NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf - formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

snprintf(), **vsnprintf**():
_BSD_SOURCE || _XOPEN_SOURCE >= 500 || _ISOC99_SOURCE ||
_POSIX_C_SOURCE >= 200112L;
or *cc -std=c99*

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The functions **printf**() and **vprintf**() write output to *stdout*, the standard output stream; **fprintf**() and **vfprintf**() write output to the given output *stream*; **sprintf**(), **snprintf**(), **vsprintf**() and **vsnprintf**() write to the character string *str*.

The functions **snprintf**() and **vsnprintf**() write at most *size* bytes (including the terminating null byte (0)) to *str*.

The functions **vprintf**(), **vfprintf**(), **vsprintf**(), **vsnprintf**() are equivalent to the functions **printf**(), **fprintf**(), **sprintf**(), **snprintf**(), respectively, except that they are called with a *va_list* instead of a variable number of arguments. These functions do not call the*va_end* macro. Because they invoke the *va_arg* macro, the value of *ap* is undefined after the call. See stdarg(3).

These eight 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.

C99 and POSIX.1-2001 specify that the results are undefined if a call to **sprintf**(), **snprintf**(), **vsprintf**(), or **vsnprintf**() would cause copying to take place between objects that overlap (e.g., if the target string array and one of the supplied input arguments refer to the same buffer). See NOTES.

Return value

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

The functions **snprintf**() and **vsnprintf**() do not write more than *size* bytes (including the terminating null byte (0)). If the output was truncated due to this limit, then the return value is the number of characters (excluding the terminating null byte) which would have been written to the final string if enough space had been available. Thus, a return value of *size* or more means that the output was truncated. (See also below under NOTES.)

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

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*, an optional *precision* and an optional *length modifier*.

The arguments must correspond properly (after type promotion) with the conversion specifier. By default, the arguments are used in the order given, where each * 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 arguments 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. 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.

The 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, the 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+ o verrides a space if both are used.

The five flag characters above are defined in the C99 standard. The Single UNIX Specification specifies one further flag character.

For decimal conversion (**i**, **d**, **u**, **f**, **F**, **g**, **G**) the output is to be grouped with thousands' grouping characters if the locale information indicates any. Note that many versions of **gcc(1)** cannot parse this option and will issue a warning. (SUSv2 did not include *%F*, but SUSv3 added it.)

glibc 2.2 adds one further flag character.

I For decimal integer conversion (**i**, **d**, **u**) the output uses the locale's alternative output digits, if any. For example, since glibc 2.2.3 this will give Arabic-Indic digits in the Persian (fa_IR) locale.

The 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.

The 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.

The length modifier

Here, integer conversion stands for **d**, **i**, **o**, **u**, **x**, or **X** conversion.

hh A following integer conversion corresponds to a *signed char* or *unsigned char* argument, or a following **n** conversion corresponds to a pointer to a *signed char* argument.

h A following integer conversion corresponds to a *short int* or *unsigned short int* argument, or a following **n** conversion corresponds to a pointer to a *short int* argument.

l (ell) A following integer conversion corresponds to a *long int* or *unsigned long int* argument, or a following **n** conversion corresponds to a pointer to a *long int* argument, or a following **c** conversion corresponds to a *wint_t* argument, or a following **s** conversion corresponds to a pointer to *wchar_t* argument.

ll (ell-ell). A following integer conversion corresponds to a *long long int* or *unsigned long long int* argument, or a following **n** conversion corresponds to a pointer to a *long long int* argument.

L A following **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion corresponds to a *long double* argument. (C99 allows %LF, but SUSv2 does not.) This is a synonym for **ll**.

j A following integer conversion corresponds to an *intmax_t* or *uintmax_t* argument, or a following **n** conversion corresponds to a pointer to an *intmax_t* argument.

z A following integer conversion corresponds to a *size_t* or *ssize_t* argument, or a following **n** conversion corresponds to a pointer to a *size_t* argument.

t A following integer conversion corresponds to a *ptrdiff_t* argument, or a following **n** conversion corresponds to a pointer to a *ptrdiff_t* argument.

SUSv3 specifies all of the above. SUSv2 specified only the length modifiers**h** (in **hd**, **hi**, **ho**, **hx**, **hX**, **hn**) and **l** (in **ld**, **li**, **lo**, **lx**, **lX**, **ln**, **lc**, **ls**) and **L** (in **Le**, **LE**, **Lf**, **Lg**, **LG**).

The conversion specifier

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.

e, **E** The *double* argument is rounded and converted in the style [-]d**.**ddd**e**±dd where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An **E** conversion uses the letter **E** (rather than **e**) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00.

f, **F** The *double* argument is rounded and converted to decimal notation in the style [-]ddd**.**ddd, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.

