

T

tag The identifier that may optionally follow the keyword `struct`, `union`, or `enum` in a structure, union, or enumerated type definition, respectively. The tag is used later to refer to that particular type; for example, in a function prototype or object definition. Structure, union, and enumeration type tags have their own name space. If the tag is omitted, all identifiers having the resultant unknown structure, union, or enumerated type must be declared at that time because that same unknown type cannot be referenced later; for example,

```
struct {
    int i;
    double d;
} sa, sb;

struct {
    int i;
    double d;
} s1, s2;
```

By definition, `sa` and `sb` have the same unknown structure type. Likewise, `s1` and `s2` have the same unknown structure type. However, the two unknown structure types are different; they are also not assignment compatible. Objects (and pointers to such objects) of such tagless types cannot be passed to nor returned from functions because there is no way to declare the unknown type in a prototype. (One exception is that a pointer to such an unknown type is assignment compatible with a `void *`.)

tag compatibility The compiler can assume that if two different parameters to a function definition are pointers to objects having different structure types, then those pointers cannot be aliases to the same object.

tan[f|l] A function that computes the tangent of its argument `x` (measured in radians).

```
#include <math.h>
double tan(double x);
float tanf(float x);
long double tanl(long double x);
```

If the magnitude of the argument is large, `tan` may produce a result with little or no significance.

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

`tanh[f|l]` A function that computes the hyperbolic tangent of its argument.

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

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

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

tentative definition^{C89} *See* definition, tentative.

termination, abnormal *See* `abort`.

termination, normal *See* `exit`.

termination, program *See* program termination.

ternary operator *See* operator, ternary.

text stream *See* stream, text.

`tgamma[f|l]`^{C99} A function that computes the gamma function of `x`.

```
#include <math.h>
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

If `x` is a negative integer or if the result cannot be represented when `x` is zero, a domain error occurs. A range error might occur if the magnitude of `x` is too large or too small. The value returned is $\Gamma(x)$.

tgmath.h^{C99} A header that includes the standard headers `math.h` and `complex.h`; it also defines a number of type-generic macros.

Except for `modf`, the functions without the suffix `f` or `l` declared in `math.h` and `complex.h` have a corresponding type-generic macro defined in this header. For example, the math function `sin` and complex function `csin` have a corresponding generic macro called `sin`. When the macro

is used, it expands to a function whose argument types are determined by those passed to the macro.

The complete set of type-generic macro names is: `acos`, `acosh`, `asin`, `asinh`, `atan`, `atan2`, `atanh`, `carg`, `cbprt`, `ceil`, `cimag`, `conj`, `copysign`, `cos`, `cosh`, `cproj`, `creal`, `erf`, `erfc`, `exp`, `exp2`, `expm1`, `fabs`, `fdim`, `floor`, `fma`, `fmax`, `fmin`, `fmod`, `frexp`, `hypot`, `ilogb`, `ldexp`, `lgamma`, `llrint`, `llround`, `log`, `log10`, `log1p`, `log2`, `logb`, `lrint`, `lround`, `nearbyint`, `nextafter`, `nexttoward`, `pow`, `remainder`, `remquo`, `rint`, `round`, `scalbln`, `scalbn`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `tgamma`, and `trunc`.

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

thousands_sep^{C89} An `lconv` structure member that is a pointer to a string containing the character used to separate groups of digits before the decimal-point character in formatted nonmonetary 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 "".

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

time A function that determines the current calendar time.

```
#include <time.h>
time_t time(time_t *timer);
```

The encoding of the type `time_t` is not specified by Standard C. The returned value is an approximation of the current calendar time. If this time is not available, `(time_t)(-1)` is returned. If `timer` is not `NULL`, the return value is also assigned to the object pointed to by `timer`.

__TIME__ A predefined macro that expands to a string containing the time of compilation in the form "`hh:mm:ss`". This macro can be used in any context where a string literal is permitted or required; for example,

```
char time[] = __TIME__;

printf("%s", __TIME__);
```

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

