

C

C standard The formal definition of the C language, preprocessor, and run time library. *See also* ANSI C; ISO C.

"C" locale^{C89} The one locale that every conforming implementation must support. The requirements for this locale are documented under each member of the structure `lconv`.

From a practical viewpoint, the "C" locale means "run in U.S.–English mode" and reflects exactly the same behavior we have always seen from the library. That is, `printf` uses a period for a decimal point, date and time formats reflect the U.S. style, and `isalpha` returns true only for the uppercase and lowercase Roman letters. By default, a C program runs in the "C" locale unless the `setlocale` function has been called (or the implementation's normal operating default locale is other than "C"). *See also* locale-specific behavior.

C++ programming language A language that was originally called "C with classes" and was developed by Bjarne Stroustrup at AT&T's Bell Laboratories, the birthplace of C and the UNIX operating system. C++ uses an object-oriented design and programming philosophy. A standard, called ISO/IEC 14892:1998, was published in 1998. Except for a few small areas, Standard C++ is a proper superset of C95; however, C99 contains numerous language and many library features not found in Standard C++. *See also* `__cplusplus`; identifier conflicts with C++; J16; WG21.

C89 The first ANSI C standard, produced in 1989 by committee X3J11.

C90 The first ISO C standard, produced in 1990 by committee WG14 in conjunction with committee X3J11. C90 is technically equivalent to C89.

C95 An amendment to C90, produced in 1995 by committee WG14 in conjunction with committee X3J11. The additions included digraphs and the header `iso646.h`, and many multibyte and wide character functions via the headers `wchar.h` and `wctype.h`. *See also* `__STDC_VERSION__`.

C99 The second ISO C standard, produced in 1999 by committee WG14 in conjunction with committee J11. Many of the new features resulted from work pioneered by NCEG and X3J11.1. *See also* `__STDC_VERSION__`.

C9X The working title used throughout the production of C99.

`cabs[f|l]`^{C99} A function that computes the complex absolute value of its argument `z`.

```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

`cacos[f|l]`^{C99} A function that computes the complex arc cosine of its argument `z`, with branch cuts outside the interval $[-1, +1]$ along the real axis.

```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

`cacosh[f|l]`^{C99} A function that returns the complex arc hyperbolic cosine of its argument `z`, with a branch cut at values less than 1 along the real axis.

```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

`calloc` A function that dynamically allocates contiguous memory for `nmemb` objects, each of whose size is `size` bytes.

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

The space allocated is initialized to “all-bits-zero.” Note that this is not guaranteed to be the same representation as floating-point zero or `NULL`. The value returned is of type `void *` and, as such, is assignment compatible with any data pointer type. Therefore, no explicit cast is needed (unlike in C++). 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.

See also `free`; `malloc`; `realloc`.

`carg[f|l]`^{C99} A function that computes the argument (or *phase angle*) of its argument `z`, with a branch cut along the negative real axis.

```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

The value returned corresponds to the interval $[-\pi, +\pi]$.

carriage-return escape sequence An escape sequence, `\r`, which is used to represent the carriage-return character. This character is not often needed. For example, on systems where records are terminated by a carriage-return/line-feed pair, new-lines are translated to and from this pair on output and input, respectively. As such, there is no need to deal with the carriage return explicitly unless a text file containing such pairs is opened in binary mode. *See also* `fopen`.

case A keyword that is used only in the context of a `switch` statement to designate a switch value label. *See also* `label`, `case`.

case conversion *See* `tolower`; `toupper`; `towctrans`; `towlower`; `toupper`; `wctrans`.

case label *See* `label`, `case`.

case sensitivity The characteristic of a language, which allows it to distinguish between upper- and lowercase letters. C is a case-sensitive language; that is, the identifiers `ABC`, `abc`, `Abc`, and `AbC` are distinct. Note, however, that prior to C99, an implementation was permitted to ignore case in handling external names (i.e., those of `extern` functions and variables) because many linkers, librarians, and assemblers are not case sensitive.

C language keywords must be written in the correct case to be recognized as such. (Note that C99 added several keywords spelled in mixed case; e.g., `_Bool`.)

casin[f|l]^{C99} A function that computes the complex arc sine of its argument `z`, with branch cuts outside the interval $[-1, +1]$ along the real axis.

```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

casinh[f|l]^{C99} A function that returns the complex arc hyperbolic sine of its argument `z`, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

cast operator The unary cast operator `()` has the general form

```
( type ) exp
```

where the value of the expression *exp* is converted to a value having type *type*. Both *type* and *exp* must have scalar type except that a `void` expression is permitted if it is being cast to type `void`. (Note that *exp* may designate an array because such an expression is first converted to a pointer, which is a scalar. Similarly, *exp* may designate a function since such an expression is first converted to a pointer.) Type qualifiers are permitted in *type* but are vacuous. The result of a cast is not an lvalue, although some implementations may make it so. This operator associates left to right.

In some sense, a compound literal looks like a cast expression.

casting The operation of explicitly converting the value of an expression to a given type. Casting a value to type `void` causes that value to be explicitly discarded. *See also* cast operator.

catan[f|l]^{C99} A function that computes the complex arc tangent of its argument *z*, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis.

