

E

E* value macros *See* **errno** value macros.

EBCDIC An acronym for Extended Binary Coded Decimal Interchange Code. This 8-bit code can represent 256 different characters and is in common use on IBM mainframes. Standard C is not character set-specific. *See also* ASCII; Unicode.

edit mask *See* conversion specifier.

EDOM A macro, defined in `errno.h`, that is used to indicate a domain error. It expands to an implementation-defined positive integer value. *See also* **errno**.

EILSEQ^{C95} A macro, defined in `errno.h`, that is used to indicate an illegal sequence error. It expands to an implementation-defined positive integer value. *See also* **errno**; error, encoding.

element type The type of an array element. Each member of an array is an element.

#elif A preprocessor directive used as a short form for a nested **#else** with **#if**. It is used as follows:

```
#elif constant-expression
```

If *constant-expression* contains any identifiers that are not currently defined as macros, they are assumed (for this directive only) to be macros defined with value 0. In C89, *constant-expression* is evaluated as though each term had type `long int`. In C99, all signed integer types and all unsigned integer types act like the types `intmax_t` and `uintmax_t`, respectively.

ellipsis^{C89} A punctuator that is used in function declarations and definitions to indicate a variable argument list. The only standard library functions that use this notation are the **printf** and **scanf** families. The standard header `stdarg.h` declares machinery for accessing arguments in a variable argument list. The behavior is undefined if you call a function that expects a variable argument list, but you do not have a prototype containing an ellipsis in scope. The rule then, is to always **#include** `<stdio.h>` before you call **printf** and **scanf** even though these functions have an `int` return type. *See also* conversion, function arguments. C99 added the capability to have macros with a variable number of arguments. *See also* `__VA_ARGS__`.

#else A preprocessor directive used to indicate the false path of a conditional compilation directive set. Used with **#if**, **#ifdef**, **#ifndef**, **#elif**, and **#endif**. It has the following form:

```
#else
```

else statement *See if/else.*

empty statement *See null statement.*

encoding scheme A set of rules for parsing a stream of bytes into a group of coded characters. The meaning of a multibyte character can vary if the encoding scheme provides state-dependent encoding via shift sequences. Standard C requires that no encoding scheme have a byte with value zero as the second or subsequent byte of a multibyte character. This restriction allows many of the traditional string manipulation functions to be used transparently with strings containing multibyte characters.

Encoding schemes for wide characters are quite simple—all characters have the same internal width and each character has a unique value. (Note, however, that in Unicode, some characters, such as accented letters not given their own alternate encoding, are encoded as two wide characters: the letter, followed by the accent.)

Examples of encoding schemes are: ASCII, EBCDIC, EUC, JIS, Shift-JIS, ISO/IEC 10646.UCS-2 (also known as Unicode), and ISO/IEC 10646.UCS-4. *See also* `__STDC_ISO_10646__`.

#endif A preprocessor directive used to indicate the end of a conditional compilation directive set containing one or more of the following: **#if**, **#ifdef**, **#ifndef**, **#elif**, and **#else**. It has the following form:

```
#endif
```

end-of-file indicator One of the members stored in a FILE object. It is used by the standard I/O library functions in conjunction with the file system to indicate whether or not the end of file has been reached for the corresponding file. This member is accessed by `fEOF` and `clearerr`. The end-of-file indicator is cleared by `fopen`, `freopen`, `ungetc`, and `ungetwc`.

end-of-file macro *See EOF; WEOF.*

end-of-line indicator The character sequence used in a particular environment to indicate the end of a line of source or data input. For example, it might be a carriage-return/line-feed pair, or just a line-feed. In any case, C treats this indicator as a single new-line character.

entry An identifier reserved for future use as a keyword in the original definition of C. It is not part of Standard C.

enum The keyword used to specify an enumerated type and to declare identifiers for objects of that type; for example,

```
enum color {red, green = 4, blue};
enum color car_color;
```

The identifier `color` is the enumerated type tag, which is optional. The identifiers `red`, `green`, and `blue` are enumeration constants. (For convenience, a comma is permitted after the final enumeration constant.) Unless it contains an initializer, the first such constant in an enumeration has the value 0. Subsequent constants take on the value of one more than the previous constant except when overridden by an initializer. Constants within the same enumeration may have the same value, and the range of values represented need not be continuous.

enumerated type A set of related integer values, each of which is referred to as an enumeration constant. The type optionally may have a tag (and is declared somewhat like a structure or union). An object of an enumerated type maps to an implementation-defined integer type. The type checking for operations on enumerated objects is very weak. *See also* `enum`.

