

# F

**F suffix** *See* constant, floating.

**f suffix** For the use of this construct in a floating-point constant, *see* constant, floating. For use with math library function names in C89, *see* future library directions.

**fabs[f|l]** A function that computes the absolute value of its floating-point argument *x*.

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

The `float` and `long double` versions were an invention of C89, where they were optional; however, in C99, they are required.

**false** One of the two possible truth values, true and false. In C, an expression tests true if its value is nonzero; otherwise it tests false. Logical tests, such as `if (x)`, are equivalent to, and treated as, `if (x != 0)`. As such, logical tests may be performed on pointer expressions as well as arithmetic expressions because a zero-valued pointer expression represents the null pointer constant. *See also* false.

By definition, logical, relational, and equality expressions have type `int` and value 0 (false) or 1 (true).

**false**<sup>C99</sup> A macro, defined in `stdbool.h`, that expands to the integer constant 0. It is intended for use in contexts involving the `bool` macro (or its underlying type, `_Bool`.) *See also* true.

**fclose** A function that causes the stream pointed to by `stream` to be flushed and the corresponding file to be closed.

```
#include <stdio.h>
int fclose(FILE *stream);
```

If `fclose` successfully closes the stream, it returns a zero. If the stream was already closed or an error occurs, EOF is returned. If a program terminates abnormally, there is no guarantee that streams open for output will have their buffers flushed. It is permissible to close the files pointed to by `stdin`, `stdout`, and `stderr`.

**fdim[f|l]**<sup>C99</sup> A function that determines the positive difference between its arguments.

```
#include <math.h>
double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);
```

A range error may occur.

**FE\_ALL\_EXCEPT**<sup>C99</sup> A macro, defined in `fenv.h`, that is the bitwise-OR of all floating-point exception macros defined by the implementation.

**feclearexcept**<sup>C99</sup> A function that clears the supported floating-point exceptions represented by its argument.

```
#include <fenv.h>
void feclearexcept(int excepts);
```

**FE\_DFL\_ENV**<sup>C99</sup> A macro, defined in `fenv.h`, that represents the default floating-point environment, as installed at program startup. It expands to an expression of type `const fenv_t *`.

**FE\_DIVBYZERO**<sup>C99</sup> *See* floating-point exception macro.

**FE\_DOWNWARD**<sup>C99</sup> *See* rounding direction.

**fegetenv**<sup>C99</sup> A function that stores the current floating-point environment in the object pointed to by `envp`.

```
#include <fenv.h>
void fegetenv(fenv_t *envp);
```

**fegetexceptflag**<sup>C99</sup> A function that stores an implementation-defined representation of the states of the floating-point status flags indicated by the argument `excepts` in the object pointed to by `flagp`.

```
#include <fenv.h>
void fegetexceptflag(fexcept_t *flagp, int excepts);
```

**fegetround**<sup>C99</sup> A function that returns the current rounding direction.

```
#include <fenv.h>
int fegetround(void);
```

The value returned should correspond to one of the rounding direction macros. It will be a negative value if there is no such rounding direction macro or the current rounding direction cannot be determined.

**feholdexcept**<sup>C99</sup> A function that saves the current floating-point environment in the object pointed to by `envp`, clears the floating-point status flags, and then installs a non-stop mode, if available, for all floating-point exceptions.

```
#include <fenv.h>
int feholdexcept(fenv_t *envp);
```

**FE\_INEXACT**<sup>C99</sup> See floating-point exception macro.

**FE\_INVALID**<sup>C99</sup> See floating-point exception macro.

**FENV\_ACCESS pragma**<sup>C99</sup> See `#pragma STDC FP_ACCESS`.

**fenv.h**<sup>C99</sup> A header that provides a number of types, macros, and functions which give access to the current floating-point environment. The following table lists their names and purpose.

<i>Name</i>	<i>Purpose</i>
<b>FE_ALL_EXCEPT</b>	All FP exceptions Ored together
<b>FE_DFL_ENV</b>	Default FP environment
<b>FE_DIVBYZERO</b>	FP exception macro
<b>FE_DOWNWARD</b>	Rounding direction
<b>FE_INEXACT</b>	FP exception macro
<b>FE_INVALID</b>	FP exception macro
<b>FE_OVERFLOW</b>	FP exception macro
<b>FE_TONEAREST</b>	Rounding direction
<b>FE_TOWARDZERO</b>	Rounding direction
<b>FE_UNDERFLOW</b>	FP exception macro
<b>FE_UPWARD</b>	Rounding direction
<b>feclearexcept</b>	Clear given FP exceptions
<b>fegetenv</b>	Store current FP environment
<b>fegetexceptflag</b>	Store FP status
<b>fegetround</b>	Get rounding mode
<b>feholdexcept</b>	Save current FP environment
<b>fenv_t</b>	Type that represents an FP environment
<b>feraiseexcept</b>	Raise given FP exceptions
<b>fesetenv</b>	Set current FP environment
<b>fesetexceptflag</b>	Set given FP exception flags
<b>fesetround</b>	Set rounding mode
<b>fetestexcept</b>	Test FP status flags
<b>feupdateenv</b>	Save currently raised FP exceptions
<b>fexcept_t</b>	Type that represents FP status flags

**fenv\_t**<sup>C99</sup> A type, defined in `fenv.h`, that is capable of representing a floating-point environment.

**feof** A function that tests the end-of-file indicator for the file pointed to by **stream**.

```
#include <stdio.h>
int feof(FILE *stream);
```

Zero is returned if the end-of-file indicator is clear; nonzero if it is set.

