# vprintf, vfprintf, vsprintf, vsnprintf, vprintf_s, vfprintf_s, vsprintf_s, vsnprintf_s

Defined in header <stdio.h>

| | | |
|---|---|---|
| `int vprintf( const char *format, va_list vlist );` | (1) | (until C99) |
| `int vprintf( const char *restrict format, va_list vlist );` | | (since C99) |
| `int vfprintf( FILE *stream, const char *format, va_list vlist );` | (2) | (until C99) |
| `int vfprintf( FILE *restrict stream, const char *restrict format, va_list vlist );` | | (since C99) |
| `int vsprintf( char *buffer, const char *format, va_list vlist );` | (3) | (until C99) |
| `int vsprintf( char *restrict buffer, const char *restrict format, va_list vlist );` | | (since C99) |
| `int vsnprintf( char *restrict buffer, size_t bufsz, const char *restrict format, va_list vlist );` | (4) | (since C99) |
| `int vprintf_s( const char *restrict format, va_list arg);` | (5) | (since C11) |
| `int vfprintf_s( FILE *restrict stream, const char *restrict format, va_list arg);` | (6) | (since C11) |
| `int vsprintf_s( char *restrict buffer, rsize_t bufsz, const char *restrict format, va_list arg);` | (7) | (since C11) |
| `int vsnprintf_s(char *restrict buffer, rsize_t bufsz, const char *restrict format, va_list arg);` | (8) | (since C11) |

Loads the data from the locations, defined by vlist, converts them to character string equivalents and writes the results to a variety of sinks.

1) Writes the results to stdout.
2) Writes the results to a file stream stream.
3) Writes the results to a character string buffer.
4) Writes the results to a character string buffer. At most bufsz - 1 characters are written. The resulting character string will be terminated with a null character, unless bufsz is zero. If bufsz is zero, nothing is written and buffer may be a null pointer, however the return value (number of bytes that would be written not including the null terminator) is still calculated and returned.

5-8) Same as (1-4), except that the following errors are detected at runtime and call the currently installed constraint handler function:

- the conversion specifier %n is present in format
- any of the arguments corresponding to %s is a null pointer
- format or buffer is a null pointer
- bufsz is zero or greater than RSIZE_MAX
- encoding errors occur in any of string and character conversion specifiers
- (for vsprintf_s only), the string to be stored in buffer (including the trailing null)) would be exceed bufsz

As with all bounds-checked functions, vprintf_s , vfprintf_s, vsprintf_s, and vsnprintf_s are only guaranteed to be available if `__STDC_LIB_EXT1__` is defined by the implementation and if the user defines `__STDC_WANT_LIB_EXT1__` to the integer constant 1 before including stdio.h.

## Parameters

**stream** - output file stream to write to
**buffer** - pointer to a character string to write to
**bufsz** - up to bufsz - 1 characters may be written, plus the null terminator
**format** - pointer to a null-terminated character string specifying how to interpret the data
**vlist** - variable argument list containing the data to print.

The **format** string consists of ordinary multibyte characters (except %), which are copied unchanged into the output stream, and conversion specifications. Each conversion specification has the following format:

- introductory % character
- (optional) one or more flags that modify the behavior of the conversion:
  - **-**: the result of the conversion is left-justified within the field (by default it is right-justified)

- **+**: the sign of signed conversions is always prepended to the result of the conversion (by default the result is preceded by minus only when it is negative)
- *space*: if the result of a signed conversion does not start with a sign character, or is empty, space is prepended to the result. It is ignored if + flag is present.
- **#** : *alternative form* of the conversion is performed. See the table below for exact effects otherwise the behavior is undefined.
- **0** : for integer and floating point number conversions, leading zeros are used to pad the field instead of *space* characters. For integer numbers it is ignored if the precision is explicitly specified. For other conversions using this flag results in undefined behavior. It is ignored if - flag is present.

