NAME

printf, **fprintf**, **sprintf**, **snprintf**, **asprintf**, **dprintf**, **vprintf**, **vfprintf**, **vsprintf**, **vsnprintf**, **vasprintf**, **vdprintf** - formatted output conversion

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

#include <stdio.h>

int
printf(*const char * restrict format, ...*);

int
fprintf(*FILE * restrict stream, const char * restrict format, ...*);

int
sprintf(*char * restrict str, const char * restrict format, ...*);

int
snprintf(*char * restrict str, size_t size, const char * restrict format, ...*);

int
asprintf(*char **ret, const char *format, ...*);

int
dprintf(*int fd, const char * restrict format, ...*);

#include <stdarg.h>

int
vprintf(*const char * restrict format, va_list ap*);

int
vfprintf(*FILE * restrict stream, const char * restrict format, va_list ap*);

int
vsprintf(*char * restrict str, const char * restrict format, va_list ap*);

int

vsnprintf(*char * restrict str, size_t size, const char * restrict format, va_list ap*);

int
vasprintf(*char **ret, const char *format, va_list ap*);

int
vdprintf(*int fd, const char * restrict format, va_list ap*);

DESCRIPTION

The **printf**() family of functions produces output according to a *format* as described below. The **printf**() and **vprintf**() functions write output to stdout, the standard output stream; **fprintf**() and **vfprintf**() write output to the given output *stream*; **dprintf**() and **vdprintf**() write output to the given file descriptor; **sprintf**(), **snprintf**(), **vsprintf**(), and **vsnprintf**() write to the character string *str*; and **asprintf**() and **vasprintf**() dynamically allocate a new string with malloc(3).

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.

The **asprintf**() and **vasprintf**() functions set **ret* to be a pointer to a buffer sufficiently large to hold the formatted string. This pointer should be passed to free(3) to release the allocated storage when it is no longer needed. If sufficient space cannot be allocated, **asprintf**() and **vasprintf**() will return -1 and set *ret* to be a NULL pointer.

The **snprintf**() and **vsnprintf**() functions will write at most *size*-1 of the characters printed into the output string (the *size*'th character then gets the terminating '\0'); if the return value is greater than or equal to the *size* argument, the string was too short and some of the printed characters were discarded. The output is always null-terminated, unless *size* is 0.

The **sprintf**() and **vsprintf**() functions effectively assume a *size* of INT_MAX + 1.

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. The arguments must correspond properly (after type promotion) with the conversion specifier. After the **%**, the following appear in sequence:

- An optional field, consisting of a decimal digit string followed by a **$**, specifying the next argument to access. If this field is not provided, the argument following the last argument accessed will be used. Arguments are numbered starting at **1**. If unaccessed arguments in the format string are

interspersed with ones that are accessed the results will be indeterminate.

- Zero or more of the following flags:

  ‘#’ The value should be converted to an "alternate form". For **c**, **d**, **i**, **n**, **p**, **s**, and **u** conversions, this option has no effect. For **b** and **B** conversions, a non-zero result has the string ‘0b’ (or ‘0B’ for **B** conversions) prepended to it. For **o** conversions, the precision of the number is increased to force the first character of the output string to a zero. For **x** and **X** conversions, a non-zero 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.

  ‘**0**’ (zero) Zero padding. For all conversions except **n**, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (**b**, **B**, **d**, **i**, **o**, **u**, **i**, **x**, and **X**), the **0** flag is ignored.

  ‘**-**’ A negative field width flag; the converted value is to be left adjusted on the field boundary. Except for **n** conversions, 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.

  ‘ ’ (space) A blank should be left before a positive number produced by a signed conversion (**a**, **A**, **d**, **e**, **E**, **f**, **F**, **g**, **G**, or **i**).

  ‘**+**’ A sign must always be placed before a number produced by a signed conversion. A **+** overrides a space if both are used.

  ‘’’ (apostrophe)
  Decimal conversions (**d**, **u**, or **i**) or the integral portion of a floating point conversion (**f** or **F**) should be grouped and separated by thousands using the non-monetary separator returned by localeconv(3).

- An optional decimal digit string 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) to fill out the field width.

- An optional precision, in the form of a period **.** followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for **b**, **B**, **d**, **i**, **o**, **u**, **x**, and **X** conversions, the number of digits to appear after the decimal-point 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** conversions.