**FE\_OVERFLOW**<sup>C99</sup> *See* floating-point exception macro.

**ferror**<sup>C99</sup> A function that raises the supported floating-point exceptions represented by its argument.

```
#include <fenv.h>
void ferror(int excepts);
```

**ferror** A function that tests the error indicator for the file pointed to by **stream**.

```
#include <stdio.h>
int ferror(FILE *stream);
```

Zero is returned if the error indicator is clear; nonzero if it is set.

**fesetenv**<sup>C99</sup> A function that establishes the floating-point environment represented by the object pointed to by **envp**.

```
#include <fenv.h>
void fesetenv(const fenv_t *envp);
```

**fesetexceptflag**<sup>C99</sup> A function that sets the floating-point status flags indicated by the argument **excepts** to the states stored in the object pointed to by **flagp**. No exceptions are raised.

```
#include <fenv.h>
void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

**fesetround**<sup>C99</sup> A function that establishes the rounding direction represented by its argument **round**, provided that argument equals the value of a supported rounding direction macro.

```
#include <fenv.h>
int fesetround(int round);
```

**fetestexcept**<sup>C99</sup> A function that determines which of a specified subset of the floating-point exception flags are currently set.

```
#include <fenv.h>
int fetestexcept(int excepts);
```

`FE_TONEAREST`<sup>C99</sup> *See* rounding direction.

`FE_TOWARDZERO`<sup>C99</sup> *See* rounding direction.

`FE_UNDERFLOW`<sup>C99</sup> *See* floating-point exception macro.

`feupdateenv`<sup>C99</sup> A function that saves the currently raised floating-point exceptions in its automatic storage, installs the floating-point environment represented by the object pointed to by `envp`, and then raises the saved floating-point exceptions.

```
#include <fenv.h>
void feupdateenv(const fenv_t *envp);
```

`FE_UPWARD`<sup>C99</sup> *See* rounding direction.

`fexcept_t`<sup>C99</sup> A type, defined in `fenv.h`, that represents the floating-point status flags.

`fflush` A function that flushes an open stream's I/O buffer. If the stream were being written, any unwritten data in the output buffer is written.

```
#include <stdio.h>
int fflush(FILE *stream);
```

`fflush` should be used only with streams open for output or open for update and currently in output mode. If the stream is not open for output, or if it is open for update and the immediately previous operation is other than output, the behavior is undefined. However, some implementations permit input streams to be `fflushed` reliably. A zero value is returned on success; an EOF is returned if a write error occurs. If a program terminates abnormally, there is no guarantee that streams open for output will have their buffers flushed. If `stream` is the null pointer, all files currently open for output are flushed.

`fgetc` A function that gets the next character (if any) from the file pointed to by `stream`.

```
#include <stdio.h>
int fgetc(FILE *stream);
```

The character is read as an `unsigned char` and returned as an `int`. If end-of-file is detected, `fgetc` returns EOF, and the end-of-file indicator is set for that stream. (`feof` can be used to test this indicator.) If a read error occurs, EOF is returned and the error indicator is set for that stream. (`ferror` can be used to test this indicator.) *See also* `fgetcwc`; `getwc`.

**fgetpos**<sup>C89</sup> A function that stores the current value of the stream's file position indicator in the object pointed to by **pos**.

```
#include <stdio.h>
int fgetpos(FILE * restrict stream,
            fpos_t * restrict pos);
```

**fgetpos** was invented to handle very large files whose file position indicator cannot be represented in a **long int** (as required by **ftell**).

The object stored in **pos** is suitable for use by **fsetpos** to restore the file to that previous position. **fgetpos** returns zero on success. On failure, a nonzero value is returned, and **errno** is set to an implementation-defined positive value.

**fgets** A function that reads at most  $n - 1$  characters from the file pointed to by **stream** into the array pointed to by **s**. A **'\0'** is appended to **array** after the last read character.

```
#include <stdio.h>
char *fgets(char * restrict s, int n,
            FILE * restrict stream);
```

If a new-line is encountered, or end-of-file occurs, no more characters are read. If seen, a new-line is included in the array (unlike **gets**). If **fgets** succeeds, it returns **s**. If an end-of-file condition is encountered and no characters have been read yet, **NULL** is returned and the contents of the array pointed to by **s** are unchanged. **NULL** is also returned on a read error; however, the contents of the array are then indeterminate. *See also fgets.*

**fgetwc**<sup>C95</sup> A function that gets the next wide character (if any) from the file pointed to by **stream**.

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc(FILE *stream);
```

The wide character is read as a **wchar\_t** and returned as a **wint\_t**. If end-of-file is detected, **fgetwc** returns **WEOF**, and the end-of-file indicator is set for that stream. If a read error occurs, **WEOF** is returned, and the error indicator is set for that stream. *See also fgetc; getc.*

**fgetws**<sup>C95</sup> A function that reads at most  $n - 1$  wide characters from the file pointed to by **stream** into the array pointed to by **s**. A wide **'\0'** is appended to **array** after the last read character.

```
#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws(wchar_t * restrict s, int n,
    FILE * restrict stream);
```

*See also* `fgets`.

**field** An item input by the `scanf` function family or output by the `printf` function family. Each field has a corresponding conversion specifier.

