27 Additional C99 Support for Mathematics

Simplicity does not precede complexity, but follows it.

This chapter completes our coverage of the standard library by describing five headers that are new in C99. These headers, like some of the older ones, provide support for working with numbers. However, the new headers are more specialized than the old ones. Some of them will appeal primarily to engineers, scientists, and mathematicians, who may need complex numbers as well as greater control over the representation of numbers and the way floating-point arithmetic is performed.

The first two sections discuss headers related to the integer types. The `<stdint.h>` header (Section 27.1) declares integer types that have a specified number of bits. The `<inttypes.h>` header (Section 27.2) provides macros that are useful for reading and writing values of the `<stdint.h>` types.

The next two sections describe C99's support for complex numbers. Section 27.3 includes a review of complex numbers as well as a discussion of C99's complex types. Section 27.4 then covers the `<complex.h>` header, which supplies functions that perform mathematical operations on complex numbers.

The headers discussed in the last two sections are related to the floating types. The `<tgmath.h>` header (Section 27.5) provides type-generic macros that make it easier to call library functions in `<complex.h>` and `<math.h>`. The functions in the `<fenv.h>` header (Section 27.6) give programs access to floating-point status flags and control modes.

27.1 The `<stdint.h>` Header (C99): Integer Types

The `<stdint.h>` header declares integer types containing a specified number of bits. In addition, it defines macros that represent the minimum and maximum values of these types as well as of integer types declared in other headers.

<limits.h> header ➤23.2

(These macros augment the ones in the `<limits.h>` header.) `<stdint.h>` also defines parameterized macros that construct integer constants with specific types. There are no functions in `<stdint.h>`.

The primary motivation for the `<stdint.h>` header lies in an observation made in Section 7.5, which discussed the role of type definitions in making programs portable. For example, if `i` is an `int` variable, the assignment

```
i = 100000;
```

is fine if `int` is a 32-bit type but will fail if `int` is a 16-bit type. The problem is that the C standard doesn't specify exactly how many bits an `int` value has. The standard *does* guarantee that the values of the `int` type must include all numbers between –32767 and +32767 (which requires at least 16 bits), but that's all it has to say on the matter. In the case of the variable `i`, which needs to be able to store 100000, the traditional solution is to declare `i` to be of some type `T`, where `T` is a type name created using `typedef`. The declaration of `T` can then be adjusted based on the sizes of integers in a particular implementation. (On a 16-bit machine, `T` would need to be `long int`, but on a 32-bit machine, it can be `int`.) This is the strategy that Section 7.5 discusses.

If your compiler supports C99, there's a better technique. The `<stdint.h>` header declares names for types based on the ***width*** of the type (the number of bits used to store values of the type, including the sign bit, if any). The `typedef` names declared in `<stdint.h>` may refer to basic types (such as `int`, `unsigned int`, and `long int`) or to extended integer types that are supported by a particular implementation.

sign bit ➤7.1

`<stdint.h>` Types

The types declared in `<stdint.h>` fall into five groups:

- ***Exact-width integer types.*** Each name of the form `intN_t` represents a signed integer type with *N* bits, stored in two's-complement form. (Two's complement, a technique used to represent signed integers in binary, is nearly universal among modern computers.) For example, a value of type `int16_t` would be a 16-bit signed integer. A name of the form `uintN_t` represents an unsigned integer type with *N* bits. An implementation is required to provide both `intN_t` and `uintN_t` for *N* = 8, 16, 32, and 64 if it supports integers with these widths.
- ***Minimum-width integer types.*** Each name of the form `int_leastN_t` represents a signed integer type with at least *N* bits. A name of the form `uint_leastN_t` represents an unsigned integer type with *N* or more bits. `<stdint.h>` is required to provide at least the following minimum-width types:

```
int_least8_t     uint_least8_t
int_least16_t    uint_least16_t
```

```
int_least32_t    uint_least32_t
int_least64_t    uint_least64_t
```

- ***Fastest minimum-width integer types.*** Each name of the form `int_fastN_t` represents the fastest signed integer type with at least *N* bits. (The meaning of "fastest" is up to the implementation. If there's no reason to classify a particular type as the fastest, the implementation may choose any signed integer type with at least *N* bits.) Each name of the form `uint_fastN_t` represents the fastest unsigned integer type with *N* or more bits. `<stdint.h>` is required to provide at least the following fastest minimum-width types:

```
int_fast8_t      uint_fast8_t
int_fast16_t     uint_fast16_t
int_fast32_t     uint_fast32_t
int_fast64_t     uint_fast64_t
```

- ***Integer types capable of holding object pointers.*** The `intptr_t` type represents a signed integer type that can safely store any `void *` value. More precisely, if a `void *` pointer is converted to `intptr_t` type and then back to `void *`, the resulting pointer and the original pointer will compare equal. The `uintptr_t` type is an unsigned integer type with the same property as `intptr_t`. The `<stdint.h>` header isn't required to provide either type.
- ***Greatest-width integer types.*** `intmax_t` is a signed integer type that includes all values that belong to any signed integer type. `uintmax_t` is an unsigned integer type that includes all values that belong to any unsigned integer type. `<stdint.h>` is required to provide both types, which might be wider than `long long int`.

The names in the first three groups are declared using `typedef`.

An implementation may provide exact-width integer types, minimum-width integer types, and fastest minimum-width integer types for values of *N* in addition to the ones listed above. Also, *N* isn't required to be a power of 2 (although it will normally be a multiple of 8). For example, an implementation might provide types named `int24_t` and `uint24_t`.

Limits of Specified-Width Integer Types

For each signed integer type declared in `<stdint.h>`, the header defines macros that specify the type's minimum and maximum values. For each unsigned integer type, `<stdint.h>` defines a macro that specifies the type's maximum value. The first three rows of Table 27.1 show the values of these macros for the exact-width integer types. The remaining rows show the constraints imposed by the C99 standard on the minimum and maximum values of the other `<stdint.h>` types. (The precise values of these macros are implementation-defined.) All macros in the table represent constant expressions.

Table 27.1 <stdint.h> Limit Macros for Specified-Width Integer Types

Name	*Value*	*Description*
INT*N*_MIN	$-(2^{N-1})$	Minimum int*N*_t value
INT*N*_MAX	$2^{N-1}-1$	Maximum int*N*_t value
UINT*N*_MAX	$2^{N}-1$	Maximum uint*N*_t value
INT_LEAST*N*_MIN	$\le -(2^{N-1}-1)$	Minimum int_least*N*_t value
INT_LEAST*N*_MAX	$\ge 2^{N-1}-1$	Maximum int_least*N*_t value
UINT_LEAST*N*_MAX	$\ge 2^{N}-1$	Maximum uint_least*N*_t value
INT_FAST*N*_MIN	$\le -(2^{N-1}-1)$	Minimum int_fast*N*_t value
INT_FAST*N*_MAX	$\ge 2^{N-1}-1$	Maximum int_fast*N*_t value
UINT_FAST*N*_MAX	$\ge 2^{N}-1$	Maximum uint_fast*N*_t value
INTPTR_MIN	$\le -(2^{15}-1)$	Minimum intptr_t value
INTPTR_MAX	$\ge 2^{15}-1$	Maximum intptr_t value
UINTPTR_MAX	$\ge 2^{16}-1$	Maximum uintptr_t value
INTMAX_MIN	$\le -(2^{63}-1)$	Minimum intmax_t value
INTMAX_MAX	$\ge 2^{63}-1$	Maximum intmax_t value
UINTMAX_MAX	$\ge 2^{64}-1$	Maximum uintmax_t value

Limits of Other Integer Types

When the C99 committee created the <stdint.h> header, they decided that it would be a good place to put macros describing the limits of integer types besides the ones declared in <stdint.h> itself. These types are ptrdiff_t, size_t, and wchar_t (which belong to <stddef.h>), sig_atomic_t (declared in <signal.h>), and wint_t (declared in <wchar.h>). Table 27.2 lists these macros and shows the value of each (or any constraints on the value imposed by the C99 standard). In some cases, the constraints on the minimum and maximum values of a type depend on whether the type is signed or unsigned. The macros in Table 27.2, like the ones in Table 27.1, represent constant expressions.

<stddef.h> header ➤21.4
<signal.h> header ➤24.3
<wchar.h> header ➤25.5

Macros for Integer Constants

The <stdint.h> header also provides function-like macros that are able to convert an integer constant (expressed in decimal, octal, or hexadecimal, but without a U and/or L suffix) into a constant expression belonging to a minimum-width integer type or greatest-width integer type.

integer constants ➤7.1

For each int_least*N*_t type declared in <stdint.h>, the header defines a parameterized macro named INT*N*_C that converts an integer constant to this type (possibly using the integer promotions). For each uint_least*N*_t type, there's a similar parameterized macro named UINT*N*_C. These macros are useful for initializing variables, among other things. For example, if i is a variable of type int_least32_t, writing

integer promotions ➤7.4

Table 27.2 <stdint.h> Limit Macros for Other Integer Types

Name	*Value*	*Description*
PTRDIFF_MIN	≤–65535	Minimum ptrdiff_t value
PTRDIFF_MAX	≥+65535	Maximum ptrdiff_t value
SIG_ATOMIC_MIN	≤–127 (if signed) 0 (if unsigned)	Minimum sig_atomic_t value
SIG_ATOMIC_MAX	≥+127 (if signed) ≥255 (if unsigned)	Maximum sig_atomic_t value
SIZE_MAX	≥65535	Maximum size_t value
WCHAR_MIN	≤–127 (if signed) 0 (if unsigned)	Minimum wchar_t value
WCHAR_MAX	≥+127 (if signed) ≥255 (if unsigned)	Maximum wchar_t value
WINT_MIN	≤–32767 (if signed) 0 (if unsigned)	Minimum wint_t value
WINT_MAX	≥+32767 (if signed) ≥65535 (if unsigned)	Maximum wint_t value

```
i = 100000;
```

is problematic, because the constant 100000 might be too large to represent using type int (if int is a 16-bit type). However, the statement

```
i = INT32_C(100000);
```

is safe. If int_least32_t represents the int type, then INT32_C(100000) has type int. But if int_least32_t corresponds to long int, then INT32_C(100000) has type long int.

<stdint.h> has two other parameterized macros. INTMAX_C converts an integer constant to type intmax_t, and UINTMAX_C converts an integer constant to type uintmax_t.

27.2 The <inttypes.h> Header (C99)

Format Conversion of Integer Types

The <inttypes.h> header is closely related to the <stdint.h> header, the topic of Section 27.1. In fact, <inttypes.h> includes <stdint.h>, so programs that include <inttypes.h> don't need to include <stdint.h> as well. The <inttypes.h> header extends <stdint.h> in two ways. First, it defines macros that can be used in ...printf and ...scanf format strings for input/output of the integer types declared in <stdint.h>. Second, it provides functions for working with greatest-width integers.

Q&A

Macros for Format Specifiers

The types declared in the `<stdint.h>` header can be used to make programs more portable, but they create new headaches for the programmer. Consider the problem of displaying the value of the variable `i`, where `i` has type `int_least32_t`. The statement

```
printf("i = %d\n", i);
```

may not work, because `i` doesn't necessarily have `int` type. If `int_least32_t` is another name for the `long int` type, then the correct conversion specification is `%ld`, not `%d`. In order to use the ...`printf` and ...`scanf` functions in a portable manner, we need a way to write conversion specifications that correspond to each of the types declared in `<stdint.h>`. That's where the `<inttypes.h>` header comes in. For each `<stdint.h>` type, `<inttypes.h>` provides a macro that expands into a string literal containing the proper conversion specifier for that type.

Each macro name has three parts:

- The name begins with either `PRI` or `SCN`, depending on whether the macro will be used in a call of a ...`printf` function or a ...`scanf` function.
- Next comes a one-letter conversion specifier (`d` or `i` for a signed type; `o`, `u`, `x`, or `X` for an unsigned type).
- The last part of the name indicates which `<stdint.h>` type is involved. For example, the name of a macro that corresponds to the `int_least`*N*`_t` type would end with `LEAST`*N*.

Let's return to our previous example, which involved displaying an integer of type `int_least32_t`. Instead of using `d` as the conversion specifier, we'll switch to the `PRIdLEAST32` macro. To use the macro, we'll split the `printf` format string into three pieces and replace the `d` in `%d` by `PRIdLEAST32`:

```
printf("i = %" PRIdLEAST32 "\n", i);
```

The value of `PRIdLEAST32` is probably either `"d"` (if `int_least32_t` is the same as the `int` type) or `"ld"` (if `int_least32_t` is the same as `long int`). Let's assume that it's `"ld"` for the sake of discussion. After macro replacement, the statement becomes

```
printf("i = %" "ld" "\n", i);
```

Once the compiler joins the three string literals into one (which it will do automatically), the statement will have the following appearance:

```
printf("i = %ld\n", i);
```

Note that we can still include flags, a field width, and other options in our conversion specification; `PRIdLEAST32` supplies only the conversion specifier and possibly a length modifier, such as the letter `l`.

Table 27.3 lists the `<inttypes.h>` macros.

Table 27.3
Format-Specifier Macros in <inttypes.h>

...printf *Macros for Signed Integers*				
PRId*N*	PRIdLEAST*N*	PRIdFAST*N*	PRIdMAX	PRIdPTR
PRIi*N*	PRIiLEAST*N*	PRIiFAST*N*	PRIiMAX	PRIiPTR
...printf *Macros for Unsigned Integers*				
PRIo*N*	PRIoLEAST*N*	PRIoFAST*N*	PRIoMAX	PRIoPTR
PRIu*N*	PRIuLEAST*N*	PRIuFAST*N*	PRIuMAX	PRIuPTR
PRIx*N*	PRIxLEAST*N*	PRIxFAST*N*	PRIxMAX	PRIxPTR
PRIX*N*	PRIXLEAST*N*	PRIXFAST*N*	PRIXMAX	PRIXPTR
...scanf *Macros for Signed Integers*				
SCNd*N*	SCNdLEAST*N*	SCNdFAST*N*	SCNdMAX	SCNdPTR
SCNi*N*	SCNiLEAST*N*	SCNiFAST*N*	SCNiMAX	SCNiPTR
...scanf *Macros for Unsigned Integers*				
SCNo*N*	SCNoLEAST*N*	SCNoFAST*N*	SCNoMAX	SCNoPTR
SCNu*N*	SCNuLEAST*N*	SCNuFAST*N*	SCNuMAX	SCNuPTR
SCNx*N*	SCNxLEAST*N*	SCNxFAST*N*	SCNxMAX	SCNxPTR

Functions for Greatest-Width Integer Types

```
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
intmax_t strtoimax(const char * restrict nptr,
                   char ** restrict endptr,
                   int base);
uintmax_t strtoumax(const char * restrict nptr,
                    char ** restrict endptr,
                    int base);
intmax_t wcstoimax(const wchar_t * restrict nptr,
                   wchar_t ** restrict endptr,
                   int base);
uintmax_t wcstoumax(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr,
                    int base);
```

In addition to defining macros, the <inttypes.h> header provides functions for working with greatest-width integers, which were introduced in Section 27.1. A greatest-width integer has type intmax_t (the widest signed integer type supported by an implementation) or uintmax_t (the widest unsigned integer type). These types might be the same width as the long long int type, but they could be wider. For example, long long int might be 64 bits wide and intmax_t and uintmax_t might be 128 bits wide.

imaxabs
imaxdiv

The imaxabs and imaxdiv functions are greatest-width versions of the integer arithmetic functions declared in <stdlib.h>. The imaxabs function returns the absolute value of its argument. Both the argument and the return value have type intmax_t. The imaxdiv function divides its first argument by its

<stdlib.h> header ➤26.2

second, returning an `imaxdiv_t` value. `imaxdiv_t` is a structure that contains both a quotient member (named `quot`) and a remainder member (`rem`); both members have type `intmax_t`.

strtoimax
strtoumax

The `strtoimax` and `strtoumax` functions are greatest-width versions of the numeric conversion functions of `<stdlib.h>`. The `strtoimax` function is the same as `strtol` and `strtoll`, except that it returns a value of type `intmax_t`. The `strtoumax` function is equivalent to `strtoul` and `strtoull`, except that it returns a value of type `uintmax_t`. Both `strtoimax` and `strtoumax` return zero if no conversion could be performed. Both functions store `ERANGE` in `errno` if a conversion produces a value that's outside the range of the function's return type. In addition, `strtoimax` returns the smallest or largest `intmax_t` value (`INTMAX_MIN` or `INTMAX_MAX`); `strtoumax` returns the largest `uintmax_t` value, `UINTMAX_MAX`.

wcstoimax
wcstoumax

`<wchar.h>` header ➤25.5

The `wcstoimax` and `wcstoumax` functions are greatest-width versions of the wide-string numeric conversion functions of `<wchar.h>`. The `wcstoimax` function is the same as `wcstol` and `wcstoll`, except that it returns a value of type `intmax_t`. The `wcstoumax` function is equivalent to `wcstoul` and `wcstoull`, except that it returns a value of type `uintmax_t`. Both `wcstoimax` and `wcstoumax` return zero if no conversion could be performed. Both functions store `ERANGE` in `errno` if a conversion produces a value that's outside the range of the function's return type. In addition, `wcstoimax` returns the smallest or largest `intmax_t` value (`INTMAX_MIN` or `INTMAX_MAX`); `wcstoumax` returns the largest `uintmax_t` value, `UINTMAX_MAX`.

27.3 Complex Numbers (C99)

Complex numbers are used in scientific and engineering applications as well as in mathematics. C99 provides several complex types, allows operators to have complex operands, and adds a header named `<complex.h>` to the standard library. There's a catch, though: complex numbers aren't supported by all implementations of C99. Section 14.3 discussed the difference between a *hosted* C99 implementation and a *freestanding* implementation. A hosted implementation must accept any program that conforms to the C99 standard, whereas a freestanding implementation doesn't have to compile programs that use complex types or standard headers other than `<float.h>`, `<iso646.h>`, `<limits.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, and `<stdint.h>`. Thus, a freestanding implementation may lack both complex types and the `<complex.h>` header.

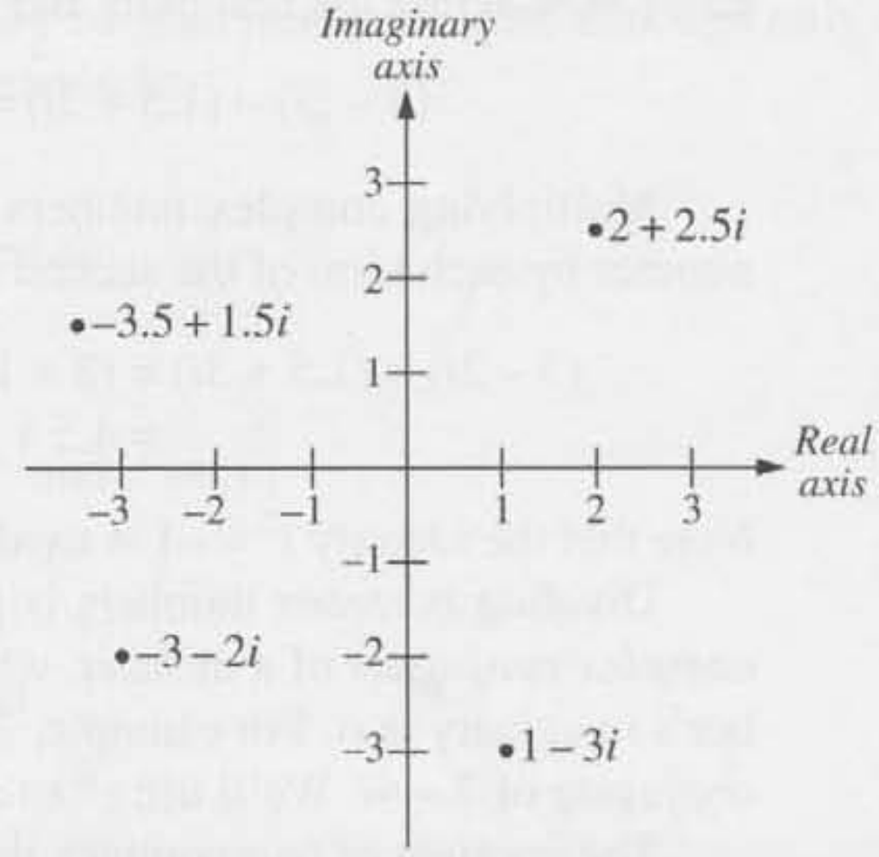
We'll start with a review of the mathematical definition of complex numbers and complex arithmetic. We'll then look at C99's complex types and the operations that can be performed on values of these types. Coverage of complex numbers continues in Section 27.4, which describes the `<complex.h>` header.

Definition of Complex Numbers

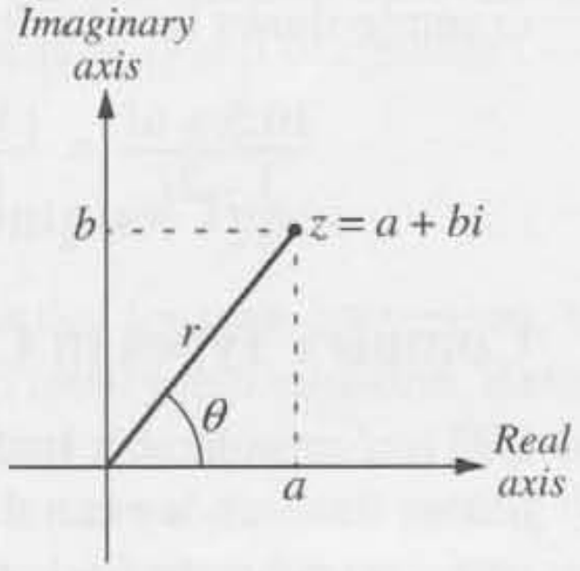
Let i be the square root of -1 (a number such that $i^2 = -1$). i is known as the ***imaginary unit;*** engineers often represent it by the symbol j instead of i. A ***complex number*** has the form $a + bi$, where a and b are real numbers. a is said to be the ***real part*** of the number, and b is the ***imaginary part.*** Note that the complex numbers include the real numbers as a special case (when $b = 0$).

