Synopsis

```
#include <stdio.h>

int fprintf(
  FILE * restrict stream, const char * restrict format, ... );
int printf(
  const char * restrict format, ...);
int sprintf(
  char * restrict s,
  const char * restrict format, ...);
int snprintf(
  char * restrict s, size_t n,
  const char * restrict format, ...);     // C99
```

```
#include <stdio.h>
#include <wchar.h>

int fwprintf(
  FILE * restrict stream,
  const wchar_t * restrict format, ... );
int wprintf(
  const wchar_t * restrict format, ...);
int swprintf(
  wchar_t *s, size_t n, const wchar_t *format, ...);
```

The function **fprintf** performs output formatting, sending the output to the stream specified as the first argument. The second argument is a format control string. Additional arguments may be required depending on the contents of the control string. A series of output characters is generated as directed by the control string; these characters are sent to the specified stream.

The **printf** function is related to **fprintf**, but sends the characters to the standard output stream **stdout**.

The **sprintf** function causes the output characters to be stored into the string buffer **s**. A final null character is output to **s** after all characters specified by the control string have been output. It is the programmer's responsibility to ensure that the **sprintf** destination string area is large enough to contain the output generated by the formatting operation. However, the swprintf function, unlike sprintf, includes a count of the maximum number of wide characters (including the terminating null character) to be written to the output string **s**. In C99, **sprintf** was added to provide the count for the nonwide function.

The value returned by these functions is **EOF** if an error occurred during the output operation; otherwise the result is some value other than **EOF**. In Standard C and most current implementations, the functions return the number of characters sent to the output stream if no error occurs. In the case of **sprintf,** the count does not include the terminating null character. (Standard C allows these functions to return any negative value if an error occurs.)

C89 (Amendment 1) specifies three wide-character versions of these functions; **fwprintf, wprintf,** and **swprintf**. The output of these functions is conceptually a wide string, and they convert their additional arguments to wide strings under control of the conversion operators. We denote these functions as the **wprintf** family of functions, or just **wprintf** functions, to distinguish them from the original byte-oriented **printf** functions. Under Amendment 1 also, the **l** size specifier may be applied to the **c** and **s** conversion operators in both the **printf** and **wprintf** functions.

C99 introduces the **a** and **A** conversion operators for hexadecimal floating-point conversions and the **hh, 11, j, z,** and **t** length modifiers.

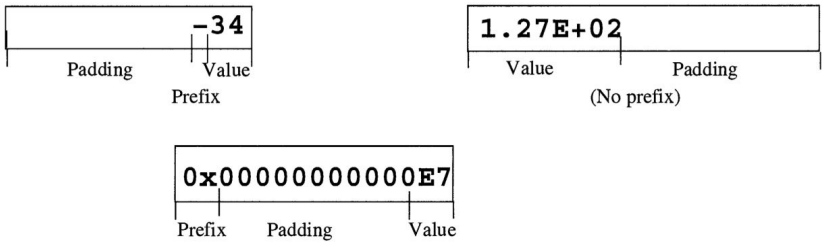
References EOF 15.1; hexadecimal floating-point format 2.7.2; **scanf** 15.8; **stdout** 15.4; wide characters 2.1.4

15.11.1 Output Format

The control string is simply text to be copied verbatim, except that the string may contain conversion specifications. In Standard C, the control string is an (uninterpreted) multibyte character sequence beginning and ending in its initial shift state. In the **wprintf** functions, it is a wide-character string.

A conversion specification may call for the processing of some number of additional arguments, resulting in a formatted conversion operation that generates output characters not explicitly contained in the control string. There should be exactly the right number of arguments, each of exactly the right type, to satisfy the conversion specifications in the control string. Extra arguments are ignored, but the result from having too few arguments is unpredictable. If any conversion specification is malformed, then the effects are unpredictable. The conversion specifications for output are similar to those used for input by **fscanf** and related functions; the differences are discussed in Section 15.8.2. There is a sequence point just after the actions called for by each conversion specification.