**\_\_FILE\_\_** A predefined object-like macro that expands to the name of the source file as a string. This macro can be used in any context where a string literal is permitted or required; for example,

```
char fname[] = __FILE__;
printf("%s", __FILE__);
```

Because adjacent string literals are concatenated, the following is also permitted:

```
printf(">%s<\n", "xxx" __FILE__ "xxx");
```

Some implementations include the file's full path name (such as device and directory as well as name), others do not. This macro cannot be the subject of `#undef`.

Note that there is no wide string version of this macro, so it was difficult to get the filename string concatenated with wide strings; however, C99 allows wide and single-byte strings to be concatenated directly, so that operation becomes trivial. *See also* `#line`.

**file** A term that has the usual data processing meaning. When a file is opened by a standard library function, a stream is associated with it. The file is specified in all subsequent operations on, and accesses to, that file via a stream, which is represented as a `FILE` pointer.

**FILE** An object type capable of containing the “current context” of an open file. This information includes buffering details, error and end-of-file flags, file position indicator, and other unspecified members. A program never needs to create `FILE` objects directly. Instead, they are created and managed by library functions.

**file access functions** The `stdio.h` functions `fclose`, `fflush`, `fopen`, `freopen`, `setbuf`, and `setvbuf`.

**file closing** *See* `abort`; `exit`; `_Exit`; `fclose`; `freopen`.

**file creation** *See* `fopen`; `tmpfile`.

**file name** The names of files for which the attributes and format of those on external media (such as disks or tapes) is specific to each operating system. As such, Standard C places no requirements on file names nor does it rely on them. Functions such as `fopen` simply expect a file name to be represented as a null-terminated array of `char`. *See also* `FILENAME_MAX`.

While most implementations actually represent headers as text files (with the same name) on disk, it is not required.

The file names used by `tmpfile` and `tmpnam` are unspecified.

**file opening** *See* `fopen`; `freopen`; `tmpfile`.

**file operation functions** The `stdio.h` functions `remove`, `rename`, `tmpfile`, and `tmpnam`.

**file pointer** A pointer to an object of type `FILE`.

**file position indicator** A member that is part of each `FILE` object and has type `fpos_t`. When files on devices supporting positioning requests are opened for read or write, the file position indicator is set to the start of the file. If the file is opened for append, the indicator is set either to the start or to the end of the file, as defined by the implementation. This indicator is maintained by the file positioning and I/O functions and is not accessed directly by application programmers. *See also* file positioning functions.

**file positioning functions** The `stdio.h` functions `fgetpos`, `fseek`, `fsetpos`, `ftell`, and `rewind`.

**file, source** The file containing the text of the program to be translated. *See also* translation unit.

`FILENAME_MAX`<sup>C89</sup> A macro, defined in `stdio.h`, that expands to an integer constant expression that represents the maximum length of a file name string that can be used with the implementation.

**flexible array member**<sup>C99</sup> The last member whose type is an incomplete array, which is present in a structure that has more than one named member; for example,

```
struct message {
    int count;
    char text[];    // flexible array member
};
```

Such structures are typically used as follows:

```
message *pm;
```

```
pm = malloc(sizeof(struct message) + (N * sizeof(char)));
```

where N is the the number of elements represented by count.

**float** A keyword used for one of the three floating-point types. (The other two are **double** and **long double**.) Traditionally, **double** meant “double precision” while **float** implied “single precision.” **float** expressions traditionally were widened to **double** when used in expressions and as arguments to functions. However, Standard C allows them to be used without widening if no prototype is in scope or, if a prototype is in scope yet it contains an ellipses in the appropriate place. *See also* conversion, function arguments; floating type.

**float** **\_Complex**<sup>C99</sup> *See* complex.

**float.h**<sup>C89,C99</sup> A header that contains a family of macros that describe the floating-point properties of the target system. While Standard C requires certain minima (or maxima), it is intended that an implementation will document its actual values. The macros have FLT, DBL, and LDBL prefixes which designate the types **float**, **double**, and **long double**, respectively. (Note that several generic floating-point attributes have the prefix FLT, even though they are not **float**-specific.) The following is the complete set of macros:

DBL_DIG	FLT_DIG	LDBL_DIG
DBL_EPSILON	FLT_EPSILON	LDBL_EPSILON
DBL_MANT_DIG	FLT_MANT_DIG	LDBL_MANT_DIG
DBL_MAX	FLT_MAX	LDBL_MAX
DBL_MAX_10_EXP	FLT_MAX_10_EXP	LDBL_MAX_10_EXP
DBL_MAX_EXP	FLT_MAX_EXP	LDBL_MAX_EXP
DBL_MIN	FLT_MIN	LDBL_MIN
DBL_MIN_10_EXP	FLT_MIN_10_EXP	LDBL_MIN_10_EXP
DBL_MIN_EXP	FLT_MIN_EXP	LDBL_MIN_EXP
DECIMAL_DIG <sup>C99</sup>		
FLT_EVAL_METHOD <sup>C99</sup>	FLT_RADIX	FLT_ROUNDS

All integer value macros except FLT\_ROUNDS are guaranteed to expand to a translation-time constant expression suitable for use with **#if**. *See also* environmental limits.