Why are complex numbers useful? For one thing, they allow solutions to problems that are otherwise unsolvable. Consider the equation $x^2 + 1 = 0$, which has no solution if x is restricted to the real numbers. If complex numbers are allowed, there are two solutions: $x = i$ and $x = -i$.

Complex numbers can be thought of as points in a two-dimensional space known as the ***complex plane.*** Each complex number—a point in the complex plane—is represented by Cartesian coordinates, where the real part of the number corresponds to the x-coordinate of the point, and the imaginary part corresponds to the y-coordinate. For example, the complex numbers $2 + 2.5i$, $1 - 3i$, $-3 - 2i$, and $-3.5 + 1.5i$ can be plotted as follows:



An alternative system known as ***polar coordinates*** can also be used to specify a point on the complex plane. With polar coordinates, a complex number z is represented by the values r and θ, where r is the length of a line segment from the origin to z, and θ is the angle between this segment and the real axis:



r is called the ***absolute value*** of z. (The absolute value is also known as the *norm, modulus,* or *magnitude*.) θ is said to be the ***argument*** (or *phase angle*) of z. The absolute value of $a + bi$ is given by the following equation:

$$|a + bi| = \sqrt{a^2 + b^2}$$

For additional information about converting from Cartesian coordinates to polar coordinates and vice versa, see the Programming Projects at the end of the chapter.

Complex Arithmetic

The sum of two complex numbers is found by separately adding the real parts of the two numbers and the imaginary parts. For example,

$$(3 - 2i) + (1.5 + 3i) = (3 + 1.5) + (-2 + 3)i = 4.5 + i$$

The difference of two complex numbers is computed in a similar manner, by separately subtracting the real parts and the imaginary parts. For example,

$$(3 - 2i) - (1.5 + 3i) = (3 - 1.5) + (-2 - 3)i = 1.5 - 5i$$

Multiplying complex numbers is done by multiplying each term of the first number by each term of the second and then summing the products:

$$\begin{aligned}(3 - 2i) \times (1.5 + 3i) &= (3 \times 1.5) + (3 \times 3i) + (-2i \times 1.5) + (-2i \times 3i) \\ &= 4.5 + 9i - 3i - 6i^2 = 10.5 + 6i\end{aligned}$$

Note that the identity $i^2 = -1$ is used to simplify the result.

Dividing complex numbers is a bit harder. First, we need the concept of the ***complex conjugate*** of a number, which is found by switching the sign of the number's imaginary part. For example, $7 - 4i$ is the conjugate of $7 + 4i$, and $7 + 4i$ is the conjugate of $7 - 4i$. We'll use z^* to denote the conjugate of a complex number z.

The quotient of two complex numbers y and z is given by the formula

$$y/z = yz^*/zz^*$$

It turns out that zz^* is always a real number, so dividing zz^* into yz^* is easy (just divide both the real part and the imaginary part of yz^* separately). The following example shows how to divide $10.5 + 6i$ by $3 - 2i$:

$$\frac{10.5 + 6i}{3 - 2i} = \frac{(10.5 + 6i)(3 + 2i)}{(3 - 2i)(3 + 2i)} = \frac{19.5 + 39i}{13} = 1.5 + 3i$$

Complex Types in C99

C99 has considerable built-in support for complex numbers. Without including any library headers, we can declare variables that represent complex numbers and then perform arithmetic and other operations on these variables.

C99 provides three complex types, which were first introduced in Section 7.2: `float _Complex`, `double _Complex`, and `long double _Complex`. These types can be used in the same way as other types in C: to declare variables, parameters, return types, array elements, members of structures and unions, and so forth. For example, we could declare three variables as follows:

```
float _Complex x;
double _Complex y;
long double _Complex z;
```

Each of these variables is stored just like an array of two ordinary floating-point numbers. Thus, `y` is stored as two adjacent `double` values, with the first value containing the real part of `y` and the second containing the imaginary part.

C99 also allows implementations to provide imaginary types (the keyword `_Imaginary` is reserved for this purpose) but doesn't make this a requirement.

Operations on Complex Numbers

Complex numbers may be used in expressions, although only the following operators allow complex operands:

- Unary + and -
- Logical negation (`!`)
- `sizeof`
- Cast
- Multiplicative (`*` and `/` only)
- Additive (`+` and `-`)
- Equality (`==` and `!=`)
- Logical *and* (`&&`)
- Logical *or* (`||`)
- Conditional (`?:`)
- Simple assignment (`=`)
- Compound assignment (`*=`, `/=`, `+=`, and `-=` only)
- Comma (`,`)

Some notable omissions from the list include the relational operators (`<`, `<=`, `>`, and `>=`), along with the increment (`++`) and decrement (`--`) operators.

Conversion Rules for Complex Types

Section 7.4 described the C99 rules for type conversion, but without covering the complex types. It's now time to rectify that situation. Before we get to the conversion rules, though, we'll need some new terminology. For each floating type there is a ***corresponding real type.*** In the case of the real floating types (`float`, `double`, and `long double`), the corresponding real type is the same as the original type.

For the complex types, the corresponding real type is the original type without the word `_Complex`. (The corresponding real type for `float _Complex` is `float`, for example.)

We're now ready to discuss the general rules that govern type conversions involving complex types. I'll group them into three categories.

- ***Complex to complex.*** The first rule concerns conversions from one complex type to another, such as converting from `float _Complex` to `double _Complex`. In this situation, the real and imaginary parts are converted separately, using the rules for the corresponding real types (see Section 7.4). In our example, the real part of the `float _Complex` value would be converted to `double`, yielding the real part of the `double _Complex` value; the imaginary part would be converted to `double` in a similar fashion.
- ***Real to complex.*** When a value of a real type is converted to a complex type, the real part of the number is converted using the rules for converting from one real type to another. The imaginary part of the result is set to positive or unsigned zero.
- ***Complex to real.*** When a value of a complex type is converted to a real type, the imaginary part of the number is discarded; the real part is converted using the rules for converting from one real type to another.

One particular set of type conversions, known as the usual arithmetic conversions, are automatically applied to the operands of most binary operators. There are special rules for performing the usual arithmetic conversions when at least one of the two operands has a complex type:

1. If the corresponding real type of either operand is `long double`, convert the other operand so that its corresponding real type is `long double`.
2. Otherwise, if the corresponding real type of either operand is `double`, convert the other operand so that its corresponding real type is `double`.
3. Otherwise, one of the operands must have `float` as its corresponding real type. Convert the other operand so that its corresponding real type is also `float`.

A real operand still belongs to a real type after conversion, and a complex operand still belongs to a complex type.

Normally, the goal of the usual arithmetic conversions is to convert both operands to a common type. However, when a real operand is mixed with a complex operand, performing the usual arithmetic conversions causes the operands to have a common real type, but not necessarily the *same* type. For example, adding a `float` operand and a `double _Complex` operand causes the `float` operand to be converted to `double` rather than `double _Complex`. The type of the result will be the complex type whose corresponding real type matches the common real type. In our example, the type of the result will be `double _Complex`.

27.4 The <complex.h> Header (C99): Complex Arithmetic

As we saw in Section 27.3, C99 has significant built-in support for complex numbers. The <complex.h> header provides additional support in the form of mathematical functions on complex numbers, as well as some very useful macros and a pragma. Let's look at the macros first.

<complex.h> Macros

The <complex.h> header defines the macros shown in Table 27.4.

Table 27.4 <complex.h> Macros

Name	*Value*
complex	_Complex
_Complex_I	Imaginary unit; has type const float _Complex
I	_Complex_I

complex serves as an alternative name for the awkward _Complex keyword. We've seen a situation like this before with the Boolean type: the C99 committee chose a new keyword (_Bool) that shouldn't break existing programs, but provided a better name (bool) as a macro defined in the <stdbool.h> header. Programs that include <complex.h> may use complex instead of _Complex, just as programs that include <stdbool.h> may use bool rather than _Bool.

<stdbool.h> header ➤21.5

The I macro plays an important role in C99. There's no special language feature for creating a complex number from its real part and imaginary part. Instead, a complex number can be constructed by multiplying the imaginary part by I and adding the real part:

```
double complex dc = 2.0 + 3.5 * I;
```

The value of the variable dc is $2 + 3.5i$.

Note that both _Complex_I and I represent the imaginary unit *i*. Presumably most programmers will use I rather than _Complex_I. However, since I might already be used in existing code for some other purpose, _Complex_I is available as a backup. If the name I causes a conflict, it can always be undefined:

```
#include <complex.h>
#undef I
```

The programmer might then define a different—but still short—name for *i*, such as J:

```
#define J _Complex_I
```

Also note that the type of `_Complex_I` (and hence the type of `I`) is `float _Complex`, not `double _Complex`. When it's used in expressions, `I` will automatically be widened to `double _Complex` or `long double _Complex` if necessary.

The `CX_LIMITED_RANGE` Pragma

#pragma directive ➤14.5

The `<complex.h>` header provides a pragma named `CX_LIMITED_RANGE` that allows the compiler to use the following standard formulas for multiplication, division, and absolute value:

$$(a+bi)\times(c+di) = (ac-bd)+(bc+ad)i$$

$$(a+bi)/(c+di) = [(ac+bd)+(bc-ad)i]/(c^2+d^2)$$

$$|a+bi| = \sqrt{a^2+b^2}$$

Using these formulas may cause anomalous results in some cases because of overflow or underflow; moreover, the formulas don't handle infinities properly. Because of these potential problems, C99 doesn't use the formulas without the programmer's permission.

The `CX_LIMITED_RANGE` pragma has the following appearance:

```
#pragma STDC CX_LIMITED_RANGE on-off-switch
```

where *on-off-switch* is either `ON`, `OFF`, or `DEFAULT`. If the pragma is used with the value `ON`, it allows the compiler to use the formulas listed above. The value `OFF` causes the compiler to perform the calculations in a way that's safer but possibly slower. The default setting, indicated by the `DEFAULT` choice, is equivalent to `OFF`.

The duration of the `CX_LIMITED_RANGE` pragma depends on where it's used in a program. When it appears at the top level of a source file, outside any external declarations, it remains in effect until the next `CX_LIMITED_RANGE` pragma or the end of the file. The only other place that a `CX_LIMITED_RANGE` pragma might appear is at the beginning of a compound statement (possibly the body of a function); in that case, the pragma remains in effect until the next `CX_LIMITED_RANGE` pragma (even one inside a nested compound statement) or the end of the compound statement. At the end of a compound statement, the state of the switch returns to its value before the compound statement was entered.

`<complex.h>` Functions

The `<complex.h>` header provides functions similar to those in the C99 version of `<math.h>`. The `<complex.h>` functions are divided into groups, just as they were in `<math.h>`: trigonometric, hyperbolic, exponential and logarithmic, and power and absolute-value. The only functions that are unique to complex numbers are the manipulation functions, the last group discussed in this section.

Each <complex.h> function comes in three versions: a float complex version, a double complex version, and a long double complex version. The name of the float complex version ends with f, and the name of the long double complex version ends with l.

Before we delve into the <complex.h> functions, a few general comments are in order. First, as with the <math.h> functions, the <complex.h> functions expect angle measurements to be in radians, not degrees. Second, when an error occurs, the <complex.h> functions may store a value in the errno variable, but aren't required to.

errno variable ➤24.2

There's one last thing we'll need before tackling the <complex.h> functions. The term ***branch cut*** often appears in descriptions of functions that might conceivably have more than one possible return value. In the realm of complex numbers, choosing which value to return creates a branch cut: a curve (often just a line) in the complex plane around which a function is discontinuous. Branch cuts are usually not unique, but rather are determined by convention. An exact definition of branch cuts takes us further into complex analysis than I'd like to go, so I'll simply reproduce the restrictions from the C99 standard without further explanation.

Trigonometric Functions

```
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);

double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);

double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);

double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);

double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);

double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

cacos The cacos function computes the complex arc cosine, with branch cuts outside the interval [–1, +1] along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis.

casin The `casin` function computes the complex arc sine, with branch cuts outside the interval [–1, +1] along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis.

catan The `catan` function computes the complex arc tangent, with branch cuts outside the interval [$-i$, $+i$] along the imaginary axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis.

ccos
csin
ctan

The `ccos` function computes the complex cosine, the `csin` function computes the complex sine, and the `ctan` function computes the complex tangent.

Hyperbolic Functions

```
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);

double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);

double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);

double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);

double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);

double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

cacosh The `cacosh` function computes the complex arc hyperbolic cosine, with a branch cut at values less than 1 along the real axis. The return value lies in a half-strip of nonnegative values along the real axis and in the interval [$-i\pi$, $+i\pi$] along the imaginary axis.

casinh The `casinh` function computes the complex arc hyperbolic sine, with branch cuts outside the interval [$-i$, $+i$] along the imaginary axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval [$-i\pi/2$, $+i\pi/2$] along the imaginary axis.

catanh The `catanh` function computes the complex arc hyperbolic tangent, with branch cuts outside the interval [–1, +1] along the real axis. The return value lies in

a strip mathematically unbounded along the real axis and in the interval [$-i\pi/2$, $+i\pi/2$] along the imaginary axis.

ccosh *csinh* *ctanh*

The `ccosh` function computes the complex hyperbolic cosine, the `csinh` function computes the complex hyperbolic sine, and the `ctanh` function computes the complex hyperbolic tangent.

Exponential and Logarithmic Functions

```
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);

double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

cexp

The `cexp` function computes the complex base-e exponential value.

clog

The `clog` function computes the complex natural (base-e) logarithm, with a branch cut along the negative real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval [$-i\pi$, $+i\pi$] along the imaginary axis.

Power and Absolute-Value Functions

```
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);

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);

double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

cabs

The `cabs` function computes the complex absolute value.

cpow

The `cpow` function returns `x` raised to the power `y`, with a branch cut for the first parameter along the negative real axis.

csqrt

The `csqrt` function computes the complex square root, with a branch cut along the negative real axis. The return value lies in the right half-plane (including the imaginary axis).

Manipulation Functions

```
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);

double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);

double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);

double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);

double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

carg The `carg` function returns the argument (phase angle) of `z`, with a branch cut along the negative real axis. The return value lies in the interval $[-\pi, +\pi]$.

cimag The `cimag` function returns the imaginary part of `z`.

conj The `conj` function returns the complex conjugate of `z`.

cproj The `cproj` function computes a projection of `z` onto the Riemann sphere. The return value is equal to `z` unless one of its parts is infinite, in which case `cproj` returns `INFINITY + I * copysign(0.0, cimag(z))`.

creal The `creal` function returns the real part of `z`.

PROGRAM

Finding the Roots of a Quadratic Equation

The roots of the quadratic equation

$$ax^2 + bx + c = 0$$

are given by the ***quadratic formula:***

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In general, the value of x will be a complex number, because the square root of $b^2 - 4ac$ is imaginary if $b^2 - 4ac$ (known as the ***discriminant***) is less than 0.

For example, suppose that $a = 5$, $b = 2$, and $c = 1$, which gives us the equation

$$5x^2 + 2x + 1 = 0$$

The value of the discriminant is $4 - 20 = -16$, so the roots of the equation will be

complex numbers. The following program, which uses several <complex.h> functions, computes and displays the roots.

quadratic.c

```
/* Finds the roots of the equation 5x**2 + 2x + 1 = 0 */

#include <complex.h>
#include <stdio.h>

int main(void)
{
  double a = 5, b = 2, c = 1;
  double complex discriminant_sqrt = csqrt(b * b - 4 * a * c);
  double complex root1 = (-b + discriminant_sqrt) / (2 * a);
  double complex root2 = (-b - discriminant_sqrt) / (2 * a);

  printf("root1 = %g + %gi\n", creal(root1), cimag(root1));
  printf("root2 = %g + %gi\n", creal(root2), cimag(root2));

  return 0;
}
```

Here's the output of the program:

```
root1 = -0.2 + 0.4i
root2 = -0.2 + -0.4i
```

The quadratic.c program shows how to display a complex number by extracting the real and imaginary parts and then writing each as a floating-point number. printf lacks conversion specifiers for complex numbers, so there's no easier technique. There's also no shortcut for reading complex numbers; a program will need to obtain the real and imaginary parts separately and then combine them into a single complex number.

27.5 The <tgmath.h> Header (C99): Type-Generic Math

The <tgmath.h> header provides parameterized macros with names that match functions in <math.h> and <complex.h>. These ***type-generic macros*** can detect the types of the arguments passed to them and substitute a call of the appropriate version of a <math.h> or <complex.h> function.

In C99, there are multiple versions of many math functions, as we saw in Sections 23.3, 23.4, and 27.4. For example, the sqrt function comes in a double version (sqrt), a float version (sqrtf), and a long double version (sqrtl), as well as three versions for complex numbers (csqrt, csqrtf, and csqrtl). By using <tgmath.h>, the programmer can simply invoke sqrt without having to worry about which version is needed: the call sqrt(x) could be a call of any of the six versions of sqrt, depending on the type of x.

One advantage of using <tgmath.h> is that calls of math functions become easier to write (and read!). More importantly, a call of a type-generic macro won't have to be modified in the future should the type of its argument(s) change.

The <tgmath.h> header includes both <math.h> and <complex.h>, by the way, so including <tgmath.h> provides access to the functions in both headers.

Type-Generic Macros

The type-generic macros defined in the <tgmath.h> header fall into three groups, depending on whether they correspond to functions in <math.h>, <complex.h>, or both headers.

Table 27.5 lists the type-generic macros that correspond to functions in both <math.h> and <complex.h>. Note that the name of each type-generic macro matches the name of the "unsuffixed" <math.h> function (acos as opposed to acosf or acosl, for example).

Table 27.5
Type-Generic Macros in <tgmath.h> (Group 1)

<math.h> *Function*	<complex.h> *Function*	*Type-Generic Macro*
acos	cacos	acos
asin	casin	asin
atan	catan	atan
acosh	cacosh	acosh
asinh	casinh	asinh
atanh	catanh	atanh
cos	ccos	cos
sin	csin	sin
tan	ctan	tan
cosh	ccosh	cosh
sinh	csinh	sinh
tanh	ctanh	tanh
exp	cexp	exp
log	clog	log
pow	cpow	pow
sqrt	csqrt	sqrt
fabs	cabs	fabs

The macros in the second group (Table 27.6) correspond only to functions in <math.h>. Each macro has the same name as the unsuffixed <math.h> function. Passing a complex argument to any of these macros causes undefined behavior.

Table 27.6
Type-Generic Macros in <tgmath.h> (Group 2)

atan2	fma	llround	remainder
cbrt	fmax	log10	remquo
ceil	fmin	log1p	rint
copysign	fmod	log2	round
erf	frexp	logb	scalbn
erfc	hypot	lrint	scalbln
exp2	ilogb	lround	tgamma
expm1	ldexp	nearbyint	trunc
fdim	lgamma	nextafter	
floor	llrint	nexttoward	

The macros in the final group (Table 27.7) correspond only to functions in <complex.h>.

Table 27.7
Type-Generic Macros in <tgmath.h> (Group 3)

carg	conj	creal
cimag	cproj	

Between the three tables, all functions in <math.h> and <complex.h> that have multiple versions are accounted for, with the exception of modf. Q&A

Invoking a Type-Generic Macro

To understand what happens when a type-generic macro is invoked, we first need the concept of a ***generic parameter.*** Consider the prototypes for the three versions of the nextafter function (from <math.h>):

```
double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);
```

The types of both x and y change depending on the version of nextafter, so both parameters are generic. Now consider the prototypes for the three versions of the nexttoward function:

```
double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);
```

The first parameter is generic, but the second is not (it always has type long double). Generic parameters always have type double (or double complex) in the unsuffixed version of a function.

When a type-generic macro is invoked, the first step is to determine whether it should be replaced by a <math.h> function or a <complex.h> function. (This step doesn't apply to the macros in Table 27.6, which are always replaced by a <math.h> function, or the macros in Table 27.7, which are always replaced by a <complex.h> function.) The rule is simple: if any argument corresponding to a generic parameter is complex, then a <complex.h> function is chosen; otherwise, a <math.h> function is selected.

The next step is to deduce which version of the <math.h> function or <complex.h> function is being called. Let's assume that the function being called belongs to <math.h>. (The rules for the <complex.h> case are analogous.) The following rules are used, in the order listed:

1. If any argument corresponding to a generic parameter has type long double, the long double version of the function is called.
2. If any argument corresponding to a generic parameter has type double or any integer type, the double version of the function is called.
3. Otherwise, the float version of the function is called.

Rule 2 is a little unusual: it states that an integer argument causes the double version of a function to be called, not the float version, which you might expect. Q&A

As an example, assume that the following variables have been declared:

```
int i;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

For each macro invocation in the left column below, the corresponding function call appears in the right column:

Macro Invocation	*Equivalent Function Call*
`sqrt(i)`	`sqrt(i)`
`sqrt(f)`	`sqrtf(f)`
`sqrt(d)`	`sqrt(d)`
`sqrt(ld)`	`sqrtl(ld)`
`sqrt(fc)`	`csqrtf(fc)`
`sqrt(dc)`	`csqrt(dc)`
`sqrt(ldc)`	`csqrtl(ldc)`

Note that writing `sqrt(i)` causes the `double` version of `sqrt` to be called, not the `float` version.

These rules also cover macros with more than one parameter. For example, the macro invocation `pow(ld, f)` will be replaced by the call `powl(ld, f)`. Both of `pow`'s parameters are generic; because one of the arguments has type `long double`, rule 1 states that the `long double` version of `pow` will be called.

27.6 The `<fenv.h>` Header (C99): Floating-Point Environment

IEEE Standard 754 is the most widely used representation for floating-point numbers. (This standard is also known as IEC 60559, which is how the C99 standard refers to it.) The purpose of the `<fenv.h>` header is to give programs access to the floating-point status flags and control modes specified in the IEEE standard. Although `<fenv.h>` was designed in a general fashion that allows it to work with other floating-point representations, supporting the IEEE standard was the reason for the header's creation.

