

# D

`__DATE__` A predefined macro that expands to a string containing the date of compilation in the form "Mmm dd yyyy". If the day is less than 10, its first position is a space. Examples are `Mar 2 1991` and `Mar 22 1991`. If the date is not available, an implementation-defined valid date is supplied. This macro can be used in any context where a string literal is permitted or required, for example,

```
char date[] = __DATE__;  
printf("%s", __DATE__);
```

And because adjacent string literals are concatenated, the following is permitted:

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

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

This macro cannot be the subject of `#undef`.

**date and time header** *See* `time.h`.

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

`DBL_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 `double` type.

`DBL_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 `double` value.

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

`DBL_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.

`DBL_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.

- DBL\_MIN**<sup>C89</sup> A macro, defined in `float.h`, that designates the minimum normalized positive `double` number.
- DBL\_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.
- DBL\_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.
- DECIMAL\_DIG**<sup>C99</sup> A macro, defined in `float.h`, that specifies the number of decimal digits such that any floating-point number in the widest supported floating type can be rounded to a floating-point number having that many decimal digits and back again without change to the value.

**decimal constant** *See* constant, integer.

**decimal\_point**<sup>C89</sup> An `lconv` structure member that is a pointer to a string containing the decimal-point character used to format nonmonetary quantities. In the "C" locale this member must have the value `"."`. *See also* `mon_decimal_point`.

**decimal point** *See* radix point.

**declaration** A construct that declares the attributes of one or more identifiers. A definition (such as `static int i;`) is also a declaration, but a declaration (such as `extern int j;`) is not necessarily a definition. Prior to C99, declarations could optionally occur at the start of any block, prior to any statements. C99 permits them to occur after statements, so long as they occur prior to their first use. They also may occur outside of function definitions. A declaration must be terminated with a semicolon.

**declaration specifiers** Those parts of a declaration excluding any initializer; that is, the storage class specifier, type specifier, and type qualifier.

**declarator** The portion of a declaration that declares one identifier. For example, the declaration `int i, j[5], k(void);` contains three declarators: for `i`, `j`, and `k`. *See also* declaration.

**declarator, full**<sup>C99</sup> A declarator that is not part of another declarator. A sequence point exists at the end of a full declarator.

**declarator, punctuation in** Those punctuation characters permitted in declarators:

`()` is used in two different ways: to indicate the derived type "function returning" and to indicate precedence. For example, `int (*fp)(void)`,

declares `fp` to be a pointer to a function having no arguments. Without the grouping parentheses, we would have `int *fp(void)`, in which case `fp` would be a function having no arguments and returning a pointer to an `int`.

`*` is used to indicate the derived type “pointer to” as in `int *pi` and `long (*pf)(int)`, for example.

`[ ]` is used to indicate the derived type “array of.”

**declarator type derivation** *See* derived type.

**decrement operator** A unary operator, `--`, that may be used as either a prefix or postfix operator. The operand must have a scalar type and must be a modifiable lvalue. The value of `x--` is the value of `x` before it is decremented by 1, whereas the value of `--x` is the value of `x` after it is decremented by 1. The postfix version of this operator associates left to right while the prefix version associates right to left. (Prior to C89, the prefix and postfix versions had the same precedence. However, C89 elevated the precedence of the postfix version. This broke no correct existing code. It did, however, permit previously invalid constructs to be valid. For example, `p--->m` is now valid.) There is a corresponding increment operator `++`.

**default** A keyword used as a special label, but only in the context of a `switch` statement. A `switch` passes control to the `default` label if the controlling expression does not match any of the associated case label values. You cannot `goto` a `default` label because `goto` can be used only to transfer control to user-defined labels. *See also* labeled statement.