The sequence of characters or wide characters output for a conversion specification may be conceptually divided into three elements: the converted value proper, which reflects the value of the converted argument; the prefix, which, if present, is typically +, -, or a space; and the padding, which is a sequence of spaces or zero digits added if necessary to increase the width of the output sequence to a specified minimum. The prefix always precedes the converted value. Depending on the conversion specification, the padding may precede the prefix, separate the prefix from the converted value, or follow the converted value. Examples are shown in the following figure; the enclosing boxes show the extent of the output governed by the conversion specification.



15.11.2 Conversion Specifications

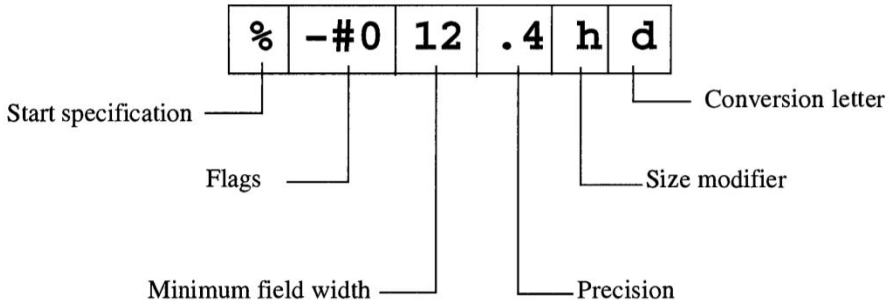
In what follows, the terms characters, letters, and so on are to be understood as normal characters or letters (bytes) in the case of the **printf** functions and wide characters or letters in the case of the **wprintf** functions. For example, in **wprintf**, conversion specifications begin with the wide-character percent sign, %.

A conversion specification begins with a percent sign character, %, and has the following elements in order:

1. Zero or more flag characters (-, +, 0, #, or space), which modify the meaning of the conversion operation.
2. An optional minimum field width expressed as a decimal integer constant.
3. An optional precision specification expressed as a period optionally followed by a decimal integer.
4. An optional size specification expressed as one of the letters 11,1, L,h, hh, j, z, or t.
5. The conversion operation, a single character from the set a, A, c, d, e, E, f, g, G, i,n, 0, p, s, u, x, X, and %.

The size specification letters L and h, and the conversion operations i, p, and n, were introduced in C89. The size specification letters ll, hh, j, z, and t, and the conversion operations **a** and **A**, were introduced in C99.

The conversion letter terminates the specification. The conversion specification **%-#012.4hd** is shown next broken into its constituent elements:



15.11.3 Conversion Flags

The optional flag characters modify the meaning of the main conversion operation:

-	Left-justify the value within the field width.
0	Use 0 for the pad character rather than space.
+	Always produce a sign, either + or -.
space	Always produce either the sign - or a space.
#	Use a variant of the main conversion operation.

The effects of the flag characters are described in more detail now.

The - flag If a minus-sign flag is present, then the converted value will be left-justified within the field—that is, any padding will be placed to the right of the converted value. If no minus sign is present, the converted value will be right-justified within the field. This flag is relevant only when an explicit minimum field width is specified and the converted value is smaller than that minimum width; otherwise the value will fill the field without padding.

The 0 flag If a 0 (zero) flag is present, then 0 will be used as the pad character if padding is to be placed to the left of the converted value. The 0 flag is relevant only when an explicit minimum field width is specified and the converted value is smaller than that minimum width. In integer conversions, this flag is superseded by the precision specification.

If no zero-digit flag is present, then a space will be used as the pad character. Space is always used as the pad character if padding is to be placed to the right of the converted value even if the - flag character is present.

The + flag If a+ flag is present, then the result of a signed conversion will always begin with a sign—that is, an explicit + will precede a converted positive value. (Negative values are always preceded by — regardless of whether a plus-sign flag is specified.) This flag is only relevant for the conversion operations a, A, d, e, E, f, g, G, andi.