```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

catanh[f|l]^{C99} A function that returns the complex arc hyperbolic tangent of its argument *z*, with branch cuts outside the interval $[-1, +1]$ along the real axis.

```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanh1(long double complex z);
```

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

category^{C89} One of a number of components that, when combined, describe a locale. *See also* `LC_*` macros; locale, mixed.

cbirt[f|l]^{C99} A function that returns the real cube root of its argument `x`.

```
#include <math.h>
double cbrt(double x);
float cbrtf(float x);
long double cbrt1(long double x);
```

ccos[f|l]^{C99} A function that computes the complex cosine of its argument `z`.

```
#include <complex.h>
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

ccosh[f|l]^{C99} A function that computes the complex hyperbolic cosine of its argument `z`.

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

ceil[f|l] A function that computes the smallest integer value not less than its argument `x`.

```
#include <math.h>
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

The integer value computed is returned as a floating-point value.

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

cerf[f|l]^{C99} An optional complex error function that, if provided, must be declared in `complex.h`.

cerfc[f|l]^{C99} An optional complex complement function that, if provided, must be declared in `complex.h`.

cexp[f|l]^{C99} A function that returns the complex base-*e* exponential of its argument `z`.

```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

cexp2[f|l]^{C99} An optional complex base-2 exponential function that, if provided, must be declared in `complex.h`.

cexpm1[f|l]^{C99} An optional complex base-*e* exponential minus one function that, if provided, must be declared in `complex.h`.

char An integer type keyword. A `char` is big enough to hold any character in the basic execution character set. Standard C requires it to be at least eight bits. Traditionally, `char` expressions were widened to `int` when used in expressions and as arguments to functions. However, Standard C allows them to be used as arguments without widening, provided a corresponding function prototype is in scope. *See also* `char`, `plain`; conversion, function arguments.

char, plain The type `char` used without either of the modifiers `signed` or `unsigned`. It is implementation defined as to whether a plain `char` is signed or unsigned. Some compilers provide an option to set the signedness.

character An abstract idea that means different things to different people. To convey an exact meaning, chose the appropriate term from the following list: `char`, multibyte character, single-byte character, or wide character.

Each character in a wide character set occupies the same amount of storage. Strictly speaking, ASCII and EBCDIC are wide character sets; each character simply occupies a single 8-bit byte. In the Unicode wide character set, each character occupies two 8-bit bytes.

character constant *See* `constant`, `character`.

character handling header *See* `ctype.h`.

character I/O functions The `stdio.h` functions `fgetc`, `fgets`, `fputc`, `fputs`, `getc`, `getchar`, `gets`, `putc`, `putchar`, `puts`, and `ungetc`.

character, multibyte^{C89} *See* `multibyte character`.

character, pushback *See* `ungetc`; `ungetwc`.

character set, execution^{C89} The set of characters available for use during the execution of a program. The basic execution character set must include all of the characters in the basic source character set, the null character, and the control characters representing alert, backspace, carriage return, and new line. The members of the extended-execution character set are locale specific and may be single-byte or multibyte characters.

character set, execution, extended^{C89} *See* character set, execution.

character set, source^{C89} The set of characters available for use in writing source code. The basic source character set must include the 52 upper- and lowercase letters from the English alphabet, the digits 0–9, the graphic characters ! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { | } ~, the space character, and control characters representing horizontal tab, vertical tab, and form feed. If any other characters are seen except inside character constants, comments, header names, and string literals, the behavior is undefined. The extended source character set may include other single-byte or multibyte characters.

character, single-byte A bit representation that fits into a byte. All source characters required to write C source are characters that can be represented in a single `char`. *See also* character, wide; multibyte character.

character string literal *See* string literal.

character testing functions The `is*` family of functions in `ctype.h` and its wide character counterpart `wctype.h`.

character, wide^{C89} A wide character is a character encoded in some manner into an object of some integer type. Standard C defines this type as `wchar_t` in a number of headers. Do not confuse a wide character with a multibyte character. C89 introduced minimal support for very large character sets in string literals, character constants, header names, and comments, and defined several functions in `stdlib.h` for conversion to and from multibyte representation. C95 significantly increased this level of support via the addition of several new headers and many functions.

`CHAR_BIT`^{C89} A macro, defined in `limits.h`, that designates the number of bits in a `char`. (Standard C requires that it be at least eight.) This macro expands to an integer constant expression suitable for use with a `#if` directive.

`CHAR_MAX`^{C89} A macro, defined in `limits.h`, that designates the maximum value for an object of type `char`. (Must be either `SCHAR_MAX` or `UCHAR_MAX` depending on whether a plain `char` is signed.) This macro expands to an integer constant expression suitable for use with a `#if` directive.

`CHAR_MIN`^{C89} A macro, defined in `limits.h`, that designates the minimum value for an object of type `char`. (Must be either `SCHAR_MIN` or zero depending on whether a plain `char` is signed.) This macro expands to an integer constant expression suitable for use with a `#if` directive.

`cimag[f|l]`^{C99} A function that returns the imaginary part of its argument `z` as a real.