(SUSv2 does not know about **F** and says that character string representations for infinity and NaN may be made available. SUSv3 adds a specification for**F**. The C99 standard specifies [-]inf or [-]infinity for infinity, and a string starting with nan for NaN, in the case of **f** conversion, and [-]INF or [-]INFINITY or NAN* in the case of **F** conversion.)

g, **G** The *double* argument is converted in style **f** or **e** (or **F** or **E** for **G** conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style **e** is used if the exponent from its 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 appears only if it is followed by at least one digit.

a, **A** (C99; not in SUSv2, but added in SUSv3) For **a** conversion, the *double* argument is converted to hexadecimal notation (using the letters abcdef) in the style [-]**0x**h**.**hhhh**p**±; for **A** conversion the prefix **0X**, the letters ABCDEF, and the exponent separator **P** is used. There is one hexadecimal digit before the decimal point, and the number of digits after it is equal to the precision. The default precision suffices for an exact representation of the value if an exact representation in base 2 exists and otherwise is sufficiently large to distinguish values of type *double*. The digit before the decimal point is unspecified for nonnormalized numbers, and nonzero but otherwise unspecified for normalized numbers.

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.

C (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for **lc**. Don't use.

S (Not in C99 or C11, but in SUSv2, SUSv3, and SUSv4.) Synonym for **ls**. Don't use.

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

n The number of characters written so far is stored into the integer pointed to by the corresponding argument. That argument shall be an *int ** , or variant whose size matches the (optionally) supplied integer length modifier. No argument is converted. The behavior is undefined if the conversion specification includes any flags, a field width, or a precision.

m (Glibc extension.) Print output of *strerror(errno)*. No argument is required.

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

CONFORMING TO

The **fprintf**(), **printf**(), **sprintf**(), **vprintf**(), **vfprintf**(), and **vsprintf**() functions conform to C89 and C99. The **snprintf**() and **vsnprintf**() functions conform to C99.

Concerning the return value of **snprintf**(), SUSv2 and C99 contradict each other: when **snprintf**() is called with *size*=0 then SUSv2 stipulates an unspecified return value less than 1, while C99 allows *str* to be NULL in this case, and gives the return value (as always) as the number of characters that would have been written in case the output string has been large enough. SUSv3 and later align their specification of **snprintf**() with C99.

glibc 2.1 adds length modifiers **hh**, **j**, **t**, and **z** and conversion characters **a** and **A**.

glibc 2.2 adds the conversion character **F** with C99 semantics, and the flag character **I**.

NOTES

Some programs imprudently rely on code such as the following

sprintf(buf, %s some further text, buf);

to append text to *buf*. However, the standards explicitly note that the results are undefined if source and destination buffers overlap when calling **sprintf**(), **snprintf**(), **vsprintf**(), and **vsnprintf**(). Depending on the version of **gcc(1)** used, and the compiler options employed, calls such as the above will **not** produce the expected results.

The glibc implementation of the functions **snprintf**() and **vsnprintf**() conforms to the C99 standard, that is, behaves as described above, since glibc version 2.1. Until glibc 2.0.6, they would return -1 when the output was truncated.

BUGS

Because **sprintf**() and **vsprintf**() assume an arbitrarily long string, callers must be careful not to overflow the actual space; this is often impossible to assure. Note that the length of the strings produced is locale-dependent and difficult to predict. Use **snprintf**() and **vsnprintf**() instead (or asprintf(3) and vasprintf(3)).

Code such as **printf(***foo***);** often indicates a bug, since *foo* may contain a % character. If*fo o* comes from untrusted user input, it may contain **%n**, causing the **printf**() call to write to memory and creating a security hole.

EXAMPLE

To print *Pi* to five decimal places:

```
#include <math.h>
#include <stdio.h>
fprintf(stdout, pi = %.5fn, 4 * atan(1.0));
```

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

```
#include <stdio.h>
fprintf(stdout, %s, %s %d, %.2d:%.2dn,
weekday, month, day, hour, min);
```

Many countries use the day-month-year order. Hence, an internationalized version must be able to print the arguments in an order specified by the format:

```
#include <stdio.h>
fprintf(stdout, format,
weekday, month, day, hour, min);
```

where *format* depends on locale, and may permute the arguments. With the value:

```
%1$s, %3$d. %2$s, %4$d:%5$.2dn
```

one might obtain Sonntag, 3. Juli, 10:02.

To allocate a sufficiently large string and print into it (code correct for both glibc 2.0 and glibc 2.1):

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

char *
make_message(const char *fmt, ...)
{
int n;
int size = 100; /* Guess we need no more than 100 bytes */
char *p, *np;
va_list ap;

p = malloc(size);
if (p == NULL)
return NULL;

while (1) {

/* Try to print in the allocated space */
```

```
va_start(ap, fmt);
n = vsnprintf(p, size, fmt, ap);
va_end(ap);

/* Check error code */

if (n < 0) {
free(p);
return NULL;
}

/* If that worked, return the string */

if (n < size)
return p;

/* Else try again with more space */

size = n + 1; /* Precisely what is needed */

np = realloc(p, size);
if (np == NULL) {
free(p);
return NULL;
} else {
p = np;
}
}
}
```

If truncation occurs in glibc versions prior to 2.0.6, this is treated as an error instead of being handled gracefully.

SEE ALSO

printf(1), asprintf(3), dprintf(3), scanf(3), setlocale(3), wcrtomb(3), wprintf(3), locale(5)

COLOPHON

This page is part of release 3.74 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at http://www.kernel.org/doc/man-pages/.