**default argument promotions** *See* conversion, function arguments.

**#define** A preprocessor directive used to define an object-like or function-like macro. *See also* `defined`; `#ifdef`; `#ifndef`; `#undef`. It is used as follows:

```
#define identifier [replacement-list]
#define identifier( [identifier-list] ) \
    [replacement-list]
```

**defined** A preprocessor operator that makes complex conditional compilation selection criteria much simpler to read and write. It is used to test if the macro specified as its operand is defined, and evaluates to 1 on true and 0 on false, for example,

```
#if defined M1 && !defined M2
    /* ... */
#endif
```

is equivalent to:

```
#ifndef M1
    #ifndef M2
        /* ... */
    #endif
#endif
```

Optional parentheses are permitted around the macro name operand of the **defined** operator.

Note that **defined** is not a keyword. It only has a special meaning when used in the context of a **#if** or **#elif** directive. In all other contexts this name may be safely used as an identifier.

**defined** cannot be the subject of a **#undef** or **#define** directive.

**definition** A declaration that causes storage to be reserved for an object or function named by an identifier. A definition (such as **static int i;**) is also a declaration, but a declaration (such as **extern int j;**) is not necessarily a definition.

**definition, tentative**<sup>C89</sup> A definition that might be augmented by another definition for the same identifier, later in the same translation unit. A declaration of an identifier (that names an object) having file scope and no initializer, and without a storage class specifier or the specifier **static**, is a tentative definition.

**delete** 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 **delete** as an identifier in new C code you write.

**deprecated** Synonym for *obsolescent*.

**dereference** *See* *indirection*.

**derived type** A type derived from another type. For example, **char \*[10]** is derived from the type **char \***, which is in turn derived from the base type **char**. There are only three ways to derive a type from another type:  $T2$  is an array of  $T1$ ;  $T2$  is a pointer to  $T1$ ; and  $T2$  is a function returning  $T1$ . Not all possible derivations are permitted. For example, a function cannot return an array or a function, and you cannot have an array of functions, as indicated by the following table.

**Valid Derived Type Combinations**

<i>Derived type</i>	Pointer	Array	Function
Pointer to	yes	yes	yes
Array of	yes	yes	no
Function returning	yes	no	no

**designated initializer**<sup>C99</sup> An initializer for an array that can be written such that the numbers of the elements to be initialized explicitly are specified, as in the following:

```
enum Color {RED, GREEN, BLUE};
int counts[] = {[BLUE] = 10, [GREEN] = 15, [RED] = 20};
```

In the previous example, the element number is specified as an integer constant expression, as in `[BLUE] = 10`. The initializer list may be incomplete, as follows:

```
int values[10] = {1, 3, 5, [7] = 2, 4, 6};
```

In this case, elements 0, 1, 2, 7, 8, and 9 are given explicit values, while the elements 3, 4, 5, and 6 take on the value zero.

Members of a union or structure can also be designated explicitly, allowing a union to be initialized through any member (not just the first) and allowing a structure's member initializers to be specified in any order. An example is:

```
struct Point
{
    int x;
    int y;
};

struct Point p1 = {.y = 5, .x = 7};
```

Designated initializers can be used with compound literals.

**diagnostic message** An error message. A standard-conforming implementation must issue at least one diagnostic message for each translation unit containing one or more syntax rule or constraint violations. The format of any such message(s) is implementation defined. Note, though, that formal validation and procurement agencies may impose further requirements here. For example, in the U.S., FIPS validation requires that such messages indicate the name of the translation unit and as near as possible, the offending source line number.

Strictly speaking, an implementation that never issues more than one message per compilation (where the text of that message is “Yes, there’s at least 1 error there.”) could be a conforming implementation; however, marketplace forces likely would encourage the vendor to provide more and better quality messages.

**diagnostics header** See `assert.h`.

**difftime**<sup>C89</sup> A function that computes the difference between two calendar times using `time1 - time0`.

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

The return value indicates the number of seconds separating the two times.

**digraph**<sup>C95</sup> An alternate spelling for a source token that cannot be written solely using characters in the ISO-646 character set. The following digraphs are defined:

<b>Digraphs</b>	
<i>Spelling</i>	<i>Equivalent Token</i>
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

**direct I/O functions** The `stdio.h` functions `fread` and `fwrite`.

**div**<sup>C89</sup> A function that computes the quotient and remainder when `numer` is divided by `denom`.

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behavior is undefined. The value returned has the structure type `div_t`, which contains the two `int` members, `quot` and `rem`, in either order. See also `ldiv`; `lldiv`.