The space flag If a space flag is present and the first character in the converted value resulting from a signed conversion is not a sign (+ or —), then a space will be added before the converted value. The adding of this space on the left is independent of any padding that may be placed to the left or right under control of the - flag character. If both the space and + flags appear in a single conversion specification, the space flag is ignored because the + flag ensures that the converted value will always begin with a sign. This flag is relevant only for the conversion operations a, A, d, e, E, f, g, G, and i.

The # flag If a # flag is present, then an alternate form of the main conversion operation is used. This flag is relevant only for the conversion operations a, A, e, E, f, g, G, i, o, x, and X. The modifications implied by the # flag are described in conjunction with the relevant conversion operations.

15.11.4 Minimum Field Width.

An optional minimum field width, expressed as a decimal integer constant, may be specified. The constant must be a nonempty sequence of decimal digits that does not begin with a zero digit (which would be taken to be the 0 flag). If the converted value (including prefix) results in fewer characters than the specified field width, then pad characters are used to pad the value to the specified width. If the converted value results in more characters than the specified field width, then the field is expanded to accommodate it without padding.

The field width may also be specified by an asterisk, *, in which case an argument of type int is consumed and specifies the minimum field width. The result of specifying a negative width is unpredictable.

Example

The following two calls to **printf** result in the same output:

```
int width=5, value;
…
printf("%5d", value);
printf("%*d", width, value);
```

15.11.5 Precision

An optional precision specification may be specified and expressed as a period followed by an optional decimal integer. The precision specification is used to control:

1. the minimum number of digits to be printed for **d, i, o, u, x,** and **X** conversions
2. the number of digits to the right of the decimal point in **e, E,** and **f** conversions
3. the number of significant digits in the g and G conversions
4. the maximum number of characters to be written from a string in the s conversion

If the period appears but the integer is missing, then the integer is assumed to be zero, which usually has a different effect than omitting the entire precision specification.

The precision may also be specified by an asterisk following the period, in which case an argument of type **int** is consumed and specifies the precision. If both the field width and precision are specified with asterisks, then the field width argument precedes the precision argument.

15.11.6 Size Specification

An optional size modifier, one of the letter sequences **ll** (ell-ell)**, l** (ell)**, L, h, hh, j, z,** or **t,** may precede some conversion operations.

The letter **l**, in conjunction with the conversion operations **d, i, o, u, x,** and **X**, indicates that the conversion argument has type long or unsigned long. In conjunction with the n conversion, it specifies that the argument has type long *. In C89, the modifier 1 may also be used with c, in which case the argument is of typewint_t, or with s, in which case it specifies that the argument has type wchar_t *. The modifier 1 has no effect when used with a, A, e, E, f, F, g, and G; compare this with the L modifier and be careful which you use.

The modifier 11, in conjunction with the conversion operations **d, i, o, u, x,** and **X**, indicates that the conversion argument has type **long long int** or **unsigned long long int**. In conjunction with the n conversion, the 11 modifier specifies that the argument has type **long long int *.** The 11 size modifier was introduced in C99.

The letter **h**, in conjunction with the conversion operations **d, i, o, u, x,** and **X**, indicates that the conversion argument has type **short** or **unsigned short**. That is, although the argument would have been converted to **int** or **unsigned** by the argument promotions, it should be converted to **short** or **unsigned short** before conversion. In conjunction with the n conversion, the h modifier specifies that the argument has type **short *.** The h size modifier was introduced in C89.

The modifier hh, in conjunction with the conversion operations **d, i, o, u, x,** and **X**, indicates that the conversion argument has type char or unsigned char. That is, although the argument would have been converted to int or unsigned by the argument promotions, it should be converted to char or unsigned char before conversion. In conjunction with the n conversion, the hh modifier specifies that the argument has type signed char *. The hh size modifier is available in C99.