```
__TIME__
```

```
printf(">%s<\n", "----" __TIME__ "----");
```

Note that there is no wide string version of this macro, so it was difficult to get the time 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`.

time components See broken-down time; `tm`.

time conversion functions The `time.h` functions `asctime`, `ctime`, `gmtime`, `localtime`, and `strftime`.

time manipulation functions The `time.h` functions `clock`, `difftime`, `mkttime`, and `time`.

time.h A header that defines several macros and types and declares functions that manipulate time information.

`time.h` contains definitions or declarations for the identifiers in the following table:

<i>Name</i>	<i>Purpose</i>
<code>asctime</code>	Convert time
<code>clock</code>	Get processor time
<code>clock_t</code>	Time type
<code>CLOCKS_PER_SEC</code>	Number/second from <code>clock</code>
<code>ctime</code>	Convert time
<code>difftime</code>	Get difference between times
<code>gmtime</code>	Convert to UTC (GMT) time
<code>localtime</code>	Convert calendar time to local time
<code>mkttime</code>	Construct a time from components
<code>NULL</code>	Null pointer constant
<code>size_t</code>	Size/count type
<code>strftime</code>	Format a time into a string
<code>time</code>	Get calendar time
<code>time_t</code>	Time type
<code>struct tm</code>	A calendar time type

`time_t`^{C89} A type, defined in `time.h`, that is an implementation-defined arithmetic type capable of representing times.

`tm` A structure type, defined in `time.h` and `wchar.h`, that contains the individual components of a calendar time. Collectively, they are known as the broken-down time. The following members must be present in the structure, in any order. Other implementation-defined members also may be present.

```

struct tm {
    int tm_sec;    seconds after the minute
    int tm_min;    minutes after the hour
    int tm_hour;   hours since midnight
    int tm_mday;   day of the month
    int tm_mon;    months since January
    int tm_year;   years since 1900
    int tm_wday;   days since Sunday
    int tm_yday;   days since January 1
    int tm_isdst;  daylight saving time flag
};

```

where

```

tm_isdst > 0    daylight savings in effect
          == 0   daylight savings not in effect
          < 0    information not available

```

tm_* See tm.

tmpfile A function that creates a temporary binary file that is removed when it is closed or at normal program termination.

```

#include <stdio.h>
FILE *tmpfile(void);

```

tmpfile opens the file for update as if it had been opened by **fopen** using the mode "wb+". If this mode is inappropriate, use **setvbuf** or **setbuf** to change it. Alternatively, you can get a unique filename from **tmpnam** and **fopen** it yourself.

If a temporary file cannot be created, **NULL** is returned; otherwise a **FILE** pointer is returned. See also **FOPEN_MAX**.

TMP_MAX A macro, defined in **stdio.h**, that expands to an integer constant expression representing the maximum number of unique filenames that can be generated by **tmpnam**. Standard C requires **TMP_MAX** to be at least 25.

tmpnam A function that generates and returns a filename that is guaranteed not to be that of any existing file. A file by that name then can be opened using **fopen**.

```

#include <stdio.h>
char *tmpnam(char *s);

```

If **s** is **NULL**, **tmpnam** leaves its result in its own area and returns a pointer to that area. Subsequent calls to **tmpnam** may modify that area. If **s** is

not NULL, it is assumed to be a pointer to an array of at least `L_tmpnam` characters; `tmpnam` puts the result in that array and returns the address of the first character in that array. `tmpnam` has no way to communicate an error, so if you give it a non-NULL address that points to an area smaller than `L_tmpnam` characters, the behavior is undefined. *See also* `TMP_MAX`.

to prefix *See* future library directions.

token The fundamental unit of source code in a program. C89 defines six token types: keywords, identifiers, constants, string literals, punctuators, and operators. C99 dropped operator, moving such tokens into the punctuator category instead. A token cannot contain another token. Adjacent tokens may be separated by an arbitrary amount of white space. A token is sometimes referred to by the name “lexical element.”

token-pasting operator^{C89} A binary preprocessor operator, `##`, that is defined only for function-like macros. It allows two preprocessing tokens to be concatenated to form a new preprocessing token. For example, given the following macro definition:

```
#define M(value) name ## value
```

the macro call `M(4)` expands to `name4`. The precedence and associativity of the `#` and `##` preprocessor operators are unspecified.

tolower A locale-specific function that returns the lowercase equivalent of its argument `c`, provided it is an uppercase character. Otherwise, the argument is returned unchanged.

```
#include <ctype.h>
int tolower(int c);
```

In the "C" locale, only those characters testing true for `isupper` are converted to their corresponding lowercase equivalent. *See also* `tolower`.

In non-"C" locales, the mapping from uppercase to lowercase does not need to be one for one. For example, an uppercase letter might be represented as two lowercase letters taken together, or, perhaps, it may not even have a lowercase equivalent. If the mapping is one to many (i.e., two different results could be produced), it is implementation defined as to which alternative is returned. If, in the human language, more than one character at a time is translated to a single output character, these functions will fail to produce the human language correct results since the decision as to the character to return is based on processing a single input character at a time.

toupper A locale-specific function that returns the uppercase equivalent of its argument `c`, provided it is a lowercase character. Otherwise, the argument is returned unchanged.

