NAME

printf, fprintf, sprintf – print formatted output

SYNOPSIS

```
#include <stdio.h>
int printf(const char *format, /* args */ ... );
int fprintf(FILE *strm, const char *format, /* args */ ... );
int sprintf(char *s, const char *format, /* args */ ... );
```

MT-LEVEL

See the **NOTES** section of this page.

DESCRIPTION

printf( ) places output on the standard output stream **stdout**.

fprintf( ) places output on *strm*.

sprintf( ) places output, followed by the null character (**\0**), in consecutive bytes starting at *s*. It is the user's responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the **\0** in the case of **sprintf( )**) or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its *args* under control of the *format*. The *format* is a character string that contains three types of objects defined below:

1. plain characters that are simply copied to the output stream;
2. escape sequences that represent non-graphic characters;
3. conversion specifications.

The following escape sequences produce the associated action on display devices capable of the action:

\a Alert. Ring the bell.

\b Backspace. Move the printing position to one character before the current position, unless the current position is the start of a line.

\f Form feed. Move the printing position to the initial printing position of the next logical page.

\n Newline. Move the printing position to the start of the next line.

\r Carriage return. Move the printing position to the start of the current line.

\t Horizontal tab. Move the printing position to the next implementation-defined horizontal tab position on the current line.

\v Vertical tab. Move the printing position to the start of the next implementation-defined vertical tab position.

All forms of the **printf( )** functions allow for the insertion of a language-dependent decimal-point character. The decimal-point character is defined by the program's locale (category **LC_NUMERIC**). In the C locale, or in a locale where the decimal-point character is not defined, the decimal-point character defaults to a period (**.**).

Each conversion specification is introduced by the character **%**. After the character **%**, the following appear in sequence:

An optional field, consisting of a decimal digit string followed by a **$**, specifying the next *args* to be converted. If this field is not provided, the *args* following the last *args* converted will be used.

Zero or more *flags*, which modify the meaning of the conversion specification.

An optional string of decimal digits to specify a minimum *field width*. If the converted value has fewer characters than the field width, it will be padded on the left (or right, if the left-adjustment flag (–), described below, has been given) to the field width. If the format is **%s** or **%ws** (wide-character string), then the field width should be interpreted as the minimum columns of screen display. E.g. **%10s** means if the converted value has a screen width of 7 columns, then 3 spaces would be padded on the right.

An optional precision that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, or **X** conversions (the field is padded with leading zeros), the number of digits to appear after the decimal-point character for the **e**, **E**, and **f** conversions, the maximum number of significant digits for the **g** and **G** conversions, or the maximum number of characters to be printed from a string in **s** or **ws** conversions. The precision takes the form of a period (**.**) followed by a decimal digit string; a null digit string is treated as zero. Padding specified by the precision overrides the padding specified by the field width.

An optional **h** specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **short int** or **unsigned short int** argument (the argument will be promoted according to the integral promotions and its value converted to **short int** or **unsigned short int** before printing); an optional **h** specifies that a following **n** conversion specifier applies to a pointer to a **short int** argument. An optional **l** (ell) specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long int** or **unsigned long int** argument; an optional **l** (ell) specifies that a following **n** conversion specifier applies to a pointer to a **long int** argument. An optional **ll** (ell ell) specifies that a following **d**, **i**, **o**, **u**, **x**, or **X** conversion specifier applies to a **long long int** or **unsigned long long int** argument; an optional **ll** (ell ell) specifies that a following **n** conversion specifier applies to a pointer to a **long long int** argument. An optional **L** specifies that a following **e**, **E**, **f**, **g**, or **G** conversion specifier applies to a **long double** argument. If an **h**, **l**, or **L** appears before any other conversion specifier, the behavior is undefined.

A conversion character (see below) that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (∗) instead of a digit string. In this case, an integer *args* supplies the field width or precision. The *args* that is actually converted is not fetched until the conversion letter is seen, so the *args* specifying field width or precision must appear before the *args* (if any) to be converted. If the *precision* argument is negative, it will be changed to zero. A negative field width argument is taken as a – flag, followed by a positive field width.

In format strings containing the **digits*$ form of a conversion specification, a field width or precision may also be indicated by the sequence **digits*$, giving the position in the argument list of an integer *args* containing the field width or precision.

When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*–1)th, be specified in the format string.

The *flag* characters and their meanings are:

– The result of the conversion will be left-justified within the field. (It will be right-justified if this flag is not specified.)

+ The result of a signed conversion will always begin with a sign (+ or –). (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space If the first character of a signed conversion is not a sign, a space will be placed before the result. This means that if the space and + flags both appear, the space flag will be ignored.

The value is to be converted to an alternate form. For **c**, **d**, **i**, **s**, and **u** conversions, the flag has no effect. For an **o** conversion, it increases the precision to force the first digit of the result to be a zero. For **x** (or **X**) conversion, a non-zero result will have **0x** (or **0X**) prepended to it. For **e**, **E**, **f**, **g**, and **G** conversions, the result will always contain a decimal-point character, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For **g** and **G** conversions, trailing zeros will not be removed from the result as they normally are.