The letter L, in conjunction with the conversion operations a, A, e, E, f, F, g, and G, indicates that the argument has type long double. The L size modifier was introduced in C89. Be careful to use L and not l for long double since **l** has no effect on these operations.

The modifier j, in conjunction with the conversion operations **d, i, o, u, x,** and **X**, indicates that the conversion argument has type **intmax_t** or **uintmax_t.** In conjunction with the n conversion, the j modifier specifies that the argument has type **intmax_t *.** The j size modifier was introduced in C99.

The modifier z, in conjunction with the conversion operations **d, i, o, u, x,** and **X**, indicates that the conversion argument has type **size_t**. In conjunction with the n conversion, the z modifier specifies that the argument has **typesize_t *.** The z size modifier was introduced in C99.

The modifier t, in conjunction with the conversion operations **d, i, o, u, x,** and **X**, indicates that the conversion argument has type ptrdiff_t. In conjunction with the n conversion, the t modifier specifies that the argument has type **ptrdiff_t *.** Thet size modifier was introduced in C99.

15.11.7 Conversion Operations

The conversion operation is expressed as a single character: a, A, c, d, e, E, f,g, G, i, n, o, p, s, u, x, X, or %. The specified conversion determines the permitted flag and size characters, the expected argument type, and how the output looks. Table 15-6 summarizes the conversion operations. Each operation is then discussed individually.

The d andiconversions. Signed decimal conversion is performed. The argument should be of type int if no size modifier is used, type **short** if **h** is used, or type **Long** if **l** is used. The i operator is present in Standard C for compatibility with **fscanf**; it is recognized on output for uniformity, where it is identical to the **d** operator.

The converted value consists of a sequence of decimal digits that represents the absolute value of the argument. This sequence is as short as possible, but not shorter than the specified precision. The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros. If the precision is 1 (the default), then the converted value will not have a leading 0 unless the argument is 0, in which case a single 0 is output. If the precision is 0 and the argument is 0, then the converted value is empty (the null string).

The prefix is computed as follows. If the argument is negative, the prefix is a minus sign. If the argument is non-negative and the + flag is specified, then the prefix is a plus sign. If the argument is non-negative, the space flag is specified, and the + flag is not specified, then the prefix is a space. Otherwise, the prefix is empty. The **#** flag is not relevant to the **d** and **i** conversions. Table 15—7 shows examples of the **d** conversion.

Table 15–6 Output conversion specifications

Conversion	Defined flags - + # 0 *space*	Size modifier	Argument type	Default precision[a]	Output
d, i[b]	- + 0 *space*	*none*	**int**	1	dd...d
		h	**short**		-dd...d
		l	**long**		+dd...d
u	- + 0 *space*	*none*	**unsigned int**	1	dd...d
		h	**unsigned short**		
		l	**unsigned long**		
o	- + # 0 *space*	*none*	**unsigned int**	1	oo...o
		h	**unsigned short**		**0**oo...o
		l	**unsigned long**		
x, X	- + # 0 *space*	*none*	**unsigned int**	1	hh...h
		h	**unsigned short**		**0x**hh...h
		l	**unsigned long**		**0X**hh...h
f	- + # 0 *space*	*none*	**double**	6	d...d.d...d
		l	**double**		-d...d.d...d
		L	**long double**		+d...d.d...d
e, E	- + # 0 *space*	*none*	**double**	6	d.d...d**e**+dd
		l	**double**		-d.d...d**E**-dd
		L	**long double**		
g, G	- + # 0 *space*	*none*	**double**	6	*like* **e**, **E**,
		l	**double**		*or* **f**
		L	**long double**		
a, A[c]	- + # 0 *space*	*none*	**double**	6	**0x**h.h...h**p+**dd
		l	**double**		**-0X**h.h...h**P-**
		L	**long double**		dd
c	-	*none*	**int**	1	c
		l[d]	**wint_t**		
s	-	*none*	**char ***	×	cc...c
		l[c]	**wchar_t ***		
p[b]	*impl. defined*	*none*	**void ***	1	*impl. defined*
n[b]		*none*	**int ***	n/a	*none*
		h	**short ***		
		l	**long ***		
%		*none*	*none*	n/a	**%**

a - Default precision, if none is specified.
b - Introduced in C89. The conversions **i** and **d** are equivalent on output.
c - Introduced in C99
d - Introduced in C99 (Amendment 1).