```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

class A term used in object-oriented programming that is also a keyword in C++. It 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 `class` as an identifier in the new C code you write. This use of `class` is not to be confused with the storage class keywords in C.

clearerr A function that clears both the end-of-file and error indicators for the file pointed to by `stream`.

```
#include <stdio.h>
void clearerr(FILE *stream);
```

clogamma[f|l]^{C99} An optional complex natural log of gamma function that, if provided, must be declared in `complex.h`.

clock A function that determines the amount of processor time used from the beginning of an implementation-defined era.

```
#include <time.h>
clock_t clock(void);
```

The era is related to the time at which the program started running and may be an approximation. The intent here is that `clock` gives the CPU time from the start of the program to the time `clock` is called.

The value returned has type `clock_t`, whose units are implementation defined. However, by definition, when this value is divided by `CLOCKS_PER_SEC`, the result is in seconds. If the processor time cannot be determined, or if the time cannot be represented in the return type, the value returned is `(clock_t)(-1)`.

CLOCKS_PER_SEC^{C89} A macro, defined in `time.h`, that expands to the number of `clock_t` intervals in a second. For example, if `clock_t` is measured in milliseconds, the value of `CLOCKS_PER_SEC` will be 1,000.

clock_t^{C89} A type, defined in `time.h`, that is an implementation-defined arithmetic type (not necessarily integer) capable of representing times. When two values of this type are obtained from calls during the same program execution, subtracting the first value from the second gives the time elapsed between those calls. *See also* `clock`; `CLOCKS_PER_SEC`.

clog[f|l]^{C99} A function that returns the complex natural logarithm of its argument `z`, with a branch cut along the negative real axis.

```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

clog10[f|l]^{C99} An optional complex base-10 log function that, if provided, must be declared in `complex.h`.

clog1p[f|l]^{C99} An optional complex natural log function that, if provided, must be declared in `complex.h`.

clog2[f|l]^{C99} An optional complex base-2 log function that, if provided, must be declared in `complex.h`.

colon A terminator for `case` and `default` labels in a `switch`, as well as for labels that are the object of a `goto` statement. It also precedes the width in a bit-field declaration, and is used as the second character of the conditional operator.

comma operator A binary operator, `,`, that causes its operands to be evaluated left to right. In the following example:

```
exp1, exp2
```

exp1 is evaluated, and its value is discarded, so to be useful, this expression must contain a side effect. Then *exp2* is evaluated, and its value and type become the value and type of the whole expression. There is a sequence point at the comma. This operator associates left to right. (Except when used in the first and third expressions in a `for` statement, the comma operator is best restricted to macro definitions and not otherwise used overtly.) If a comma operator is used in the same context as a comma punctuator, it may be necessary to distinguish between the two. In the following example:

```
g(a, b, c);
h((a, b), c);
```

results in a call to `g` with three arguments and a call to `h` with only two. In the second case, the first comma is acting as an operator (as indicated by the grouping parentheses). The second comma is a punctuator separating the two arguments.

comma punctuator A token used as a punctuator, most often to separate macro and function arguments, declarators in a declaration, and initializer list expressions.

command line The set of arguments specified when a program is invoked at the operating system level or via a call to `system`. The count and contents of these arguments are made available to `main` via `argc` and `argv`. The existence of a command-line processor can be determined via the library routine `system`. Certain command-line processor-related issues regarding to arguments are outside the scope of the C standard; for example: Can an argument contain white space or double quotes? Will its letters' casing be preserved?

comment Text placed within code, that does not affect that code's operation, and that is intended to explain that code. The character sequences `/*` and `*/` delimit comments that are permitted to span multiple source lines but need not actually do so. C99 added C++'s `//`-style comments, which end at the next new-line. During source translation, an implementation recognizes each comment and replaces it with a single space. As such, `/*...*/`-style comments can be used anywhere white space is permitted; that is, between any two adjacent tokens. For example, the lines

```
/* */ a /* */ = /* */ b /* */ ; /* */  
a = b;
```

contain identical token sequences. `/*...*/`-style comments do not nest in Standard C, although nesting is permitted in some nonstandard implementations. To disable a block of code containing `/*...*/`-style comments, use the following approach:

```
#if 0  
    a = b + c; /* ... */  
#endif
```

Consider the following code fragment:

```
int i = 20, j, *pi = &i;  
  
j = 100/*pi;
```

At first glance it appears that 100 is being divided by the value of the `int` to which `pi` points. However, during tokenizing, the characters `/*` are recognized as the (presumably unintended) start of a comment. To resolve this, a space should follow the slash, or (better style suggests) the expression should be written as `100/(pi)` instead.

common extensions Extensions that are widely implemented but are not universal. Extensions may include keywords (such as `fortran`), pre-processor directives (such as `#module`), or library functions (such as `open`)

and `close`). Support for `envp` and the allowance of `$` characters in identifiers are other examples.

common initial sequence A set of one or more initial members in two different structures, that have compatible types (and, for members that are bit-fields, the same widths).

common warning A warning issued by a translator to help the programmer locate nonsyntactic problems. However, these are not required by Standard C. Examples are “a statement can never be reached,” “a function is called without a prototype in scope,” and “an unrecognized pragma was found.” As to whether an implementation issues such warnings is a marketplace issue and is referred to as “Quality of Implementation.”

comparison functions The `string.h` functions `memcmp`, `strcmp`, `strcoll`, `strncmp`, and `strxfrm`, and their wide character counterparts in `wchar.h`, `wmemcmp`, `wscmp`, `wscoll`, `wcsncmp`, and `wcsxfrm`. Functions `bsearch` and `qsort` also take an argument that is a pointer to one of these functions or to a user-defined comparison function.

compatible type *See* `type`, compatible.

compiler A term used generically to mean a C language translator; it includes such tools as interpreters and incremental compilers.

`compl`^{C95} A macro, defined in `iso646.h`, that expands to the token `~`. It allows programmers using source character sets (such as ISO 646) that are missing certain characters necessary for writing C programs to enter those characters using identifiers instead. Note that in C++, this name is a keyword.

complement operator A unary operator, `~`, that produces the bit-wise one’s-complement of its operand. The operand must have integer type. The usual arithmetic conversions are performed on the operand, and the result has the promoted type. This operator associates right to left.

`complex`^{C99} A macro, defined in `complex.h`, that expands to the keyword `_Complex`. For normal usage of the complex types, it is strongly recommended that you include `complex.h` and use this macro rather than using the `_Complex` keyword directly. There are three complex types: `float _Complex`, `double _Complex`, and `long double _Complex`, which can be written instead as `float complex`, `double complex`, and `long double complex`, respectively. *See also* `__STDC_IEC_559_COMPLEX__`.

`_Complex`^{C99} A keyword that provides support for a family of complex types. Since it was invented by C99, by which time quite a few programs already contained type synonyms using some form of the word `complex`, this keyword was spelled using one of the forms reserved for implementers.

Unless you must mix both existing homegrown complex machinery and this new keyword in the same source file, it is strongly recommended that you include `complex.h` and use its macro `complex` instead.

Note that freestanding implementations are not required to provide complex types.

complex arithmetic^{C99} Arithmetic based on objects having a real and an imaginary part. (In pre-C99 implementations, `math.h` is often extended to include functions and macros to assist with complex number processing.) See also `complex`; `_Complex`; `complex.h`.

complex.h^{C99} A header that declares or defines the following names, all pertaining to complex arithmetic support:

<i>Name</i>	<i>Purpose</i>
<code>cabs[f l]</code>	Complex absolute value
<code>cacos[f l]</code>	Complex arc cosine
<code>cacosh[f l]</code>	Complex hyperbolic arc cosine
<code>carg[f l]</code>	Complex argument (<i>phase angle</i>)
<code>casin[f l]</code>	Complex arc sine
<code>casinh[f l]</code>	Complex hyperbolic arc sine
<code>catan[f l]</code>	Complex arc tangent
<code>catanh[f l]</code>	Complex hyperbolic arc tangent
<code>ccos[f l]</code>	Complex cosine
<code>ccosh[f l]</code>	Complex hyperbolic cosine
<code>cexp[f l]</code>	Complex base- <i>e</i> exponential
<code>cimag[f l]</code>	Obtain imaginary part
<code>clog[f l]</code>	Complex natural logarithm
<code>complex</code>	Synonym for the type <code>_Complex</code>
<code>_Complex_I</code>	The constant <i>i</i>
<code>conj[f l]</code>	Complex conjugate
<code>cpow[f l]</code>	Complex power
<code>cproj[f l]</code>	Complex Riemann sphere projection
<code>creal[f l]</code>	Obtain real part
<code>csin[f l]</code>	Complex sine
<code>csinh[f l]</code>	Complex hyperbolic sine
<code>csqrt[f l]</code>	Complex square root
<code>ctan[f l]</code>	Complex tangent
<code>ctanh[f l]</code>	Complex hyperbolic tangent
<code>I</code>	The constant <i>i</i>
<code>imaginary</code>	Synonym for the type <code>_Imaginary</code>
<code>_Imaginary_I</code>	The constant <i>i</i>

See future library directions.

`_Complex_I`^{C99} A macro, defined in `complex.h`, that expands to a constant expression of type `const float _Complex`, having a value of the imaginary unit i .

compliance The degree to which an implementation conforms to the requirements of a definition such as Standard C. While compliance can easily be claimed, it is usually subject to verification by using a formally accepted validation suite. *See also* implementation, conforming; program, conforming; program, strictly conforming.

composite type^{C89} *See* type, composite.

compound assignment operator *See* assignment operator, compound.

compound literal^{C99} An expression consisting of a parenthesized type name followed by a brace-enclosed list of initializers. This results in an unnamed object whose value is given by the initializer list.

If the type name specifies an array of unknown size, the size is determined by the initializer list, and the type of the compound literal is that of the completed array type. If the type name specifies an object type, the type of the compound literal is that specified by the type name. In either case, the result is an lvalue and can be modified.

If a compound literal occurs outside the body of a function, the resulting object has static storage duration, and its initializer must be made up of constant expressions; otherwise, it has automatic storage duration associated with the enclosing block, and its initializer need not be made up of constant expressions.

For example, the file scope definition