- (optional) integer value or * that specifies minimum field width. The result is padded with *space* characters (by default), if required, on the left when right-justified, or on the right if left-justified. In the case when * is used, the width is specified by an additional argument of type `int`, which appears before the argument to be converted and the argument supplying precision if one is supplied. If the value of the argument is negative, it results with the - flag specified and positive field width. (Note: This is the minimum width: The value is never truncated.)

- (optional) . followed by integer number or *, or neither that specifies *precision* of the conversion. In the case when * is used, the *precision* is specified by an additional argument of type `int`, which appears before the argument to be converted, but after the argument supplying minimum field width if one is supplied. If the value of this argument is negative, it is ignored. If neither a number nor * is used, the precision is taken as zero. See the table below for exact effects of *precision*.

- (optional) *length modifier* that specifies the size of the argument (in combination with the conversion format specifier, it specifies the type of the corresponding argument)

- conversion format specifier

The following format specifiers are available:

| Conversion Specifier | Explanation | Expected Argument Type | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Length Modifier → | hh (C99) | h | (none) | l | ll (C99) | j (C99) | z (C99) | t (C99) | L |
| % | writes literal %. The full conversion specification must be %% . | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A | N/A |
| c | writes a **single character**.<br>The argument is first converted to `unsigned char`. If the **l** modifier is used, the argument is first converted to a character string as if by **%ls** with a `wchar_t[2]` argument. | N/A | N/A | `int` | `wint_t` | N/A | N/A | N/A | N/A | N/A |
| s | writes a **character string**<br>The argument must be a pointer to the initial element of an array of characters. *Precision* specifies the maximum number of bytes to be written. If *Precision* is not specified, writes every byte up to and not including the first null terminator. If the **l** specifier is used, the argument must be a pointer to the initial element of an array of `wchar_t`, which is converted to char array as if by a call to wcrtomb with zero-initialized conversion state. | N/A | N/A | `char*` | `wchar_t*` | N/A | N/A | N/A | N/A | N/A |
| d<br>i | converts a **signed integer** into decimal representation *[-]dddd*.<br>*Precision* specifies the minimum number of digits to appear. The default precision is `1`.<br>If both the converted value and the precision are `0` the conversion results in no characters. | `signed char` | `short` | `int` | `long` | `long long` | `intmax_t` | signed `size_t` | `ptrdiff_t` | N/A |
| o | converts an **unsigned integer** into octal representation *oooo*.<br>*Precision* specifies the minimum number of digits to appear. The default precision is `1`. If both the converted value and the precision are `0` the conversion results in no characters. In the *alternative implementation* precision is increased if necessary, to write one leading zero. In that case if both the converted value and the precision are `0`, single `0` is written. | `unsigned char` | `unsigned short` | `unsigned int` | `unsigned long` | `unsigned long long` | `uintmax_t` | `size_t` | unsigned version of `ptrdiff_t` | N/A |
| x<br>X | converts an **unsigned integer** into hexadecimal representation *hhhh*.<br>For the **x** conversion letters abcdef are used.<br>For the **X** conversion letters ABCDEF are used.<br>*Precision* specifies the minimum number of digits to appear. The default precision is `1`. If both the converted value and the precision are `0` the conversion results in no characters. In the *alternative implementation* **0x** or **0X** is prefixed to results if the converted value is nonzero. | | | | | | | | | N/A |
| u | converts an **unsigned integer** into decimal representation *dddd*.<br>*Precision* specifies the minimum number of digits to appear. The default precision is `1`. If both the converted value and the precision are `0` the conversion results in no characters. | | | | | | | | | N/A |
| f<br>F | converts **floating-point number** to the decimal notation in the style *[-]ddd.ddd*.<br>*Precision* specifies the exact number of digits to appear after the decimal point character. The default precision is `6`. In the *alternative implementation* decimal point character is written even if no digits follow it. For infinity and not-a-number conversion style see notes. | N/A | N/A | `double` | `double` (C99) | N/A | N/A | N/A | N/A | `long double` |
| e<br>E | converts **floating-point number** to the decimal exponent notation.<br>For the **e** conversion style *[-]d.ddd***e***±dd* is used.<br>For the **E** conversion style *[-]d.ddd***E***±dd* is used.<br>The exponent contains at least two digits, more digits are used only if necessary. If the value is `0`, the exponent is also `0`. *Precision* specifies the exact number of digits to appear after the decimal point character. The default precision is `6`. In the *alternative implementation* decimal point character is written even if no digits follow it. For infinity and not-a-number conversion style see notes. | N/A | N/A | | | N/A | N/A | N/A | N/A | |
| a<br>A<br>(C99) | converts **floating-point number** to the hexadecimal exponent notation.<br>For the **a** conversion style *[-]***0x***h.hhh***p***±d* is used.<br>For the **A** conversion style *[-]***0X***h.hhh***P***±d* is used.<br>The first hexadecimal digit is not 0 if the argument is a normalized | N/A | N/A | | | N/A | N/A | N/A | N/A | |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | floating point value. If the value is `0`, the exponent is also `0`. *Precision* specifies the exact number of digits to appear after the hexadecimal point character. The default precision is sufficient for exact representation of the value. In the *alternative implementation* decimal point character is written even if no digits follow it. For infinity and not-a-number conversion style see notes. | | | | | | | | | |
| **g**<br>**G** | converts **floating-point number** to decimal or decimal exponent notation depending on the value and the *precision*.<br>For the **g** conversion style conversion with style **e** or **f** will be performed. For the **G** conversion style conversion with style **E** or **F** will be performed. Let P equal the precision if nonzero, `6` if the precision is not specified, or `1` if the precision is `0`. Then, if a conversion with style E would have an exponent of X:<br>▪ if *P > X ≥ −4*, the conversion is with style **f** or **F** and precision *P − 1 − X*.<br>▪ otherwise, the conversion is with style **e** or **E** and precision *P − 1*.<br>Unless *alternative representation* is requested the trailing zeros are removed, also the decimal point character is removed if no fractional part is left. For infinity and not-a-number conversion style see notes. | N/A | N/A | | | N/A | N/A | N/A | N/A | |
| **n** | returns the **number of characters written** so far by this call to the function.<br>The result is *written* to the value pointed to by the argument. The specification may not contain any *flag*, *field width*, or *precision*. | `signed char*` | `short*` | `int*` | `long*` | `long long*` | `intmax_t*` | `signed size_t*` | `ptrdiff_t*` | N/A |
| **p** | writes an implementation defined character sequence defining a **pointer**. | N/A | N/A | `void*` | N/A | N/A | N/A | N/A | N/A | N/A |