The u conversion. Unsigned decimal conversion is performed. The argument should be of type unsigned if no size modifier is used, type **unsigned short** if **h** is used, or type **unsigned long** if **l** is used.

The converted value consists of a sequence of decimal digits that represents the value of the argument. This sequence is as short as possible, but not shorter than the specified precision.

Table 15–7 Examples of the **d** conversion

Sample format	Sample output Value = 45	Sample output Value = –45
%12d	45	-45
%012d	000000000045	-00000000045
% 012d	00000000045	-00000000045
%+12d	+45	-45
%+012d	+00000000045	-00000000045
%-12d	45	-45
%- 12d	45	-45
%-+12d	+45	-45
%12.4d	0045	-0045
%-12.4d	0045	-0045

The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros. If the precision is 1 (the default), then the converted value will not have a leading 0 unless the argument is 0, in which case a single 0 is output. If the precision and argument are 0, then the converted value is empty (the null string). The prefix is always empty. The +, space, and # flags are not relevant to the u conversion operation. Table 15-8 shows examples of the u conversion.

Table 15–8 Examples of the **u** conversion

Sample format	Sample output Value = 45	Sample output Value = –45
%14u	45	4294967251
%014u	00000000000045	00004294967251
%#14u	45	4294967251
%#014u	00000000000045	00004294967251
%-14u	45	4294967251
%-#14u	45	4294967251
%14.4u	0045	4294967251
%-14.4u	0045	4294967251

The o conversion. Unsigned octal conversion is performed. The argument should be of type unsigned if no size modifier is used, type unsigned short if h is used, or type unsigned long if l is used.

The converted value consists of a sequence of octal digits that represents the value of the argument. This sequence is as short as possible, but not shorter than the specified precision. The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros. If the precision is 1 (the default), then the converted value will not have a leading 0 unless the argument is 0, in which case a single 0 is output. If the precision is 0 and the argument is 0, then the converted value is empty (the null string).

If the # flag is present, then the prefix is 0. If the # flag is not present, then the prefix is empty. The + and space flags are not relevant to the o conversion operation. Table 15—9 shows examples of the o conversion.

Table 15–9 Examples of the **o** conversion

Sample format	Sample output Value = 45	Sample output Value = –45
%14o	55	37777777723
%014o	00000000000055	00037777777723
%#14o	055	037777777723
%#014o	00000000000055	00037777777723
%-14o	55	377777777723
%-#14o	055	037777777723
%14.4o	0055	37777777723
%-#14.4o	00055	037777777723

The x and X conversions. Unsigned hexadecimal conversion is performed. The argument should be of type unsigned if no size modifier is used, type **unsigned short** if **h** is used, or type **unsigned long** if **l** is used.

The converted value consists of a sequence of hexadecimal digits that represents the value of the argument. This sequence is as short as possible, but not shorter than the specified precision. The x operation uses **0123456789abcdef** as digits, whereas the X operation uses **0123456789ABCDEF**. The converted value will have leading zeros if necessary to satisfy the precision specification; these leading zeros are independent of any padding, which might also introduce leading zeros. If the precision is 1, then the converted value will not have a leading 0 unless the argument is 0, in which case a single 0 is output. If the precision is 0 and the argument is 0, then the converted value is empty (the null string). If no precision is specified, then a precision of 1 is assumed.

If the # flag is present, then the prefix is 0x (for the x operation) or 0X (for the X operation). If the # flag is not present, then the prefix is empty. The + and space flags are not relevant. Table 15-10 shows examples of x and X conversions.