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

Prior to C99, if either operand is negative the behavior is implementation defined. That is, the result could be the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient. In such cases, C99 requires truncation toward zero.

*See also* assignment operator, compound.

**division, integer, with negative values** *See* `div`; division operator; `ldiv`; `lldiv`.

**division operator** A binary operator, `/`, that causes the value of its left operand to be divided by the value of its second. Both operands must have arithmetic type. The order of evaluation of the operands is unspecified. The usual arithmetic conversions are performed on the operands. This operator associates left to right.

Prior to C99, if either operand is negative the behavior is implementation defined. That is, the result could be the largest integer less than or equal to the algebraic quotient or the smallest integer greater than or equal to the algebraic quotient. In such cases, C99 requires truncation toward zero.

*See also* `div`; `ldiv`; `lldiv`.

`div_t`<sup>C89</sup> A type, defined in `stdlib.h`, that is a structure type used as the implementation-defined return type of the `div` function. One possible definition for `div_t` is:

```
typedef struct {
    int quot;
    int rem;
} div_t;
```

The ordering of the members does not need to be specified. *See also* `ldiv_t`; `lldiv_t`.

**domain error** An error that occurs if an argument input to a math function is outside the domain over which that function is defined. In this case, an implementation-defined value is returned, and, in C89, `errno` is set to the macro `EDOM`. C99 changed things slightly. Specifically, if `math_errhandling & MATH_ERRNO` is nonzero, `errno` is set to `EDOM`; if `math_errhandling & MATH_ERREXCEPT` is nonzero, the “invalid” floating-point exception is raised.

**dot operator** A binary operator, `.`, that is used to select the right operand from the structure or union designated by the left operand. The order of evaluation of the operands is unspecified. The type of the left operand must be structure or union, and the right operand must be the name of

a member in that structure or union. This operator produces an lvalue if the left operand is an lvalue. (One example where the expression is not an lvalue is `f().m`.) A dot expression can almost always be rewritten using the arrow operator. For example, `s.m` is equivalent to `(&s)->m`. Similarly, if `s` is a structure or union having storage class `register`, `s.m` is permitted while `(&s)->m` is not because you cannot take the address of a register variable. The type of the result of the dot operator is the type of the named member. This operator associates left to right.

**double** A keyword used for one of the three floating-point types. (The other two are `float` and `long double`.) Traditionally, `double` meant “double precision” while `float` implied “single precision.” However, `double` is permitted to have the same precision as `float`. Some compilers accept `long float` as a synonym for `double`; however, this is not permitted by Standard C. *See also* floating type.

`double _Complex`<sup>C99</sup> *See* `complex`.

`double _Imaginary`<sup>C99</sup> *See* `imaginary`.

**double quote escape sequence** An escape sequence, `\"`, which represents the double quote character. A double quote character `"` can be included in a string literal only in its escape sequence form. However, in a character constant, it can occur either as `'\"'` or `'\"'`.

`double_t`<sup>C99</sup> A type, defined in `math.h`, that is a floating type and is at least as wide as `double` and `float_t`. If the macro `FLT_EVAL_METHOD` evaluates to 0 or 1, `double_t` is `double`. If `FLT_EVAL_METHOD` evaluates to 2, `double_t` is `long double`. For other values of `FLT_EVAL_METHOD`, its exact type is implementation defined.

**double type conversion** *See* `conversion`, usual arithmetic.

**do/while** Two keywords used to implement a loop that executes at least once. The general form is

```
do
    statement
while ( expression );
```

The loop body *statement* is executed, then *expression* is evaluated. If it evaluates to true, this process is repeated. *expression* is a full expression.

**dynamic\_cast** 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 `dynamic_cast` as an identifier in new C code you write.