**float** **\_Imaginary**<sup>C99</sup> *See* imaginary.

**floating-point exception macro**<sup>C99</sup> A macro, defined in fenv.h, that indicates that implementation supports the corresponding floating-point exception. The defined set contains FE\_DIVBYZERO, FE\_INEXACT,

FE\_INVALID, FE\_OVERFLOW, and FE\_UNDERFLOW. Other implementation-defined macros whose names begin with FE\_ and an uppercase letter, may also be defined. These macros must expand to integer constant expressions such that bitwise-ORing of all combinations of the macros results in distinct values.

**floating suffix, f or F**<sup>C89</sup> See constant, floating.

**floating suffix, l or L**<sup>C89</sup> See constant, floating.

**floating type** A type that is represented using an exponent and fractional part. There are three such types: `float`, `double`, and `long double` (which was an invention of C89). An object of type `long double` must have as least as much range and precision as that of type `double`, which, in turn, must have at least as much as `float`. As such, two or more of the three types could map to the same representation. The header `float.h` can be used to determine the attributes of an implementation's floating-point types. All floating types are arithmetic types.

**float\_t**<sup>C99</sup> A type, defined in `math.h`, that is a floating type at least as wide as `float`. If the macro `FLT_EVAL_METHOD` evaluates to 0, `float_t` is `float`. If `FLT_EVAL_METHOD` evaluates to 1, `float_t` is `double`. If `FLT_EVAL_METHOD` evaluates to 2, `float_t` is `long double`. The exact type is implementation defined for other values of `FLT_EVAL_METHOD`. See `double_t`

**floor[f|l]** A function that computes the largest integer value not greater than its argument `x`.

```
#include <math.h>
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

The integer value computed is returned as a `double`.

The `float` and `long double` versions were an invention of C89, where they were optional; however, in C99, they are required.

**FLT\_DIG**<sup>C89</sup> A macro, defined in `float.h`, that designates the number of decimal digits, such that a `float` value of that significance can be rounded into a floating-point number and back again without change in those decimal digits.

**FLT\_EPSILON**<sup>C89</sup> A macro, defined in `float.h`, that designates the difference between 1.0 and the least value greater than 1.0 that is representable in the `float` type.

`FLT_EVAL_METHOD`<sup>C89</sup> A macro, defined in `float.h`, that indicates how floating-point operations are evaluated with respect to range and precision. Its value can be one of the following:

- 1 Indeterminable.
  - 0 Evaluate all operations and constants to the range and precision of the type.
  - 1 Evaluate operations and constants of type `float` and `double` to the range and precision of `double`, and `long double` to `long double`.
  - 2 Evaluate all operations and constants to the range and precision of `long double`.
- Other** All other negative values indicate implementation-defined behavior.

`FLT_MANT_DIG`<sup>C89</sup> A macro, defined in `float.h`, that designates the number of base-`FLT_RADIX` digits in the floating-point significand of a `float` value.

`FLT_MAX`<sup>C89</sup> A macro, defined in `float.h`, that designates the maximum representable finite `float` number.

`FLT_MAX_10_EXP`<sup>C89</sup> A macro, defined in `float.h`, that designates the maximum integer such that 10 raised to that power is in a given range of representable finite floating-point numbers.

`FLT_MAX_EXP`<sup>C89</sup> A macro, defined in `float.h`, that designates the maximum integer such that `FLT_RADIX` raised to that power minus 1 is a representable finite floating-point number.

`FLT_MIN`<sup>C89</sup> A macro, defined in `float.h`, that designates the minimum normalized positive `float` number.

`FLT_MIN_10_EXP`<sup>C89</sup> A macro, defined in `float.h`, that designates the minimum negative integer such that 10 raised to that power is in a given range of normalized floating-point numbers.

`FLT_MIN_EXP`<sup>C89</sup> A macro, defined in `float.h`, that designates the minimum negative integer such that `FLT_RADIX` raised to that power minus 1 is a normalized floating-point number.

`FLT_RADIX`<sup>C89</sup> A macro, defined in `float.h`, that designates the radix of exponent representation. This macro expands to a translation-time constant expression suitable for use with `#if`.

`FLT_ROUNDS`<sup>C89</sup> A macro, defined in `float.h`, that designates the current mode of rounding behavior. Standard C defines the following modes:

- 1 Indeterminable

- 0 Toward zero
- 1 To nearest
- 2 Toward positive infinity
- 3 Toward negative infinity

All other values specify implementation-defined rounding behavior.

`fma[f|l]`<sup>C99</sup> A function that computes a fused multiply-and-add,  $(x \times y) + z$ , as a single operation.

```
#include <math.h>
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y,
                 long double z);
```

`fma` may provide a more accurate result than performing a multiply followed by a separate add since only the final result is rounded, as opposed to rounding the result of the multiply, followed by rounding the result of the add.

`fmax[f|l]`<sup>C99</sup> A function that determines the maximum numeric value of its arguments.

```
#include <math.h>
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

If one argument is a NaN and the other numeric, the numeric value is returned.

`fmin[f|l]`<sup>C99</sup> A function that determines the minimum numeric value of its arguments.

```
#include <math.h>
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

If one argument is a NaN and the other numeric, the numeric value is returned.

`fmod[f|l]` A function that computes the floating-point remainder of  $x/y$ .

```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

The value returned is  $x - i \times y$ , where  $i$  is an integer such that, if  $y$  is nonzero, the result has the same sign as  $x$  and a magnitude less than that of  $y$ . If  $y$  is 0, it is implementation defined whether or not a domain error occurs, or if `fmod` returns 0.

The `float` and `long double` versions were an invention of C89, where they were optional; however, in C99, they are required.