Table 15–10 Examples of the **x** and **X** conversions

Sample format	Sample output Value = 45	Sample output Value = –45
%12x	2d	ffffffd3
%012x	00000000002d	0000ffffffd3
%#12X	0X2D	0XFFFFFFD3
%#012X	0X000000002D	0X00FFFFFFD3
%-12x	2d	ffffffd3
%-#12x	0x2d	0xffffffd3
%12.4x	002d	ffffffd3
%-#12.4x	0x002d	ffffffd3

The c conversion. The argument is printed as a character or wide character. One argument is consumed. The +, space, and # flags, and the precision specification, are not relevant to the c conversion operation. The conversions applied to the argument character depend on whether the 1 size specifier is present and whether printf or wprintf is used. The possibilities are listed in Table 15-13. Table 15-12 shows examples of the c conversion.

Table 15–11 Conversions of the **c** specifier

Function	Size specifier	Argument type	Conversion
printf	none	**int**	argument is converted to **unsigned char** and copied to the output
	l	**wint_t**	argument is converted to **wchar_t**, converted to a multibyte characters as if by **wcrtomb**[a], and output
wprintf	none	**int**	argument is converted to a wide character as if by **btowc** and copied to the output
	l	**wint_t**	argument is converted to **wchar_t** and copied to the output

[a] The conversion state for the **wcrtomb** function is set to zero before the character is converted.

Table 15–12 Examples of the **c** conversion

Sample format	Sample output Value = '*'
%12c	*
%012c	00000000000*
%-12c	*

The s conversion. The argument is printed as a string. One argument is consumed. If the 1 size specifier is not present, the argument must be a pointer to an array of any character type. If 1 is present, the argument must have type **wchar_t** * and designate a sequence of wide characters. The prefix is always empty. The +, space, and # flags are not relevant to the **s** conversion.

If no precision specification is given, then the converted value is the sequence of characters in the string argument up to but not including the terminating null character or null wide character. If a precision specification p is given, then the converted value is the first p characters of the output string or up to but not including the terminating null character, whichever is shorter. When a precision specification is given, the argument string need not end in a null character as long as it contains enough characters to yield the maximum number of output characters. When writing multibyte characters (printf, with 1), in no case will a partial multibyte character be written, so the actual number of bytes written may be less than p.

The conversions that occur on the argument string depend on whether the l size specifier is present and whether the printf or wprintf functions are used. The possibilities are listed in Table 15-13. Table 15-14 shows examples of the **s** conversion.

Table 15–13 Conversions of the **s** specifier

Function	Size specifier	Argument type	Conversion
`printf`	none	`char *`	characters from the argument string are copied to the output
	`l`	`wchar_t *`	wide characters from the argument string are converted to multibyte characters as if by `wcrtomb`[a]
`wprintf`	none	`char *`	multibyte characters from the argument string are converted to wide characters as if by `mbrtowc`[a]
	`l`	`wchar_t *`	wide characters from the argument string are copied to the output

a - The conversion state for the wertomb or mbrtowc function is set to zero before the first character is converted. Subsequent conversions use the state as modified by the preceding characters.

Table 15–14 Examples of the **s** conversion

Sample format	Sample output Value = `"zap"`	Sample output Value = `"longish"`
`%12s`	`zap`	`longish`
`%12.5s`	`zap`	`longi`
`%012s`	`000000000zap`	`00000longish`
`%-12s`	`zap`	`longish`

The p conversion. The argument must have type void *, and it is printed in an implementation-defined format. For most computers, this will probably be the same as the format produced by the o, x, or X conversions. This conversion operator is found in Standard C, but is otherwise uncommon.

The n conversion. The argument must have type int * if no size modifier is used, type long * if the 1 specifier is used, or type short * if the h specifier is used.Instead of outputting characters, this conversion operator causes the number of characters output so far to be written into the designated integer. This conversion operator is found in Standard C, but is otherwise uncommon.

