

M

macro An identifier associated with a sequence of tokens (called its replacement list) by the preprocessor directive `#define`. During preprocessing, each occurrence of a macro name is replaced by its corresponding definition. As such, a macro is simply an abbreviation for its text value. Sometimes, a macro is referred to as a symbolic constant because its name is typically chosen to be symbolic of its underlying definition; for example,

```
#define PI 3.1415926
#define Max_Value 100
#define Clear_Screen() printf("\033[2J")
```

Note that a macro can be defined with no value; for example,

```
#define NAME
```

defines `NAME` to have no value, in which case all occurrences of that identifier are removed from the source during preprocessing.

Because one macro's definition can involve other, previously defined macros, subtle errors can occur that are difficult to debug. Most compilers provide a command-line option to allow you to save the output from the preprocessor so you can see exactly how the macros expanded.

See also macro, function-like; macro, object-like; macro, predefined; macro, replacement; `#undef`.

macro, function-like A macro defined with an argument list, which can be empty; for example,

```
#define Clear_Screen() printf("\033[2J")
#define INTSWAP(a,b) {int t = (a); (a) = (b); (b) = t;}
#define Isdigit(c) ((c) >= '0' && (c) <= '9')
```

The `(` token must immediately follow the name of the macro being defined. However, horizontal white space may be used to separate other preprocessing tokens in the definition; for example,

```
#define M( a ) ( a )
```

defines a function-like macro with one argument while:

```
#define M ( a ) ( a )
```

defines an object-like macro with the definition (a) (a).

macro, object-like A macro defined without an argument list. This form of macro often is called a symbolic constant; for example,

```
#define PI 3.1415926
#define Max_Value 100
```

See also macro, function-like.

macro, predefined A macro automatically defined within the translator. C89 defined five such macros called `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__`, and `__STDC__`. C95 added `__STDC_VERSION__`. C99 added `__STDC_HOSTED__` as well as the conditionally defined names `__STDC_IEC_559__`, `__STDC_IEC_559_COMPLEX__`, and `__STDC_ISO_10646__`. These macros cannot be the subject of `#undef` and they cannot be redefined. Implementations often have their own predefined macros as well. For example, a compiler running under the UNIX operating system might predefine the macros `UNIX` or `BSD`. A conforming implementation, however, must spell any extra predefined macros with a leading underscore followed by another underscore or an uppercase letter.

Many compilers support a translation-time option of defining macros before processing begins. This amounts to letting the user define a predefined macro. *See also* `_cplusplus`.

macro, redefinition of^{C89} The redefining of a macro with another macro by the same name without an intervening `#undef`. A function-like macro may be redefined as a function-like macro provided all such definitions contain the same number and spelling of parameters and their replacement lists are identical. (Two replacement lists are identical if they contain the same set of tokens and the same number of sets of separating white space. The white space characters used may vary, however.) Similarly, an object-like macro may be redefined as an object-like macro provided all such definitions contain identical replacement lists. The following are valid redefinitions:

```
#define A 10 +tab5
#define A 10tab+ 5
#define A 10 + 5

#define B(a, b) a +tabb
#define B(a, b) atab+ b
#define B(a, b) a + b
```

However, given the previous definitions, the following redefinitions are invalid:

```
#define A 10+tab5
#define A 10+ 5
#define A 10+5
#define A 15

#define B(a, b) a+tabb
#define B(a, b) a+b
#define B(x, y) x + y
```

This is because the absence of white space does not match the presence of white space.

macro replacement The process by which a macro is replaced by its definition. For object-like macros, the macro name at each call is simply replaced by the replacement list from its definition. For function-like macros, each call is replaced by the corresponding replacement list, however, each formal parameter is replaced by each actual argument in the process. If a macro expands to a set of tokens that includes the name of the macro originally being expanded, it is not further replaced. Standard C requires that macro expansion not be infinitely recursive. *See also* stringize operator; token-pasting operator.

macro, safe A function-like macro whose definition only evaluates each argument once. *See also* macro, unsafe.

macro, unsafe A function-like macro can be defined so that when it is expanded, the resulting text has one or more arguments appearing multiple times; for example,

```
#define Isdigit(c) ((c) >= '0' && (c) <= '9')
```

However, invoking this macro with an argument that contains a side effect can cause unexpected behavior. For example, `Isdigit(x++)`, where `x` contains the value `'9'`, tests false because `x` would be incremented before it is tested the second time. Also, `x` would be incremented twice. As such, this macro definition is said to be unsafe. Either it must be called with arguments that have no side effects or it must be scrapped and a function version used instead.