**fopen** A function that opens the file whose name is pointed to by `filename`, in the mode specified by `mode`.

```
#include <stdio.h>
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

`mode` points to a string whose initial character contents must be one of the following sequences: "r", "w", "a", "rb", "wb", "ab", "r+", "w+", "a+", "rb+", "wb+", "ab+", "r+b", "w+b", or "a+b". Mode `r` signifies read, `w` signifies write (or create if the file does not already exist), and `a` signifies append mode. In the absence of mode `b`, the file is deemed to be a text stream, while `b` signifies a binary stream. The `+` mode indicates that the file is to be open for update. The set of modes "r?+" is equivalent to the set "r?".

If `fopen` succeeds, it returns a `FILE` pointer to the opened stream. On failure, it returns `NULL`. Note that an implementation may limit the number of currently open files—`FOPEN_MAX` specifies the number permitted—in which case `fopen` will fail if you attempt to exceed this number. The error and end-of-file indicators are cleared.

The standard streams `stderr`, `stdin`, and `stdout` are opened automatically at program startup. *See also* `FOPEN_MAX`; `freopen`.

`FOPEN_MAX`<sup>C89</sup> A macro, defined in `stdio.h`, that expands to an integer constant expression representing the minimum number of files that the implementation guarantees you can have open simultaneously. Standard C requires `FOPEN_MAX` to be at least eight, including `stdin`, `stdout`, and `stderr`. Earlier versions of it were called `SYS_MAX` and `OPEN_MAX`. Files created by `tmpfile` count against this limit.

**for** A looping construct that evaluates its criteria before each iteration of the loop like `while` and unlike `do/while` which always executes at least once. It has the following form:

```
for ( [ exp1 ]; [ exp2 ]; [ exp3 ] )
    statement
```

First, *exp1* is evaluated for the side effects it contains. Then, *exp2* is evaluated. If it tests false, the body of the **for** statement is bypassed. If it tests true, *statement* is executed and *exp3* is evaluated for its side effects. The process is then repeated starting with *exp2*.

All three expressions are optional. If the first is missing, there is no initialization. If the second is missing, it is as if a true expression were present. If the third is missing, there is nothing to do at the end of each iteration. A **for** construct can always be rewritten as a **while** construct and vice versa. Each of *exp1*, *exp2*, and *exp3* is a full expression.

C99 permits *exp1* to be a declaration, whose identifiers' scope ends at the end of the loop body.

**form-feed** One of the white space characters allowed in source text and as input to certain library functions.

**form-feed escape sequence** An escape sequence, `\f`, which represents the form-feed character.

**formatted I/O** An input/output capability provided by the **scanf** and **printf** functions families, respectively. The **printf** family provides formatted output capabilities to the standard output stream (usually directed to the screen or terminal printer), to files, and to memory. The **scanf** family handles formatted input from these same places.

**FP\_\* value macros** See number classification macro.

**fpclassify**<sup>C99</sup> A macro, defined in `math.h`, that classifies its argument value as having one of the following categories: infinite, NaN, normal, subnormal, zero, or implementation defined.

```
#include <math.h>
int fpclassify(real-floating-type x);
```

The value returned is one of the number classification macros.

**FP\_CONTRACT pragma**<sup>C99</sup> See `#pragma STDC FP_CONTRACT`.

**FP\_FAST\_FMA**<sup>C99</sup> An optional macro, defined in `math.h`, which indicates that, in general, the `fma` function executes about as fast as, or faster than, a multiply and an add of `double` operands. See also the macros `FP_FAST_FMAF` and `FP_FAST_FMAL`.

**FP\_FAST\_FMAF**<sup>C99</sup> An optional macro, defined in `math.h`, which indicates that, in general, the `fma` function executes about as fast as, or faster than,

a multiply and an add of `float` operands. *See also* `FP_FAST_FMA` and `FP_FAST_FMAL`.

`FP_FAST_FMAL`<sup>C99</sup> An optional macro, defined in `math.h`, which indicates that, in general, the `fma` function executes about as fast as, or faster than, a multiply and an add of `long double` operands. *See also* `FP_FAST_FMA` and `FP_FAST_FMAF`.

`FP_ILOGBO`<sup>C99</sup> A macro, defined in `math.h`, that expands to one of the integer constant expressions `INT_MIN` or `-INT_MAX`. This macro's value is returned by the function `ilogb` if its argument is zero.

`FP_ILOGBNAN`<sup>C99</sup> A macro, defined in `math.h`, that expands to one of the integer constant expressions `INT_MIN` or `INT_MAX`. This macro's value is returned by the function `ilogb` if its argument is a NaN.

`FP_INFINITE`<sup>C99</sup> *See* number classification macro.

`FP_NAN`<sup>C99</sup> *See* number classification macro.

`FP_NORMAL`<sup>C99</sup> *See* number classification macro.

`fpos_t`<sup>C89</sup> A type, defined in `stdio.h`, that is large enough to hold the largest possible file position indicator for the implementation. *See* `fgetpos`; `fsetpos`.

`fprintf` A function that writes formatted output to the file specified by `stream` in a format specified by `format`. It is a more general version of `printf`.