The floating point conversion functions convert infinity to `inf` or `infinity`. Which one is used is implementation defined.

Not-a-number is converted to `nan` or `nan(`*`char_sequence`*`)`. Which one is used is implementation defined.

The conversions **F**, **E**, **G**, **A** output `INF`, `INFINITY`, `NAN` instead.

Even though `%c` expects `int` argument, it is safe to pass a `char` because of the integer promotion that takes place when a variadic function is called.

The correct conversion specifications for the fixed-width character types (`int8_t`, etc) are defined in the header `<inttypes.h>` (although `PRIdMAX`, `PRIuMAX`, etc is synonymous with `%jd`, `%ju`, etc).

The memory-writing conversion specifier `%n` is a common target of security exploits where format strings depend on user input and is not supported by the bounds-checked `printf_s` family of functions.

There is a sequence point after the action of each conversion specifier; this permits storing multiple `%n` results in the same variable or, as an edge case, printing a string modified by an earlier `%n` within the same call.

If a conversion specification is invalid, the behavior is undefined.

## Return value

1-3) The number of characters written if successful or negative value if an error occurred.

4) The number of characters written if successful or negative value if an error occurred. If the resulting string gets truncated due to `buf_size` limit, function returns the total number of characters (not including the terminating null-byte) which would have been written, if the limit was not imposed.

5,6) number of characters transmitted to the output stream or negative value if an output error, a runtime constrants violation error, or an encoding error occurred.