A discussion of why programs might need access to status flags and control modes is beyond the scope of this book. For good examples, see "What every computer scientist should know about floating-point arithmetic" by David Goldberg (*ACM Computing Surveys*, vol. 23, no. 1 (March 1991): 5–48), which can be found on the Web.

Floating-Point Status Flags and Control Modes

Section 7.2 discussed some of the basic properties of IEEE Standard 754. Section 23.4, which covered the C99 additions to the <math.h> header, gave additional detail. Some of that discussion, particularly concerning exceptions and rounding directions, is directly relevant to the <fenv.h> header. Before we continue, let's review some of the material from Section 23.4 as well as define a few new terms.

A ***floating-point status flag*** is a system variable that's set when a floating-point exception is raised. In the IEEE standard, there are five types of floating-point exceptions: *overflow, underflow, division by zero, invalid operation* (the result of an arithmetic operation was NaN), and *inexact* (the result of an arithmetic operation had to be rounded). Each exception has a corresponding status flag.

The <fenv.h> header declares a type named fexcept_t that's used for working with the floating-point status flags. An fexcept_t object represents the collective value of these flags. Although fexcept_t can simply be an integer type, with single bits representing individual flags, the C99 standard doesn't make this a requirement. Other alternatives exist, including the possibility that fexcept_t is a structure, with one member for each exception. This member could store additional information about the corresponding exception, such as the address of the floating-point instruction that caused the exception to be raised.

A ***floating-point control mode*** is a system variable that may be set by a program to change the future behavior of floating-point arithmetic. The IEEE standard requires a "directed-rounding" mode that controls the rounding direction when a number can't be represented exactly using a floating-point representation. There are four rounding directions: (1) *Round toward nearest.* Rounds to the nearest representable value. If a number falls halfway between two values, it's rounded to the "even" value (the one whose least significant bit is zero). (2) *Round toward zero.* (3) *Round toward positive infinity.* (4) *Round toward negative infinity.* The default rounding direction is round toward nearest. Some implementations of the IEEE standard provide two additional control modes: a mode that controls rounding precision and a "trap enablement" mode that determines whether a floating-point processor will trap (or stop) when an exception is raised.

The term ***floating-point environment*** refers to the combination of floating-point status flags and control modes supported by a particular implementation. A value of type fenv_t represents an entire floating-point environment. The fenv_t type, like the fexcept_t type, is declared in <fenv.h>.

<fenv.h> Macros

The <fenv.h> header potentially defines the macros listed in Table 27.8. Only two of these macros (FE_ALL_EXCEPT and FE_DFL_ENV) are required, however. An implementation may define additional macros not listed in the table; the names of these macros must begin with FE_ and an uppercase letter.

Table 27.8 `<fenv.h>` Macros

Name	*Value*	*Description*

`FE_DIVBYZERO`
`FE_INEXACT`
`FE_INVALID`
`FE_OVERFLOW`
`FE_UNDERFLOW`
	Integer constant expressions whose bits do not overlap	Defined only if the corresponding floating-point exception is supported by the implementation. An implementation may define additional macros that represent floating-point exceptions.

`FE_ALL_EXCEPT`	See description	Bitwise *or* of all floating-point exception macros defined by the implementation. Has the value 0 if no such macros are defined.

`FE_DOWNWARD`
`FE_TONEAREST`
`FE_TOWARDZERO`
`FE_UPWARD`
	Integer constant expressions with distinct nonnegative values	Defined only if the corresponding rounding direction can be retrieved and set via the `fegetround` and `fesetround` functions. An implementation may define additional macros that represent rounding directions.

`FE_DFL_ENV`	A value of type `const fenv_t *`	Represents the default (program start-up) floating-point environment. An implementation may define additional macros that represent floating-point environments.

The `FENV_ACCESS` Pragma

#pragma directive ➤14.5

The `<fenv.h>` header provides a pragma named `FENV_ACCESS` that's used to notify the compiler of a program's intention to use the functions provided by this header. Knowing which portions of a program will use the capabilities of `<fenv.h>` is important for the compiler, because some common optimizations can't be performed if control modes don't have their customary settings or may change during program execution.

The `FENV_ACCESS` pragma has the following appearance:

```
#pragma STDC FENV_ACCESS on-off-switch
```

where *on-off-switch* is either `ON`, `OFF`, or `DEFAULT`. If the pragma is used with the value `ON`, it informs the compiler that the program might test floating-point status flags or alter a floating-point control mode. The value `OFF` indicates that flags won't be tested and default control modes are in effect. The meaning of `DEFAULT` is implementation-defined; it represents either `ON` or `OFF`.

The duration of the `FENV_ACCESS` pragma depends on where it's used in a program. When it appears at the top level of a source file, outside any external declarations, it remains in effect until the next `FENV_ACCESS` pragma or the end of the file. The only other place that an `FENV_ACCESS` pragma might appear is at the beginning of a compound statement (possibly the body of a function); in that case, the pragma remains in effect until the next `FENV_ACCESS` pragma (even one inside a nested compound statement) or the end of the compound statement. At the end of a compound statement, the state of the switch returns to its value before the compound statement was entered.

It's the programmer's responsibility to use the `FENV_ACCESS` pragma to indicate regions of a program in which low-level access to floating-point hardware

is needed. Undefined behavior occurs if a program tests floating-point status flags or runs under non-default control modes in a region for which the value of the pragma switch is OFF.

Typically, an FENV_ACCESS pragma that specifies the ON switch would be placed at the beginning of a function body:

```
void f(double x, double y)
{
  #pragma STDC FENV_ACCESS ON
  ...
}
```

The function f may test floating-point status flags or change control modes as needed. At the end of f's body, the pragma switch will return to its previous state.

When a program goes from an FENV_ACCESS "off" region to an "on" region during execution, the floating-point status flags have unspecified values and the control modes have their default settings.

Floating-Point Exception Functions

```
int feclearexcept(int excepts);
int fegetexceptflag(fexcept_t *flagp, int excepts);
int feraiseexcept(int excepts);
int fesetexceptflag(const fexcept_t *flagp,
                    int excepts);
int fetestexcept(int excepts);
```

The <fenv.h> functions are divided into three groups. Functions in the first group deal with the floating-point status flags. Each of the five functions has an int parameter named excepts, which is the bitwise *or* of one or more of the floating-point exception macros (the first group of macros listed in Table 27.8). For example, the argument passed to one of these functions might be FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW, to represent the combination of these three status flags. The argument may also be zero, to indicate that no flags are selected.

feclearexcept

The feclearexcept function attempts to clear the floating-point exceptions represented by excepts. It returns zero if excepts is zero or if all specified exceptions were successfully cleared; otherwise, it returns a nonzero value.

fegetexceptflag

The fegetexceptflag function attempts to retrieve the states of the floating-point status flags represented by excepts. This data is stored in the fexcept_t object pointed to by flagp. The fegetexceptflag function returns zero if the states of the status flags were successfully stored; otherwise, it returns a nonzero value.

feraiseexcept

The feraiseexcept function attempts to raise supported floating-point exceptions represented by excepts. It is implementation-defined whether feraiseexcept also raises the *inexact* floating-point exception whenever it

raises the *overflow* or *underflow* exception. (Implementations that conform to the IEEE standard will have this property.) `feraiseexcept` returns zero if `excepts` is zero or if all specified exceptions were successfully raised; otherwise, it returns a nonzero value.

fesetexceptflag

The `fesetexceptflag` function attempts to set the floating-point status flags represented by `excepts`. The states of the flags are stored in the `fexcept_t` object pointed to by `flagp`; this object must have been set by a previous call of `fegetexceptflag`. Moreover, the second argument in the prior call of `fegetexceptflag` must have included all floating-point exceptions represented by `excepts`. The `fesetexceptflag` function returns zero if `excepts` is zero or if all specified exceptions were successfully set; otherwise, it returns a nonzero value.

fetestexcept

The `fetestexcept` function tests only those floating-point status flags represented by `excepts`. It returns the bitwise *or* of the floating-point exception macros corresponding to the flags that are currently set. For example, if the value of `excepts` is `FE_INVALID | FE_OVERFLOW | FE_UNDERFLOW`, the `fetestexcept` function might return `FE_INVALID | FE_UNDERFLOW`, indicating that, of the exceptions represented by `FE_INVALID`, `FE_OVERFLOW`, and `FE_UNDERFLOW`, only the flags for `FE_INVALID` and `FE_UNDERFLOW` are currently set.

Rounding Functions

```
int fegetround(void);
int fesetround(int round);
```

The `fegetround` and `fesetround` functions are used to determine the rounding direction and modify it. Both functions rely on the rounding-direction macros (the third group in Table 27.8).

fegetround

The `fegetround` function returns the value of the rounding-direction macro that matches the current rounding direction. If the current rounding direction can't be determined or doesn't match any rounding-direction macro, `fegetround` returns a negative number.

fesetround

When passed the value of a rounding-direction macro, the `fesetround` function attempts to establish the corresponding rounding direction. If the call is successful, `fesetround` returns zero; otherwise, it returns a nonzero value.

Environment Functions

```
int fegetenv(fenv_t *envp);
int feholdexcept(fenv_t *envp);
int fesetenv(const fenv_t *envp);
int feupdateenv(const fenv_t *envp);
```

The last four functions in `<fenv.h>` deal with the entire floating-point environment, not just the status flags or control modes. Each function returns zero if it succeeds at the operation it was asked to perform. Otherwise, it returns a nonzero value.

fegetenv The `fegetenv` function attempts to retrieve the current floating-point environment from the processor and store it in the object pointed to by `envp`.

feholdexcept The `feholdexcept` function (1) stores the current floating-point environment in the object pointed to by `envp`, (2) clears the floating-point status flags, and (3) attempts to install a non-stop mode—if available—for all floating-point exceptions (so that future exceptions won't cause a trap or stop).

fesetenv The `fesetenv` function attempts to establish the floating-point environment represented by `envp`, which either points to a floating-point environment stored by a previous call of `fegetenv` or `feholdexcept`, or is equal to a floating-point environment macro such as `FE_DFL_ENV`. Unlike the `feupdateenv` function, `fesetenv` doesn't raise any exceptions. If a call of `fegetenv` is used to save the current floating-point environment, then a later call of `fesetenv` can restore the environment to its previous state.

feupdateenv The `feupdateenv` function attempts to (1) save the currently raised floating-point exceptions, (2) install the floating-point environment pointed to by `envp`, and (3) raise the saved exceptions. `envp` either points to a floating-point environment stored by a previous call of `fegetenv` or `feholdexcept`, or is equal to a floating-point environment macro such as `FE_DFL_ENV`.

Q & A

Q: If the `<inttypes.h>` header includes the `<stdint.h>` header, why do we need the `<stdint.h>` header at all? [p. 709]

A: The primary reason that `<stdint.h>` exists as a separate header is so that programs in a freestanding implementation may include it. (C99 requires conforming implementations—both hosted and freestanding—to provide the `<stdint.h>` header, but `<inttypes.h>` is required only for hosted implementations.) Even in a hosted environment, it may be advantageous to include `<stdint.h>` rather than `<inttypes.h>` to avoid defining all the macros that belong to the latter.

freestanding implementation ➤ 14.3

***Q: There are three versions of the `modf` function in `<math.h>`, so why isn't there a type-generic macro named `modf`? [p. 725]**

A: Let's take a look at the prototypes for the three versions of `modf`:

```
double modf(double value, double *iptr);
float modff(float value, float *iptr);
long double modfl(long double value, long double *iptr);
```

`modf` is unusual in that it has a pointer parameter, and the type of the pointer isn't the same among the three versions of the function. (`frexp` and `remquo` have a

pointer parameter, but it always has `int *` type.) Having a type-generic macro for `modf` would pose some difficult problems. For example, the meaning of `modf(d, &f)`, where `d` has type `double` and `f` has type `float`, is unclear: are we calling the `modf` function or the `modff` function? Rather than develop a complicated set of rules for a single function (and probably taking into account that `modf` isn't a very popular function), the C99 committee chose not to provide a type-generic `modf` macro.

Q: When a `<tgmath.h>` macro is invoked with an integer argument, the `double` version of the corresponding function is called. Shouldn't the `float` version be called, according to the usual arithmetic conversions? [p. 725]

usual arithmetic conversions ➤7.4

A: We're dealing with a macro, not a function, so the usual arithmetic conversions don't come into play. The C99 committee had to create a rule for determining which version of a function would be called when an integer argument is passed to a `<tgmath.h>` macro. Although the committee at one point considered having the `float` version called (for consistency with the usual arithmetic conversions), they eventually decided that choosing the `double` version was better. First, it's safer: converting an integer to `float` may cause a loss of accuracy, especially for integer types whose width is 32 bits or more. Second, it causes fewer surprises for the programmer. Suppose that `i` is an integer variable. If the `<tgmath.h>` header isn't included, the call `sin(i)` calls the `sin` function. On the other hand, if `<tgmath.h>` *is* included, the call `sin(i)` invokes the `sin` macro; because `i` is an integer, the preprocessor replaces the `sin` macro with the `sin` function, and the end result is the same.

Q: When a program invokes one of the type-generic macros in `<tgmath.h>`, how does the implementation determine which function to call? Is there a way for a macro to test the types of its arguments?

A: One unusual aspect of `<tgmath.h>` is that its macros need to be able to test the types of the arguments that are passed to them. C has no features for testing types, so it would normally be impossible to write such a macro. The `<tgmath.h>` macros rely on special facilities provided by a particular compiler to make such testing possible. We don't know what these facilities are, and they're not guaranteed to be portable from one compiler to another.

Exercises

Section 27.1

1. (C99) Locate the declarations of the `intN_t` and `uintN_t` types in the `<stdint.h>` header installed on your system. Which values of *N* are supported?

2. (C99) Write the parameterized macros `INT32_C(n)`, `UINT32_C(n)`, `INT64_C(n)`, and `UINT64_C(n)`, assuming that the `int` type and `long int` types are 32 bits wide and the `long long int` type is 64 bits wide. *Hint:* Use the ## preprocessor operator to attach

a suffix to `n` containing a combination of `L` and/or `U` characters. (See Section 7.1 for a discussion of how to use the `L` and `U` suffixes with integer constants.)

Section 27.2

3. (C99) In each of the following statements, assume that the variable `i` has the indicated original type. Using macros from the `<inttypes.h>` header, modify each statement so that it will work correctly if the type of `i` is changed to the indicated new type.

(a) `printf("%d", i);` Original type: `int` New type: `int8_t`
(b) `printf("%12.4d", i);` Original type: `int` New type: `int32_t`
(c) `printf("%-6o", i);` Original type: `unsigned int` New type: `uint16_t`
(d) `printf("%#x", i);` Original type: `unsigned int` New type: `uint64_t`

Section 27.5

4. (C99) Assume that the following variable declarations are in effect:

```
int i;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

Each of the following is an invocation of a macro in `<tgmath.h>`. Show what it will look like after preprocessing, when the macro has been replaced by a function from `<math.h>` or `<complex.h>`.

(a) `tan(i)`
(b) `fabs(f)`
(c) `asin(d)`
(d) `exp(ld)`
(e) `log(fc)`
(f) `acosh(dc)`
(g) `nexttoward(d, ld)`
(h) `remainder(f, i)`
(i) `copysign(d, ld)`
(j) `carg(i)`
(k) `cimag(f)`
(l) `conj(ldc)`

Programming Projects

1. (C99) Make the following modifications to the `quadratic.c` program of Section 27.4:

(a) Have the user enter the coefficients of the polynomial (the values of the variables `a`, `b`, and `c`).

(b) Have the program test the discriminant before displaying the values of the roots. If the discriminant is negative, have the program display the roots in the same way as before. If it's nonnegative, have the program display the roots as real numbers (without an imaginary part). For example, if the quadratic equation is $x^2 + x - 2 = 0$, the output of the program would be

```
root1 = 1
root2 = -2
```

(c) Modify the program so that it displays a complex number with a negative imaginary part as $a - bi$ instead of $a + -bi$. For example, the output of the program with the original coefficients would be