Except where explicitly exempted, Standard C permits all library functions to exist as macros also, provided they are safe macros.

main A function required within each hosted program, that marks the program's logical entry point. (Freestanding C programs do not need to have a function called `main`.) Although `main` has special semantics, it is just another function and therefore can have arguments passed to it. `main` can even be called recursively if you can find a good reason to do

so. Standard C requires a conforming implementation to support `main` defined in both of the following ways:

```
int main(void) { /* ... */ }
int main(int argc, char *argv[]) { /* ... */ }
```

Other implementation-defined forms are permitted; for example, a common extension is to have a third argument of type `char *` called `envp`, which provides access to an array of environment variable strings much like `getenv`.

malloc A function that dynamically allocates contiguous memory of `size` bytes.

```
#include <stdlib.h>
void *malloc(size_t size);
```

The space allocated has no guaranteed initial value.

The value returned is of type `void *` and therefore is assignment compatible with any object or incomplete pointer type. Therefore, no explicit cast is needed. This value is the address of the beginning of the allocated memory and is guaranteed to be suitably aligned for use in storing any object.

If the memory cannot be allocated, `NULL` is returned.

If `size` is zero, it is implementation defined as to whether `NULL` or a unique pointer is returned.

By giving `realloc` a `NULL` first argument, `realloc` can be used to produce the same effect as `malloc` with the same size.

See also `calloc`; `free`.

manifest constant A synonym for object-like macro. Standard library examples are `NULL` and `EOF`.

`MATH_ERREXCEPT`^{C99} A macro, defined in `math.h`, that expands to the integer constant 2. *See* `math_errhandling`.

`math_errhandling`^{C99} A macro, defined in `math.h`, that expands to an expression that has type `int` and the value `MATH_ERRNO`, `MATH_ERREXCEPT` or the bitwise OR of both. The value of `math_errhandling` is constant for the duration of the program. It is unspecified whether `math_errhandling` is really a macro or an identifier with external linkage.

If the expression `math_errhandling & MATH_ERREXCEPT` can be nonzero, the implementation must define the macros `FE_DIVBYZERO`, `FE_INVALID`, and `FE_OVERFLOW` in `fenv.h`.

`MATH_ERRNO`^{C99} A macro, defined in `math.h`, that expands to the integer constant 1. See `math_errhandling`.

math.h A header that contains support for the math library functions. (Other math-related names are declared in `complex.h`, `fenv.h`, and `tg-math.h`.) C89 requires a standard-conforming implementation to support only a `double` version of each function. However, if `float` or `long double` versions are also supported, the function names have a suffix of `f` or `l`, respectively (as indicated below by the optional suffix `f` or `l`). C99 requires all three versions. `math.h` contains definitions or declarations for the following identifiers:

<i>Name</i>	<i>Purpose</i>
<code>acos[f l]</code>	Arc cosine
<code>acosh[f l]</code> ^{C99}	Arc hyperbolic cosine
<code>asin[f l]</code>	Arc sine
<code>asinh[f l]</code> ^{C99}	Arc hyperbolic sine
<code>atan[f l]</code>	Arc tan
<code>atan2[f l]</code>	Principal arc tan
<code>atanh[f l]</code> ^{C99}	Arc hyperbolic tan
<code>cbrt[f l]</code> ^{C99}	Cube root
<code>ceil[f l]</code>	Highest ceiling
<code>copysign[f l]</code> ^{C99}	Magnitude of x with sign of y
<code>cos[f l]</code>	Cosine
<code>cosh[f l]</code>	Hyperbolic cosine
<code>double_t</code> ^{C99}	Type-widening type
<code>erf[f l]</code> ^{C99}	Error function
<code>erfc[f l]</code> ^{C99}	Complementary error function
<code>exp[f l]</code>	Exponential using base e
<code>exp2[f l]</code> ^{C99}	Exponential using base 2
<code>expm1[f l]</code> ^{C99}	Exponential using base e , minus 1
<code>fabs[f l]</code>	Floating absolute
<code>fdim[f l]</code> ^{C99}	Positive difference
<code>float_t</code> ^{C99}	Type-widening type
<code>floor[f l]</code>	Lowest floor
<code>fma[f l]</code> ^{C99}	Multiply-and-add
<code>fmax[f l]</code> ^{C99}	Maximum value
<code>fmin[f l]</code> ^{C99}	Minimum value
<code>fmod[f l]</code>	Floating remainder
<code>FP_FAST_FMA</code> ^{C99}	Fma macro, double
<code>FP_FAST_FMAF</code> ^{C99}	Fma macro, float
<code>FP_FAST_FMAL</code> ^{C99}	Fma macro, long double
<code>FP_ILOGB0</code> ^{C99}	<code>ilogb</code> return value