```
#include <stdio.h>
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
```

A call to `printf` is equivalent to a call to `fprintf` using the stream `stdout`. For a discussion of `format` and the value returned, *see* `printf`. *See also* `fwprintf`.

`FP_SUBNORMAL`<sup>C99</sup> *See* number classification macro.

`fputc` A function that writes the character specified by `c` (converted to `unsigned char`) to the file pointed to by `stream`.

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

On success, the character written is returned. If a write error occurs, EOF is returned, and the error indicator is set for that stream. (`ferror` can be used to test this indicator.) *See also* `fputc`; `putc`; `putwc`.

**fputs** A function that writes the string pointed to by **s** to the file pointed to by **stream**.

```
#include <stdio.h>
int fputs(const char * restrict s,
          FILE * restrict stream);
```

The '\0' terminating the string is not written. Unlike **puts**, **fputs** does not append a new-line to the output. **fputs** returns **EOF** if an error occurs; otherwise, it returns a nonnegative value. *See also* **fputc**.

**fputcw**<sup>C95</sup> A function that writes the wide character specified by **c** to the file pointed to by **stream**.

```
#include <stdio.h>
#include <wchar.h>
wint_t fputcw(wchar_t c, FILE *stream);
```

On success, the wide character written is returned. If a write error occurs, **WEOF** is returned, and the error indicator is set for that stream. *See also* **fputc**.

**fputws**<sup>C95</sup> A function that writes the wide string pointed to by **s** to the file pointed to by **stream**.

```
#include <stdio.h>
#include <wchar.h>
int fputws(const wchar_t * restrict s,
           FILE * restrict stream);
```

The wide null terminating the string is not written. **EOF** is returned if an error occurs; otherwise, a nonnegative value is returned. *See also* **fputs**.

**FP\_ZERO**<sup>C99</sup> *See* number classification macro.

**frac\_digits**<sup>C89</sup> An **lconv** structure member that is a nonnegative number representing the number of fractional digits (those after the decimal point) to be displayed in a formatted monetary quantity. A value of **CHAR\_MAX** indicates that the value is not available in the current locale. In the "C" locale this member must have the value **CHAR\_MAX**.

**fread** A function that reads up to **nmemb** elements each of size **size** into the array pointed to by **ptr** from the file pointed to by **stream**.

```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

If an error occurs, the file position indicator's value is indeterminate. If a partial element is read, its value is indeterminate. The value returned is the number of elements successfully read. This number may be less than `nmemb` if end-of-file is detected or if an error occurs. If either `size` or `nmemb` is 0, a zero is returned and the contents of the array pointed to by `ptr` remains intact. *See also* `fwrite`.

**free** A function that causes the space (previously allocated by `calloc`, `malloc`, or `realloc`) pointed to by `ptr` to be freed.

```
#include <stdlib.h>
void free(void *ptr);
```

If `ptr` is `NULL`, `free` does nothing. Otherwise, if `ptr` is not a value previously returned by one of these three allocation functions, the behavior is undefined. The value of a pointer that refers to space that has been freed is indeterminate, and such pointers should not be dereferenced. Note that `free` has no way to communicate an error if one is detected.

**freestanding environment**<sup>C89</sup> *See* environment, freestanding; cross-compilation.

**freopen** A function that is almost identical to `fopen` except that `freopen` recycles an existing `FILE` pointer that points to a currently open file.

```
#include <stdio.h>
FILE *freopen(const char * restrict filename,
               const char * restrict mode, FILE * restrict stream);
```

The arguments `filename` and `mode` are the same as for `fopen`. If `freopen` succeeds, it returns the value of `stream`; otherwise it returns `NULL`. `freopen` first tries to close the file associated with `stream`. Any failure encountered in this attempt is ignored. The error and end-of-file indicators are cleared.

**frexp[f|l]** A function that breaks a floating-point number value into a normalized fraction and an integer power of 2.

```
#include <math.h>
double frexp(double value, int *exp);
float frexpf(float value, int *exp);
long double frexpl(long double value, int *exp);
```

The integer power is stored at the location pointed to by `exp` and the fractional part is the return value. The value returned has magnitude of

0 or is in the interval  $[1/2, 1)$ , and `value` equals  $x \times 2^{*exp}$ . If `value` is 0, both the return value and `*exp` are 0.

The `float` and `long double` versions were an invention of C89, where they were optional; however, in C99, they are required.

**friend** A C++ keyword that is not part of Standard C. If you think you might wish to move C code to a C++ environment in the future, you should refrain from using `friend` as an identifier in new C code you write.

**fscanf** A function that reads formatted input from the file specified by `stream` in a format specified by `format`. It is a more general version of `scanf`.

```
#include <stdio.h>
int fscanf(FILE * restrict stream,
           const char * restrict format, ...);
```

Note, that all arguments must be passed by address. A call to `scanf` is equivalent to a call to `fscanf` using the stream `stdin`. For a discussion of `format` and the value returned, *see* `scanf`. *See also* `fwscanf`.

**fseek** A function that sets the current value of the stream's file position indicator to an `offset` based on the value of `whence`.

```
#include <stdio.h>
int fseek(FILE *stream, long int offset,
          int whence);
```

`whence` may be any one of the three macros `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, which represent, respectively, the start of the file, the current file position, and the end of the file. On success, `fseek` clears the end-of-file indicator, discards any pushed back characters for that stream and returns zero. On failure, it returns a nonzero value.

For very large files whose file position indicator cannot be represented in a `long int` (as required by `fseek`), use `fsetpos`. *See also* `ftell`.

**fsetpos**<sup>C89</sup> A function that sets the file position indicator for the file pointed to by `stream` to the value of the object pointed to by `pos`.