```
root1 = -0.2 + 0.4i
root2 = -0.2 - 0.4i
```

2. (C99) Write a program that converts a complex number in Cartesian coordinates to polar form. The user will enter a and b (the real and imaginary parts of the number); the program will display the values of r and θ.

3. (C99) Write a program that converts a complex number in polar coordinates to Cartesian form. After the user enters the values of r and θ, the program will display the number in the form $a + bi$, where

   $a = r \cos \theta$
   $b = r \sin \theta$

4. (C99) Write a program that displays the nth roots of unity when given a positive integer n. The nth roots of unity are given by the formula $e^{2\pi ik/n}$, where k is an integer between 0 and $n-1$.

APPENDIX A
C Operators

Precedence	*Name*	*Symbol(s)*	*Associativity*
1	Array subscripting	`[]`	Left
1	Function call	`()`	Left
1	Structure and union member	`.  ->`	Left
1	Increment (postfix)	`++`	Left
1	Decrement (postfix)	`--`	Left
2	Increment (prefix)	`++`	Right
2	Decrement (prefix)	`--`	Right
2	Address	`&`	Right
2	Indirection	`*`	Right
2	Unary plus	`+`	Right
2	Unary minus	`-`	Right
2	Bitwise complement	`~`	Right
2	Logical negation	`!`	Right
2	Size	`sizeof`	Right
3	Cast	`()`	Right
4	Multiplicative	`*  /  %`	Left
5	Additive	`+  -`	Left
6	Bitwise shift	`<<  >>`	Left
7	Relational	`<  >  <=  >=`	Left
8	Equality	`==  !=`	Left
9	Bitwise *and*	`&`	Left
10	Bitwise exclusive *or*	`^`	Left
11	Bitwise inclusive *or*	`|`	Left
12	Logical *and*	`&&`	Left
13	Logical *or*	`||`	Left
14	Conditional	`?:`	Right

15	Assignment	`=  *=  /=  %=`
		`+=  -=  <<=  >>=`
		`&=  ^=  |=`
			Right

16	Comma	`,`	Left

APPENDIX B
C99 versus C89

This appendix lists many of the most significant differences between C89 and C99. (The smaller differences are too numerous to mention here.) The headings indicate which chapter contains the primary discussion of each C99 feature. Some of the changes attributed to C99 actually occurred earlier, in Amendment 1 to the C89 standard; these changes are marked "Amendment 1."

2 C Fundamentals

// comments C99 adds a second kind of comment, which begins with `//`.

identifiers C89 requires compilers to remember the first 31 characters of identifiers; in C99, the requirement is 63 characters. Only the first six characters of names with external linkage are significant in C89. Moreover, the case of letters may not matter. In C99, the first 31 characters are significant, and the case of letters is taken into account.

keywords Five keywords are new in C99: `inline`, `restrict`, `_Bool`, `_Complex`, and `_Imaginary`.

returning from `main` In C89, if a program reaches the end of the `main` function without executing a `return` statement, the value returned to the operating system is undefined. In C99, if `main` is declared to return an `int`, the program returns 0 to the operating system.

4 Expressions

/ and % operators The C89 standard states that if either operand is negative, the result of an integer division can be rounded either up or down. Moreover, if `i` or `j` is negative, the sign of `i % j` depends on the implementation. In C99, the result of a division is always truncated toward zero and the value of `i % j` has the same sign as `i`.

5 Selection Statements

`_Bool` type C99 provides a Boolean type named `_Bool`; C89 has no Boolean type.

6 Loops

`for` statements In C99, the first expression in a `for` statement can be replaced by a declaration, allowing the statement to declare its own control variable(s).

7 Basic Types

`long long` integer types C99 provides two additional standard integer types, `long long int` and `unsigned long long int`.

extended integer types In addition to the standard integer types, C99 allows implementation-defined extended signed and unsigned integer types.

`long long` integer constants C99 provides a way to indicate that an integer constant has type `long long int` or `unsigned long long int`.

types of integer constants C99's rules for determining the type of an integer constant are different from those in C89.

hexadecimal floating constants C99 provides a way to write floating constants in hexadecimal.

implicit conversions The rules for implicit conversions in C99 are somewhat different from the rules in C89, primarily because of C99's additional basic types.

8 Arrays

designated initializers C99 supports designated initializers, which can be used to initialize arrays, structures, and unions.

variable-length arrays In C99, the length of an array may be specified by an expression that's not constant, provided that the array doesn't have static storage duration and its declaration doesn't contain an initializer.

9 Functions

no default return type If the return type of a function is omitted in C89, the function is presumed to return a value of type `int`. In C99, it's illegal to omit the return type of a function.

mixed declarations and statements In C89, declarations must precede statements within a block (including the body of a function). In C99, declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

declaration or definition required prior to function call
C99 requires that either a declaration or a definition of a function be present prior to any call of the function. C89 doesn't have this requirement; if a function is called without a prior declaration or definition, the compiler assumes that the function returns an `int` value.

variable-length array parameters
C99 allows variable-length array parameters. In a function declaration, the * symbol may appear inside brackets to indicate a variable-length array parameter.

`static` *array parameters*
C99 allows the use of the word `static` in the declaration of an array parameter, indicating a minimum length for the first dimension of the array.

compound literals
C99 supports the use of compound literals, which allow the creation of unnamed array and structure values.

declaration of `main`
C99 allows `main` to be declared in an implementation-defined manner, with a return type other than `int` and/or parameters other than those specified by the standard.

`return` *statement without expression*
In C89, executing a `return` statement without an expression in a non-`void` function causes undefined behavior (but only if the program attempts to use the value returned by the function). In C99, such a statement is illegal.

14 The Preprocessor

additional predefined macros
C99 provides several new predefined macros.

empty macro arguments
C99 allows any or all of the arguments in a macro call to be empty, provided that the call contains the correct number of commas.

macros with a variable number of arguments
In C89, a macro must have a fixed number of arguments, if it has any at all. C99 allows macros that take an unlimited number of arguments.

`__func__` *identifier*
In C99, the `__func__` identifier behaves like a string variable that stores the name of the currently executing function.

standard pragmas
In C89, there are no standard pragmas. C99 has three: `CX_LIMITED_RANGE`, `FENV_ACCESS`, and `FP_CONTRACT`.

`_Pragma` *operator*
C99 provides the `_Pragma` operator, which is used in conjunction with the `#pragma` directive.

16 Structures, Unions, and Enumerations

structure type compatibility
In C89, structures defined in different files are compatible if their members have the same names and appear in the same order, with corresponding members having

compatible types. C99 also requires that either both structures have the same tag or neither has a tag.

trailing comma in enumerations

In C99, the last constant in an enumeration may be followed by a comma.

17 Advanced Uses of Pointers

restricted pointers

C99 has a new keyword, `restrict`, that can appear in the declaration of a pointer.

flexible array members

C99 allows the last member of a structure to be an array of unspecified length.

18 Declarations

block scopes for selection and iteration statements

In C99, selection statements (`if` and `switch`) and iteration statements (`while`, `do`, and `for`)—along with the "inner" statements that they control—are considered to be blocks.

array, structure, and union initializers

In C89, a brace-enclosed initializer for an array, structure, or union must contain only constant expressions. In C99, this restriction applies only if the variable has static storage duration.

inline functions

C99 allows functions to be declared `inline`.

21 The Standard Library

`<stdbool.h>` *header*

The `<stdbool.h>` header, which defines the `bool`, `true`, and `false` macros, is new in C99.

22 Input/Output

`...printf` *conversion specifications*

The conversion specifications for the `...printf` functions have undergone a number of changes in C99, with new length modifiers, new conversion specifiers, the ability to write infinity and NaN, and support for wide characters. Also, the `%le`, `%lE`, `%lf`, `%lg`, and `%lG` conversions are legal in C99; they caused undefined behavior in C89.

`...scanf` *conversion specifications*

In C99, the conversion specifications for the `...scanf` functions have new length modifiers, new conversion specifiers, the ability to read infinity and NaN, and support for wide characters.

`snprintf` *function*

C99 adds the `snprintf` function to the `<stdio.h>` header.

23 Library Support for Numbers and Character Data

additional macros in `<float.h>` *header*

C99 adds the `DECIMAL_DIG` and `FLT_EVAL_METHOD` macros to the `<float.h>` header.

additional macros in `<limits.h>` *header*

In C99, the `<limits.h>` header contains three new macros that describe the characteristics of the `long long int` types.

`math_errhandling` *macro*

C99 gives implementations a choice of how to inform a program that an error has occurred in a mathematical function: via a value stored in `errno`, via a floating-point exception, or both. The value of the `math_errhandling` macro (defined in `<math.h>`) indicates how errors are signaled by a particular implementation.

additional functions in `<math.h>` *header*

C99 adds two new versions of most `<math.h>` functions, one for `float` and one for `long double`. C99 also adds a number of completely new functions and function-like macros to `<math.h>`.

24 Error Handling

`EILSEQ` *macro*

C99 adds the `EILSEQ` macro to the `<errno.h>` header.

25 International Features

digraphs

Digraphs, which are two-character symbols that can be used as substitutes for the `[`, `]`, `{`, `}`, `#`, and `##` tokens, are new in C99. (Amendment 1)

`<iso646.h>` *header*

The `<iso646.h>` header, which defines macros that represent operators containing the characters `&`, `|`, `~`, `!`, and `^`, is new in C99. (Amendment 1)

universal character names

Universal character names, which provide a way to embed UCS characters in the source code of a program, are new in C99.

`<wchar.h>` *header*

The `<wchar.h>` header, which provides functions for wide-character input/output and wide string manipulation, is new in C99. (Amendment 1)

`<wctype.h>` *header*

The `<wctype.h>` header, the wide-character version of `<ctype.h>`, is new in C99. `<wctype.h>` provides functions for classifying and changing the case of wide characters. (Amendment 1)

26 Miscellaneous Library Functions

`va_copy` *macro*

C99 adds a function-like macro named `va_copy` to the `<stdarg.h>` header.

additional functions in `<stdio.h>` *header*

C99 adds the `vsnprintf`, `vfscanf`, `vscanf`, and `vsscanf` functions to the `<stdio.h>` header.

additional functions in `<stdlib.h>` *header*

C99 adds five numeric conversion functions, the `_Exit` function, and `long long` versions of the `abs` and `div` functions to the `<stdlib.h>` header.

additional `strftime` *conversion specifiers*

C99 adds a number of new `strftime` conversion specifiers. It also allows the use of an `E` or `O` character to modify the meaning of certain conversion specifiers.

27 Additional C99 Support for Mathematics

<stdint.h> header The <stdint.h> header, which declares integer types with specified widths, is new in C99.

<inttypes.h> header The <inttypes.h> header, which provides macros that are useful for input/output of the integer types in <stdint.h>, is new in C99.

complex types C99 provides three complex types: `float _Complex`, `double _Complex`, and `long double _Complex`.

<complex.h> header The <complex.h> header, which provides functions that perform mathematical operations on complex numbers, is new in C99.

<tgmath.h> header The <tgmath.h> header, which provides type-generic macros that make it easier to call library functions in <math.h> and <complex.h>, is new in C99.

<fenv.h> header The <fenv.h> header, which gives programs access to floating-point status flags and control modes, is new in C99.

APPENDIX C
C89 versus K&R C

This appendix lists most of the significant differences between C89 and K&R C (the language described in the first edition of Kernighan and Ritchie's *The C Programming Language*). The headings indicate which chapter of this book discusses each C89 feature. This appendix doesn't address the C library, which has changed much over the years. For other (less important) differences between C89 and K&R C, consult Appendices A and C in the second edition of K&R.

Most of today's C compilers can handle all of C89, but this appendix is useful if you to happen to encounter older programs that were originally written for pre-C89 compilers.

2 C Fundamentals

identifiers In K&R C, only the first eight characters of an identifier are significant.

keywords K&R C lacks the keywords `const`, `enum`, `signed`, `void`, and `volatile`. In K&R C, the word `entry` is a keyword.

4 Expressions

unary + K&R C doesn't support the unary + operator.

5 Selection Statements

`switch` In K&R C, the controlling expression (and case labels) in a `switch` statement must have type `int` after promotion. In C89, the expression and labels may be of any integral type, including `unsigned int` and `long int`.

7 Basic Types

unsigned types K&R C provides only one unsigned type (`unsigned int`).

`signed` K&R C doesn't support the `signed` type specifier.

number suffixes K&R C doesn't support the `U` (or `u`) suffix to specify that an integer constant is unsigned, nor does it support the `F` (or `f`) suffix to indicate that a floating constant is to be stored as a `float` value instead of a `double` value. In K&R C, the `L` (or `l`) suffix can't be used with floating constants.

`long float` K&R C allows the use of `long float` as a synonym for `double`; this usage isn't legal in C89.

`long double` K&R C doesn't support the `long double` type.

escape sequences The escape sequences `\a`, `\v`, and `\?` don't exist in K&R C. Also, K&R C doesn't support hexadecimal escape sequences.

`size_t` In K&R C, the `sizeof` operator returns a value of type `int`; in C89, it returns a value of type `size_t`.

usual arithmetic conversions K&R C requires that `float` operands be converted to `double`. Also, K&R C specifies that combining a shorter unsigned integer with a longer signed integer always produces an unsigned result.

9 Functions

function definitions In a C89 function definition, the types of the parameters are included in the parameter list:

```
double square(double x)
{
  return x * x;
}
```

K&R C requires that the types of parameters be specified in separate lists:

```
double square(x)
double x;
{
  return x * x;
}
```

function declarations A C89 function declaration (prototype) specifies the types of the function's parameters (and the names as well, if desired):

```
double square(double x);
double square(double);      /* alternate form */
int rand(void);             /* no parameters  */
```

A K&R C function declaration omits all information about parameters:

```
double square();
int rand();
```

function calls When a K&R C definition or declaration is used, the compiler doesn't check that the function is called with arguments of the proper number and type. Furthermore, the arguments aren't automatically converted to the types of the corresponding parameters. Instead, the integral promotions are performed, and `float` arguments are converted to `double`.

`void` K&R C doesn't support the `void` type.

12 Pointers and Arrays

pointer subtraction Subtracting two pointers produces an `int` value in K&R C but a `ptrdiff_t` value in C89.

13 Strings

string literals In K&R C, adjacent string literals aren't concatenated. Also, K&R C doesn't prohibit the modification of string literals.

string initialization In K&R C, an initializer for a character array of length n is limited to $n - 1$ characters (leaving room for a null character at the end). C89 allows the initializer to have length n.

14 The Preprocessor

`#elif`, `#error`, `#pragma` K&R C doesn't support the `#elif`, `#error`, and `#pragma` directives.

`#`, `##`, `defined` K&R C doesn't support the `#`, `##`, and `defined` operators.

16 Structures, Unions, and Enumerations

structure and union members and tags In C89, each structure and union has its own name space for members; structure and union tags are kept in a separate name space. K&R C uses a single name space for members and tags, so members can't have the same name (with some exceptions), and members and tags can't overlap.

whole-structure operations K&R C doesn't allow structures to be assigned, passed as arguments, or returned by functions.

enumerations K&R C doesn't support enumerations.

17 Advanced Uses of Pointers

`void *` In C89, `void *` is used as a "generic" pointer type; for example, `malloc` returns a value of type `void *`. In K&R C, `char *` is used for this purpose.

pointer mixing K&R C allows pointers of different types to be mixed in assignments and comparisons. In C89, pointers of type `void *` can be mixed with pointers of other types, but any other mixing isn't allowed without casting. Similarly, K&R C allows the mixing of integers and pointers in assignments and comparisons; C89 requires casting.

pointers to functions If `pf` is a pointer to a function, C89 permits using either `(*pf)(...)` or `pf(...)` to call the function. K&R C allows only `(*pf)(...)`.

18 Declarations

`const` *and* `volatile` K&R C doesn't support the `const` and `volatile` type qualifiers.

initialization of arrays, structures, and unions K&R C doesn't allow the initialization of automatic arrays and structures, nor does it allow initialization of unions (regardless of storage duration).

25 International Features

wide characters K&R C doesn't support wide character constants and wide string literals.

trigraph sequences K&R C doesn't support trigraph sequences.

26 Miscellaneous Library Functions

variable arguments K&R C doesn't provide a portable way to write functions with a variable number of arguments, and it lacks the . . . (ellipsis) notation.

APPENDIX D
Standard Library Functions

This appendix describes all library functions supported by C89 and C99.* When using this appendix, please keep the following points in mind:

- In the interest of brevity and clarity, I've omitted many details. Some functions (notably `printf` and `scanf` and their variants) are covered in depth elsewhere in the book, so their descriptions here are minimal. For more information about a function (including examples of how it's used), see the section(s) listed in italic at the lower right corner of the function's description.
- As in other parts of the book, italics are used to indicate C99 differences. The names and prototypes of functions that were added in C99 are shown in italics. Changes to C89 prototypes (the addition of the word `restrict` to the declaration of certain parameters) are also italicized.
- Function-like macros are included in this appendix (with the exception of the type-generic macros in `<tgmath.h>`). Each prototype for a macro is followed by the word *macro*.
- In C99, some `<math.h>` functions have three versions (one each for `float`, `double`, and `long double`). All three are grouped into a single entry, under the name of the `double` version. For example, there's only one entry (under `acos`) for the `acos`, `acosf`, and `acosl` functions. The name of each additional version (`acosf` and `acosl`, in this example) appears to the left of its prototype. The `<complex.h>` functions, which also come in three versions, are treated in a similar fashion.
- Most of the `<wchar.h>` functions are wide-character versions of functions found in other headers. Unless there's a significant difference in behavior, the

*This material is adapted from international standard ISO/IEC 9899:1999.

description of each wide-character function simply refers the reader to the corresponding function found elsewhere.

- If some aspect of a function's behavior is described as *implementation-defined*, that means that it depends on how the C library is implemented. The function will always behave consistently, but the results may vary from one system to another. (In other words, check the manual to see what happens.) *Undefined* behavior, on the other hand, is bad news: not only may the behavior vary between systems, but the program may act strangely or even crash.
- The descriptions of many `<math.h>` functions refer to the terms *domain error* and *range error*. The way in which these errors are indicated changed between C89 and C99. For the C89 treatment of these errors, see Section 23.3. For the C99 treatment, see Section 23.4.
- The behavior of the following functions is affected by the current locale:

`<ctype.h>`	All functions
`<stdio.h>`	Formatted input/output functions
`<stdlib.h>`	Multibyte/wide-character conversion functions, numeric conversion functions
`<string.h>`	`strcoll`, `strxfrm`
`<time.h>`	`strftime`
`<wchar.h>`	`wcscoll`, `wcsftime`, `wcsxfrm`, formatted input/output functions, numeric conversion functions, extended multibyte/wide-character conversion functions
`<wctype.h>`	All functions

The `isalpha` function, for example, usually checks whether a character lies between `a` and `z` or `A` and `Z`. In some locales, other characters are considered alphabetic as well.

---

abort *Abort Program* `<stdlib.h>`

```
void abort(void);
```

Raises the `SIGABRT` signal. If the signal isn't caught (or if the signal handler returns), the program terminates abnormally and returns an implementation-defined code indicating unsuccessful termination. Whether output buffers are flushed, open streams are closed, or temporary files are removed is implementation-defined.

26.2

---

abs *Integer Absolute Value* `<stdlib.h>`

```
int abs(int j);
```

Returns Absolute value of `j`. The behavior is undefined if the absolute value of `j` can't be represented. *26.2*

---

acos *Arc Cosine* `<math.h>`

```
double acos(double x);
```

acosf *`float acosf(float x);`*

acosl *`long double acosl(long double x);`*

Returns Arc cosine of x; the return value is in the range 0 to π. A domain error occurs if x isn't between –1 and +1. *23.3*

---

acosh *Arc Hyperbolic Cosine (C99)* <math.h>

```
double acosh(double x);
float acoshf(float x);
long double acoshl(long double x);
```

Returns Arc hyperbolic cosine of x; the return value is in the range 0 to +∞. A domain error occurs if x is less than 1. *23.4*

---

asctime *Convert Broken-Down Time to String* <time.h>

```
char *asctime(const struct tm *timeptr);
```

Returns A pointer to a null-terminated string of the form

```
Sun Jun  3 17:48:34 2007\n
```

constructed from the broken-down time in the structure pointed to by `timeptr`. *26.3*

---

asin *Arc Sine* <math.h>

```
double asin(double x);
float asinf(float x);
long double asinl(long double x);
```

Returns Arc sine of x; the return value is in the range –π/2 to +π/2. A domain error occurs if x isn't between –1 and +1. *23.3*

---

asinh *Arc Hyperbolic Sine (C99)* <math.h>

```
double asinh(double x);
float asinhf(float x);
long double asinhl(long double x);
```

Returns Arc hyperbolic sine of x. *23.4*

---

assert *Assert Truth of Expression* <assert.h>

```
void assert(scalar expression);
```
macro

If the value of `expression` is nonzero, `assert` does nothing. If the value is zero, `assert` writes a message to `stderr` (specifying the text of `expression`, the name of the source file containing the `assert`, and the line number of the `assert`); it then terminates the program by calling `abort`. To disable `assert`, define the macro `NDEBUG` before including <assert.h>. *C99 changes:* The argument is allowed to have any scalar type; C89 specifies that the type is `int`. Also, C99 requires that the message written by `assert` include the name of the function in which the `assert` appears; C89 doesn't have this requirement. *24.1*

---

atan *Arc Tangent* <math.h>

```
double atan(double x);
float atanf(float x);
```

atanl `long double atanl(long double x);`

Returns Arc tangent of x; the return value is in the range $-\pi/2$ to $+\pi/2$. *23.3*

---

atan2 — *Arc Tangent of Quotient* — <math.h>

```
double atan2(double y, double x);
float atan2f(float y, float x);
long double atan2l(long double y, long double x);
```

Returns Arc tangent of y/x; the return value is in the range $-\pi$ to $+\pi$. A domain error may occur if x and y are both zero. *23.3*

---

atanh — *Arc Hyperbolic Tangent (C99)* — <math.h>

```
double atanh(double x);
float atanhf(float x);
long double atanhl(long double x);
```

Returns Arc hyperbolic tangent of x. A domain error occurs if x is not between –1 and +1. A range error may occur if x is equal to –1 or +1. *23.4*

---

atexit — *Register Function to Be Called at Program Exit* — <stdlib.h>

```
int atexit(void (*func)(void));
```

Registers the function pointed to by `func` as a termination function. The function will be called if the program terminates normally (via `return` or `exit` but not `abort`).

Returns Zero if successful, nonzero if unsuccessful (an implementation-dependent limit has been reached). *26.2*

---

atof — *Convert String to Floating-Point Number* — <stdlib.h>

```
double atof(const char *nptr);
```

Returns A `double` value corresponding to the longest initial part of the string pointed to by `nptr` that has the form of a floating-point number. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented. *26.2*

---

atoi — *Convert String to Integer* — <stdlib.h>

```
int atoi(const char *nptr);
```

Returns An `int` value corresponding to the longest initial part of the string pointed to by `nptr` that has the form of an integer. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented. *26.2*

---

atol — *Convert String to Long Integer* — <stdlib.h>

```
long int atol(const char *nptr);
```

Returns A `long int` value corresponding to the longest initial part of the string pointed to by `nptr` that has the form of an integer. Returns zero if no conversion

could be performed. The function's behavior is undefined if the number can't be represented. *26.2*

---

atoll *Convert String to Long Long Integer (C99)* `<stdlib.h>`

```
long long int atoll(const char *nptr);
```

Returns A `long long int` value corresponding to the longest initial part of the string pointed to by `nptr` that has the form of an integer. Returns zero if no conversion could be performed. The function's behavior is undefined if the number can't be represented. *26.2*

---

bsearch *Binary Search* `<stdlib.h>`

```
void *bsearch(const void *key, const void *base,
              size_t memb, size_t size,
              int (*compar)(const void *,
                            const void *));
```

Searches for the value pointed to by `key` in the sorted array pointed to by `base`. The array has `nmemb` elements, each `size` bytes long. `compar` is a pointer to a comparison function. When passed pointers to the key and an array element, in that order, the comparison function must return a negative, zero, or positive integer, depending on whether the key is less than, equal to, or greater than the array element.

Returns A pointer to an array element that tests equal to the key. Returns a null pointer if the key isn't found. *26.2*

---

btowc *Convert Byte to Wide Character (C99)* `<wchar.h>`

```
wint_t btowc(int c);
```

Returns Wide-character representation of `c`. Returns `WEOF` if `c` is equal to `EOF` or if `c` (when cast to `unsigned char`) isn't a valid single-byte character in the initial shift state. *25.5*

---

cabs *Complex Absolute Value (C99)* `<complex.h>`

```
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

Returns Complex absolute value of `z`. *27.4*

---

cacos *Complex Arc Cosine (C99)* `<complex.h>`

```
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

Returns Complex arc cosine of `z`, with branch cuts outside the interval [–1, +1] along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval [0, π] along the real axis. *27.4*

cacosh *Complex Arc Hyperbolic Cosine (C99)* <complex.h>

```
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

Returns Complex arc hyperbolic cosine of z, with a branch cut at values less than 1 along the real axis. The return value lies in a half-strip of nonnegative values along the real axis and in the interval [$-i\pi$, $+i\pi$] along the imaginary axis. *27.4*

---

calloc *Allocate and Clear Memory Block* <stdlib.h>

```
void *calloc(size_t nmemb, size_t size);
```

Allocates a block of memory for an array with nmemb elements, each with `size` bytes. The block is cleared by setting all bits to zero.