<i>Name</i>	<i>Purpose</i>
FP_ILOGBNAN ^{C99}	ilogb return value
FP_INFINITE ^{C99}	Number classification macro
FP_NAN ^{C99}	Number classification macro
FP_NORMAL ^{C99}	Number classification macro
FP_SUBNORMAL ^{C99}	Number classification macro
FP_ZERO ^{C99}	Number classification macro
fpclassify ^{C99}	Classifies a value
frexp[f l]	Breakdown floating number
HUGE_VAL	Huge floating value of type double
HUGE_VALF ^{C99}	Huge floating value of type float
HUGE_VALL ^{C99}	Huge floating value of type long double
hypot[f l] ^{C99}	Hypotenuse
ilogb[f l] ^{C99}	Extracts an exponent
INFINITY ^{C99}	Positive infinity of type float
isfinite ^{C99}	Test if finite
isgreater ^{C99}	Test if greater
isgreaterequal ^{C99}	Test if greater or equal
isinf ^{C99}	Test if infinite
isless ^{C99}	Test if less
islessequal ^{C99}	Test if less or equal
islessgreater ^{C99}	Test if less or greater
isnan ^{C99}	Test if Not-a-Number
isnormal ^{C99}	Test if normal
isunordered ^{C99}	Test if unordered
ldexp[f l]	Load exponent
lgamma[f l] ^{C99}	Natural log of gamma
llrint[f l] ^{C99}	Rounding function
llround[f l] ^{C99}	Rounding function
log[f l]	Base <i>e</i> logarithm
log10[f l]	Base 10 logarithm
log1p[f l] ^{C99}	Base <i>e</i> logarithm of 1 plus arg
log2[f l] ^{C99}	Base 2 logarithm
logb[f l] ^{C99}	Extracts an exponent
lrint[f l] ^{C99}	Rounding function
lround[f l] ^{C99}	Rounding function
MATH_ERREXCEPT ^{C99}	math_errhandling flag
math_errhandling ^{C99}	Math error flags value
MATH_ERRNO ^{C99}	math_errhandling flag
modf[f l]	Breakdown floating number
NAN ^{C99}	Not-a-Number of type float
nan[f l] ^{C99}	Generate a NaN value
nearbyint[f l] ^{C99}	Rounding function
nextafter[f l] ^{C99}	Manipulation function
nexttoward[f l] ^{C99}	Manipulation function

<i>Name</i>	<i>Purpose</i>
<code>pow[f l]</code>	Power
<code>remainder[f l]^{C99}</code>	IEEE remainder
<code>remquo[f l]^{C99}</code>	Remainder function
<code>rint[f l]^{C99}</code>	Rounding function
<code>round[f l]^{C99}</code>	Rounding function
<code>scalbln[f l]^{C99}</code>	Scale-by function
<code>scalbn[f l]^{C99}</code>	Scale-by function
<code>signbit^{C99}</code>	Tests sign bit
<code>sin[f l]</code>	Sine
<code>sinh[f l]</code>	Hyperbolic sine
<code>sqrt[f l]</code>	Square root
<code>tan[f l]</code>	Tan
<code>tanh[f l]</code>	Hyperbolic tan
<code>tgamma[f l]^{C99}</code>	Gamma function
<code>trunc[f l]^{C99}</code>	Rounding function

See also future library directions.