```
#include <ctype.h>
int toupper(int c);
```

In the "C" locale, only those characters testing true for `islower` are converted to their uppercase equivalent. In non-"C" locales, the mapping from lowercase to uppercase does not need to be one for one. For example, a lowercase letter might be represented as two uppercase letters taken together, or, perhaps, it may not even have an uppercase equivalent. *See also* `tolower`; `toupper`.

towctrans^{C95} A function that maps the wide character `wc` using the mapping described by `desc`.

```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

The setting of the `LC_CTYPE` category must be the same as that during the call to `wctrans` that produced `desc`.

towlower^{C95} A locale-specific function that returns the lowercase equivalent of its wide character argument `wc`, provided it is an uppercase character. Otherwise, the argument is returned unchanged.

```
#include <wctype.h>
wint_t tolower(wint_t wc);
```

In the "C" locale, only those characters testing true for `isupper` are converted to their corresponding lowercase equivalent. In non-"C" locales, the mapping from uppercase to lowercase does not need to be one for one. If more than one candidate exists, this function returns one of the corresponding wide characters; however, for a given locale, it always returns the same one. *See also* `tolower`.

toupper^{C95} A locale-specific function that returns the uppercase equivalent of its wide character argument `wc`, provided it is a lowercase character. Otherwise, the argument is returned unchanged.

```
#include <wctype.h>
wint_t toupper(wint_t wc);
```

In the "C" locale, only those characters testing true for `islower` are converted to their uppercase equivalent. In non-"C" locales, the mapping

from lowercase to uppercase does not need to be one-for-one. If more than one candidate exists, this function returns one of the corresponding wide characters; however, for a given locale, it always returns the same one. *See also* `toupper`.

translation, phases of^{C89} *See* phases of translation.

translation limits *See* environmental limits.

translation unit The set of source lines resulting after a source file is processed, all referenced headers have been included, conditional pre-processing directives have been executed, and all macros have been expanded.

trap representation The bit pattern contained within an object that does not correspond to a value of the object's type.

trigonometric functions The `math.h` functions `acos`, `asin`, `atan`, `atan2`, `cos`, `sin`, and `tan`, and their `float` and `long double` counterparts.

trigraph^{C89} A three-character sequence beginning with `??` that permits certain punctuation characters to have an alternative representation. Trigraphs were invented by X3J11 to allow C source to be mechanically converted to and from machines supporting the ISO-646 character set (which is missing certain characters needed to write C source). Trigraphs are processed in translation phase 1. The complete set of trigraphs and their meanings are as follows:

<i>Sequence</i>	<i>Meaning</i>
??!	
??'	^
??([
??)]
??-	~
??/	\
??<	{
??=	#
??>	}

Following is a program written without trigraphs.

```
#include <stdio.h>

int main()
{
    static int i[10][20] =
        {{1, 2, 4}};
    int j;

    j = ~(i[0][0]
          | i[0][1]
          ^ i[0][2]);
    printf("j = %d\n", j);

    return 0;
}
```

The equivalent program with trigraphs is as follows:

```
??=include <stdio.h>

int main()
??<
    static int i??(10??)??(20??)
        = ??<??<1, 2, 4??>??>;
    int j;

    j = ??-(i??(0??)??(0??)
            ??! i??(0??)??(1??)
            ??' i??(0??)??(2??));
    printf("j = %d??/n", j);

    return 0;
??>
```

true 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.

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

true^{C99} A macro, defined in `stdbool.h`, that expands to the integer constant 1. It is intended for use in contexts involving the `bool` macro (or its underlying type, `_Bool`.) *See also* `false`.

trunc[f|l]^{C99} A function that rounds its argument to the integer value, in floating format, nearest to but no larger in magnitude than the argument.

```
#include <math.h>
double trunc(double x);
float truncf(float x);
long double trunc1(long double x);
```

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

type Describes the meaning of a value stored in an object or returned by a function. *See also* type, function; type, incomplete; type, object.

type, compatible Two types are compatible if they are the same. Two structure, union, or enumeration types declared in separate translation units are compatible if they have the same number of members, the same member names, and compatible member types; for two structures, the members shall be in the same order. For two structures or unions, the bit-fields shall have the same widths; for two enumerations, the members shall have the same values.