Returns A pointer to the beginning of the block. Returns a null pointer if a block of the requested size can't be allocated. *17.3*

---

carg *Complex Argument (C99)* <complex.h>

```
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

Returns Argument (phase angle) of z, with a branch cut along the negative real axis. The return value lies in the interval [$-\pi$, $+\pi$]. *27.4*

---

casin *Complex Arc Sine (C99)* <complex.h>

```
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

Returns Complex arc sine of z, with branch cuts outside the interval [–1, +1] along the real axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval [$-\pi/2$, $+\pi/2$] along the real axis. *27.4*

---

casinh *Complex Arc Hyperbolic Sine (C99)* <complex.h>

```
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

Returns Complex arc hyperbolic sine of z, with branch cuts outside the interval [$-i$, $+i$] along the imaginary axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval [$-i\pi/2$, $+i\pi/2$] along the imaginary axis. *27.4*

---

catan *Complex Arc Tangent (C99)* <complex.h>

```
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

Returns Complex arc tangent of z, with branch cuts outside the interval $[-i, +i]$ along the imaginary axis. The return value lies in a strip mathematically unbounded along the imaginary axis and in the interval $[-\pi/2, +\pi/2]$ along the real axis. *27.4*

---

catanh — *Complex Arc Hyperbolic Tangent (C99)* — <complex.h>

```
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

Returns Complex arc hyperbolic tangent of z, with branch cuts outside the interval $[-1, +1]$ along the real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi/2, +i\pi/2]$ along the imaginary axis. *27.4*

---

cbrt — *Cube Root (C99)* — <math.h>

```
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);
```

Returns Real cube root of x. *23.4*

---

ccos — *Complex Cosine (C99)* — <complex.h>

```
double complex ccos(double complex z);
float complex ccosf(float complex z);
long double complex ccosl(long double complex z);
```

Returns Complex cosine of z. *27.4*

---

ccosh — *Complex Hyperbolic Cosine (C99)* — <complex.h>

```
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

Returns Complex hyperbolic cosine of z. *27.4*

---

ceil — *Ceiling* — <math.h>

```
double ceil(double x);
float ceilf(float x);
long double ceill(long double x);
```

Returns Smallest integer that is greater than or equal to x. *23.3*

---

cexp — *Complex Base-e Exponential (C99)* — <complex.h>

```
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

Returns Complex base-*e* exponential of z. *27.4*

---

cimag — *Imaginary Part of Complex Number (C99)* — <complex.h>

```
double cimag(double complex z);
```

cimagf `float cimagf(float complex z);`
cimagl `long double cimagl(long double complex z);`

Returns Imaginary part of z. *27.4*

---

clearerr *Clear Stream Error* <stdio.h>

```
void clearerr(FILE *stream);
```

Clears the end-of-file and error indicators for the stream pointed to by `stream`. *22.3*

---

clock *Processor Clock* <time.h>

```
clock_t clock(void);
```

Returns Elapsed processor time (measured in "clock ticks") since the beginning of program execution. (To convert into seconds, divide by `CLOCKS_PER_SEC`.) Returns `(clock_t)(-1)` if the time is unavailable or can't be represented. *26.3*

---

clog *Complex Natural Logarithm (C99)* <complex.h>

```
double complex clog(double complex z);
```
clogf `float complex clogf(float complex z);`
clogl `long double complex clogl(long double complex z);`

Returns Complex natural (base-*e*) logarithm of z, with a branch cut along the negative real axis. The return value lies in a strip mathematically unbounded along the real axis and in the interval $[-i\pi, +i\pi]$ along the imaginary axis. *27.4*

---

conj *Complex Conjugate (C99)* <complex.h>

```
double complex conj(double complex z);
```
conjf `float complex conjf(float complex z);`
conjl `long double complex conjl(long double complex z);`

Returns Complex conjugate of z. *27.4*

---

copysign *Copy Sign (C99)* <math.h>

```
double copysign(double x, double y);
```
copysignf `float copysignf(float x, float y);`
copysignl `long double copysignl(long double x, long double y);`

Returns A value with the magnitude of x and the sign of y. *23.4*

---

cos *Cosine* <math.h>

```
double cos(double x);
```
cosf `float cosf(float x);`
cosl `long double cosl(long double x);`

Returns Cosine of x (measured in radians). *23.3*

---

cosh *Hyperbolic Cosine* <math.h>

```
double cosh(double x);
```
coshf `float coshf(float x);`

coshl `long double coshl(long double x);`

Returns Hyperbolic cosine of x. A range error occurs if the magnitude of x is too large. *23.3*

---

cpow Complex Power (C99) <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);
```

Returns x raised to the power y, with a branch cut for the first parameter along the negative real axis. *27.4*

---

cproj Complex Projection (C99) <complex.h>

```
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

Returns Projection of z onto the Riemann sphere. z is returned unless one of its parts is infinite, in which case the return value is `INFINITY + I * copysign(0.0, cimag(z))`. *27.4*

---

creal Real Part of Complex Number (C99) <complex.h>

```
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

Returns Real part of z. *27.4*

---

csin Complex Sine (C99) <complex.h>

```
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

Returns Complex sine of z. *27.4*

---

csinh Complex Hyperbolic Sine (C99) <complex.h>

```
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

Returns Complex hyperbolic sine of z. *27.4*

---

csqrt Complex Square Root (C99) <complex.h>

```
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

Returns Complex square root of z, with a branch cut along the negative real axis. The return value lies in the right half-plane (including the imaginary axis). *27.4*

---

ctan *Complex Tangent (C99)* <complex.h>

```
double complex ctan(double complex z);
ctanf  float complex ctanf(float complex z);
ctanl  long double complex ctanl(long double complex z);
```

Returns Complex tangent of z. *27.4*

---

ctanh *Complex Hyperbolic Tangent (C99)* <complex.h>

```
double complex ctanh(double complex z);
ctanhf  float complex ctanhf(float complex z);
ctanhl  long double complex ctanhl(long double complex z);
```

Returns Complex hyperbolic tangent of z. *27.4*

---

ctime *Convert Calendar Time to String* <time.h>

```
char *ctime(const time_t *timer);
```

Returns A pointer to a string describing a local time equivalent to the calendar time pointed to by `timer`. Equivalent to `asctime(localtime(timer))`. *26.3*

---

difftime *Time Difference* <time.h>

```
double difftime(time_t time1, time_t time0);
```

Returns Difference between `time0` (the earlier time) and `time1`, measured in seconds. *26.3*

---

div *Integer Division* <stdlib.h>

```
div_t div(int numer, int denom);
```

Returns A `div_t` structure containing members named `quot` (the quotient when `numer` is divided by `denom`) and `rem` (the remainder). The behavior is undefined if either part of the result can't be represented. *26.2*

---

erf *Error Function (C99)* <math.h>

```
double erf(double x);
erff  float erff(float x);
erfl  long double erfl(long double x);
```

Returns erf(x), where erf is the Gaussian error function. *23.4*

---

erfc *Complementary Error Function (C99)* <math.h>

```
double erfc(double x);
erfcf  float erfcf(float x);
erfcl  long double erfcl(long double x);
```

Returns erfc(x) = 1 – erf(x), where erf is the Gaussian error function. A range error occurs if x is too large. *23.4*

exit *Exit from Program* `<stdlib.h>`

```
void exit(int status);
```

Calls all functions registered with `atexit`, flushes all output buffers, closes all open streams, removes any files created by `tmpfile`, and terminates the program. The value of `status` indicates whether the program terminated normally. The only portable values for `status` are `0` and `EXIT_SUCCESS` (both indicate successful termination) plus `EXIT_FAILURE` (unsuccessful termination).

9.5, 26.2

_Exit *Exit from Program (C99)* `<stdlib.h>`

```
void _Exit(int status);
```

Causes normal program termination. Doesn't call functions registered with `atexit` or signal handlers registered with `signal`. The status returned is determined in the same way as for `exit`. Whether output buffers are flushed, open streams are closed, or temporary files are removed is implementation-defined.

26.2

---

exp *Base-e Exponential* `<math.h>`

```
double exp(double x);
float expf(float x);
long double expl(long double x);
```

Returns e raised to the power x. A range error occurs if the magnitude of x is too large.

23.3

---

exp2 *Base-2 Exponential (C99)* `<math.h>`

```
double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);
```

Returns 2 raised to the power x. A range error occurs if the magnitude of x is too large.

23.4

---

expm1 *Base-e Exponential Minus 1 (C99)* `<math.h>`

```
double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);
```

Returns e raised to the power x, minus 1. A range error occurs if x is too large. *23.4*

---

fabs *Floating Absolute Value* `<math.h>`

```
double fabs(double x);
float fabsf(float x);
long double fabsl(long double x);
```

Returns Absolute value of x. *23.3*

fclose *Close File* `<stdio.h>`

```
int fclose(FILE *stream);
```

Closes the stream pointed to by `stream`. Flushes any unwritten output remaining in the stream's buffer. Deallocates the buffer if it was allocated automatically.

Returns Zero if successful, `EOF` if an error was detected. *22.2*

---

fdim *Positive Difference (C99)* `<math.h>`

```
double fdim(double x, double y);
fdimf  float fdimf(float x, float y);
fdiml  long double fdiml(long double x, long double y);
```

Returns Positive difference of `x` and `y`:

$$\begin{cases} x - y & \text{if } x > y \\ +0 & \text{if } x \le y \end{cases}$$

A range error may occur. *23.4*

---

feclearexcept *Clear Floating-Point Exceptions (C99)* `<fenv.h>`

```
int feclearexcept(int excepts);
```

Attempts to clear the floating-point exceptions represented by `excepts`.

Returns Zero if `excepts` is zero or if all specified exceptions were successfully cleared; otherwise, returns a nonzero value. *27.6*

---

fegetenv *Get Floating-Point Environment (C99)* `<fenv.h>`

```
int fegetenv(fenv_t *envp);
```

Attempts to store the current floating-point environment in the object pointed to by `envp`.

Returns Zero if the environment was successfully stored; otherwise, returns a nonzero value. *27.6*

---

fegetexceptflag *Get Floating-Point Exception Flags (C99)* `<fenv.h>`

```
int fegetexceptflag(fexcept_t *flagp, int excepts);
```

Attempts to retrieve the states of the floating-point status flags represented by `excepts` and store them in the object pointed to by `flagp`.

Returns Zero if the states of the status flags were successfully stored; otherwise, returns a nonzero value. *27.6*

---

fegetround *Get Floating-Point Rounding Direction (C99)* `<fenv.h>`

```
int fegetround(void);
```

Returns Value of the rounding-direction macro that represents the current rounding direction. Returns a negative value if the current rounding direction can't be determined or doesn't match any rounding-direction macro. *27.6*

feholdexcept *Save Floating-Point Environment (C99)* `<fenv.h>`

```
int feholdexcept(fenv_t *envp);
```

Saves the current floating-point environment in the object pointed to by `envp`, clears the floating-point status flags, and attempts to install a non-stop mode for all floating-point exceptions.

Returns Zero if non-stop floating-point exception handling was successfully installed; otherwise, returns a nonzero value. *27.6*

---

feof *Test for End-of-File* `<stdio.h>`

```
int feof(FILE *stream);
```

Returns A nonzero value if the end-of-file indicator is set for the stream pointed to by `stream`; otherwise, returns zero. *22.3*

---

feraiseexcept *Raise Floating-Point Exceptions (C99)* `<fenv.h>`

```
int feraiseexcept(int excepts);
```

Attempts to raise supported floating-point exceptions represented by `excepts`.

Returns Zero if `excepts` is zero or if all specified exceptions were successfully raised; otherwise, returns a nonzero value. *27.6*

---

ferror *Test for File Error* `<stdio.h>`

```
int ferror(FILE *stream);
```

Returns A nonzero value if the error indicator is set for the stream pointed to by `stream`; otherwise, returns zero. *22.3*

---

fesetenv *Set Floating-Point Environment (C99)* `<fenv.h>`

```
int fesetenv(const fenv_t *envp);
```

Attempts to establish the floating-point environment represented by the object pointed to by `envp`.

Returns Zero if the environment was successfully established; otherwise, returns a nonzero value. *27.6*

---

fesetexceptflag *Set Floating-Point Exception Flags (C99)* `<fenv.h>`

```
int fesetexceptflag(const fexcept_t *flagp,
                    int excepts);
```

Attempts to set the floating-point status flags represented by `excepts` to the states stored in the object pointed to by `flagp`.

Returns Zero if `excepts` is zero or if all specified exceptions were successfully set; otherwise, returns a nonzero value. *27.6*

---

fesetround *Set Floating-Point Rounding Direction (C99)* `<fenv.h>`

```
int fesetround(int round);
```

Attempts to establish the rounding direction represented by `round`.

Returns Zero if the requested rounding direction was established; otherwise, returns a nonzero value. *27.6*

`fetestexcept` *Test Floating-Point Exception Flags (C99)* `<fenv.h>`

```
int fetestexcept(int excepts);
```

Returns Bitwise *or* of the floating-point exception macros corresponding to the currently set flags for the exceptions represented by `excepts`. *27.6*

`feupdateenv` *Update Floating-Point Environment (C99)* `<fenv.h>`

```
int feupdateenv(const fenv_t *envp);
```

Attempts to save the currently raised floating-point exceptions, install the floating-point environment represented by the object pointed to by `envp`, and then raise the saved exceptions.

Returns Zero if all actions were successfully carried out; otherwise, returns a nonzero value. *27.6*

`fflush` *Flush File Buffer* `<stdio.h>`

```
int fflush(FILE *stream);
```

Writes any unwritten data in the buffer associated with `stream`, which points to a stream that was opened for output or updating. If `stream` is a null pointer, `fflush` flushes all streams that have unwritten data stored in a buffer.

Returns Zero if successful, `EOF` if a write error occurs. *22.2*

`fgetc` *Read Character from File* `<stdio.h>`

```
int fgetc(FILE *stream);
```

Reads a character from the stream pointed to by `stream`.

Returns Character read from the stream. If `fgetc` encounters the end of the stream, it sets the stream's end-of-file indicator and returns `EOF`. If a read error occurs, `fgetc` sets the stream's error indicator and returns `EOF`. *22.4*

`fgetpos` *Get File Position* `<stdio.h>`

```
int fgetpos(FILE * restrict stream,
            fpos_t * restrict pos);
```

Stores the current position of the stream pointed to by `stream` in the object pointed to by `pos`.

Returns Zero if successful. If the call fails, returns a nonzero value and stores an implementation-defined positive value in `errno`. *22.7*

`fgets` *Read String from File* `<stdio.h>`

```
char *fgets(char * restrict s, int n,
            FILE * restrict stream);
```

Reads characters from the stream pointed to by `stream` and stores them in the array pointed to by `s`. Reading stops at the first new-line character (which is stored in the string), when `n - 1` characters have been read, or at end-of-file. `fgets` appends a null character to the string.

Returns `s` (a pointer to the array in which the input is stored). Returns a null pointer if a read error occurs or `fgets` encounters the end of the stream before it has stored any characters. *22.5*

`fgetwc` *Read Wide Character from File (C99)* `<wchar.h>`

```
wint_t fgetwc(FILE *stream);
```

Wide-character version of `fgetc`. *25.5*

`fgetws` *Read Wide String from File (C99)* `<wchar.h>`

```
wchar_t *fgetws(wchar_t * restrict s, int n,
                FILE * restrict stream);
```

Wide-character version of `fgets`. *25.5*

`floor` *Floor* `<math.h>`

```
double floor(double x);
float floorf(float x);
long double floorl(long double x);
```

Returns Largest integer that is less than or equal to x. *23.3*

`fma` *Floating Multiply-Add (C99)* `<math.h>`

```
double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y,
                 long double z);
```

Returns $(x \times y) + z$. The result is rounded only once, using the rounding mode corresponding to `FLT_ROUNDS`. A range error may occur. *23.4*

`fmax` *Floating Maximum (C99)* `<math.h>`

```
double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);
```

Returns Maximum of `x` and `y`. If one argument is a NaN and the other is numeric, the numeric value is returned. *23.4*

`fmin` *Floating Minimum (C99)* `<math.h>`

```
double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);
```

Returns Minimum of `x` and `y`. If one argument is a NaN and the other is numeric, the numeric value is returned. *23.4*

fmod *Floating Modulus* <math.h>

```
double fmod(double x, double y);
float fmodf(float x, float y);
long double fmodl(long double x, long double y);
```

Returns Remainder when x is divided by y. If y is zero, either a domain error occurs or zero is returned. *23.3*

fopen *Open File* <stdio.h>

```
FILE *fopen(const char * restrict filename,
            const char * restrict mode);
```

Opens the file whose name is pointed to by `filename` and associates it with a stream. `mode` specifies the mode in which the file is to be opened. Clears the error and end-of-file indicators for the stream.

Returns A file pointer to be used when performing subsequent operations on the file. Returns a null pointer if the file can't be opened. *22.2*

fpclassify *Floating-Point Classification (C99)* <math.h>

```
int fpclassify(real-floating x);
```

macro

Returns Either `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, or `FP_ZERO`, depending on whether `x` is infinity, not a number, normal, subnormal, or zero, respectively. *23.4*

fprintf *Formatted File Write* <stdio.h>

```
int fprintf(FILE * restrict stream,
            const char * restrict format, ...);
```

Writes output to the stream pointed to by `stream`. The string pointed to by `format` specifies how subsequent arguments will be displayed.

Returns Number of characters written. Returns a negative value if an error occurs. *22.3*

fputc *Write Character to File* <stdio.h>

```
int fputc(int c, FILE *stream);
```

Writes the character `c` to the stream pointed to by `stream`.

Returns `c` (the character written). If a write error occurs, `fputc` sets the stream's error indicator and returns `EOF`. *22.4*

fputs *Write String to File* <stdio.h>

```
int fputs(const char * restrict s,
          FILE * restrict stream);
```

Writes the string pointed to by `s` to the stream pointed to by `stream`.

Returns A nonnegative value if successful. Returns `EOF` if a write error occurs. *22.5*

fputwc *Write Wide Character to File (C99)* <wchar.h>

```
wint_t fputwc(wchar_t c, FILE *stream);
```

Wide-character version of `fputc`. *25.5*

---

fputws *Write Wide String to File (C99)* <wchar.h>

```
int fputws(const wchar_t * restrict s,
           FILE * restrict stream);
```

Wide-character version of `fputs`. *25.5*

---

fread *Read Block from File* <stdio.h>

```
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict stream);
```

Attempts to read `nmemb` elements, each `size` bytes long, from the stream pointed to by `stream` and store them in the array pointed to by `ptr`.

Returns Number of elements actually read. This number will be less than `nmemb` if `fread` encounters end-of-file or a read error occurs. Returns zero if either `nmemb` or `size` is zero. *22.6*

---

free *Free Memory Block* <stdlib.h>

```
void free(void *ptr);
```

Releases the memory block pointed to by `ptr`. (If `ptr` is a null pointer, the call has no effect.) The block must have been allocated by a call of `calloc`, `malloc`, or `realloc`. *17.4*

---

freopen *Reopen File* <stdio.h>

```
FILE *freopen(const char * restrict filename,
              const char * restrict mode,
              FILE * restrict stream);
```

Closes the file associated with `stream`, then opens the file whose name is pointed to by `filename` and associates it with `stream`. The `mode` parameter has the same meaning as in a call of `fopen`. *C99 change:* If `filename` is a null pointer, `freopen` attempts to change the stream's mode to that specified by `mode`.

Returns Value of `stream` if the operation succeeds. Returns a null pointer if the file can't be opened. *22.2*

---

frexp *Split into Fraction and Exponent* <math.h>

```
double frexp(double value, int *exp);
```

frexpf `float frexpf(float value, int *exp);`

frexpl `long double frexpl(long double value, int *exp);`

Splits `value` into a fractional part f and an exponent n in such a way that

`value` $= f \times 2^n$

f is normalized so that either $0.5 \le f < 1$ or $f = 0$. Stores n in the object pointed to by `exp`.

Returns f, the fractional part of `value`. *23.3*

`fscanf` *Formatted File Read* `<stdio.h>`

```
int fscanf(FILE * restrict stream,
           const char * restrict format, ...);
```

Reads input items from the stream pointed to by `stream`. The string pointed to by `format` specifies the format of the items to be read. The arguments that follow `format` point to objects in which the items are to be stored.

Returns Number of input items successfully read and stored. Returns `EOF` if an input failure occurs before any items can be read. *22.3*

`fseek` *File Seek* `<stdio.h>`

```
int fseek(FILE *stream, long int offset, int whence);
```

Changes the file position indicator for the stream pointed to by `stream`. If `whence` is `SEEK_SET`, the new position is the beginning of the file plus `offset` bytes. If `whence` is `SEEK_CUR`, the new position is the current position plus `offset` bytes. If `whence` is `SEEK_END`, the new position is the end of the file plus `offset` bytes. The value of `offset` may be negative. For text streams, either `offset` must be zero or `whence` must be `SEEK_SET` and `offset` a value obtained by a previous call of `ftell`. For binary streams, `fseek` may not support calls in which `whence` is `SEEK_END`.

Returns Zero if the operation is successful, nonzero otherwise. *22.7*

`fsetpos` *Set File Position* `<stdio.h>`

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Sets the file position indicator for the stream pointed to by `stream` according to the value pointed to by `pos` (obtained from a previous call of `fgetpos`).

Returns Zero if successful. If the call fails, returns a nonzero value and stores an implementation-defined positive value in `errno`. *22.7*

`ftell` *Determine File Position* `<stdio.h>`

```
long int ftell(FILE *stream);
```

Returns Current file position indicator for the stream pointed to by `stream`. If the call fails, returns `-1L` and stores an implementation-defined positive value in `errno`. *22.7*

`fwide` *Get and Set Stream Orientation (C99)* `<wchar.h>`

```
int fwide(FILE *stream, int mode);
```

Determines the current orientation of a stream and, if desired, attempts to set its orientation. If `mode` is greater than zero, `fwide` tries to make the stream wide-oriented if it has no orientation. If `mode` is less than zero, it tries to make the

stream byte-oriented if it has no orientation. If mode is zero, the orientation is not changed.

Returns A positive value if the stream has wide orientation after the call, a negative value if it has byte orientation, or zero if it has no orientation. *25.5*

---

fwprintf *Wide-Character Formatted File Write (C99)* <wchar.h>

```
int fwprintf(FILE * restrict stream,
             const wchar_t * restrict format, ...);
```

Wide-character version of fprintf. *25.5*

---

fwrite *Write Block to File* <stdio.h>

```
size_t fwrite(const void * restrict ptr, size_t size,
              size_t nmemb, FILE * restrict stream);
```

Writes nmemb elements, each size bytes long, from the array pointed to by ptr to the stream pointed to by stream.

Returns Number of elements actually written. This number will be less than nmemb if a write error occurs. In C99, returns zero if either nmemb or size is zero. *22.6*

---

fwscanf *Wide-Character Formatted File Read (C99)* <wchar.h>

```
int fwscanf(FILE * restrict stream,
            const wchar_t * restrict format, ...);
```

Wide-character version of fscanf. *25.5*

---

getc *Read Character from File* <stdio.h>

```
int getc(FILE *stream);
```

Reads a character from the stream pointed to by stream. *Note:* getc is normally implemented as a macro; it may evaluate stream more than once.

Returns Character read from the stream. If getc encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getc sets the stream's error indicator and returns EOF. *22.4*

---

getchar *Read Character* <stdio.h>

```
int getchar(void);
```

Reads a character from the stdin stream. *Note:* getchar is normally implemented as a macro.

Returns Character read from the stream. If getchar encounters the end of the stream, it sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getchar sets the stream's error indicator and returns EOF. *7.3, 22.4*

---

getenv *Get Environment String* <stdlib.h>

```
char *getenv(const char *name);
```

Searches the operating system's environment list to see if any string matches the one pointed to by name.

Returns A pointer to the string associated with the matching name. Returns a null pointer if no match is found. *26.2*

---

gets *Read String* `<stdio.h>`

```
char *gets(char *s);
```

Reads characters from the `stdin` stream and stores them in the array pointed to by `s`. Reading stops at the first new-line character (which is discarded) or at end-of-file. `gets` appends a null character to the string.

Returns `s` (a pointer to the array in which the input is stored). Returns a null pointer if a read error occurs or `gets` encounters the end of the stream before it has stored any characters. *13.3, 22.5*

---

getwc *Read Wide Character from File (C99)* `<wchar.h>`

```
wint_t getwc(FILE *stream);
```

Wide-character version of `getc`. *25.5*

---

getwchar *Read Wide Character (C99)* `<wchar.h>`