MB_CUR_MAX^{C89} A macro, defined in `stdlib.h`, that expands to a positive integer expression of type `size_t` whose value is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category `LC_CTYPE`) and whose value is never greater than `MB_LEN_MAX` (defined in `limits.h`).

mblen^{C89} A function that computes the number of bytes in the multibyte character pointed to by `s`.

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

See also `mbtowc`.

MB_LEN_MAX^{C89} A macro, defined in `limits.h`, that designates the maximum number of bytes in a multibyte character for any supported locale. (It must be at least 1.) This macro expands to an integer constant expression suitable for use with a `#if` directive. *See also* `MB_CUR_MAX`.

mbrlen^{C95} A function that is equivalent to the call

```
mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)
```

where `internal` is `mbrlen`'s `mbstate_t` object, except that the expression `ps` is evaluated only once.

```
#include <wchar.h>
size_t mbrlen(const char * restrict s, size_t n,
              mbstate_t * restrict ps);
```

The value returned is between zero and `n`, inclusive, `(size_t)(-2)`, or `(size_t)(-1)`.

`mbrtowc`^{C95} A function that is a restartable version of `mbtowc`.

```
#include <wchar.h>
size_t mbrtowc(wchar_t * restrict pwc,
               const char * restrict s, size_t n,
               mbstate_t * restrict ps);
```

`mbsinit`^{C95} A function that determines whether the pointed-to `mbstate_t` object describes an initial conversion state, if `ps` is non-null.

```
#include <wchar.h>
int mbsinit(const mbstate_t *ps);
```

The return value is nonzero if `ps` is null or if the pointed-to object describes an initial conversion state; otherwise, it is zero.

`mbsrtowcs`^{C95} A function that is a restartable version of `mbstowcs`.

```
#include <wchar.h>
size_t mbsrtowcs(wchar_t * restrict dst,
                 const char ** restrict src,
                 size_t len, mbstate_t * restrict ps);
```

`mbstate_t`^{C95} An object type, defined in `wchar.h`, other than an array type, that can hold the conversion state information necessary to convert between sequences of multibyte characters and wide characters.

`mbstowcs`^{C89} A function that converts a sequence of multibyte characters into a string of corresponding wide characters.

```
#include <stdlib.h>
int mbstowcs(wchar_t * restrict pwcs,
             const char * restrict s, size_t n);
```

`mbtowc`^{C89} A function that computes the number of bytes in the multibyte character pointed to by `s`. It then determines the code for a value of type `wchar_t` that corresponds to that multibyte character.

```
#include <stdlib.h>
int mbtowc(wchar_t * restrict pwc,
           const char * restrict s, size_t n);
```

mem prefix *See* future library directions.

member An identifier declared as part of a structure or union layout. Such members are referenced using the dot or arrow operator.

member selection operators *See* arrow operator; dot operator.

memchr A function that searches the first *n* characters of a string *s* for a character *c*.

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

If *c* is found, **memchr** returns a pointer to that location within *s*, otherwise, it returns `NULL`. *c* is converted to `unsigned char` before the search begins. *See also* `wmemchr`.

memcmp A function that compares *n* characters at the location pointed to by *s2* to the characters at the location pointed to by *s1*.

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

An integer less than, equal to, or greater than zero is returned to indicate whether the first *n* characters starting at *s1* have binary values less than, equal to, or greater than, respectively, those starting at *s2*. *See also* `wmemcmp`.

memcpy A function that copies *n* characters from the location pointed to by *s2* to the characters at the location pointed to by *s1*.

```
#include <string.h>
void *memcpy(void * restrict s1, const void * restrict s2,
             size_t n);
```

The value of *s1* is returned. If the objects located at *s1* and *s2* overlap, the behavior of **memcpy** is undefined. To copy overlapping objects, use `memmove` instead. *See also* `wmemcpy`.

memmove^{C89} A function that copies *n* characters from the location pointed to by *s2* to the characters at the location pointed to by *s1*.

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

The value of *s1* is returned. **memmove** works correctly even if the objects located at *s1* and *s2* overlap. If the objects are known to not overlap, it may be more efficient to use `memcpy` instead. *See also* `wmemmove`.

memory management functions The `stdlib.h` functions `calloc`, `free`, `malloc`, and `realloc`.

memset A function that sets the first `n` characters of the object pointed to by `s` to the value `c`.