enumeration constant An identifier used inside the definition of an enumerated type. It is a translation-time constant expression of type `int` and may have an explicit initializer. Enumeration constants share the same name space as variables, functions, and typedef names. Unlike members defined within structures and unions, enumeration constants are never qualified with `.` and `->` operators. Instead, they can be used in any context where a translation-time constant expression is permitted (except in `#if` and `#elif` directives). *See also* `enum`.

enumeration tag *See* `tag`.

environment *See* environment list.

environment, freestanding^{C89} A system in which a program does not run under the control of an operating system. Often, freestanding programs do not have access to a command-line processor or a file system. (An operating system is a special case of such an environment.) A C89 translator for a freestanding environment must provide the headers `float.h`, `limits.h`, and `stddef.h`. A C99 translator for a freestanding environment must also provide `iso646.h`, `stdarg.h`, `stdbool.h`, and `stdint.h`. Any library facilities available to the freestanding program are implementation defined. *See also* environment, hosted; `__STDC_HOSTED__`.

environment functions The `stdlib.h` functions `abort`, `atexit`, `_Exit`, `exit`, `getenv`, and `system`.

environment, hosted^{C89} A system in which a program runs under the control of an operating system. Typically, hosted programs have access to a command-line processor and a file system. A conforming hosted C translator must implement all of the standard runtime library. *See also* environment, freestanding; `__STDC_HOSTED__`.

environment list A list of environment strings. What (if any) strings exist and how they are created or changed is implementation defined. This mechanism is often used on systems as a way to pass information (such as directory and file names) from an operating system shell or command level to an application program. Standard C provides `getenv` to access this list. *See also* `envp`.

environment variables The entries in an environment list.

environmental considerations Issues pertaining to translation and execution character sets, device character display semantics, signals and interrupts, and environmental limits.

environmental limits^{C89,C99} Minimum “capacity” requirements placed on a conforming translator. A conforming implementation must be able to handle translation units of a minimum complexity. The values shown below are in pairs with the first representing C99 and the second (in parentheses) representing C89.

According to Standard C, “The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:

- 127 (15) nesting levels of compound statements, iteration control structures, and selection control structures
- 63 (8) nesting levels of conditional inclusion
- 12 (12) pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration
- 63 (31) nesting levels of parenthesized declarators within a full declarator
- 63 (32) nesting levels of parenthesized expressions within a full expression
- 63 (31) significant initial characters in an internal identifier or a macro name
- 31 (6) significant initial characters in an external identifier
- 4095 (511) external identifiers in one translation unit
- 511 (127) identifiers with block scope declared in one block

- 4,085 (1,024) macro identifiers simultaneously defined in one translation unit
- 127 (31) parameters in one function definition
- 127 (31) arguments in one function call
- 127 (31) parameters in one macro definition
- 127 (31) arguments in one macro invocation
- 4,095 (509) characters in a logical source line
- 4,095 (509) characters in a character-string literal or wide-string literal (after concatenation)
- 65,535 (32,767) bytes in an object (in a hosted environment only)
- 15 (8) nesting levels for headers
- 1,023 (257) `case` labels for a `switch` statement (excluding those for any nested `switch` statements)
- 1,023 (127) members in a single structure or union
- 1,023 (127) enumeration constants in a single enumeration
- 63 (15) levels of nested structure or union definitions in a single structure declaration list”

Standard C also requires an implementer to document the integer and floating type properties in `limits.h` and `float.h`.

envp An array of pointers to the strings specified as environment variables. It is commonly used as the third argument passed to `main`. `envp` is not defined by Standard C but is a permitted extension.

EOF A macro, defined in `stdio.h`, that is the negative integer constant expression returned by numerous functions to indicate an end-of-file condition. *See also* end-of-file indicator.

equality The testing of two expressions to see if their values are equal. This requires the `==` operator, which should not be confused with the assignment operator `=`.

equality operator A binary operator, `==`, that compares the values of its two operands for equality. Both operands must have scalar types and must be either both arithmetic, both pointers to qualified or unqualified versions of compatible types, one a pointer to `void` and the other a pointer to an object or incomplete type, or one a pointer and the other the null pointer constant. The order of evaluation of the operands is unspecified. The result has type `int` and value 0 (if false) or 1 (if true). This operator associates left to right. Note that the equality operator must not be confused with the assignment operator `=`. The two are

easily confused and can be interchanged syntactically. However, their semantics are quite different. For example, in

```
if (a = b) /* ... */
```

the value of **b** is assigned to **a** and the value of **a** is tested for truth. In the following:

```
if (a == b) /* ... */
```

the values of **a** and **b** are compared for equality.

equals punctuator A punctuator used in initializers in object definitions and enumeration constants.