```
wint_t getwchar(void);
```

Wide-character version of `getchar`. *25.5*

---

gmtime *Convert Calendar Time to Broken-Down UTC Time* `<time.h>`

```
struct tm *gmtime(const time_t *timer);
```

Returns A pointer to a structure containing a broken-down UTC time equivalent to the calendar time pointed to by `timer`. Returns a null pointer if the calendar time can't be converted to UTC. *26.3*

---

hypot *Hypotenuse (C99)* `<math.h>`

```
double hypot(double x, double y);
hypotf  float hypotf(float x, float y);
hypotl  long double hypotl(long double x, long double y);
```

Returns $\sqrt{x^2 + y^2}$ (the hypotenuse of a right triangle with legs `x` and `y`). A range error may occur. *23.4*

---

ilogb *Unbiased Exponent (C99)* `<math.h>`

```
int ilogb(double x);
ilogbf  int ilogbf(float x);
ilogbl  int ilogbl(long double x);
```

Returns Exponent of `x` as a signed integer; equivalent to calling the corresponding `logb` function and casting the returned value to type `int`. Returns `FP_ILOGB0` if `x` is zero, `INT_MAX` if `x` is infinite, and `FP_ILOGBNAN` if `x` is a NaN; a domain error or range error may occur in these cases. *23.4*

---

imaxabs *Greatest-Width Integer Absolute Value (C99)* `<inttypes.h>`

```
intmax_t imaxabs(intmax_t j);
```

Returns Absolute value of `j`. The behavior is undefined if the absolute value of `j` can't be represented. *27.2*

`imaxdiv` *Greatest-Width Integer Division (C99)* `<inttypes.h>`

```
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

Returns A structure of type `imaxdiv_t` containing members named `quot` (the quotient when `numer` is divided by `denom`) and `rem` (the remainder). The behavior is undefined if either part of the result can't be represented. *27.2*

`isalnum` *Test for Alphanumeric* `<ctype.h>`

```
int isalnum(int c);
```

Returns A nonzero value if `c` is alphanumeric and zero otherwise. (`c` is alphanumeric if either `isalpha(c)` or `isdigit(c)` is true.) *23.5*

`isalpha` *Test for Alphabetic* `<ctype.h>`

```
int isalpha(int c);
```

Returns A nonzero value if `c` is alphabetic and zero otherwise. In the `"C"` locale, `c` is alphabetic if either `islower(c)` or `isupper(c)` is true. *23.5*

`isblank` *Test for Blank (C99)* `<ctype.h>`

```
int isblank(int c);
```

Returns A nonzero value if `c` is a blank character that is used to separate words within a line of text. In the `"C"` locale, the blank characters are space (`' '`) and horizontal tab (`'\t'`). *23.5*

`iscntrl` *Test for Control Character* `<ctype.h>`

```
int iscntrl(int c);
```

Returns A nonzero value if `c` is a control character and zero otherwise. *23.5*

`isdigit` *Test for Digit* `<ctype.h>`

```
int isdigit(int c);
```

Returns A nonzero value if `c` is a decimal digit and zero otherwise. *23.5*

`isfinite` *Test for Finite Number (C99)* `<math.h>`

```
int isfinite(real-floating x);                                  macro
```

Returns A nonzero value if `x` is finite (zero, subnormal, or normal, but not infinite or NaN) and zero otherwise. *23.4*

`isgraph` *Test for Graphical Character* `<ctype.h>`

```
int isgraph(int c);
```

Returns A nonzero value if `c` is a printing character (except a space) and zero otherwise. *23.5*

`isgreater` *Test for Greater Than (C99)* `<math.h>`

```
int isgreater(real-floating x, real-floating y);                macro
```

Returns `(x) > (y)`. Unlike the `>` operator, `isgreater` doesn't raise the *invalid* floating-point exception if one or both of the arguments is a NaN. *23.4*

`isgreaterequal` *Test for Greater Than or Equal (C99)* `<math.h>`

`int isgreaterequal(real-floating x, real-floating y);` *macro*

Returns `(x) >= (y)`. Unlike the `>=` operator, `isgreaterequal` doesn't raise the *invalid* floating-point exception if one or both of the arguments is a NaN. *23.4*

`isinf` *Test for Infinity (C99)* `<math.h>`

`int isinf(real-floating x);` *macro*

Returns A nonzero value if `x` is infinity (positive or negative) and zero otherwise. *23.4*

`isless` *Test for Less Than (C99)* `<math.h>`

`int isless(real-floating x, real-floating y);` *macro*

Returns `(x) < (y)`. Unlike the `<` operator, `isless` doesn't raise the *invalid* floating-point exception if one or both of the arguments is a NaN. *23.4*

`islessequal` *Test for Less Than or Equal (C99)* `<math.h>`

`int islessequal(real-floating x, real-floating y);` *macro*

Returns `(x) <= (y)`. Unlike the `<=` operator, `islessequal` doesn't raise the *invalid* floating-point exception if one or both of the arguments is a NaN. *23.4*

`islessgreater` *Test for Less Than or Greater Than (C99)* `<math.h>`

`int islessgreater(real-floating x, real-floating y);` *macro*

Returns `(x) < (y) || (x) > (y)`. Unlike this expression, `islessgreater` doesn't raise the *invalid* floating-point exception if one or both of the arguments is a NaN; also, `x` and `y` are evaluated only once. *23.4*

`islower` *Test for Lower-Case Letter* `<ctype.h>`

```
int islower(int c);
```

Returns A nonzero value if `c` is a lower-case letter and zero otherwise. *23.5*

`isnan` *Test for NaN (C99)* `<math.h>`

`int isnan(real-floating x);` *macro*

Returns A nonzero value if `x` is a NaN value and zero otherwise. *23.4*

`isnormal` *Test for Normal Number (C99)* `<math.h>`

`int isnormal(real-floating x);` *macro*

Returns A nonzero value if `x` has a normal value (not zero, subnormal, infinite, or NaN) and zero otherwise. *23.4*

`isprint` *Test for Printing Character* `<ctype.h>`

```
int isprint(int c);
```

Returns A nonzero value if `c` is a printing character (including a space) and zero otherwise. *23.5*

---

ispunct *Test for Punctuation Character* `<ctype.h>`

```
int ispunct(int c);
```

Returns A nonzero value if `c` is a punctuation character and zero otherwise. All printing characters except the space (`' '`) and the alphanumeric characters are considered punctuation. *C99 change:* In the `"C"` locale, all printing characters except those for which `isspace` or `isalnum` is true are considered punctuation. *23.5*

---

isspace *Test for White-Space Character* `<ctype.h>`

```
int isspace(int c);
```

Returns A nonzero value if `c` is a white-space character and zero otherwise. In the `"C"` locale, the white-space characters are space (`' '`), form feed (`'\f'`), new-line (`'\n'`), carriage return (`'\r'`), horizontal tab (`'\t'`), and vertical tab (`'\v'`). *23.5*

---

isunordered *Test for Unordered (C99)* `<math.h>`

```
int isunordered(real-floating x, real-floating y);
```
macro

Returns 1 if `x` and `y` are unordered (at least one is a NaN) and 0 otherwise. *23.4*

---

isupper *Test for Upper-Case Letter* `<ctype.h>`

```
int isupper(int c);
```

Returns A nonzero value if `c` is an upper-case letter and zero otherwise. *23.5*

---

iswalnum *Test for Alphanumeric Wide Character (C99)* `<wctype.h>`

```
int iswalnum(wint_t wc);
```

Returns A nonzero value if `wc` is alphanumeric and zero otherwise. (`wc` is alphanumeric if either `iswalpha(wc)` or `iswdigit(wc)` is true.) *25.6*

---

iswalpha *Test for Alphabetic Wide Character (C99)* `<wctype.h>`

```
int iswalpha(wint_t wc);
```

Returns A nonzero value if `wc` is alphabetic and zero otherwise. (`wc` is alphabetic if `iswupper(wc)` or `iswlower(wc)` is true, or if `wc` is one of a locale-specific set of alphabetic wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true.) *25.6*

---

iswblank *Test for Blank Wide Character (C99)* `<wctype.h>`

```
int iswblank(wint_t wc);
```

Returns A nonzero value if `wc` is a standard blank wide character or one of a locale-specific set of wide characters for which `iswspace` is true and that are used to separate words within a line of text. In the `"C"` locale, `iswblank` returns true only for the standard blank characters: space (`L' '`) and horizontal tab (`L'\t'`). *25.6*

iswcntrl *Test for Control Wide Character (C99)* <wctype.h>

```
int iswcntrl(wint_t wc);
```

Returns A nonzero value if wc is a control wide character and zero otherwise. *25.6*

---

iswctype *Test Type of Wide Character (C99)* <wctype.h>

```
int iswctype(wint_t wc, wctype_t desc);
```

Returns A nonzero value if the wide character wc has the property described by desc. (desc must be a value returned by a call of wctype; the current setting of the LC_CTYPE category must be the same during both calls.) Returns zero otherwise. *25.6*

---

iswdigit *Test for Digit Wide Character (C99)* <wctype.h>

```
int iswdigit(wint_t wc);
```

Returns A nonzero value if wc corresponds to a decimal digit and zero otherwise. *25.6*

---

iswgraph *Test for Graphical Wide Character (C99)* <wctype.h>

```
int iswgraph(wint_t wc);
```

Returns A nonzero value if iswprint(wc) is true and iswspace(wc) is false. Returns zero otherwise. *25.6*

---

iswlower *Test for Lower-Case Wide Character (C99)* <wctype.h>

```
int iswlower(wint_t wc);
```

Returns A nonzero value if wc corresponds to a lower-case letter or is one of a locale-specific set of wide characters for which none of iswcntrl, iswdigit, iswpunct, or iswspace is true. Returns zero otherwise. *25.6*

---

iswprint *Test for Printing Wide Character (C99)* <wctype.h>

```
int iswprint(wint_t wc);
```

Returns A nonzero value if wc is a printing wide character and zero otherwise. *25.6*

---

iswpunct *Test for Punctuation Wide Character (C99)* <wctype.h>

```
int iswpunct(wint_t wc);
```

Returns A nonzero value if wc is a printing wide character that is one of a locale-specific set of punctuation wide characters for which neither iswspace nor iswalnum is true. Returns zero otherwise. *25.6*

---

iswspace *Test for White-Space Wide Character (C99)* <wctype.h>

```
int iswspace(wint_t wc);
```

Returns A nonzero value if wc is one of a locale-specific set of white-space wide characters for which none of iswalnum, iswgraph, or iswpunct is true. Returns zero otherwise. *25.6*

iswupper *Test for Upper-Case Wide Character (C99)* `<wctype.h>`

```
int iswupper(wint_t wc);
```

Returns A nonzero value if `wc` corresponds to an upper-case letter or is one of a locale-specific set of wide characters for which none of `iswcntrl`, `iswdigit`, `iswpunct`, or `iswspace` is true. Returns zero otherwise. *25.6*

---

iswxdigit *Test for Hexadecimal-Digit Wide Character (C99)* `<wctype.h>`

```
int iswxdigit(wint_t wc);
```

Returns A nonzero value if `wc` corresponds to a hexadecimal digit (`0`–`9`, `a`–`f`, `A`–`F`) and zero otherwise. *25.6*

---

isxdigit *Test for Hexadecimal Digit* `<ctype.h>`

```
int isxdigit(int c);
```

Returns A nonzero value if `c` is a hexadecimal digit (`0`–`9`, `a`–`f`, `A`–`F`) and zero otherwise. *23.5*

---

labs *Long Integer Absolute Value* `<stdlib.h>`

```
long int labs(long int j);
```

Returns Absolute value of `j`. The behavior is undefined if the absolute value of `j` can't be represented. *26.2*

---

ldexp *Combine Fraction and Exponent* `<math.h>`

```
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

Returns $x \times 2^{exp}$. A range error may occur. *23.3*

---

ldiv *Long Integer Division* `<stdlib.h>`

```
ldiv_t ldiv(long int numer, long int denom);
```

Returns An `ldiv_t` structure containing members named `quot` (the quotient when `numer` is divided by `denom`) and `rem` (the remainder). The behavior is undefined if either part of the result can't be represented. *26.2*

---

lgamma *Logarithm of Gamma Function (C99)* `<math.h>`

```
double lgamma(double x);
float lgammaf(float x);
long double lgammal(long double x);
```

Returns $\ln(|\Gamma(x)|)$, where Γ is the gamma function. A range error occurs if `x` is too large and may occur if `x` is a negative integer or zero. *23.4*

---

llabs *Long Long Integer Absolute Value (C99)* `<stdlib.h>`

```
long long int llabs(long long int j);
```

Returns Absolute value of `j`. The behavior is undefined if the absolute value of `j` can't be represented. *26.2*

`lldiv` *Long Long Integer Division (C99)* `<stdlib.h>`

```
lldiv_t lldiv(long long int numer,
              long long int denom);
```

Returns An `lldiv_t` structure containing members named `quot` (the quotient when `numer` is divided by `denom`) and `rem` (the remainder). The behavior is undefined if either part of the result can't be represented. *26.2*

`llrint` *Round to Long Long Integer Using Current Direction (C99)* `<math.h>`

```
long long int llrint(double x);
long long int llrintf(float x);
long long int llrintl(long double x);
```

Returns `x` rounded to the nearest integer using the current rounding direction. If the rounded value is outside the range of the `long long int` type, the result is unspecified and a domain or range error may occur. *23.4*

`llround` *Round to Nearest Long Long Integer (C99)* `<math.h>`

```
long long int llround(double x);
long long int llroundf(float x);
long long int llroundl(long double x);
```

Returns `x` rounded to the nearest integer, with halfway cases rounded away from zero. If the rounded value is outside the range of the `long long int` type, the result is unspecified and a domain or range error may occur. *23.4*

`localeconv` *Get Locale Conventions* `<locale.h>`

```
struct lconv *localeconv(void);
```

Returns A pointer to a structure containing information about the current locale. *25.1*

`localtime` *Convert Calendar Time to Broken-Down Local Time* `<time.h>`

```
struct tm *localtime(const time_t *timer);
```

Returns A pointer to a structure containing a broken-down local time equivalent to the calendar time pointed to by `timer`. Returns a null pointer if the calendar time can't be converted to local time. *26.3*

`log` *Natural Logarithm* `<math.h>`

```
double log(double x);
float logf(float x);
long double logl(long double x);
```

Returns Logarithm of `x` to the base *e*. A domain error occurs if `x` is negative. A range error may occur if `x` is zero. *23.3*

log10 *Common Logarithm* <math.h>

```
double log10(double x);
float log10f(float x);
long double log10l(long double x);
```

Returns Logarithm of x to the base 10. A domain error occurs if x is negative. A range error may occur if x is zero. *23.3*

log1p *Natural Logarithm of 1 Plus Argument (C99)* <math.h>

```
double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);
```

Returns Logarithm of 1 + x to the base *e*. A domain error occurs if x is less than –1. A range error may occur if x is equal to –1. *23.4*

log2 *Base-2 Logarithm (C99)* <math.h>

```
double log2(double x);
float log2f(float x);
long double log2l(long double x);
```

Returns Logarithm of x to the base 2. A domain error occurs if x is negative. A range error may occur if x is zero. *23.4*

logb *Radix-Independent Exponent (C99)* <math.h>

```
double logb(double x);
float logbf(float x);
long double logbl(long double x);
```

Returns $\log_r(|x|)$, where *r* is the radix of floating-point arithmetic (defined by the macro FLT_RADIX, which typically has the value 2). A domain error or range error may occur if x is zero. *23.4*

longjmp *Nonlocal Jump* <setjmp.h>

```
void longjmp(jmp_buf env, int val);
```

Restores the environment stored in env and returns from the call of setjmp that originally saved env. If val is nonzero, it will be setjmp's return value; if val is 0, setjmp returns 1. *24.4*

lrint *Round to Long Integer Using Current Direction (C99)* <math.h>

```
long int lrint(double x);
long int lrintf(float x);
long int lrintl(long double x);
```

Returns x rounded to the nearest integer using the current rounding direction. If the rounded value is outside the range of the long int type, the result is unspecified and a domain or range error may occur. *23.4*

lround — Round to Nearest Long Integer (C99) — <math.h>

```
long int lround(double x);
long int lroundf(float x);
long int lroundl(long double x);
```

Returns x rounded to the nearest integer, with halfway cases rounded away from zero. If the rounded value is outside the range of the `long int` type, the result is unspecified and a domain or range error may occur. *23.4*

malloc — Allocate Memory Block — <stdlib.h>

```
void *malloc(size_t size);
```

Allocates a block of memory with `size` bytes. The block is not cleared.

Returns A pointer to the beginning of the block. Returns a null pointer if a block of the requested size can't be allocated. *17.2*

mblen — Length of Multibyte Character — <stdlib.h>

```
int mblen(const char *s, size_t n);
```

Returns If `s` is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. If `s` points to a null character, returns zero. Otherwise, returns the number of bytes in the multibyte character pointed to by `s`; returns –1 if the next `n` or fewer bytes don't form a valid multibyte character. *25.2*

mbrlen — Length of Multibyte Character – Restartable (C99) — <wchar.h>

```
size_t mbrlen(const char * restrict s, size_t n,
              mbstate_t * restrict ps);
```

Determines the number of bytes in the array pointed to by `s` that are required to complete a multibyte character. `ps` should point to an object of type `mbstate_t` that contains the current conversion state. A call of `mbrlen` is equivalent to

```
mbrtowc(NULL, s, n, ps)
```

except that if `ps` is a null pointer, the address of an internal object is used instead.

Returns See `mbrtowc`. *25.5*

mbrtowc — Convert Multibyte Character to Wide Character – Restartable (C99) — <wchar.h>

```
size_t mbrtowc(wchar_t * restrict pwc,
               const char * restrict s, size_t n,
               mbstate_t * restrict ps);
```

If `s` is a null pointer, a call of `mbrtowc` is equivalent to

```
mbrtowc(NULL, "", 1, ps)
```

Otherwise, `mbrtowc` examines up to `n` bytes in the array pointed to by `s` to see if

they complete a valid multibyte character. If so, the multibyte character is converted into a wide character. If `pwc` isn't a null pointer, the wide character is stored in the object pointed to by `pwc`. The value of `ps` should be a pointer to an object of type `mbstate_t` that contains the current conversion state. If `ps` is a null pointer, `mbrtowc` uses an internal object to store the conversion state. If the result of the conversion is the null wide character, the `mbstate_t` object used during the call is left in the initial conversion state.

Returns 0 if the conversion produces a null wide character. Returns a number between 1 and `n` if the conversion produces a wide character other than null, where the value returned is the number of bytes used to complete the multibyte character. Returns `(size_t)(-2)` if the `n` bytes pointed to by `s` weren't enough to complete a multibyte character. Returns `(size_t)(-1)` and stores `EILSEQ` in `errno` if an encoding error occurs. *25.5*

`mbsinit` *Test for Initial Conversion State (C99)* `<wchar.h>`

```
int mbsinit(const mbstate_t *ps);
```

Returns A nonzero value if `ps` is a null pointer or it points to an `mbstate_t` object that describes an initial conversion state; otherwise, returns zero. *25.5*

`mbsrtowcs` *Convert Multibyte String to Wide String – Restartable (C99)* `<wchar.h>`

```
size_t mbsrtowcs(wchar_t * restrict dst,
                 const char ** restrict src,
                 size_t len, mbstate_t * restrict ps);
```

Converts a sequence of multibyte characters from the array indirectly pointed to by `src` into a sequence of corresponding wide characters. `ps` should point to an object of type `mbstate_t` that contains the current conversion state. If the argument corresponding to `ps` is a null pointer, `mbsrtowcs` uses an internal object to store the conversion state. If `dst` isn't a null pointer, the converted characters are stored in the array that it points to. Conversion continues up to and including a terminating null character, which is also stored. Conversion stops earlier if a sequence of bytes is encountered that doesn't form a valid multibyte character or—if `dst` isn't a null pointer—when `len` wide characters have been stored in the array. If `dst` isn't a null pointer, the object pointed to by `src` is assigned either a null pointer (if a terminating null character was reached) or the address just past the last multibyte character converted (if any). If the conversion ends at a null character and if `dst` isn't a null pointer, the resulting state is the initial conversion state.

Returns Number of multibyte characters successfully converted, not including any terminating null character. Returns `(size_t)(-1)` and stores `EILSEQ` in `errno` if an invalid multibyte character is encountered. *25.5*

`mbstowcs` *Convert Multibyte String to Wide String* `<stdlib.h>`

```
size_t mbstowcs(wchar_t * restrict pwcs,
                const char * restrict s, size_t n);
```

Converts the sequence of multibyte characters pointed to by `s` into a sequence of wide characters, storing at most `n` wide characters in the array pointed to by `pwcs`. Conversion ends if a null character is encountered; it is converted into a null wide character.

Returns Number of array elements modified, not including the null wide character, if any. Returns `(size_t)(-1)` if an invalid multibyte character is encountered. *25.2*

---

mbtowc *Convert Multibyte Character to Wide Character* <stdlib.h>

```
int mbtowc(wchar_t * restrict pwc,
           const char * restrict s, size_t n);
```

If `s` isn't a null pointer, converts the multibyte character pointed to by `s` into a wide character; at most `n` bytes will be examined. If the multibyte character is valid and `pwc` isn't a null pointer, stores the value of the wide character in the object pointed to by `pwc`.

Returns If `s` is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. If `s` points to a null character, returns zero. Otherwise, returns the number of bytes in the multibyte character pointed to by `s`; returns –1 if the next `n` or fewer bytes don't form a valid multibyte character. *25.2*

---

memchr *Search Memory Block for Character* <string.h>

```
void *memchr(const void *s, int c, size_t n);
```

Returns A pointer to the first occurrence of the character `c` among the first `n` characters of the object pointed to by `s`. Returns a null pointer if `c` isn't found. *23.6*

---

memcmp *Compare Memory Blocks* <string.h>