- An optional length modifier, that specifies the size of the argument. The following length modifiers are valid for the **b**, **B**, **d**, **i**, **n**, **o**, **u**, **x**, or **X** conversion:

Modifier	**d**, **i**	**b**, **B**, **o**, **u**, **x**, **X**	**n**
hh	*signed char*	*unsigned char*	*signed char **
h	*short*	*unsigned short*	*short **
l (ell)	*long*	*unsigned long*	*long **
ll (ell ell)	*long long*	*unsigned long long*	*long long **
j	*intmax_t*	*uintmax_t*	*intmax_t **
t	*ptrdiff_t*	(see note)	*ptrdiff_t **
w*N*	*intN_t*	*uintN_t*	*intN_t **
wf*N*	*int_fastN_t*	*uint_fastN_t*	*int_fastN_t **
z	(see note)	*size_t*	(see note)
q *(deprecated)*	*quad_t*	*u_quad_t*	*quad_t **

Note: the **t** modifier, when applied to a **b**, **B**, **o**, **u**, **x**, or **X** conversion, indicates that the argument is of an unsigned type equivalent in size to a *ptrdiff_t*. The **z** modifier, when applied to a **d** or **i** conversion, indicates that the argument is of a signed type equivalent in size to a *size_t*. Similarly, when applied to an **n** conversion, it indicates that the argument is a pointer to a signed type equivalent in size to a *size_t*.

The following length modifier is valid for the **a**, **A**, **e**, **E**, **f**, **F**, **g**, or **G** conversion:

Modifier	**a**, **A**, **e**, **E**, **f**, **F**, **g**, **G**
l (ell)	*double* (ignored, same behavior as without it)
L	*long double*

The following length modifier is valid for the **c** or **s** conversion:

Modifier	**c**	**s**
l (ell)	*wint_t*	*wchar_t **

- A character that specifies the type of conversion to be applied.

A field width or precision, or both, may be indicated by an asterisk ‘*’ or an asterisk followed by one or more decimal digits and a ‘$’ instead of a digit string. In this case, an *int* argument supplies the field width or precision. A negative field width is treated as a left adjustment flag followed by a positive field

width; a negative precision is treated as though it were missing. If a single format directive mixes positional (nn$) and non-positional arguments, the results are undefined.

The conversion specifiers and their meanings are:

bBdiouxX The *int* (or appropriate variant) argument is converted to unsigned binary (**b** and **B**), signed decimal (**d** and **i**), 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.

DOU The *long int* argument is converted to signed decimal, unsigned octal, or unsigned decimal, as if the format had been **ld**, **lo**, or **lu** respectively. These conversion characters are deprecated, and will eventually disappear.

eE 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.

For **a**, **A**, **e**, **E**, **f**, **F**, **g**, and **G** conversions, positive and negative infinity are represented as inf and -inf respectively when using the lowercase conversion character, and INF and -INF respectively when using the uppercase conversion character. Similarly, NaN is represented as nan when using the lowercase conversion, and NAN when using the uppercase conversion.

fF 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.

gG 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.

aA The *double* argument is rounded and converted to hexadecimal notation in the style [-]0x*h.hhhp*[+-]*d*, where the number of digits after the hexadecimal-point character is equal to the precision specification. If the precision is missing, it is taken as enough to represent the floating-point number exactly, and no rounding occurs. If the precision is zero, no hexadecimal-point character appears. The **p** is a literal character ‘p’, and the exponent consists of a positive or negative sign followed by a decimal number representing an exponent of 2. The **A** conversion uses the prefix "0X" (rather than "0x"), the letters "ABCDEF" (rather than "abcdef") to represent the hex digits, and the letter ‘P’ (rather than ‘p’) to separate the mantissa and exponent.

Note that there may be multiple valid ways to represent floating-point numbers in this hexadecimal format. For example, 0x1.92p+1, 0x3.24p+0, 0x6.48p-1, and 0xc.9p-2 are all equivalent. FreeBSD 8.0 and later always prints finite non-zero numbers using ‘1’ as the digit before the hexadecimal point. Zeroes are always represented with a mantissa of 0 (preceded by a ‘-’ if appropriate) and an exponent of +0.

C Treated as **c** with the **l** (ell) modifier.

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

If the **l** (ell) modifier is used, the *wint_t* argument shall be converted to a *wchar_t*, and the (potentially multi-byte) sequence representing the single wide character is written, including any shift sequences. If a shift sequence is used, the shift state is also restored to the original state after the character.

S Treated as **s** with the **l** (ell) modifier.

s The *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 NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.

If the **l** (ell) modifier is used, the *wchar_t* * argument is expected to be a pointer to an array of wide characters (pointer to a wide string). For each wide character in the string, the (potentially multi-byte) sequence representing the wide character is written, including any shift sequences. If any shift sequence is used, the shift state is also restored to the original state after the string. Wide characters from the array are written up to (but not including) a terminating wide NUL character; if a precision is specified, no more than the number of bytes specified are written (including shift sequences). Partial characters are never written.

If a precision is given, no null character need be present; if the precision is not specified, or is greater than the number of bytes required to render the multibyte representation of the string, the array must contain a terminating wide NUL character.