The f and F conversions. Signed decimal floating-point conversion is performed. One argument is consumed, which should be of type double if no size modifier is used or type long double if L is used. If an argument of type float is supplied, it is converted to type double by the usual argument promotions, so it does work to use %f to print a number of type float.

The converted value consists of a sequence of decimal digits, possibly with an embedded decimal point, that represents the approximate absolute value of the argument.

At least one digit appears before the decimal point. The precision specifies the number of digits to appear after the decimal point. If the precision is 0, then no digits appear after the decimal point. Moreover, the decimal point also does not appear unless the # flag is present. If no precision is specified, then a precision of 6 is assumed.

If the floating-point value cannot be represented exactly in the number of digits produced, then the converted value should be the result of rounding the exact floating-point value to the number of decimal places produced. (Some C implementations do not perform correct rounding in all cases.)

In C99, if the floating-point value represents infinity, then the converted value using the f operator is one of inf, **-inf**, **infinity**, or **-infinity**. (Which one is chosen is implementation-defined.) If the floating-point value represents NaN, then the converted value using the f operator is one of **nan, -nan**, **nan** (...), or **-nan**(...), where "..." is an implementation-defined sequence of letters, digits, or underscores. The **F** operator converts infinity and **NaN** using uppercase letters. The # and 0 flags have no effect on conversion of infinity or **NaN**.

The prefix is computed as follows. If the argument is negative, the prefix is a minus sign. If the argument is non-negative and the + flag is specified, then the prefix is a plus sign. If the argument is non-negative, the space flag is specified, and the + flag is not specified, then the prefix is a space. Otherwise, the prefix is empty. Table 15—15 shows examples of the **f** conversion.

Table 15–15 Examples of the `f` conversion

Sample format	Sample output Value = 12.678	Sample output Value = –12.678
`%10.2f`	`12.68`	`-12.68`
`%010.2f`	`000000012.68`	`-00000012.68`
`% 010.2f`	`00000012.68`	`-00000012.68`
`%+10.2f`	`+12.68`	`-12.68`
`%+010.2f`	`+00000012.68`	`-00000012.68`
`%-10.2f`	`12.68`	`-12.68`
`%- 10.2f`	`12.68`	`-12.68`
`%-+10.4f`	`+12.6780`	`-12.6780`

The e and E conversions. Signed decimal floating-point conversion is performed. One argument is consumed, which should be of type double if no size specifier is used or type long double if L is used. An argument of type float is permitted, as for the f conversion. The e conversion is described; the E conversion differs only in that the letter E appears whenever e appears in the e conversion.

The converted value consists of a decimal digit, then possibly a decimal point and more decimal digits, then the letter e, then a plus or minus sign, then finally at least two more decimal digits. Unless the value is zero, the part before the letter e represents a value between 1.0 and 9.99.... The part after the letter e represents an exponent value as a signed decimal integer. The value of the first part, multiplied by 10 raised to the value of the second

part, is approximately equal to the absolute value of the argument. The number of exponent digits is the same for all values and is the maximum number needed to represent the range of the implementation's floating-point types. Table 15-16 shows examples of **e** and **E** conversions.

Table 15–16 Examples of **e** and **E** conversions

Sample format	Sample output Value = 12.678	Sample output Value = –12.678
%10.2e	1.27e+01	-1.27e+01
%010.2e	00001.27e+01	-0001.27e+01
% 010.2e	0001.27e+01	-0001.27e+01
%+10.2E	+1.27E+01	-1.27E+01
%+010.2E	+0001.27E+01	-0001.27E+01
%-10.2e	1.27e+01	-1.27e+01
%- 10.2e	1.27e+01	-1.27e+01
%-+10.2e	+1.27e+01	-1.27e+01

The precision specifies the number of digits to appear after the decimal point; if not supplied, then 6 is assumed. If the precision is 0, then no digits appear after the decimal point. Moreover, the decimal point also does not appear unless the # flag is present. If the floating-point value cannot be represented exactly in the number of digits produced, then the converted value is obtained by rounding the exact floating-point value. The prefix is computed as for the £ conversion. Values of infinity or NaN are converted as specified for the f and F conversions.