```
int memcmp(const void *s1, const void *s2, size_t n);
```

Returns A negative, zero, or positive integer, depending on whether the first `n` characters of the object pointed to by `s1` are less than, equal to, or greater than the first `n` characters of the object pointed to by `s2`. *23.6*

---

memcpy *Copy Memory Block* <string.h>

```
void *memcpy(void * restrict s1,
             const void * restrict s2, size_t n);
```

Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. The behavior is undefined if the objects overlap.

Returns `s1` (a pointer to the destination). *23.6*

---

memmove *Copy Memory Block* <string.h>

```
void *memmove(void *s1, const void *s2, size_t n);
```

Copies `n` characters from the object pointed to by `s2` into the object pointed to by `s1`. Will work properly if the objects overlap.

Returns `s1` (a pointer to the destination). *23.6*

memset *Initialize Memory Block* <string.h>

```
void *memset(void *s, int c, size_t n);
```

Stores `c` in each of the first `n` characters of the object pointed to by `s`.

Returns `s` (a pointer to the object). *23.6*

---

mktime *Convert Broken-Down Local Time to Calendar Time* <time.h>

```
time_t mktime(struct tm *timeptr);
```

Converts a broken-down local time (stored in the structure pointed to by `timeptr`) into a calendar time. The members of the structure aren't required to be within their legal ranges; also, the values of `tm_wday` (day of the week) and `tm_yday` (day of the year) are ignored. `mktime` stores values in `tm_wday` and `tm_yday` after adjusting the other members to bring them into their proper ranges.

Returns A calendar time corresponding to the structure pointed to by `timeptr`. Returns `(time_t)(-1)` if the calendar time can't be represented. *26.3*

---

modf *Split into Integer and Fractional Parts* <math.h>

```
double modf(double value, double *iptr);
modff  float modff(float value, float *iptr);
modfl  long double modfl(long double value, long double *iptr);
```

Splits `value` into integer and fractional parts; stores the integer part in the object pointed to by `iptr`.

Returns Fractional part of `value`. *23.3*

---

nan *Create NaN (C99)* <math.h>

```
double nan(const char *tagp);
nanf  float nanf(const char *tagp);
nanl  long double nanl(const char *tagp);
```

Returns A "quiet" NaN whose binary pattern is determined by the string pointed to by `tagp`. Returns zero if quiet NaNs aren't supported. *23.4*

---

nearbyint *Round to Integral Value Using Current Direction (C99)* <math.h>

```
double nearbyint(double x);
nearbyintf  float nearbyintf(float x);
nearbyintl  long double nearbyintl(long double x);
```

Returns `x` rounded to an integer (in floating-point format) using the current rounding direction. Doesn't raise the *inexact* floating-point exception. *23.4*

---

nextafter *Next Number After (C99)* <math.h>

```
double nextafter(double x, double y);
nextafterf  float nextafterf(float x, float y);
nextafterl  long double nextafterl(long double x, long double y);
```

Returns Next representable value after x in the direction of y. Returns the value just before x if y < x or the value just after x if x < y. Returns y if x equals y. A range error may occur if the magnitude of x is the largest representable finite value and the result is infinite or not representable. *23.4*

---

nexttoward *Next Number Toward (C99)* <math.h>

```
double nexttoward(double x, long double y);
```
nexttowardf `float nexttowardf(float x, long double y);`
nexttowardl `long double nexttowardl(long double x, long double y);`

Returns Next representable value after x in the direction of y (see nextafter). Returns y converted to the function's type if x equals y. *23.4*

---

perror *Print Error Message* <stdio.h>

```
void perror(const char *s);
```

Writes the following message to the stderr stream:

string : *error-message*

string is the string pointed to by s and *error-message* is an implementation-defined message that matches the one returned by the call strerror(errno). *24.2*

---

pow *Power* <math.h>

```
double pow(double x, double y);
```
powf `float powf(float x, float y);`
powl `long double powl(long double x, long double y);`

Returns x raised to the power y. A domain or range error may occur in certain cases, which vary between C89 and C99. *23.3*

---

printf *Formatted Write* <stdio.h>

```
int printf(const char * restrict format, ...);
```

Writes output to the stdout stream. The string pointed to by format specifies how subsequent arguments will be displayed.

Returns Number of characters written. Returns a negative value if an error occurs. *3.1, 22.3*

---

putc *Write Character to File* <stdio.h>

```
int putc(int c, FILE *stream);
```

Writes the character c to the stream pointed to by stream. *Note:* putc is normally implemented as a macro; it may evaluate stream more than once.

Returns c (the character written). If a write error occurs, putc sets the stream's error indicator and returns EOF. *22.4*

---

putchar *Write Character* <stdio.h>

```
int putchar(int c);
```

Writes the character c to the stdout stream. *Note:* putchar is normally implemented as a macro.

Returns c (the character written). If a write error occurs, `putchar` sets the stream's error indicator and returns `EOF`. *7.3, 22.4*

---

puts *Write String* `<stdio.h>`

```
int puts(const char *s);
```

Writes the string pointed to by `s` to the `stdout` stream, then writes a new-line character.

Returns A nonnegative value if successful. Returns `EOF` if a write error occurs. *13.3, 22.5*

---

putwc *Write Wide Character to File (C99)* `<wchar.h>`

```
wint_t putwc(wchar_t c, FILE *stream);
```

Wide-character version of `putc`. *25.5*

---

putwchar *Write Wide Character (C99)* `<wchar.h>`

```
wint_t putwchar(wchar_t c);
```

Wide-character version of `putchar`. *25.5*

---

qsort *Sort Array* `<stdlib.h>`

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

Sorts the array pointed to by `base`. The array has `nmemb` elements, each `size` bytes long. `compar` is a pointer to a comparison function. When passed pointers to two array elements, the comparison function must return a negative, zero, or positive integer, depending on whether the first array element is less than, equal to, or greater than the second. *17.7, 26.2*

---

raise *Raise Signal* `<signal.h>`

```
int raise(int sig);
```

Raises the signal whose number is `sig`.

Returns Zero if successful, nonzero otherwise. *24.3*

---

rand *Generate Pseudo-Random Number* `<stdlib.h>`

```
int rand(void);
```

Returns A pseudo-random integer between 0 and `RAND_MAX` (inclusive). *26.2*

---

realloc *Resize Memory Block* `<stdlib.h>`

```
void *realloc(void *ptr, size_t size);
```

`ptr` is assumed to point to a block of memory previously obtained from `calloc`, `malloc`, or `realloc`. `realloc` allocates a block of `size` bytes, copying the contents of the old block if necessary.

Returns A pointer to the beginning of the new memory block. Returns a null pointer if a block of the requested size can't be allocated. *17.3*

remainder *Remainder (C99)* <math.h>

```
double remainder(double x, double y);
remainderf  float remainderf(float x, float y);
remainderl  long double remainderl(long double x, long double y);
```

Returns x – *n*y, where *n* is the integer nearest the exact value of x/y. (If x/y is halfway between two integers, *n* is even.) If x – *n*y = 0, the return value has the same sign as x. If y is zero, either a domain error occurs or zero is returned. *23.4*

---

remove *Remove File* <stdio.h>

```
int remove(const char *filename);
```

Deletes the file whose name is pointed to by `filename`.

Returns Zero if successful, nonzero otherwise. *22.2*

---

remquo *Remainder and Quotient (C99)* <math.h>

```
double remquo(double x, double y, int *quo);
remquof  float remquof(float x, float y, int *quo);
remquol  long double remquol(long double x, long double y,
                             int *quo);
```

Computes both the remainder and the quotient when x is divided by y. The object pointed to by quo is modified so that it contains *n* low-order bits of the integer quotient |x/y|, where *n* is implementation-defined but must be at least three. The value stored in this object will be negative if x/y < 0.

Returns Same value as the corresponding `remainder` function. If y is zero, either a domain error occurs or zero is returned. *23.4*

---

rename *Rename File* <stdio.h>

```
int rename(const char *old, const char *new);
```

Changes the name of a file. `old` and `new` point to strings containing the old name and new name, respectively.

Returns Zero if the renaming is successful. Returns a nonzero value if the operation fails (perhaps because the old file is currently open). *22.2*

---

rewind *Rewind File* <stdio.h>

```
void rewind(FILE *stream);
```

Sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. Clears the error and end-of-file indicators for the stream. *22.7*

---

rint *Round to Integral Value Using Current Direction (C99)* <math.h>

```
double rint(double x);
rintf  float rintf(float x);
rintl  long double rintl(long double x);
```

Returns x rounded to an integer (in floating-point format) using the current rounding direc-

tion. May raise the *inexact* floating-point exception if the result has a different value than x. *23.4*

---

round *Round to Nearest Integral Value (C99)* <math.h>

```
double round(double x);
float roundf(float x);
long double roundl(long double x);
```

Returns x rounded to the nearest integer (in floating-point format). Halfway cases are rounded away from zero. *23.4*

---

scalbln *Scale Floating-Point Number Using Long Integer (C99)* <math.h>

```
double scalbln(double x, long int n);
float scalblnf(float x, long int n);
long double scalblnl(long double x, long int n);
```

Returns x × FLT_RADIX^n, computed in an efficient way. A range error may occur. *23.4*

---

scalbn *Scale Floating-Point Number Using Integer (C99)* <math.h>

```
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
```

Returns x × FLT_RADIX^n, computed in an efficient way. A range error may occur. *23.4*

---

scanf *Formatted Read* <stdio.h>

```
int scanf(const char * restrict format, ...);
```

Reads input items from the stdin stream. The string pointed to by format specifies the format of the items to be read. The arguments that follow format point to objects in which the items are to be stored.

Returns Number of input items successfully read and stored. Returns EOF if an input failure occurs before any items can be read. *3.2, 22.3*

---

setbuf *Set Buffer* <stdio.h>

```
void setbuf(FILE * restrict stream,
            char * restrict buf);
```

If buf isn't a null pointer, a call of setbuf is equivalent to:

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

Otherwise, it's equivalent to:

```
(void) setvbuf(stream, NULL, _IONBF, 0);
```

22.2

---

setjmp *Prepare for Nonlocal Jump* <setjmp.h>

```
int setjmp(jmp_buf env);
```

macro

Stores the current environment in env for use in a later call of longjmp.

Returns Zero when called directly. Returns a nonzero value when returning from a call of longjmp. *24.4*

setlocale *Set Locale* <locale.h>

```
char *setlocale(int category, const char *locale);
```

Sets a portion of the program's locale. `category` indicates which portion is affected. `locale` points to a string representing the new locale.

Returns If `locale` is a null pointer, returns a pointer to the string associated with `category` for the current locale. Otherwise, returns a pointer to the string associated with `category` for the new locale. Returns a null pointer if the operation fails.

25.1

---

setvbuf *Set Buffer* <stdio.h>

```
int setvbuf(FILE * restrict stream,
            char * restrict buf,
            int mode, size_t size);
```

Changes the buffering of the stream pointed to by `stream`. The value of `mode` can be either `_IOFBF` (full buffering), `_IOLBF` (line buffering), or `_IONBF` (no buffering). If `buf` is a null pointer, a buffer is automatically allocated if needed. Otherwise, `buf` points to a memory block that can be used as the buffer; `size` is the number of bytes in the block. *Note:* `setvbuf` must be called after the stream is opened but before any other operations are performed on it.

Returns Zero if the operation is successful. Returns a nonzero value if `mode` is invalid or the request can't be honored. *22.2*

---

signal *Install Signal Handler* <signal.h>

```
void (*signal(int sig, void (*func)(int)))(int);
```

Installs the function pointed to by `func` as the handler for the signal whose number is `sig`. Passing `SIG_DFL` as the second argument causes default handling for the signal; passing `SIG_IGN` causes the signal to be ignored.

Returns A pointer to the previous handler for this signal; returns `SIG_ERR` and stores a positive value in `errno` if the handler can't be installed. *24.3*

---

signbit *Sign Bit (C99)* <math.h>

```
int signbit(real-floating x);
```

macro

Returns A nonzero value if the sign of `x` is negative and zero otherwise. The value of `x` may be any number, including infinity and NaN. *23.4*

---

sin *Sine* <math.h>

```
double sin(double x);
```

sinf `float sinf(float x);`

sinl `long double sinl(long double x);`

Returns Sine of `x` (measured in radians). *23.3*

---

sinh *Hyperbolic Sine* <math.h>

```
double sinh(double x);
```

sinhf `float sinhf(float x);`
sinhl `long double sinhl(long double x);`

Returns Hyperbolic sine of x. A range error occurs if the magnitude of x is too large. *23.3*

---

snprintf — *Bounded Formatted String Write (C99)* — <stdio.h>

```
int snprintf(char * restrict s, size_t n,
             const char * restrict format, ...);
```

Equivalent to `fprintf`, but stores characters in the array pointed to by `s` instead of writing them to a stream. No more than `n` – 1 characters will be written to the array. The string pointed to by `format` specifies how subsequent arguments will be displayed. Stores a null character in the array at the end of output.

Returns Number of characters that would have been stored in the array (not including the null character) had there been no length restriction. Returns a negative value if an encoding error occurs. *22.8*

---

sprintf — *Formatted String Write* — <stdio.h>

```
int sprintf(char * restrict s,
            const char * restrict format, ...);
```

Equivalent to `fprintf`, but stores characters in the array pointed to by `s` instead of writing them to a stream. The string pointed to by `format` specifies how subsequent arguments will be displayed. Stores a null character in the array at the end of output.

Returns Number of characters stored in the array, not including the null character. In C99, returns a negative value if an encoding error occurs. *22.8*

---

sqrt — *Square Root* — <math.h>

```
double sqrt(double x);
```
sqrtf `float sqrtf(float x);`
sqrtl `long double sqrtl(long double x);`

Returns Nonnegative square root of x. A domain error occurs if x is negative. *23.3*

---

srand — *Seed Pseudo-Random Number Generator* — <stdlib.h>

```
void srand(unsigned int seed);
```

Uses `seed` to initialize the sequence of pseudo-random numbers produced by calling `rand`. *26.2*

---

sscanf — *Formatted String Read* — <stdio.h>

```
int sscanf(const char * restrict s,
           const char * restrict format, ...);
```

Equivalent to `fscanf`, but reads characters from the string pointed to by `s` instead of reading them from a stream. The string pointed to by `format` specifies the format of the items to be read. The arguments that follow `format` point to objects in which the items are to be stored.

Returns Number of input items successfully read and stored. Returns `EOF` if an input failure occurs before any items could be read. *22.8*

`strcat` *String Concatenation* `<string.h>`

```
char *strcat(char * restrict s1,
             const char * restrict s2);
```

Appends characters from the string pointed to by `s2` to the string pointed to by `s1`.

Returns `s1` (a pointer to the concatenated string). *13.5, 23.6*

`strchr` *Search String for Character* `<string.h>`

```
char *strchr(const char *s, int c);
```

Returns A pointer to the first occurrence of the character `c` in the string pointed to by `s`. Returns a null pointer if `c` isn't found. *23.6*

`strcmp` *String Comparison* `<string.h>`

```
int strcmp(const char *s1, const char *s2);
```

Returns A negative, zero, or positive integer, depending on whether the string pointed to by `s1` is less than, equal to, or greater than the string pointed to by `s2`. *13.5, 23.6*

`strcoll` *String Comparison Using Locale-Specific Collating Sequence* `<string.h>`

```
int strcoll(const char *s1, const char *s2);
```

Returns A negative, zero, or positive integer, depending on whether the string pointed to by `s1` is less than, equal to, or greater than the string pointed to by `s2`. The comparison is performed according to the rules of the current locale's `LC_COLLATE` category. *23.6*

`strcpy` *String Copy* `<string.h>`

```
char *strcpy(char * restrict s1,
             const char * restrict s2);
```

Copies the string pointed to by `s2` into the array pointed to by `s1`.

Returns `s1` (a pointer to the destination). *13.5, 23.6*

`strcspn` *Search String for Initial Span of Characters Not in Set* `<string.h>`

```
size_t strcspn(const char *s1, const char *s2);
```

Returns Length of the longest initial segment of the string pointed to by `s1` that doesn't contain any character in the string pointed to by `s2`. *23.6*

`strerror` *Convert Error Number to String* `<string.h>`

```
char *strerror(int errnum);
```

Returns A pointer to a string containing an error message corresponding to the value of `errnum`. *24.2*

strftime *Write Formatted Date and Time to String* `<time.h>`

```
size_t strftime(char * restrict s, size_t maxsize,
                const char * restrict format,
                const struct tm * restrict timeptr);
```

Stores characters in the array pointed to by `s` under control of the string pointed to by `format`. The format string may contain ordinary characters, which are copied unchanged, and conversion specifiers, which are replaced by values from the structure pointed to by `timeptr`. The `maxsize` parameter limits the number of characters (including the null character) that can be stored.

Returns Number of characters stored (not including the terminating null character). Returns zero if the number of characters to be stored (including the null character) exceeds `maxsize`. *26.3*

---

strlen *String Length* `<string.h>`

```
size_t strlen(const char *s);
```

Returns Length of the string pointed to by `s`, not including the null character. *13.5, 23.6*

---

strncat *Bounded String Concatenation* `<string.h>`

```
char *strncat(char * restrict s1,
              const char * restrict s2, size_t n);
```

Appends characters from the array pointed to by `s2` to the string pointed to by `s1`. Copying stops when a null character is encountered or `n` characters have been copied.

Returns `s1` (a pointer to the concatenated string). *13.5, 23.6*

---

strncmp *Bounded String Comparison* `<string.h>`

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Returns A negative, zero, or positive integer, depending on whether the first `n` characters of the array pointed to by `s1` are less than, equal to, or greater than the first `n` characters of the array pointed to by `s2`. Comparison stops if a null character is encountered in either array. *23.6*

---

strncpy *Bounded String Copy* `<string.h>`

```
char *strncpy(char * restrict s1,
              const char * restrict s2, size_t n);
```

Copies the first `n` characters of the array pointed to by `s2` into the array pointed to by `s1`. If it encounters a null character in the array pointed to by `s2`, `strncpy` adds null characters to the array pointed to by `s1` until a total of `n` characters have been written.

Returns `s1` (a pointer to the destination). *13.5, 23.6*

strpbrk *Search String for One of a Set of Characters* `<string.h>`

```
char *strpbrk(const char *s1, const char *s2);
```

Returns A pointer to the leftmost character in the string pointed to by `s1` that matches any character in the string pointed to by `s2`. Returns a null pointer if no match is found. *23.6*

strrchr *Search String in Reverse for Character* `<string.h>`

```
char *strrchr(const char *s, int c);
```

Returns A pointer to the last occurrence of the character `c` in the string pointed to by `s`. Returns a null pointer if `c` isn't found. *23.6*

strspn *Search String for Initial Span of Characters in Set* `<string.h>`

```
size_t strspn(const char *s1, const char *s2);
```

Returns Length of the longest initial segment in the string pointed to by `s1` that consists entirely of characters in the string pointed to by `s2`. *23.6*

strstr *Search String for Substring* `<string.h>`

```
char *strstr(const char *s1, const char *s2);
```

Returns A pointer to the first occurrence in the string pointed to by `s1` of the sequence of characters in the string pointed to by `s2`. Returns a null pointer if no match is found. *23.6*

strtod *Convert String to Double* `<stdlib.h>`

```
double strtod(const char * restrict nptr,
              char ** restrict endptr);
```

Skips white-space characters in the string pointed to by `nptr`, then converts subsequent characters into a `double` value. If `endptr` isn't a null pointer, `strtod` modifies the object pointed to by `endptr` so that it points to the first leftover character. If no `double` value is found, or if it has the wrong form, `strtod` stores `nptr` in the object pointed to by `endptr`. If the number is too large or small to represent, it stores `ERANGE` in `errno`. *C99 changes:* The string pointed to by `nptr` may contain a hexadecimal floating-point number, infinity, or NaN. Whether `ERANGE` is stored in `errno` when the number is too small to represent is implementation-defined.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus `HUGE_VAL`, depending on the number's sign. Returns zero if the number is too small to represent. *C99 change:* If the number is too small to represent, `strtod` returns a value whose magnitude is no greater than the smallest normalized positive `double`. *26.2*

strtof *Convert String to Float (C99)* `<stdlib.h>`

```
float strtof(const char * restrict nptr,
             char ** restrict endptr);
```

`strtof` is identical to `strtod`, except that it converts a string to a `float` value.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus `HUGE_VALF`, depending on the number's sign. If the number is too small to represent, returns a value whose magnitude is no greater than the smallest normalized positive `float`. *26.2*

---

strtoimax *Convert String to Greatest-Width Integer (C99)* `<inttypes.h>`

```
intmax_t strtoimax(const char * restrict nptr,
                   char ** restrict endptr, int base);
```

`strtoimax` is identical to `strtol`, except that it converts a string to a value of type `intmax_t` (the widest signed integer type).

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `INTMAX_MAX` or `INTMAX_MIN`, depending on the number's sign. *27.2*

---

strtok *Search String for Token* `<string.h>`

```
char *strtok(char * restrict s1,
             const char * restrict s2);
```

Searches the string pointed to by `s1` for a "token" consisting of characters not in the string pointed to by `s2`. If a token exists, the character following it is changed to a null character. If `s1` is a null pointer, a search begun by the most recent call of `strtok` is continued; the search begins immediately after the null character at the end of the previous token.

Returns A pointer to the first character of the token. Returns a null pointer if no token could be found. *23.6*

---

strtol *Convert String to Long Integer* `<stdlib.h>`

```
long int strtol(const char * restrict nptr,
                char ** restrict endptr, int base);
```

Skips white-space characters in the string pointed to by `nptr`, then converts subsequent characters into a `long int` value. If `base` is between 2 and 36, it is used as the radix of the number. If `base` is zero, the number is assumed to be decimal unless it begins with `0` (octal) or with `0x` or `0X` (hexadecimal). If `endptr` isn't a null pointer, `strtol` modifies the object pointed to by `endptr` so that it points to the first leftover character. If no `long int` value is found, or if it has the wrong form, `strtol` stores `nptr` in the object pointed to by `endptr`. If the number can't be represented, it stores `ERANGE` in `errno`.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `LONG_MAX` or `LONG_MIN`, depending on the number's sign. *26.2*

---

strtold *Convert String to Long Double (C99)* `<stdlib.h>`