```
int *values = (int [4]){10, 20, 30};
```

initializes `values` to point to the first element of an array of four `int`, the first having the value 10, the second, 20, and the third, 30. The initializer expressions are required to be constant, since the unnamed object has static storage duration. Therefore, the fourth element takes on the value zero. The array elements can be modified.

The block scope definitions

```
int i = 10;
const int *counts = (int []){i, i * 2, i * 3};
```

initialize `counts` to point to the first element of an array of three `int`, the first having the value 10, the second, 20, and the third, 30. The initializer expressions are not required to be constant, since the unnamed object has automatic storage duration. The array elements cannot be modified.

A compound literal can be made `const` explicitly, as follows:

```
f((const int []){i, i * 2, i * 3});
```

Consider the case in which `Point` is a structure that contains the members `x` and `y`, in that order, and we wish to create a pair of unnamed `Points` and pass them by address to a function that draws a line between those points as in the following example:

```
drawLine(&(const struct point){-4, 3},
         &(const struct point){9, 2});
```

A compound literal can contain designated initializers. The following lines illustrate the previous example written without knowing the order of `Point`'s members:

```
drawLine(&(const struct point){.x = -4, .y = 3},
         &(const struct point){.x = 9, .y = 2});
```

compound statement *See* block.

concatenation functions The `string.h` functions `strcat` and `strncat` and their wide character counterparts in `wchar.h`, `wscat`, and `wcsncat`.

conditional compilation The selection or rejection of source file lines based on the value of a translation-time integer constant expression, on the existence or nonexistence of a particular macro definition, or on a combination of both. It is effected using the preprocessor directives `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`, and the preprocessor operator `defined`. The true and false selection paths may contain any language token or preprocessor directive. As such, conditional directives may be nested—to at least 8 levels in C89 and to 15 levels in C99.

All directives in a related set must reside in the same source file. That is, in the following example:

```
#ifdef DEBUG
#include "debug.h"
#endif
```

the header `debug.h` is not permitted to contain `#elif` or `#else` directives that belong to the outer set.

conditional operator A ternary operator, `?:`, that has the following form:

```
exp1 ? exp2 : exp3
```

which evaluates to *exp2* if *exp1* tests true; otherwise it evaluates to *exp3*. (Only one of *exp2* and *exp3* is evaluated.) The type of the expression is the composite type of *exp2* and *exp3*. The first operand must have scalar type. The second and third operands must have either arithmetic types, compatible structure or union types, or `void` type; or they must be pointers to compatible types, or one a pointer and the other the null pointer constant, or one a pointer to an object or incomplete type and the other a pointer to `void`. This operator associates right to left. There is a sequence point at the ?.

conforming implementation See implementation, conforming.

conforming program See program, conforming.

`conj[f|l]`^{C99} A function that returns the complex conjugate of its argument `z`.

```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

`const`^{C89} A keyword that is used as a type qualifier. It indicates that the object to which it applies cannot be modified in the scope of this declaration. Specifically, it causes it to be treated as a nonmodifiable lvalue. `const` was adapted from C++. While `const` is useful for partitioning data into read-only and read-write sections, `const` does not guarantee that an object will be physically write protected at run time. The `const` qualifier may also be applied to the underlying object in a pointer declaration. Consider the following cases:

```
char *ncpncc;
char * const cpncc;
const char *ncpcc;
const char * const cpcc;
```

`ncpncc` is a non-`const` pointer to a non-`const` `char`. Both the pointer and the `char` to which it points can be modified via this identifier.

`cpncc` is a `const` pointer to a non-`const` `char`. Only the `char` to which it points can be modified via this identifier; `cpncc` itself cannot be modified.

`ncpcc` is a non-`const` pointer to a `const` `char`. Only the pointer can be modified via this identifier; the `char` to which it points cannot be modified.

`cpcc` is a `const` pointer to a `const` `char`. Neither the pointer nor the `char` to which it points can be modified via this identifier.

Declarations containing the `const` qualifier also may contain the qualifiers `restrict` and `volatile`; the three are not related, nor are they mutually exclusive.