Two pointer types are compatible if they are identically qualified and they point to compatible types.

Two qualified types are compatible if they are identically qualified versions of compatible types.

Two array types are compatible if they are compatible element types and, if both size specifiers are present, they are equal.

Two function types are compatible if they return compatible types. If both parameter lists are present, they must have the same number of parameters and the corresponding parameters in each must be compatible.

C99 added the following requirements: “[T]wo structure, union, or enumerated types declared in separate translation units are compatible if their tags and members satisfy the following requirements: If one is declared with a tag, the other shall be declared with the same tag. If both are complete types, then the following additional requirements apply: there shall be a one-to-one correspondence between their members such that each pair of corresponding members are declared with compatible types, and such that if one member of a corresponding pair is declared with a name, the other member is declared with the same name. For two structures, corresponding members shall be declared in the same order. For two structures or unions, corresponding bit-fields shall have

the same widths. For two enumerations, corresponding members shall have the same values.”

type, composite^{C89} A composite type can be constructed from two types that are compatible; it is a type that is compatible with both of the two types and satisfies the following conditions:

- If one type is an array of known constant size, the composite type is an array of that size; otherwise, if one type is a variable length array, the composite type is that type.
- If only one type is a function type with a parameter type list (a function prototype), the composite type is a function prototype with the parameter type list.
- If both types are function types with parameter type lists, the type of each parameter in the composite parameter type list is the composite type of the corresponding parameters.
- These rules apply recursively to the types from which the two types are derived.
- For an identifier with external or internal linkage declared in the same scope as another declaration for that identifier, the type of the identifier becomes the composite type.

In the following declarations:

```
int i[];
int i[5];

void f();
void f(int);

int j;
char c;
short s;
```

the composite type of `i` is “array of 5 ints,” the composite type of `f` is “function taking one argument of type `int` and returning no value,” and the composite type of `j ? c : s` is “`int`.”

type, const-qualified^{C89} A type that includes the `const` type qualifier.

type conversion *See conversion and its subentries.*

type definition A type synonym created via the `typedef` keyword.

type, derived *See derived type.*

type, effective Ordinarily, the type with which an object is declared. Objects can only be accessed by lvalues having the appropriate type, which is referred to as the effective type. Although objects that are allocated at run time do not have a declared type, they still have a type. This term was invented to help describe the rules for aliasing.

type, function A type that describes a function. It includes the return type and also the argument list type information if present. For example,

```
int f();
int g(void);
```

The type of `f` is “function returning an `int` and having an unknown number and type of arguments.” The type of `g` is “function returning an `int` and having no arguments.”

type, incomplete^{C89} A type that describes an object but contains insufficient information for the object’s size to be computed. In the case of `int (*p) []`; `p` is a “pointer to array of unknown size.” This type could be completed by a future (re)declaration of `p`. Other examples of incomplete types are `extern double d[]`; and `struct tag`; . The `void` type is a special incomplete type in that it can never be completed.

type, narrow The signed and unsigned versions of `char` and `short`, and `float` are often referred to as narrow types because they are converted to wide types in certain contexts. *See also* conversion, function arguments; UP rule; VP rule.

type, object A type that completely describes the value of an object. *See also* type, incomplete.

type, qualified^{C89} A type that includes the `const`, `restrict`, or `volatile` qualifier. *See also* type, unqualified.

type qualifier^{C89,C99} A keyword used to somehow qualify an object type. C89 invented: `const` and `volatile`, while C99 added `restrict`.

C99 allows the same qualifier to be used multiple types in the same declarator, without ill effect, as in `const const int i = 10`;

type, restrict-qualified^{C89} A type that includes the `restrict` qualifier.

type specifier One of the following:

```
void
char
short
int
long
float
double
signed
unsigned
struct-or-union-specifier
enum-specifier
typedef-name
```

type, unqualified^{C89} A type that does not include the `const`, `restrict`, or `volatile` qualifier. *See also* type, qualified.

type, volatile-qualified^{C89} A type that includes the `volatile` qualifier.

type, wide *See* conversion, function arguments.

typedef A keyword used to create a synonym for a type. Examples of its use are as follows:

```
typedef int counter;

typedef struct {
    int key;
    double data;
} Node;
```

Technically, `typedef` is a storage class keyword, although it has nothing to do with storage classes.

typedef name The identifier defined using `typedef` to be a synonym for a type. Typedef names share the same name space as enumeration constants, functions, and variables.

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

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

◇ ◇ ◇