded by `0`. For example, the

string `031` would be read as 31 using `%d`, and 25 using `%i`. The flag `h` in `%hi` indicates conversion to a `short` and `hh` conversion to a `char`.

- `%u` : Scan for decimal `unsigned int` (Note that in the C99 standard the input value minus sign is optional, so if a minus sign is read, no errors will arise and the result will be the two's complement of a negative number, likely a very large value. See `strtoul()`.[not in citation given]) Correspondingly, `%hu` scans for an `unsigned short` and `%hhu` for an `unsigned char`.
- `%f` : Scan a floating-point number in normal (fixed-point) notation.
- `%g`, `%G` : Scan a floating-point number in either normal or exponential notation. `%g` uses lower-case letters and `%G` uses upper-case.
- `%x`, `%X` : Scan an integer as an unsigned hexadecimal number.
- `%o` : Scan an integer as an octal number.
- `%s` : Scan a character string. The scan terminates at whitespace. A null character is stored at the end of the string, which means that the buffer supplied must be at least one character longer than the specified input length.
- `%c` : Scan a character (char). No null character is added.
- whitespace: Any whitespace characters trigger a scan for zero or more whitespace characters. The number and type of whitespace characters do not need to match in either direction.
- `%lf` : Scan as a double floating-point number.
- `%Lf` : Scan as a long double floating-point number.

The above can be used in compound with numeric modifiers and the `l`, `L` modifiers which stand for "long" in between the percent symbol and the letter. There can also be numeric values between the percent symbol and the letters, preceding the `long` modifiers if any, that specifies the number of characters to be scanned. An optional asterisk (`*`) right after the percent symbol denotes that the datum read by this format specifier is not to be stored in a variable. No argument behind the format string should be included for this dropped variable.

The `ff` modifier in printf is not present in scanf, causing differences between modes of input and output. The `ll` and `hh` modifiers are not present in the C90 standard, but are present in the C99 standard.[2]

An example of a format string is

```
"%7d%s %c%lf"
```

The above format string scans the first seven characters as a decimal integer, then reads the remaining as a string until a space, new line or tab is found, then scans the first non-whitespace character following and a double-precision floating-point number afterwards.

Error handling[edit]

`scanf` is usually used in situations when the program cannot guarantee that the input is in the expected format. Therefore a robust program must check whether the `scanf` call succeeded and take appropriate action. If the input was not in the correct format, the erroneous data will still be on the input stream and must be read and discarded before new input can be read. An alternative method of reading input, which avoids this, is to use `fgets` and then examine the string read in. The last step can be done by `sscanf`, for example.

Vulnerabilities[edit]

Like `printf`, `scanf` is vulnerable to format string attacks. Great care should be taken to ensure that the formatting string includes limitations for string and array sizes. In most cases the input string size from a user is arbitrary; it can not be determined before the `scanf` function is executed. This means that uses of `%s` placeholders without length specifiers are inherently insecure and exploitable for buffer overflows. Another potential problem is to allow dynamic formatting strings, for example formatting strings stored in configuration files or other user controlled files. In this case the allowed input length of string sizes can not be specified unless the formatting string is checked beforehand and limitations are enforced. Related to this are additional or mismatched formatting placeholders which do not match the actual vararg list. These placeholders might be partially extracted from the stack, contain undesirable or even insecure pointers depending on the particular implementation of varargs.

C library function - printf()

Previous Page

Next Page

Description

The C library function **int printf(const char *format, ...)** sends formatted output to stdout.

Declaration

Following is the declaration for printf() function.

```
int printf(const char *format, ...)
```

Parameters

- **format** − This is the string that contains the text to be written to stdout. It can optionally contain embedded format tags that are replaced by the values specified in subsequent additional arguments and formatted as requested. Format tags prototype is **%[flags][width][.precision][length]specifier**, which is explained below −

specifier	**Output**
c	Character
d or i	Signed decimal integer
e	Scientific notation (mantissa/exponent) using e character

E	Scientific notation (mantissa/exponent) using E character
f	Decimal floating point
g	Uses the shorter of %e or %f
G	Uses the shorter of %E or %f
o	Signed octal
s	String of characters
u	Unsigned decimal integer
x	Unsigned hexadecimal integer
X	Unsigned hexadecimal integer (capital letters)
p	Pointer address
n	Nothing printed
%	Character

flags	**Description**
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a -ve sign.

(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with e, E and f, it forces the written output to contain a decimal point even if no digits would follow. By default, if no digits follow, no decimal point is written. Used with g or G the result is the same as with e or E but trailing zeros are not removed.
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

width	**Description**
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	**Description**
.number	For integer specifiers (d, i, o, u, x, X) − precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For e, E and f specifiers − this is the number of digits to be printed after the decimal point. For g and G specifiers − This is the maximum number of significant digits to be printed. For s − this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. For c type − it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.

.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	Description
h	The argument is interpreted as a short int or unsigned short int (only applies to integer specifiers: i, d, o, u, x and X).
l	The argument is interpreted as a long int or unsigned long int for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a long double (only applies to floating point specifiers: e, E, f, g and G).

- **additional arguments** – Depending on the format string, the function may expect a sequence of additional arguments, each containing one value to be inserted instead of each %-tag specified in the format parameter (if any). There should be the same number of these arguments as the number of %-tags that expect a value.

Return Value

If successful, the total number of characters written is returned. On failure, a negative number is returned.

Example

The following example shows the usage of printf() function.

```
#include <stdio.h>


int main ()

{

   int ch;
   for( ch = 75 ; ch <= 100; ch++ )

   {

      printf("ASCII value = %d, Character = %c\n", ch , ch );

   }


   return(0);

}
```

Let us compile and run the above program to produce the following result −

```
ASCII value = 75, Character = K
ASCII value = 76, Character = L
ASCII value = 77, Character = M
ASCII value = 78, Character = N
ASCII value = 79, Character = O
ASCII value = 80, Character = P
ASCII value = 81, Character = Q
ASCII value = 82, Character = R
ASCII value = 83, Character = S
ASCII value = 84, Character = T
ASCII value = 85, Character = U
ASCII value = 86, Character = V
ASCII value = 87, Character = W
ASCII value = 88, Character = X
ASCII value = 89, Character = Y
ASCII value = 90, Character = Z
ASCII value = 91, Character = [
ASCII value = 92, Character = \
ASCII value = 93, Character = ]
ASCII value = 94, Character = ^
ASCII value = 95, Character = _
ASCII value = 96, Character = `
ASCII value = 97, Character = a
ASCII value = 98, Character = b
ASCII value = 99, Character = c
ASCII value = 100, Character = d
```