In C99, `const` may also be used inside any dimension of an array parameter (possibly along with `restrict` or `static`); for example

```
void copy(int s[const]);
```

The previous declaration is simply an alternate way of saying the following:

```
void copy(int * const s);
```

constant One of the token types in C. A constant represents a value that does not change. The kinds of constants are: character, enumeration, floating, and integer. String literals are in a token category of their own and are not considered constants.

constant, character A token of the form `'x'` where `x` is a sequence of printable graphic multibyte characters or escape sequences. Its type is `int` not `char` (unlike in C++). If the sequence contains more than one character, as in `'ab'` and `'abcd'`, the value of the resulting character constant is implementation defined. Note that `'x'` is quite different from `"x"`, the latter being an array of two `char`. *See also* constant, character, wide.

constant, character, wide^{C89} A token of the form `L'x'` where `x` is a sequence of multibyte characters. The sequence may include escape sequences. A wide character constant has type `wchar_t`. *See also* constant, character; string literal, wide.

constant, enumeration Any one of the identifiers defined as part of an enumeration type. For example, `red`, `green`, and `blue` are enumeration constants in the following enumerated type definition:

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

They have type `int` and take on the values 0, 4, and 5, respectively. Enumeration constants can be explicitly initialized as shown with `green`. As a result, there may be gaps in the set of initial values used, and there can also be duplicates. Enumeration constants share the same name space as variables and functions and are not limited to being used in the context of their parent enumerated type. To some extent, enumeration constants can be used in place of simple object-like macros.

constant expression An expression whose subexpressions are all constants. A constant expression can be evaluated at translation-time rather than at run time. There are three kinds of constant expression: address, arithmetic, and integer. Integer constant expressions are actually required in certain contexts; for example, in case labels, bit-field widths, and enumeration-constant initializers.

constant, floating A constant containing a significand part that may be followed by an exponent part and a suffix indicating its type. The exponent may be signed and it may be introduced with either `e` or `E`. The suffix `F` (or `f`) indicates the `float` type and the suffix `L` (or `l`) indicates the `long double` type. (These suffixes were an invention of C89.) The type of an unsuffixed floating constant is `double`.

Prior to C99, floating-point constants were written entirely in decimal. C99 allows the significand to be written in hexadecimal, provided a `0x` or `0X` prefix is present and the exponent is introduced by `p` (or `P`) instead of `e` (or `E`). While the exponent in a hexadecimal floating-point constant must be written in decimal, that exponent indicates the power of 2 (rather than 10) by which the significand part is to be scaled.

constant, integer A constant that begins with a digit but has no decimal point or exponent. It may have a prefix that specifies its base and a suffix that specifies its type. A prefix of `0` indicates octal base while `0x` or `0X` indicates hexadecimal. If no prefix is present, the base is taken as decimal. The suffix `L` (or `l`) indicates the `long int` type and the suffix `U` (or `u`) indicates `unsigned`. All possible combinations of `L` (or `l`) and `U` (or `u`) are permissible and equivalent. The `U` and `u` suffixes were an invention of C89.

C99 added the type `long long` along with the suffixes `ll` and `LL` (which are equivalent) that can be combined in either order with `u` or `U` to indicate the type `unsigned long long int`.

The type of an integer constant depends on its base and value as follows. It is the first of the corresponding list in which its value can be represented:

- In C89—Unsuffixed decimal: `int`, `long int`, `unsigned long int`; unsuffixed octal or hexadecimal: `int`, `unsigned int`, `long int`, `unsigned long int`; suffixed by the letter `u` or `U`: `unsigned int`, `unsigned long int`; suffixed by the letter `l` or `L`: `long int`, `unsigned long int`; suffixed by both the letters `u` or `U` and `l` or `L`: `unsigned long int`.
- In C99—Unsuffixed decimal: `int`, `long int`, `long long int`; unsuffixed octal or hexadecimal: `int`, `unsigned int`, `long int`, `unsigned long int`, `long long int`, `unsigned long long int`; suffixed by the letter `u` or `U`: `unsigned int`, `unsigned long int`,

`unsigned long long int`; decimal suffixed by the letter `l` or `L`: `long int`, `long long int`; octal or hexadecimal suffixed by the letter `l` or `L`: `long int`, `unsigned long int`, `long long int`, `unsigned long long int`; suffixed by both the letters `u` or `U` and `l` or `L`: `unsigned long int`, `unsigned long long int`; decimal suffixed by the letters `ll` or `LL`: `long long int`; octal or hexadecimal suffixed by the letters `ll` or `LL`: `long long int`, `unsigned long long int`; suffixed by both `u` or `U` and `ll` or `LL`: `unsigned long long int`.

constant, null pointer A pointer expression whose value is not, and never can be, the address of an object or function. It can be constructed from any integer constant expression having the value 0 or from such a constant cast to the type `void *`. The most common ways of representing a null pointer constant are 0, 0L, and NULL.

constant, type of See constant, floating; constant, integer.

const_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 `const_cast` as an identifier in new C code you write.

const-qualified type^{C89} A type containing the `const` qualifier.