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

On success, `fsetpos` returns zero, clears the end-of-file indicator, and discards characters pushed back via `ungetc`. On failure, a nonzero value

is returned and `errno` is set to an implementation-defined positive value. See also `fgetpos`; `fseek`.

**ftell** A function that returns the current value of the stream's file position indicator.

```
#include <stdio.h>
long int ftell(FILE *stream);
```

On failure, `-1L` is returned and `errno` is set to an implementation-defined positive value. For very large files whose file position indicator cannot be represented in a `long int` (as required by `ftell`), use `fgetpos`. See also `fseek`.

**full declarator** See declarator, full.

**full expression** See expression, full.

**fully buffered stream** A stream in which characters are intended to be sent to or received from the host environment as a block when some buffer is filled. See also `_IOFBF`; `setvbuf`.

`__func__`<sup>C99</sup> A predefined identifier that is implicitly declared in each function as if the following declaration

```
static const char __func__[] = "function-name";
```

occurred immediately following the opening brace of that function's definition; *function-name* is the name of the parent function.

**function** The basic executable module in a C program. It is synonymous with a subroutine or procedure in other languages. All C functions have the same general format, including `main`. A function may expect arguments, produce a return value, both, or neither. In C, function definitions do not nest.

**function argument** See argument.

**function body** That part of a function's definition delimited by `{` and `}` that immediately follows the declarator part that introduces the definition; for example,

```
void f(int i, double d)
{
    /* body begins here */
    ...
}
/* body ends here */
```

```

void f(i, d)
int i;
double d;
{
    /* body begins here */
    ...
}
    /* body ends here */

```

**function call** The act of invoking a function using the function call operator ().

**function call operator** A primary operator, (), that has the following general form:

$$exp1 \ ( \ [exp2 \ [ , \ exp3 \ ] \dots \ ] \ )$$

where *exp1* designates the function to be called and *exp2*, *exp3*, ... are the arguments passed to that function. The order of evaluation of all the expressions is unspecified, but there is a sequence point after all expressions have been evaluated, just before the function is actually called. The type of the result is that returned by the function—it can be any scalar, structure, or union type, or the incomplete type `void`. This operator associates left to right.

**function declarator** The part of a declaration having the following form:

$$D \ ( \ parameter\text{-}type\text{-}list \ )$$

as in `void f(int i, double d)`, or:

$$D \ ( \ identifier\text{-}list_{opt} \ )$$

as in `void f(i, d)`.

*See also* future language directions.

**function definition** A mechanism for defining a function. It has the following general form:

$$\begin{aligned}
 & declaration\text{-}specifiers_{opt} \ declarator \\
 & \quad declaration\text{-}list_{opt} \ compound\text{-}statement
 \end{aligned}$$

where *compound-statement* is the function's body. *See also* future language directions.

**function designator** An expression that has a function type. In most cases, such an expression is converted to a pointer to that function. The most

common way to designate a function is simply to use its name. However, a function also can be designated by dereferencing a pointer to it. For example, given the following declarations:

```
void f(int);
void (*pf)(int) = f;
void (**ppf)(int) = &pf;
```

the function `f` can be designated by the expressions `f`, `*pf`, and `**ppf`.  
*See also* conversion, function.

**function library** A collection of functions that may be defined by Standard C, extra functions provided by an implementer, third-party or user-written functions, or a combination of all three.

**function name** An identifier used to name a function. Function names share the same name space as enumeration constants, variables, and `typedef` names.

**function parameter** *See* parameter.

**function prototype** A function declaration that includes a parameter type list. Each parameter optionally may include an identifier, which has no effect. Prototypes were adapted by C89 from the C++ language, for example,

```
void f(int [i], double [d]);
```

is a prototype that declares `f` to be a function that takes one `int` argument and one `double` argument and returns no value. If present, the parameter names must be unique within that prototype. The “old-style” declaration for this function would be as follows:

```
void f();
```

Essentially, a prototype is an old-style function declaration that has argument list information. A new style of defining a function was also defined by C89. It also uses the prototype notation.

**function return** The explicit or implicit return from a called function to its caller. *See also* return.

**function type** A type that describes a function. This type information includes both the argument list and the return type. *See also* function designator.

**function type conversion** *See* conversion, function.

**future directions** Guidance contained in Standard C to implementers and users with regard to areas in which future revision might take place. Language standards are revised periodically, either to correct shortcomings or, more likely, to add support for new devices, application classes, and environments. (With ANSI and ISO standards, this typically takes place every 10 years.) For specific details *see* future language directions; future library directions. A future standard revision might even drop support for features declared obsolescent in previous versions.

**future language directions** Guidance contained in Standard C to implementers and users with regard to areas in which language revision might occur, due to a revision of the standard, or language may be added as an extension to a conforming implementation. The following items are taken directly from the C89 and C99 standards:

**Array parameters** C89 and C99: The use of two parameters declared with an array type (prior to their adjustment to pointer type) in separate lvalues to designate the same object is an obsolescent feature.

**Character escape sequences** C89 and C99: Lowercase letters as escape sequences are reserved for future standardization. Other characters may be used in extensions.

**External names** C89: Restriction of the significance of an external name to fewer than 31 characters or to only one case is an obsolescent feature that is a concession to existing implementations. C99: Restriction of the significance of an external name to fewer than 255 characters (considering each universal character name or extended source character as a single character) is an obsolescent feature that is a concession to existing implementations.