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

`c` is converted to an `unsigned char` before the copying begins. The value returned is `s`. *See also* `wmemset`.

minus operator, unary *See* unary minus operator.

mktime A function that converts a broken-down time (as defined in the type `struct tm` type), in the structure pointed to by `timeptr`, into a calendar time.

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

The value returned is the calendar time resulting from the conversion. If this time cannot be represented, the value returned is `(time_t)(-1)`. All the members required to be in the `tm` structure (except `tm_wday` and `tm_yday`) must be initialized. When `mktime` completes successfully, these two members have been filled in.

modf[f|l] A function that breaks a floating-point number into integer and fractional parts.

```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

Each part has the same sign as the original number. The integer part is stored in the location pointed to by `iptr`, and the fractional part is the return value.

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

modifiable lvalue^{C89} *See* `lvalue`; `lvalue`, `modifiable`; `lvalue`, `non-modifiable`.

modulus An often-used but incorrect name for the remainder when performing division. For integer remainder, *see* `remainder operator`; for floating-point remainder, *see* `modf`.

mon_decimal_point^{C89} An `lconv` structure member that is a pointer to a string containing the decimal point used to format monetary quantities. If the string consists of "", the value is not available in the current locale or is of zero length. In the "C" locale this member must have the value "".

mon_grouping^{C89} An `lconv` structure member that is a pointer to a string whose elements indicate the size of each group of digits in formatted monetary quantities. If the string consists of "", this indicates that the value is not available in the current locale or is of zero length. In the "C" locale this member must have the value "". The elements are interpreted as follows:

CHAR_MAX No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that make up the current group. The next element is examined to determine the size of the next group of digits before the current group.

mon_thousands_sep^{C89} An `lconv` structure member that is a pointer to a string containing the separator for groups of digits before the decimal point in formatted monetary quantities. If the string consists of "", the value is not available in the current locale or is of zero length. In the "C" locale this member must have the value ".".

multibyte character The result when a character from a writing system such as Chinese, Japanese, and Korean, whose characters cannot be represented in what C refers to as a byte, is encoded into one or more bytes. (Strictly speaking, since a multibyte character can have a single byte only, ASCII and EBCDIC are really multibyte character sets in which each character can be represented in a single byte.) The method of encoding the bytes is locale dependent; commonly used encoding schemes for Japanese text are EUC, JIS, and Shift-JIS. A multibyte character is stored as an array of one or more `char`, so a function taking a multibyte character argument by address or returning one by address, uses `char *` to describe that type. In fact, most library functions accepting or returning this type really traffic in multibyte characters. Multibyte characters must not be confused with wide characters. *See also* `MB_CUR_MAX`; `MB_LEN_MAX`; `mblen`; `mbtowc`; `wctomb`.

multibyte character functions The `stdlib.h` functions `mblen`, `mbtowc`, and `wctomb`, and the `wchar.h` functions `mbrtowc` and `wcrtomb`. The behavior of these functions is subject to the current locale, in particular, to the `LC_CTYPE` category.

multibyte string A string containing one or more multibyte characters. *See also* `mbstowcs`; `wcstombs`.

multibyte string functions The `stdlib.h` functions `mbstowcs` and `wcstombs`, and the `wchar.h` functions `mbsrtowcs` and `wcsrtoombs`. The behavior of these functions is subject to the current locale, in particular, to the `LC_CTYPE` category.

multiplication assignment operator A binary operator, `*=`, that permits multiplication and assignment to be combined so that `exp1 *= exp2` is equivalent to `exp1 = exp1 * exp2`, except that in the former, `exp1` is only evaluated once. The order of evaluation of the operands is unspecified. Both operands must have arithmetic type, and the left operand must be a modifiable lvalue. The type of the result is the type of `exp1`. This operator associates right to left. *See also* assignment operator, compound.

multiplication operator A binary operator, `*`, that causes the values of its operands to be multiplied together. The order of evaluation of the two operands is unspecified. Both operands must have arithmetic type. The usual arithmetic conversions are performed on the operands. This operator associates left to right.

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