constraint A syntactic and semantic restriction by which the exposition of language elements is to be interpreted. An implementation must diagnose a constraint violation. The following are examples of constraints: “The operators `[]`, `()`, and `?` : shall occur in pairs, separated by expressions,” “Each of the operands of the division, multiplication, and remainder operators shall have arithmetic type,” and “Each `#` preprocessing token in the replacement list of a function-like macro shall be followed by a parameter as the next preprocessing token in the replacement line.”

continue A statement that causes the current innermost iteration of a `for`, `while`, or `do` loop to be terminated and a new iteration (if any) to be started. It cannot be used in any other context. It is subtly different from `break` and has the following form:

```
continue;
```

contracted expression^{C99} A floating-point expression (usually consisting of a multiply and add operation) may be *contracted*; that is, evaluated as though it were an atomic operation, thereby omitting intermediate rounding errors implied by the source code and the expression evaluation method. See also `#pragma STDC FP_CONTRACT`.

control character A member of an implementation-defined set of characters that are not printing characters. *See also* control wide character; `iscntrl`; `iswcntrl`.

control wide character^{C89} A member of a locale-specific set of wide characters that are not printing wide characters. *See also* control character; `iscntrl`; `iswcntrl`.

conversion The changing of operand values from one type to another. For example, in

```
double d;
int i = 5, j = 4;

d = i + (double)j;
```

the value of `i` is implicitly converted to `double` while that of `j` is explicitly converted using a cast.

conversion, array In almost all cases, an expression designating an array is converted to an expression of type “pointer to the first element of that array.” The exceptions are when the expression is the operand of `sizeof` or the unary `&` operator, or it is a string literal used as the initializer of an array of characters or wide characters.

conversion, explicit The use of a cast operator to convert the value of an expression from one type to another.

conversion, function In almost all cases, a process in which an expression designating a function is converted to an expression of type “pointer to function.” The exceptions are when the expression is the operand of `sizeof` or the unary `&` operator. (If `f` is a function, the expression `f` is not converted in `sizeof(f)`, thus producing the constraint violation that `sizeof` cannot be applied to function types.)

conversion, function arguments By default, a conversion in which expressions of type `signed` or `unsigned char` and `short` are widened to the corresponding flavor of `int` (as determined by the value preserving rule) when passed as arguments to a function. Similarly, `float` arguments get widened to `double`.

Standard C permits (but does not require) arguments of these “narrow” types to be passed without widening provided a prototype containing the corresponding narrow types is used for both the function definition and all its declarations. In such a case, an implementation may widen all, some, or none. For example, it may widen `char` and `short` but not `float`. Note that all arguments passed that correspond to an ellipsis in a prototype are always widened.

Functions defined and declared without prototype notation always expect and get widened types. (That is why the `printf` family has no conversion specifiers for the types `char`, `short`, and `float`.)

conversion, implicit The automatic conversion of operand values from one type to another that occurs with some operators. For example, in the expression `i + d`, if `i` has type `int` and `d` has type `double`, the value of `i` is converted to `double`, the addition is performed, and the resulting type is also `double`. *See also* conversion; conversion, explicit.

conversion, integer type A conversion in which signed and unsigned `chars`, `shorts`, and `int` bit-fields may be used wherever an `int` or `unsigned int` are expected. If their value can be represented in an `int`, that is the type to which they are converted, otherwise they are converted to `unsigned int`. *See also* unsigned preserving rule; value preserving rule.

conversion, pointer A `void` pointer is assignment compatible with all object and incomplete type pointers. As such, it may be converted to and from those types. A null pointer constant can be converted to any function pointer type. All other pointer types are incompatible with each other, however, they may be able to be cast one from the other producing useful results on some systems.

conversion specifier A character sequence of the general form `%x`, where `x` may be one or more characters, used by the `printf` and `scanf` function family in interpreting formatted output and input argument lists, respectively. Sometimes called an edit mask.

conversion state^{C89} The state in which a conversion between a given multi-byte character sequence and a wide character sequence depends on the rules established by the `LC_CTYPE` category of the current locale. The current conversion state is stored in an object of type `mbstate_t`.

conversion, usual arithmetic The set of rules that dictates how operands of arithmetic type are converted when they meet across the operators `/`, `*`, `%`, `+`, `-`, `<<`, `>>`, `<`, `<=`, `>`, `>=`, `==`, `!=`, `&`, `^`, `|`, and the `:` in `?:`. The rules are as follows:

- If either operand has type `long double`, the other operand is converted to `long double`.
- If either operand has type `double`, the other operand is converted to `double`.
- If either operand has type `float`, the other operand is converted to `float`.
- Otherwise, the integer promotions are performed on both operands. Then, the following rules are applied:

- If both operands have the same type, no further conversion is needed.
- If both operands have signed integer types or both have unsigned integer types, the one with the type of lesser rank is converted to the type of the operand with greater rank.
- If the unsigned operand has greater or equal rank when compared to the other operand, the signed operand is converted to unsigned operand's type.
- If the signed operand's type can represent all of the values of the unsigned operand's type, the unsigned operand is converted to the type of the signed operand.
- Otherwise, both operands are converted to the unsigned type that corresponds to the type of the signed operand.