ERANGE A macro, defined in `errno.h`, that is used to indicate a range error. It expands to an implementation-defined positive integer value. *See also* `errno`.

`erf[f|l]`^{C99} A function that computes the error function of **x**.

```
#include <math.h>
double erf(double x);
float erff(float x);
long double erfl(long double x);
```

where the error function is defined by $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$.

`erfc[f|l]`^{C99} A function that computes the complementary error function of **x**.

```
#include <math.h>
double erfc(double x);
float erfcf(float x);
long double erfcf(long double x);
```

where the complementary error function is defined by $1 - \text{erf}(x)$; that is, $1 - \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt$. A range error occurs if **x** is too large.

errno A modifiable lvalue that is declared in `errno.h` and designates the memory location via which the standard library can communicate error values. Historically, `errno` has been declared using `extern int errno`; however, Standard C permits it to be a macro. There are only three standard values defined for `errno`: `EDOM`, `EILSEQ`, and `ERANGE`.

errno is cleared (set to 0) during program startup; however, no library routine is required to clear **errno**. Therefore, it is the programmer's responsibility to clear **errno** each time immediately before calling a library routine that is documented as having the ability to set it.

See also **errno** value macros; **perror**; **strerror**.

errno value macros Macros that define valid values for **errno**. They are required to be named **E*** where ***** starts with an uppercase letter or digit. C89 defined only two such macros: **EDOM** and **ERANGE**; C95 added **EILSEQ**. Other implementation-defined macros of this form may exist. *See also* future library directions.

errno.h A header that contains definitions or declarations for the following identifiers:

<i>Name</i>	<i>Purpose</i>
EDOM	Argument out of domain
EILSEQ ^{C95}	Illegal sequence indicator
ERANGE	Result out of range
errno	Error number storage location

It also may contain other implementation-defined macros (named **E***) against which the value of **errno** can be compared. *See also* future library directions.

#error^{C89} A preprocessor directive used to display a message on **stderr** and to terminate translation. It is used as follows:

```
#error [preprocessor-tokens]
```

error, domain *See* domain error.

error, encoding^{C95} An error that occurs if the character sequence presented to **mbrtowc** (when it is called directly, or indirectly from some other library function) does not form a valid multibyte character, or if the value passed to the underlying **wcrtomb** does not correspond to a valid multibyte character. When an encoding error occurs within the wide-character and byte I/O functions, they store the value of the macro **EILSEQ** in **errno**.

error handling functions The **stdio.h** functions **clearerr**, **feof**, **ferror**, and **perror**.

error indicator One of the members stored in a **FILE** object. It is used by the standard I/O library functions in conjunction with the file system to indicate whether or not a read/write error has occurred on the corresponding file. *See also* **clearerr**; **ferror**.

error, range *See* range error.

escape sequence A sequence of source characters used to represent the code for one logical character. While displayable characters (such as A, b, ?, and +) may be represented as themselves directly, nondisplayable characters (such as backspace and new-line) must be represented using an escape sequence (such as `\b` and `\n`, respectively). Displayable characters can also be represented using either their octal or hexadecimal internal representation. The following table lists the escape sequences and their meanings:

<i>Sequence</i>	<i>Meaning</i>
<code>\"</code>	Double quote
<code>\'</code>	Single quote
<code>\0</code>	Null character
<code>\?^{C89}</code>	Question mark
<code>\\</code>	Backslash
<code>\a^{C89}</code>	Terminal alert
<code>\b</code>	Backspace
<code>\ddd</code>	Octal value <i>ddd</i>
<code>\f</code>	Form-feed
<code>\n</code>	New-line
<code>\r</code>	Carriage return
<code>\t</code>	Horizontal tab
<code>\v</code>	Vertical tab
<code>\xh..h</code>	Hexadecimal value <i>h..h</i>

See also future language directions.

EUC A scheme commonly used to encode Japanese text in multibyte characters. It is an abbreviation for Extended UNIX Code. *See also* JIS; Shift-JIS.

evaluation The act of performing an operation on one or more operands to produce a value (e.g., `a + b`), a designator (e.g., `*pc`), a side effect (e.g., `++i`), or a combination thereof (e.g., `*pc = g(a + b++)`). *See* order of evaluation.

exception *See* signal.

exclusive OR assignment operator *See* OR assignment operator, bitwise exclusive.

exclusive OR operator *See* OR operator, bitwise exclusive.

exit A function that causes normal program termination to occur.

```
#include <stdlib.h>
void exit(int status);
```

First, the functions registered by `atexit` are called in the reverse order of their registration. Then, all open output files are flushed, open files are closed, and temporary files created by `tmpfile` are removed. Control is then returned to the host environment, to which it is given the exit code `status`. *See also* `abort`.