0 For **d**, **i**, **o**, **u**, **x**, **X**, **e**, **E**, **f**, **g**, and **G** conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the **0** and **–** flags both appear, the **0** flag will be ignored. For **d**, **i**, **o**, **u**, **x**, and **X** conversions, if a precision is specified, the **0** flag will be ignored. For other conversions, the behavior is undefined.

Each conversion character results in fetching zero or more *args*. The results are undefined if there are insufficient *args* for the format. If the format is exhausted while *args* remain, the excess *args* are ignored.

The conversion characters and their meanings are:

d,**i**,**o**,**u**,**x**,**X** The integer *arg* is converted to signed decimal (**d** or **i**), unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal notation (**x** and **X**). The **x** conversion uses the letters **abcdef** and the **X** conversion uses the letters **ABCDEF**. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f The double *args* is converted to decimal notation in the style **[**–**]***ddd***.***ddd*, where the number of digits after the decimal-point character (see

setlocale(3C)) is equal to the precision specification. If the precision is omitted from *arg*, six digits are output; if the precision is explicitly zero and the **#** flag is not specified, no decimal-point character appears. If a decimal-point character appears, at least 1 digit appears before it. The value is rounded to the appropriate number of digits.

e,**E** The double *args* is converted to the style **[**–**]***d***.***ddd***e**±*dd*, where there is one digit before the decimal-point character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is zero and the **#** flag is not specified, no decimal-point character appears. The **E** conversion character will produce a number with **E** instead of **e** introducing the exponent. The exponent always contains at least two digits. The value is rounded to the appropriate number of digits.

g,**G** The double *args* is printed in style **f** or **e** (or in style **E** in the case of a **G** conversion character), with the precision specifying the number of significant digits. If the precision is zero, it is taken as one. The style used depends on the value converted: style **e** (or **E**) will be used only if the exponent resulting from the conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result. A decimal-point character appears only if it is followed by a digit.

c The **int** *args* is converted to an **unsigned char**, and the resulting character is printed.

wc The **int** *args* is converted to a wide character (**wchar_t**), and the resulting wide character is printed.

s The *args* is taken to be a string (character pointer) and characters from the string are written up to (but not including) a terminating null character; if the precision is specified, no more than that many characters are written. If the precision is not specified, it is taken to be infinite, so all characters up to the first null character are printed. A null value for *args* will yield undefined results.

ws The *args* is taken to be a wide character string (wide character pointer) and wide characters from the string are written up to (but not including) a terminating null character; if the precision is specified, no more than that many wide characters are written. If the precision is not specified, it is taken to be infinite, so all wide characters up to the first null character are printed. A null value for *args* will yield undefined results.

p The *args* should be a pointer to **void**. The value of the pointer is converted to an implementation-defined set of sequences of printable characters, which should be the same as the set of sequences that are matched by the **%p** conversion of the **scanf** function.

n The argument should be a pointer to an integer into which is written the

number of characters written to the output standard I/O stream so far by this call to **printf( )**, **fprintf( )**, or **sprintf( )**. No argument is converted.

% Print a %; no argument is converted.

If the character after the **%** or **%***digits***$** sequence is not a valid conversion character, the results of the conversion are undefined.

If a floating-point value is the internal representation for infinity, the output is [±]*Infinity*, where *Infinity* is either **Infinity** or **Inf**, depending on the desired output string length. Printing of the sign follows the rules described above.

If a floating-point value is the internal representation for ''not-a-number,'' the output is [±]*NaN*. Printing of the sign follows the rules described above.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by **printf** and **fprintf** are printed as if the **putc( )** routine had been called.

LC_NUMERIC
Determines how numeric formats are handled. In the "C" locale, numeric handling follows the U.S. rules.

RETURN VALUES

printf( ), **fprintf( )**, and **sprintf( )** return the number of characters transmitted, or return a negative value if an error was encountered.

EXAMPLES

To print a date and time in the form **Sunday, July 3, 10:02**, where **weekday** and **month** are pointers to null-terminated strings:

```
printf("%s, %s %i, %d:%.2d", weekday, month, day, hour, min);
```

To print π to 5 decimal places:

```
printf("pi = %.5f", 4 * atan(1.0));
```

To print a list of names in columns which are 20 characters wide:

```
printf("%20s%20s%20s", lastname, firstname, middlename );
```

FILES

/usr/lib/locale/*locale***/LC_NUMERIC/numeric**
LC_NUMERIC database for *locale*

SEE ALSO

exit(2), **lseek**(2), **write**(2), **abort**(3C), **ecvt**(3C), **putc**(3S), **scanf**(3S), **setlocale**(3C), **stdio**(3S)

NOTES

sprintf( ) is MT-Safe in multi-thread applications. **printf** and **fprintf** can be used safely in a multi-thread application, as long as **setlocale**(3C) is not being called to change the locale.