The addition of complex types in C99 required the following additional rules:

- If one operand has a real type and the other has a complex type, the value of the real type is converted to a complex type as follows: the real value becomes the real part of the complex result while the imaginary part of the complex result is set to positive or unsigned zero.
- If the operands are of different complex types, the the conversion rules for the corresponding real types are followed.

conversion, void type A conversion in which the (nonexistent) value of a void expression is used as the operand in an explicit cast to type `void`.

copying functions The `string.h` functions `memcpy`, `memmove`, `strcpy`, and `strncpy`, and their wide character counterparts in `string.h`, `wmemcpy`, `wmemmove`, `wscpy`, and `wcsncpy`.

`copysign[f|l]`^{C99} A function that produces a value with the magnitude of `x` and the sign of `y`.

```
#include <math.h>
double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);
```

If `x` is a NaN, the result is a NaN with the sign of `y`.

correctly rounded result^{C99} The representation of a result that is nearest in value to what the result would be given unlimited range and precision; that is, the result with unlimited range and precision rounded to the representation used.

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

```
#include <math.h>
double cos(double x);
float cosf(float x);
long double cosl(long double x);
```

If the magnitude of the argument is large, `cos` 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.

`cosh[f|l]` A function that computes the hyperbolic cosine of its argument x .

```
#include <math.h>
double cosh(double x);
float coshf(float x);
long double coshl(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.

`_cplusplus` A macro that is predefined in C++ implementations only. It is intended to allow translation units (especially headers) containing C++-specific constructs to be shared between C and C++ programs. Its existence is tested for using `#ifdef`.

`cpow[f|l]C99` A function that computes the complex power function x^y , with a branch cut for the first parameter along the negative real axis.

```
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
    long double complex y);
```

`cproj[f|l]C99` A function that returns a projection of its argument z onto the Riemann sphere.

```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

`creal[f|l]`^{C99} A function that returns the real part of its argument z .

```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

cross-compilation A development environment in which source programs are compiled on one system (the host) but are intended to run on another (the target). The resulting program generally takes complete control of the target system; that is, it does not run under the control of an operating system. Such target systems are called “freestanding systems.” *See also* environment, freestanding; environment, hosted.

`csin[f|l]`^{C99} A function that computes the complex sine of its argument z .

```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

`csinh[f|l]`^{C99} A function that computes the complex hyperbolic sine of its argument z .

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

`csqrt[f|l]`^{C99} A function that computes the complex square root of its argument z , with a branch cut along the negative real axis.

```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

`ctan[f|l]`^{C99} A function that computes the complex tangent of its argument z .

```
#include <complex.h>
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

`ctanh[f|l]`^{C99} A function that computes the complex hyperbolic tangent of its argument z .

```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

`ctgamma[f|l]`^{C99} An optional complex true gamma function that, if provided, must be declared in `complex.h`.

`ctime` A function that converts the calendar time pointed to by `timer` to a local time in the form of a string.

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

A call to `ctime` is equivalent to

```
asctime(localtime(timer))
```

so the return value points to a string of the form identical to that returned by `asctime`.

ctype.h A header that contains various character testing and conversion functions. The `is*` family members return a zero or nonzero value based on the truth of their operation while the `to*` family members return a possibly case-converted value of their character argument.

The `ctype.h` header contains definitions or declarations for the following identifiers:

<i>Name</i>	<i>Purpose</i>
<code>isalnum</code>	Test if character is alphanumeric
<code>isalpha</code>	Test if character is alphabetic
<code>isblank</code> ^{C99}	Test if character is blank
<code>iscntrl</code>	Test if character is control
<code>isdigit</code>	Test if character is digit (0–9)
<code>isgraph</code>	Test if character is graphic
<code>islower</code>	Test if character is lowercase
<code>isprint</code>	Test if character is printable
<code>ispunct</code>	Test if character is punctuation
<code>isspace</code>	Test if character is space
<code>isupper</code>	Test if character is uppercase
<code>isxdigit</code>	Test if character is hex digit
<code>tolower</code>	Produce lowercase version
<code>toupper</code>	Produce uppercase version

The behavior of some functions is locale specific.

All functions take one `int` argument. However, the `int` argument must be either representable in an `unsigned char` or it must be the macro `EOF`. If the argument has any other value, the behavior is undefined.

See also future library directions; `wctype.h`.

currency_display^{C89} Monetary formatting information defined in a structure of type `lconv`. The macro `LC_MONETARY` can be used as the category argument to `setlocale` to select the currency display information to be used.

currency_symbol^{C89} An `lconv` structure member that is a pointer to a string containing the local currency symbol applicable to the current locale. 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 "".

CX_LIMITED_RANGE pragma^{C99} *See* `#pragma STDC CX_LIMITED_RANGE`.

◇ ◇ ◇