_Exit^{C99} A function that causes normal program termination to occur.

```
#include <stdlib.h>
void _Exit(int status);
```

Unlike with `exit`, no functions registered by `atexit` or signal handlers registered by `signal` are called. The `status` argument has the same meaning as in `exit`. It is implementation defined as to whether open streams are flushed or closed, or temporary files are removed.

exit code The `int` value returned by a user program to its calling environment when it returns from `main`, or from a call to `exit` or `abort`. The code's meaning is implementation defined except that zero and the object-like macro `EXIT_SUCCESS` signify success while `EXIT_FAILURE` indicates failure. Prior to C89, a value of 0 did not mean "success" on some systems.

Prior to C99, the exit code was undefined if you dropped through the outermost closing brace of `main`. However, C99 requires this to be equivalent to `return 0`; Explicitly `returning` from `main` without specifying a value results in an undefined exit code.

EXIT_FAILURE^{C89} A macro, defined in `stdlib.h`, that is used as the implementation-defined failure exit code value that can be used with `exit` or `abort`. It expands to an integer expression (that is not necessarily constant). *See also* `EXIT_SUCCESS`.

EXIT_SUCCESS^{C89} A macro, defined in `stdlib.h`, that is used as the implementation-defined success exit code value that can be used with `exit`. It expands to an integer expression (that is not necessarily constant). *See also* `EXIT_FAILURE`.

exp[f|l] A function that computes the exponential function of its argument `x`.

```
#include <math.h>
double exp(double x);
float expf(float x);
long double expl(long double x);
```

If the magnitude of the argument is too large, a range error occurs.

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

`exp2[f|l]`^{C99} A function that computes the base-2 exponential function of its argument `x`.

```
#include <math.h>
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

If the magnitude of the argument is too large, a range error occurs.

`explicit` 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 `explicit` as an identifier in new C code you write.

explicit conversion A conversion achieved by using a cast operator.

`expm1[f|l]`^{C99} A function that computes the base- e minus 1 exponential function of its argument `x`; that is, $e^x - 1$.

```
#include <math.h>
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

If the value of the argument is positive and too large, a range error occurs.

exponential and logarithmic functions The `math.h` functions `exp`, `exp2`, `expm1`, `frexp`, `ldexp`, `log`, `log10`, `log1p`, `log2`, and `modf`, and their `float` and `long double` counterparts for both floating and complex types.

`export` 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 `export` as an identifier in new C code you write.

expression A valid sequence of operators and operands that specifies how to compute a value (e.g., `a + b`), how to generate side effects (e.g., `f()`, `++i`, or `j--`), or both (e.g., `a + g() + ++k`).

expression, full An expression that is not part of another expression. There is a sequence point at the end of a full expression. Each of the following is a full expression: an initializer; the expression in an expression statement; the controlling expression of an `if`, `switch`, `while`, or `do` statement; each of the three (optional) expressions of a `for` statement; and the (optional) expression in a `return` statement.

expression, parenthesized Any expression contained within parentheses. The type and value of the parenthesized expression is the same as the type and value of the same expression without parentheses. A parenthesized expression is a primary expression. If expression *exp* is an lvalue, function designator, or void expression, then `(exp)` is, respectively, an lvalue, function designator, or void expression. This gives rise to the following correct, but odd-looking, expressions:

```
((i)) = 6
((printf))("Hello")
```

expression statement An expression statement has the form

```
expression ;
```

Such an expression is a full expression. Most C statements are expression statements, as in the following examples:

```
i++;
f(i, j, k);
a = b + c - g();
x += y;
```

The following are also acceptable (but vacuous) expression statements; it is a quality-of-implementation issue whether an implementation identifies such vacuous statements or generates code for them:

```
i;
10 + j * k;
*pc + 3;
```

extended integer types^{C99} Signed and unsigned integer types beyond those required by the standard. An implementation may support these, in which case they are subject to the rules of the standard with respect to signed and unsigned type behavior.

extern A keyword used to indicate a storage class that signifies an identifier is defined, either later in the same translation unit or in a separate trans-

lation unit. If a function declaration does not include a class keyword, **extern** is assumed.

external definition A non-**static** function definition or a non-**static** variable definition outside of a function definition. *See also* scope, file.

external name An identifier exported to the development environment. It may be visible to tools such as assemblers, linkers, object libraries, and debuggers; and its length of significance and case distinction may be limited. (C89 permits external name significance to be as few as 6 characters and to be monospace only, while C99 requires at least 31 characters of significance.) Also, underscores might be mapped to some other character. *See also* internal name; future language directions.

external object definition The defining instance of an external object. *See also* definition, tentative.