**Floating types** C99: Future standardization may include additional floating-point types, including those with range, precision, or both greater than `long double`.

**Function declarators** C89 and C99: The use of function declarators with empty parentheses (not prototype-format parameter type declarators) is an obsolescent feature.

**Function definitions** C89 and C99: The use of function definitions with separate parameter identifier and declaration lists (not prototype-format parameter type and identifier declarators) is an obsolescent feature.

**Pragma directives** C99: Pragmas whose first preprocessing token is `STDC` are reserved for future standardization.

**Predefined macro names** C99: Macro names beginning with `__STDC__` are reserved for future standardization.

**Storage-class specifiers** C89: The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature. C99: Declaring an identifier with internal linkage at file scope without the static storage-class specifier is an obsolescent feature.

**future library directions** Guidance contained in Standard C to implementers and users with regard to areas in which library revision might occur due to a revision of the standard or by the addition of extensions to a conforming implementation. The following list is taken directly from the C89 and C99 standards:

The following names are grouped under individual headers for convenience. **All external names described below are reserved no matter what headers are included by the program.**

**Complex arithmetic** `<complex.h>` C99: The function names `cerf`, `cerfc`, `cexp2`, `cexpm1`, `clog10`, `clog1p`, `clog2`, `clgamma`, and `ctgamma`, and the same names suffixed with `f` or `l` may be added to the declarations in the `complex.h` header.

**Character handling** `<ctype.h>` C89 and C99: Function names that begin with `is` or `to` and a lowercase letter (followed by any combination of digits, letters, and underscores) may be added to the declarations in the `ctype.h` header.

**Errors** `<errno.h>` C89 and C99: Macros that begin with `E` and a digit or `E` and an uppercase letter (followed by any combination of digits, letters, and underscores) may be added to the declarations in the `errno.h` header.

**Format conversion of integer types** `<inttypes.h>` C99: Macro names beginning with `PRI` or `SCN` followed by any lowercase letter or `X` may be added to the macros defined in the `inttypes.h` header.

**Localization** `<locale.h>` C89 and C99: Macros that begin with `LC_` and an uppercase letter (followed by any combination of digits, letters, and underscores) may be added to the definitions in the `locale.h` header.

**Mathematics** `<math.h>` C89: The names of all existing functions declared in the `math.h` header, suffixed with `f` or `l`, are reserved respectively for corresponding functions with `float` and `long double` arguments and return values. (C99 requires these functions.)

**Signal handling** `<signal.h>` C89 and C99: Macros that begin with either `SIG` and an uppercase letter or `SIG_` and an uppercase letter (followed by any combination of digits, letters, and underscores) may be added to the definitions in the `signal.h` header.

**Boolean type and values** <stdbool.h> C99: The ability to undefine and perhaps then redefine the macros `bool`, `true`, and `false` is an obsolescent feature.

**Integer types** <stdint.h> C99: Typedef names beginning with `int` or `uint` and ending with `_t` may be added to the types defined in the `stdint.h` header. Macro names beginning with `INT` or `UINT` and ending with `_MAX`, `_MIN`, or `_C` may be added to the macros defined in the `stdint.h` header.

**Input/output** <stdio.h> C89 and C99: Lowercase letters may be added to the conversion specifiers in `fprintf` and `fscanf`. Other characters may be used in extensions. C99: The use of `ungetc` on a binary stream where the file position indicator is zero prior to the call is an obsolescent feature.

**General utilities** <stdlib.h> C89 and C99: Function names that begin with `str` and a lowercase letter (followed by any combination of digits, letters, and underscores) may be added to the declarations in the `stdlib.h` header.

**String handling** <string.h> C89 and C99: Function names that begin with `str`, `mem`, or `wcs` and a lowercase letter (followed by any combination of digits, letters, and underscores) may be added to the declarations in the `string.h` header.

**Extended multibyte and wide character utilities** <wchar.h> C99: Function names that begin with `wcs` and a lowercase letter may be added to the declarations in the `wchar.h` header. Lowercase letters may be added to the conversion specifiers and length modifiers in `fwprintf` and `fwscanf`. Other characters may be used in extensions.

**Wide character classification and mapping utilities** <wctype.h> C99: Function names that begin with `is` or `to` and a lowercase letter may be added to the declarations in the `wctype.h` header.

`fwide`<sup>C95</sup> A function that determines the orientation of the stream pointed to by `stream`.

```
#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);
```

If `mode` has a positive value, the function attempts to make the stream wide oriented. If `mode` has a negative value, the function attempts to make the stream byte oriented. Otherwise, `mode` is zero and the stream's orientation is not altered.

A positive return value indicates the stream has wide orientation; a negative return value indicates the stream has byte orientation; a zero return value indicates the stream has no orientation.

**fwprintf**<sup>C95</sup> A function that is the wide character analog of **fprintf**.

```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE * restrict stream,
             const wchar_t * restrict format, ...);
```

**fwrite** A function that writes up to **nmemb** elements each of size **size** from the array pointed to by **ptr** to the file pointed to by **stream**.

```
#include <stdio.h>
size_t fwrite(const void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

If an error occurs, the file position indicator's value is indeterminate. The value returned is the number of elements successfully written. This number may be less than **nmemb** if an error occurs. *See also* **fread**.

**fwscanf**<sup>C95</sup> A function that is the wide character analog of **fscanf**.

```
#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE * restrict stream,
            const wchar_t * restrict format, ...);
```

◇ ◇ ◇