7) number of characters written to `buffer`, not counting the null character (which is always written as long as `buffer` is not a null pointer and `bufsz` is not zero and not greater than `RSIZE_MAX`), or zero on runtime constraint violations, and negative value on encoding errors

8) number of characters not including the terminating null character (which is always written as long as `buffer` is not a null pointer and `bufsz` is not zero and not greater than `RSIZE_MAX`), which would have been written to `buffer` if `bufsz` was ignored, or a negative value if a runtime constraints violation or an encoding error occurred

## Notes

All these functions invoke `va_arg` at least once, the value of `arg` is indeterminate after the return. These functions do not invoke `va_end`, and it must be done by the caller.

`vsnprintf_s`, unlike `vsprintf_s`, will truncate the result to fit within the array pointed to by `buffer`.

### Example

Run this code

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

void debug_log(const char *fmt, ...)
{
    struct timespec ts;
    timespec_get(&ts, TIME_UTC);
    char time_buf[100];
    size_t rc = strftime(time_buf, sizeof time_buf, "%D %T", gmtime(&ts.tv_sec));
    snprintf(time_buf + rc, sizeof time_buf - rc, ".%06ld UTC", ts.tv_nsec / 1000);

    va_list args1;
    va_start(args1, fmt);
    va_list args2;
    va_copy(args2, args1);
    char buf[1+vsnprintf(NULL, 0, fmt, args1)];
    va_end(args1);
    vsnprintf(buf, sizeof buf, fmt, args2);
    va_end(args2);

    printf("%s [debug]: %s\n", time_buf, buf);
}

int main(void)
{
    debug_log("Logging, %d, %d, %d", 1, 2, 3);
}
```

Possible output:

```
02/20/15 21:58:09.072683 UTC [debug]: Logging, 1, 2, 3
```

### References

- C11 standard (ISO/IEC 9899:2011):
  - 7.21.6.8 The vfprintf function (p: 326-327)
  - 7.21.6.10 The vprintf function (p: 328)
  - 7.21.6.12 The vsnprintf function (p: 329)
  - 7.21.6.13 The vsprintf function (p: 329)
  - K.3.5.3.8 The vfprintf_s function (p: 597)
  - K.3.5.3.10 The vprintf_s function (p: 598-599)
  - K.3.5.3.12 The vsnprintf_s function (p: 600)
  - K.3.5.3.13 The vsprintf_s function (p: 601)
- C99 standard (ISO/IEC 9899:1999):
  - 7.19.6.8 The vfprintf function (p: 292)
  - 7.19.6.10 The vprintf function (p: 293)
  - 7.19.6.12 The vsnprintf function (p: 294)
  - 7.19.6.13 The vsprintf function (p: 295)
- C89/C90 standard (ISO/IEC 9899:1990):
  - 4.9.6.7 The vfprintf function
  - 4.9.6.8 The vprintf function
  - 4.9.6.9 The vsprintf function

## See also

| | |
|---|---|
| **vwprintf** (C95)<br>**vfwprintf** (C95)<br>**vswprintf** (C95)<br>**vwprintf_s** (C11)<br>**vfwprintf_s** (C11)<br>**vswprintf_s** (C11)<br>**vsnwprintf_s** (C11) | prints formatted wide character output to stdout, a file stream or a buffer using variable argument list<br>(function) |
| **printf**<br>**fprintf**<br>**sprintf**<br>**snprintf** (C99)<br>**printf_s** (C11)<br>**fprintf_s** (C11)<br>**sprintf_s** (C11)<br>**snprintf_s** (C11) | prints formatted output to stdout, a file stream or a buffer<br>(function) |
| **vscanf** (C99)<br>**vfscanf** (C99)<br>**vsscanf** (C99)<br>**vscanf_s** (C11)<br>**vfscanf_s** (C11)<br>**vsscanf_s** (C11) | reads formatted input from stdin, a file stream or a buffer using variable argument list<br>(function) |
| **C++ documentation** for **vprintf, vfprintf, vsprintf, vsnprintf** | |