**DESCRIPTION**

The functions in the **printf**() family produce output according to a format as described below. The function **printf**() writes output to stdout, the standard output stream.

All of these functions write the output under the control of a format string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg**(3)) are converted for output.

**Format of the format string**

The format string is a character string, beginning and ending in its *initial shift state*, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a conversion specifier. In between there may be (in this order) zero or more flags, an optional minimum field width and an optional precision.

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each '\*' (see Field width and Precision below) and each conversion specifier asks for the next argument (and it is an error if insufficiently many arguments are given). One can also specify explicitly which argument is taken, at each place where an argument is required, by writing "%m$" instead of '%' and "\*m$" instead of '\*', where the decimal integer m denotes the position in the argument list of the desired argument, indexed starting from 1. Thus,

printf("%\*d", width, num);

and

printf("%2$\*1$d", width, num);

are equivalent. The second style allows repeated references to the same argument. The C99 standard does not include the style using '$', which comes from the Single UNIX Specification. If the style using '$' is used, it must be used throughout for all conversions taking an argument and all width and precision arguments, but it may be mixed with "%%" formats, which do not consume an argument. There may be no gaps in the numbers of argu ments specified using '$'; for example, if arguments 1 and 3 are specified, argument 2 must also be specified somewhere in the format string.

For some numeric conversions a radix character ("decimal point") or thousands' grouping character is used. The actual character used depends on the **LC\_NUMERIC** part of the locale. (See **setlocale**(3).) The POSIX locale uses '.' as radix character, and does not have a grouping character. Thus,

printf("%'.2f", 1234567.89);

results in "1234567.89" in the POSIX locale, in "1234567,89" in the nl\_NL locale, and in "1.234.567,89" in the da\_DK locale.

**Flag characters**

The character % is followed by zero or more of the following flags:

**#** The value should be converted to an "alternate form". For **o** conversions, the first character of the output string is made zero (by prefixing a 0 if it was not zero already). For **x** and **X** conversions, a nonzero result has the string "0x" (or "0X" for X conversions) prepended to it. For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the result will always contain a decimal point, even if no digits follow it (normally, a decimal point appears in the results of those conversions only if a digit follows). For **g** and **G** conversions, trailing zeros are not removed from the result as they would otherwise be. For other conversions, result is undefined.

**0** The value should be zero padded. For **d**, **i**, **o**, **u**, **x**, **X**, **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, the converted value is padded on the left with zeros rather than blanks. If the **0** and **-** flags both appear, the **0** flag is ignored. If a precision is given with a numeric conversion (**d**, **i**, **o**, **u**, **x**, and **X**), the **0** flag is ignored. For other conversions, the behavior is undefined.

**-** The converted value is to be left adjusted on the field boundary. (The default is right justification.) The converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A **-** overrides a **0** if both are given.

**‘ ‘** (a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.

**+** A sign (+ or -) should always be placed before a number produced by a signed conversion. By default, a sign is used only for negative numbers. A + overrides a space if both are used.

**Field width**

An optional decimal digit string (with nonzero first digit) specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given). Instead of a decimal digit string one may write "\*" or "\*m$" (for some decimal integer m) to specify that the field width is given in the next argument, or in the m-th argument, respectively, which must be of type int. A negative field width is taken as a '-' flag followed by a positive field width. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

**Precision**

An optional precision, in the form of a period ('.') followed by an optional decimal digit string. Instead of a decimal digit string one may write "\*" or "\*m$" (for some decimal integer m) to specify that the precision is given in the next argument, or in the m-th argument, respectively, which must be of type int. If the precision is given as just '.', the precision is taken to be zero. A negative precision is taken as if the precision were omitted. This gives the minimum number of digits to appear for **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the radix character for **a**, **A**, **e**, **E**, **f**, and **F** conversions, the maximum number of significant digits for **g** and **G** conversions, or the maximum number of characters to be printed from a string for **s** and **S** conversions.

**Conversion specifiers**

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

**d, i** The int argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

**o, u, x, X** The unsigned int argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation. The letters **abcdef** are used for **x** conversions; the letters **ABCDEF** are used for **X** conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros. The default precision is 1. When 0 is printed with an explicit precision 0, the output is empty.

**c** If no **l** modifier is present, the int argument is converted to an unsigned char, and the resulting character is written. If an **l** modifier is present, the wint\_t (wide character) argument is converted to a multibyte sequence by a call to the **wcrtomb**(3) function, with a conversion state starting in the initial state, and the resulting multibyte string is written.

**s** If no **l** modifier is present: the const char \* argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

If an **l** modifier is present: the const wchar\_t \* argument is expected to be a pointer to an array of wide characters. Wide characters from the array are converted to multibyte characters (each by a call to the **wcrtomb**(3) function, with a conversion state starting in the initial state before the first wide character), up to and including a terminating null wide character. The resulting multibyte characters are written up to (but not including) the terminating null byte. If a precision is specified, no more bytes than the number specified are written, but no partial multibyte characters are written. Note that the precision determines the number of bytes written, not the number of wide characters or screen positions. The array must contain a terminating null wide character, unless a precision is given and it is so small that the number of bytes written exceeds it before the end of the array is reached.

**p** The void \* pointer argument is printed in hexadecimal (as if by **%#x** or **%#lx**).

**%** A '%' is written. No argument is converted. The complete conversion specification is '%%'.

**RETURN VALUE**

Upon successful return, these functions return the number of characters printed (excluding the null byte used to end output to strings).

If an output error is encountered, a negative value is returned.

Inner working of ft\_printf

Conversion specification: %(flags)(field\_width)(precision)conv\_spec

*flags* = [**-0# +**]

*field\_width* = [decimal\_digit\_string *or* \*]

*precision* = .[decimal\_digit\_string *or* \*]

conv\_spec = [**dixXcsp%**]

1. Check whether the character if ‘%’
   1. NO – write character to stdout
   2. YES – check next x characters (2)
2. Check for flag character
   1. YES – check for more flag character (2)
   2. NO – check for field\_width (3)
3. Check for field\_width [check for \* and with atoi, there already isn’t 0 or – or +]
   1. NO/YES – check for precision (4)
4. Check for precision (find .)
   1. YES – check for value of precision (5)
   2. NO – check for conv\_spec (6)
5. Check for value of precision [check for \* and with atoi]
   1. NO/YES – check for conv\_spec (6)
6. Check for conv\_spec
   1. YES – write output based on conv\_spec, flags, … (continue on char after conv\_spec)
   2. NO – write ‘%’ to stdout (continue on char after ‘%’)

Functions which print to stdout

* Input:
  + Variable
  + Conversion specification info (maybe not necessary, separate function for each conversion) -> conversion for digits, char, string, ptr, hexadecimal, …
* Outputs:
  + Ptr/index of next char
  + Number of printed char