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 indicated by the *int* * (or variant) pointer argument. No argument is converted.

m Print the string representation of the error code stored in the errno variable at the beginning of the call, as returned by strerror(3). No argument is taken.

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

The decimal point character is defined in the program's locale (category LC_NUMERIC).

In no case does a non-existent or small field width cause truncation of a numeric field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

RETURN VALUES

These functions return the number of characters printed (not including the trailing '\0' used to end output to strings), except for **snprintf**() and **vsnprintf**(), which return the number of characters that would have been printed if the *size* were unlimited (again, not including the final '\0'). These functions return a negative value if an error occurs.

EXAMPLES

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:%.2d\n",
        weekday, month, day, hour, min);
```

To print pi to five decimal places:

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

To allocate a 128 byte string and print into it:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
char *newfmt(const char *fmt, ...)
{
        char *p;
        va_list ap;
        if ((p = malloc(128)) == NULL)
                return (NULL);
        va_start(ap, fmt);
        (void) vsnprintf(p, 128, fmt, ap);
        va_end(ap);
        return (p);
}
```

COMPATIBILITY

The conversion formats **%D**, **%O**, and **%U** are not standard and are provided only for backward compatibility. The conversion format **%m** is also not standard and provides the popular extension from the GNU C library.

The effect of padding the **%p** format with zeros (either by the **0** flag or by specifying a precision), and the benign effect (i.e., none) of the **#** flag on **%n** and **%p** conversions, as well as other nonsensical combinations such as **%Ld**, are not standard; such combinations should be avoided.

ERRORS

In addition to the errors documented for the write(2) system call, the **printf**() family of functions may fail if:

[EILSEQ] An invalid wide character code was encountered.

[ENOMEM] Insufficient storage space is available.

[EOVERFLOW] The *size* argument exceeds INT_MAX + 1, or the return value would be too large to be represented by an *int*.

SEE ALSO

printf(1), errno(2), fmtcheck(3), scanf(3), setlocale(3), strerror(3), wprintf(3)

STANDARDS

Subject to the caveats noted in the *BUGS* section below, the **fprintf**(), **printf**(), **sprintf**(), **vprintf**(), **vfprintf**(), and **vsprintf**() functions conform to ANSI X3.159-1989 ("ANSI C89") and ISO/IEC 9899:1999 ("ISO C99"). With the same reservation, the **snprintf**() and **vsnprintf**() functions conform to ISO/IEC 9899:1999 ("ISO C99"), while **dprintf**() and **vdprintf**() conform to IEEE Std 1003.1-2008 ("POSIX.1").

HISTORY

The functions **asprintf**() and **vasprintf**() first appeared in the GNU C library. These were implemented by Peter Wemm <*peter@FreeBSD.org*> in FreeBSD 2.2, but were later replaced with a different implementation from OpenBSD 2.3 by Todd C. Miller <*Todd.Miller@courtesan.com*>. The **dprintf**() and **vdprintf**() functions were added in FreeBSD 8.0. The **%m** format extension first appeared in the GNU C library, and was implemented in FreeBSD 12.0.

BUGS

The **printf** family of functions do not correctly handle multibyte characters in the *format* argument.

SECURITY CONSIDERATIONS

The **sprintf**() and **vsprintf**() functions are easily misused in a manner which enables malicious users to arbitrarily change a running program's functionality through a buffer overflow attack. Because **sprintf**() and **vsprintf**() assume an infinitely long string, callers must be careful not to overflow the actual space; this is often hard to assure. For safety, programmers should use the **snprintf**() interface instead. For example:

```
void
foo(const char *arbitrary_string, const char *and_another)
{
        char onstack[8];

#ifdef BAD
        /*
         * This first sprintf is bad behavior.  Do not use sprintf!
         */
        sprintf(onstack, "%s, %s", arbitrary_string, and_another);
#else
        /*
         * The following two lines demonstrate better use of
         * snprintf().
         */
        snprintf(onstack, sizeof(onstack), "%s, %s", arbitrary_string,
```

```
            and_another);
#endif
}
```

The **printf**() and **sprintf**() family of functions are also easily misused in a manner allowing malicious users to arbitrarily change a running program’s functionality by either causing the program to print potentially sensitive data "left on the stack", or causing it to generate a memory fault or bus error by dereferencing an invalid pointer.

%n can be used to write arbitrary data to potentially carefully-selected addresses. Programmers are therefore strongly advised to never pass untrusted strings as the *format* argument, as an attacker can put format specifiers in the string to mangle your stack, leading to a possible security hole. This holds true even if the string was built using a function like **snprintf**(), as the resulting string may still contain user-supplied conversion specifiers for later interpolation by **printf**().

Always use the proper secure idiom:

```
snprintf(buffer, sizeof(buffer), "%s", string);
```