The g and G conversions. Signed decimal floating-point conversion is performed. One argument is consumed, which should be of type double if no size specifier is used, or type long double if L is used. An argument of type float is permitted, as for the f conversion. Only the g conversion operator is discussed later; the G operation is identical except that wherever g uses e conversion, G uses E conversion. If the specified precision is less than 1, then a precision of 1 is used. If no precision is specified, then a precision of 6 is assumed.

The g conversion begins the same as either the f or e conversions; which one is selected depends on the value to be converted. The Standard C specification says that the e conversion is used only if the exponent resulting from the e conversion is less than — 4 or greater than or equal to the specified precision. Some other implementations use the e conversion if the exponent is less than — 3 or strictly greater than the specified precision.

The converted value (whether by f or e) is then further modified by stripping off trailing zeros to the right of the decimal point. If the result has no digits after the decimal point, then the decimal point is also removed. If the # flag is present, this stripping of zeros and the decimal point does not occur.

The prefix is computed as for the f and e conversions. Values of infinity or NaN are converted as specified for the f and F conversions.

The a and A conversions. These conversions are new in C99. Signed hexadecimal floating-point conversion is performed. One argument is consumed, which should be of type double if no size specifier is used or type long double if L is used.

An argument of type float is permitted, as for the other floating-point conversions. The a conversion is described; the A conversion differs by using uppercase letters for the hexadecimal digits, the prefix (0X), and the exponent letter (P).

The converted value consists of a hexadecimal digit, then possibly a decimal point and more hexadecimal digits, then the letter p, then a plus or minus sign, then finally one or more decimal digits. Unless the value is zero or denormalized, the leading hexadecimal digit is nonzero. The part after the letter p represents a binary exponent value as a signed decimal integer.

The precision specifies the number of hexadecimal digits to appear after the decimal point; if not supplied, then enough digits appear to distinguish values of type double. (If FLT RADIX is 2, then the default precision is enough to exactly represent the values.) If the precision is 0, then no digits appear after the decimal point; moreover, the decimal point also does not appear unless the # flag is present. If the floating-point value cannot be represented exactly in the number of hexadecimal digits produced, then the converted value is obtained by rounding the exact floating-point value. The prefix is computed as for the f conversion.

Values of infinity or NaN are converted as specified for the f and F conversions.

The % conversion. A single percent sign is printed. Because a percent sign is used to indicate the beginning of a conversion specification, it is necessary to write two of them to have one printed. No arguments are consumed, and the prefix is empty.

Standard C does not permit any flag characters, minimum width, precision, or size modifiers to be present; the complete conversion specification must be %%. However, other C implementations perform padding just as for any other conversion operation; for example, the conversion specification %05% prints 0000% in these implementations. The +, space, and # flags, the precision specification, and the size specifications are never relevant to the % conversion operation.

Example

The following two-line program is known as a quine — a self-reproducing program. When executed, it will print a copy of itself on the standard output. (The first line of the program is too long to fit on a printed line in this book, so we have split it after % cmain() by inserting a backslash and a line break.)

```
char*f="char*f=%c%s%c,q='%c',n='%cn',b='%c%c';%cmain()\
{printf(f,q,f,q,q,b,b,b,n,n);}%c",q='"',n='\n',b='\\';
main(){printf(f,q,f,q,q,b,b,b,n,n);}
```

The following one-line program is almost a quine. (We have split it after ";main() by inserting a backslash and a line break since it does not fit on a printed line.) We leave it to the reader to discover why it is not exactly a quine.

```
char*f="char*f=%c%s%c;main(){printf(f,34,f,34);}";main()\
{printf(f,34,f,34);}
```