```
long double strtold(const char * restrict nptr,
                    char ** restrict endptr);
```

`strtold` is identical to `strtod`, except that it converts a string to a `long double` value.

Returns The converted number. Returns zero if no conversion could be performed. If the number is too large to represent, returns plus or minus `HUGE_VALL`, depending on the number's sign. If the number is too small to represent, returns a value whose magnitude is no greater than the smallest normalized positive `long double`. *26.2*

`strtoll` *Convert String to Long Long Integer (C99)* `<stdlib.h>`

```
long long int strtoll(const char * restrict nptr,
                      char ** restrict endptr,
                      int base);
```

`strtoll` is identical to `strtol`, except that it converts a string to a `long long int` value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `LLONG_MAX` or `LLONG_MIN`, depending on the number's sign. *26.2*

`strtoul` *Convert String to Unsigned Long Integer* `<stdlib.h>`

```
unsigned long int strtoul(const char * restrict nptr,
                          char ** restrict endptr,
                          int base);
```

`strtoul` is identical to `strtol`, except that it converts a string to an `unsigned long int` value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `ULONG_MAX`. *26.2*

`strtoull` *Convert String to Unsigned Long Long Integer (C99)* `<stdlib.h>`

```
unsigned long long int strtoull(
  const char * restrict nptr,
  char ** restrict endptr, int base);
```

`strtoull` is identical to `strtol`, except that it converts a string to an `unsigned long long int` value.

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `ULLONG_MAX`. *26.2*

`strtoumax` *Convert String to Unsigned Greatest-Width Integer (C99)* `<inttypes.h>`

```
uintmax_t strtoumax(const char * restrict nptr,
                    char ** restrict endptr,
                    int base);
```

`strtoumax` is identical to `strtol`, except that it converts a string to a value of type `uintmax_t` (the widest unsigned integer type).

Returns The converted number. Returns zero if no conversion could be performed. If the number can't be represented, returns `UINTMAX_MAX`. *27.2*

strxfrm *Transform String* <string.h>

```
size_t strxfrm(char * restrict s1,
               const char * restrict s2, size_t n);
```

Transforms the string pointed to by `s2`, placing the first `n` characters of the result—including the null character—in the array pointed to by `s1`. Calling `strcmp` with two transformed strings should produce the same outcome (negative, zero, or positive) as calling `strcoll` with the original strings. If `n` is zero, `s1` is allowed to be a null pointer.

Returns Length of the transformed string. If this value is `n` or more, the contents of the array pointed to by `s1` are indeterminate. *23.6*

swprintf *Wide-Character Formatted String Write (C99)* <wchar.h>

```
int swprintf(wchar_t * restrict s, size_t n,
             const wchar_t * restrict format, ...);
```

Equivalent to `fwprintf`, but stores wide characters in the array pointed to by `s` instead of writing them to a stream. The string pointed to by `format` specifies how subsequent arguments will be displayed. No more than `n` wide characters will be written to the array, including a terminating null wide character.

Returns Number of wide characters stored in the array, not including the null wide character. Returns a negative value if an encoding error occurs or the number of wide characters to be written is `n` or more. *25.5*

swscanf *Wide-Character Formatted String Read (C99)* <wchar.h>

```
int swscanf(const wchar_t * restrict s,
            const wchar_t * restrict format, ...);
```

Wide-character version of `sscanf`. *25.5*

system *Perform Operating-System Command* <stdlib.h>

```
int system(const char *string);
```

Passes the string pointed to by `string` to the operating system's command processor (shell) to be executed. Program termination may occur as a result of executing the command.

Returns If `string` is a null pointer, returns a nonzero value if a command processor is available. If `string` isn't a null pointer, `system` returns an implementation-defined value (if it returns at all). *26.2*

tan *Tangent* <math.h>

```
double tan(double x);
tanf   float tanf(float x);
tanl   long double tanl(long double x);
```

Returns Tangent of `x` (measured in radians). *23.3*

tanh *Hyperbolic Tangent* `<math.h>`

```
double tanh(double x);
float tanhf(float x);
long double tanhl(long double x);
```

Returns Hyperbolic tangent of x. *23.3*

---

tgamma *Gamma Function (C99)* `<math.h>`

```
double tgamma(double x);
float tgammaf(float x);
long double tgammal(long double x);
```

Returns Γ(x), where Γ is the gamma function. A domain error or range error may occur if x is a negative integer or zero. A range error may occur if the magnitude of x is too large or too small. *23.4*

---

time *Current Time* `<time.h>`

```
time_t time(time_t *timer);
```

Returns Current calendar time. Returns `(time_t)(-1)` if the calendar time isn't available. If `timer` isn't a null pointer, also stores the return value in the object pointed to by `timer`. *26.3*

---

tmpfile *Create Temporary File* `<stdio.h>`

```
FILE *tmpfile(void);
```

Creates a temporary file that will automatically be removed when it's closed or the program ends. Opens the file in `"wb+"` mode.

Returns A file pointer to be used when performing subsequent operations on the file. Returns a null pointer if a temporary file can't be created. *22.2*

---

tmpnam *Generate Temporary File Name* `<stdio.h>`

```
char *tmpnam(char *s);
```

Generates a name for a temporary file. If `s` is a null pointer, `tmpnam` stores the file name in a static object. Otherwise, it copies the file name into the character array pointed to by `s`. (The array must be long enough to store `L_tmpnam` characters.)

Returns A pointer to the file name. Returns a null pointer if a file name can't be generated. *22.2*

---

tolower *Convert to Lower Case* `<ctype.h>`

```
int tolower(int c);
```

Returns If `c` is an upper-case letter, returns the corresponding lower-case letter. If `c` isn't an upper-case letter, returns `c` unchanged. *23.5*

---

toupper *Convert to Upper Case* `<ctype.h>`

```
int toupper(int c);
```

Returns If c is a lower-case letter, returns the corresponding upper-case letter. If c isn't a lower-case letter, returns c unchanged. *23.5*

---

towctrans *Transliterate Wide Character (C99)* <wctype.h>

```
wint_t towctrans(wint_t wc, wctrans_t desc);
```

Returns Mapped value of wc using the mapping described by desc. (desc must be a value returned by a call of wctrans; the current setting of the LC_CTYPE category must be the same during both calls.) *25.6*

---

towlower *Convert Wide Character to Lower Case (C99)* <wctype.h>

```
wint_t towlower(wint_t wc);
```

Returns If iswupper(wc) is true, returns a corresponding wide character for which iswlower is true in the current locale, if such a character exists. Otherwise, returns wc unchanged. *25.6*

---

towupper *Convert Wide Character to Upper Case (C99)* <wctype.h>

```
wint_t towupper(wint_t wc);
```

Returns If iswlower(wc) is true, returns a corresponding wide character for which iswupper is true in the current locale, if such a character exists. Otherwise, returns wc unchanged. *25.6*

---

trunc *Truncate to Nearest Integral Value (C99)* <math.h>

```
double trunc(double x);
truncf  float truncf(float x);
truncl  long double truncl(long double x);
```

Returns x rounded to the integer (in floating-point format) nearest to it but no larger in magnitude. *23.4*

---

ungetc *Unread Character* <stdio.h>

```
int ungetc(int c, FILE *stream);
```

Pushes the character c back onto the stream pointed to by stream and clears the stream's end-of-file indicator. The number of characters that can be pushed back by consecutive calls of ungetc varies; only the first call is guaranteed to succeed. Calling a file positioning function (fseek, fsetpos, or rewind) causes the pushed-back character(s) to be lost.

Returns c (the pushed-back character). Returns EOF if an attempt is made to push back EOF or to push back too many characters without a read or file positioning operation. *22.4*

---

ungetwc *Unread Wide Character (C99)* <wchar.h>

```
wint_t ungetwc(wint_t c, FILE *stream);
```

Wide-character version of ungetc. *25.5*

va_arg *Fetch Argument from Variable Argument List* `<stdarg.h>`

type `va_arg(va_list ap, type);` *macro*

Fetches an argument in the variable argument list associated with `ap`, then modifies `ap` so that the next use of `va_arg` fetches the following argument. `ap` must have been initialized by `va_start` (or `va_copy` in C99) prior to the first use of `va_arg`.

Returns Value of the argument, assuming that its type (after the default argument promotions have been applied) is compatible with *type*. *26.1*

---

va_copy *Copy Variable Argument List (C99)* `<stdarg.h>`

`void va_copy(va_list dest, va_list src);` *macro*

Copies `src` into `dest`. The value of `dest` will be the same as if `va_start` had been applied to `dest` followed by the same sequence of `va_arg` applications that was used to reach the present state of `src`. *26.1*

---

va_end *End Processing of Variable Argument List* `<stdarg.h>`

`void va_end(va_list ap);` *macro*

Ends the processing of the variable argument list associated with `ap`. *26.1*

---

va_start *Start Processing of Variable Argument List* `<stdarg.h>`

`void va_start(va_list ap, parmN);` *macro*

Must be invoked before accessing a variable argument list. Initializes `ap` for later use by `va_arg` and `va_end`. *parmN* is the name of the last ordinary parameter (the one followed by `, ...`). *26.1*

---

vfprintf *Formatted File Write Using Variable Argument List* `<stdio.h>`

```
int vfprintf(FILE * restrict stream,
             const char * restrict format,
             va_list arg);
```

Equivalent to `fprintf` with the variable argument list replaced by `arg`.

Returns Number of characters written. Returns a negative value if an error occurs. *26.1*

---

vfscanf *Formatted File Read Using Variable Argument List (C99)* `<stdio.h>`

```
int vfscanf(FILE * restrict stream,
            const char * restrict format,
            va_list arg);
```

Equivalent to `fscanf` with the variable argument list replaced by `arg`.

Returns Number of input items successfully read and stored. Returns `EOF` if an input failure occurs before any items can be read. *26.1*

---

vfwprintf *Wide-Character Formatted File Write Using Variable Argument List (C99)* `<wchar.h>`

```
int vfwprintf(FILE * restrict stream,
              const wchar_t * restrict format,
              va_list arg);
```

Wide-character version of `vfprintf`. *25.5*

`vfwscanf` *Wide-Character Formatted File Read Using Variable Argument List (C99)* `<wchar.h>`

```
int vfwscanf(FILE * restrict stream,
             const wchar_t * restrict format,
             va_list arg);
```

Wide-character version of `vfscanf`. *25.5*

`vprintf` *Formatted Write Using Variable Argument List* `<stdio.h>`

```
int vprintf(const char * restrict format, va_list arg);
```

Equivalent to `printf` with the variable argument list replaced by `arg`.

Returns Number of characters written. Returns a negative value if an error occurs. *26.1*

`vscanf` *Formatted Read Using Variable Argument List (C99)* `<stdio.h>`

```
int vscanf(const char * restrict format, va_list arg);
```

Equivalent to `scanf` with the variable argument list replaced by `arg`.

Returns Number of input items successfully read and stored. Returns `EOF` if an input failure occurs before any items can be read. *26.1*

`vsnprintf` *Bounded Formatted String Write Using Variable Argument List (C99)* `<stdio.h>`

```
int vsnprintf(char * restrict s, size_t n,
              const char * restrict format,
              va_list arg);
```

Equivalent to `snprintf` with the variable argument list replaced by `arg`.

Returns Number of characters that would have been stored in the array pointed to by `s` (not including the null character) had there been no length restriction. Returns a negative value if an encoding error occurs. *26.1*

`vsprintf` *Formatted String Write Using Variable Argument List* `<stdio.h>`

```
int vsprintf(char * restrict s,
             const char * restrict format,
             va_list arg);
```

Equivalent to `sprintf` with the variable argument list replaced by `arg`.

Returns Number of characters stored in the array pointed to by `s`, not including the null character. In C99, returns a negative value if an encoding error occurs. *26.1*

vsscanf *Formatted String Read Using Variable Argument List (C99)* <stdio.h>

```
int vsscanf(const char * restrict s,
            const char * restrict format,
            va_list arg);
```

Equivalent to `sscanf` with the variable argument list replaced by `arg`.

Returns Number of input items successfully read and stored. Returns `EOF` if an input failure occurs before any items can be read. *26.1*

---

vswprintf *Wide-Character Formatted String Write Using Variable Argument List (C99)* <wchar.h>

```
int vswprintf(wchar_t * restrict s, size_t n,
              const wchar_t * restrict format,
              va_list arg);
```

Equivalent to `swprintf` with the variable argument list replaced by `arg`.

Returns Number of wide characters stored in the array pointed to by `s`, not including the null wide character. Returns a negative value if an encoding error occurs or the number of wide characters to be written is `n` or more. *25.5*

---

vswscanf *Wide-Character Formatted String Read Using Variable Argument List (C99)* <wchar.h>

```
int vswscanf(const wchar_t * restrict s,
             const wchar_t * restrict format,
             va_list arg);
```

Wide-character version of `vsscanf`. *25.5*

---

vwprintf *Wide-Character Formatted Write Using Variable Argument List (C99)* <wchar.h>

```
int vwprintf(const wchar_t * restrict format,
             va_list arg);
```

Wide-character version of `vprintf`. *25.5*

---

vwscanf *Wide-Character Formatted Read Using Variable Argument List (C99)* <wchar.h>

```
int vwscanf(const wchar_t * restrict format,
            va_list arg);
```

Wide-character version of `vscanf`. *25.5*

---

wcrtomb *Convert Wide Character to Multibyte Character – Restartable (C99)* <wchar.h>

```
size_t wcrtomb(char * restrict s, wchar_t wc,
               mbstate_t * restrict ps);
```

If `s` is a null pointer, a call of `wcrtomb` is equivalent to

```
wcrtomb(buf, L'\0', ps)
```

where `buf` is an internal buffer. Otherwise, `wcrtomb` converts `wc` from a wide character into a multibyte character (possibly including shift sequences), which it stores in the array pointed to by `s`. The value of `ps` should be a pointer to an object of type `mbstate_t` that contains the current conversion state. If `ps` is a null pointer, `wcrtomb` uses an internal object to store the conversion state. If `wc` is a null wide character, `wcrtomb` stores a null byte, preceded by a shift sequence if necessary to restore the initial shift state, and the `mbstate_t` object used during the call is left in the initial conversion state.

Returns Number of bytes stored in the array, including shift sequences. If `wc` isn't a valid wide character, returns `(size_t) (-1)` and stores `EILSEQ` in `errno`. *25.5*

`wcscat` *Wide-String Concatenation (C99)* `<wchar.h>`

```
wchar_t *wcscat(wchar_t * restrict s1,
                const wchar_t * restrict s2);
```

Wide-character version of `strcat`. *25.5*

`wcschr` *Search Wide String for Character (C99)* `<wchar.h>`

```
wchar_t *wcschr(const wchar_t *s, wchar_t c);
```

Wide-character version of `strchr`. *25.5*

`wcscmp` *Wide-String Comparison (C99)* `<wchar.h>`

```
int wcscmp(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strcmp`. *25.5*

`wcscoll` *Wide-String Comparison Using Locale-Specific Collating Sequence (C99)* `<wchar.h>`

```
int wcscoll(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strcoll`. *25.5*

`wcscpy` *Wide-String Copy (C99)* `<wchar.h>`

```
wchar_t *wcscpy(wchar_t * restrict s1,
                const wchar_t * restrict s2);
```

Wide-character version of `strcpy`. *25.5*

`wcscspn` *Search Wide String for Initial Span of Characters Not in Set (C99)* `<wchar.h>`

```
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strcspn`. *25.5*

`wcsftime` *Write Formatted Date and Time to Wide String (C99)* `<wchar.h>`

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize,
                const wchar_t * restrict format,
                const struct tm * restrict timeptr);
```

Wide-character version of `strftime`. *25.5*

wcslen *Wide-String Length (C99)* <wchar.h>

```
size_t wcslen(const wchar_t *s);
```

Wide-character version of `strlen`. *25.5*

wcsncat *Bounded Wide-String Concatenation (C99)* <wchar.h>

```
wchar_t *wcsncat(wchar_t * restrict s1,
                 const wchar_t * restrict s2,
                 size_t n);
```

Wide-character version of `strncat`. *25.5*

wcsncmp *Bounded Wide-String Comparison (C99)* <wchar.h>

```
int wcsncmp(const wchar_t *s1, const wchar_t *s2,
            size_t n);
```

Wide-character version of `strncmp`. *25.5*

wcsncpy *Bounded Wide-String Copy (C99)* <wchar.h>

```
wchar_t *wcsncpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2,
                 size_t n);
```

Wide-character version of `strncpy`. *25.5*

wcspbrk *Search Wide String for One of a Set of Characters (C99)* <wchar.h>

```
wchar_t *wcspbrk(const wchar_t *s1,
                 const wchar_t *s2);
```

Wide-character version of `strpbrk`. *25.5*

wcsrchr *Search Wide String in Reverse for Character (C99)* <wchar.h>

```
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
```

Wide-character version of `strrchr`. *25.5*

wcsrtombs *Convert Wide String to Multibyte String – Restartable (C99)* <wchar.h>

```
size_t wcsrtombs(char * restrict dst,
                 const wchar_t ** restrict src,
                 size_t len,
                 mbstate_t * restrict ps);
```

Converts a sequence of wide characters from the array indirectly pointed to by `src` into a sequence of corresponding multibyte characters that begins in the conversion state described by the object pointed to by `ps`. If `ps` is a null pointer, `wcsrtombs` uses an internal object to store the conversion state. If `dst` isn't a null pointer, the converted characters are then stored in the array pointed to by `dst`. Conversion continues up to and including a terminating null wide character, which is also stored. Conversion stops earlier if a wide character is reached that doesn't correspond to a valid multibyte character or—if `dst` isn't a null pointer—

when the next multibyte character would exceed the limit of `len` total bytes to be stored in the array pointed to by `dst`. If `dst` isn't a null pointer, the object pointed to by `src` is assigned either a null pointer (if a terminating null wide character was reached) or the address just past the last wide character converted (if any). If the conversion ends at a null wide character, the resulting state is the initial conversion state.

Returns Number of bytes in the resulting multibyte character sequence, not including any terminating null character. Returns `(size_t)(-1)` and stores `EILSEQ` in `errno` if a wide character is encountered that doesn't correspond to a valid multibyte character. *25.5*

`wcsspn` *Search Wide String for Initial Span of Characters in Set (C99)* `<wchar.h>`

```
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strspn`. *25.5*

`wcsstr` *Search Wide String for Substring (C99)* `<wchar.h>`

```
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
```

Wide-character version of `strstr`. *25.5*

`wcstod` *Convert Wide String to Double (C99)* `<wchar.h>`

```
double wcstod(const wchar_t * restrict nptr,
              wchar_t ** restrict endptr);
```

Wide-character version of `strtod`. *25.5*

`wcstof` *Convert Wide String to Float (C99)* `<wchar.h>`

```
float wcstof(const wchar_t * restrict nptr,
             wchar_t ** restrict endptr);
```

Wide-character version of `strtof`. *25.5*

`wcstoimax` *Convert Wide String to Greatest-Width Integer (C99)* `<inttypes.h>`

```
intmax_t wcstoimax(const wchar_t * restrict nptr,
                   wchar_t ** restrict endptr,
                   int base);
```

Wide-character version of `strtoimax`. *27.2*

`wcstok` *Search Wide String for Token (C99)* `<wchar.h>`

```
wchar_t *wcstok(wchar_t * restrict s1,
                const wchar_t * restrict s2,
                wchar_t ** restrict ptr);
```

Searches the wide string pointed to by `s1` for a "token" consisting of wide characters not in the wide string pointed to by `s2`. If a token exists, the character following it is changed to a null wide character. If `s1` is a null pointer, a search begun by a previous call of `wcstok` is continued; the search begins immediately after the null wide character at the end of the previous token. `ptr` points to an object of

type `wchar_t *` that `wcstok` modifies to keep track of its progress. If `s1` is a null pointer, this object must be the same one used in a previous call of `wcstok`; it determines which wide string is to be searched and where the search is to begin.

Returns A pointer to the first wide character of the token. Returns a null pointer if no token could be found. *25.5*

`wcstol` *Convert Wide String to Long Integer (C99)* `<wchar.h>`

```
long int wcstol(const wchar_t * restrict nptr,
                wchar_t ** restrict endptr, int base);
```

Wide-character version of `strtol`. *25.5*

`wcstold` *Convert Wide String to Long Double (C99)* `<wchar.h>`

```
long double wcstold(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr);
```

Wide-character version of `strtold`. *25.5*

`wcstoll` *Convert Wide String to Long Long Integer (C99)* `<wchar.h>`

```
long long int wcstoll(const wchar_t * restrict nptr,
                      wchar_t ** restrict endptr,
                      int base);
```

Wide-character version of `strtoll`. *25.5*

`wcstombs` *Convert Wide String to Multibyte String* `<stdlib.h>`

```
size_t wcstombs(char * restrict s,
                const wchar_t * restrict pwcs,
                size_t n);
```

Converts a sequence of wide characters into corresponding multibyte characters. `pwcs` points to an array containing the wide characters. The multibyte characters are stored in the array pointed to by `s`. Conversion ends if a null character is stored or if storing a multibyte character would exceed the limit of `n` bytes.

Returns Number of bytes stored, not including the terminating null character, if any. Returns `(size_t)(-1)` if a wide character is encountered that doesn't correspond to a valid multibyte character. *25.2*

`wcstoul` *Convert Wide String to Unsigned Long Integer (C99)* `<wchar.h>`

```
unsigned long int wcstoul(
  const wchar_t * restrict nptr,
  wchar_t ** restrict endptr, int base);
```

Wide-character version of `strtoul`. *25.5*

`wcstoull` *Convert Wide String to Unsigned Long Long Integer (C99)* `<wchar.h>`

```
unsigned long long int wcstoull(
  const wchar_t * restrict nptr,
  wchar_t ** restrict endptr, int base);
```

Wide-character version of `strtoull`. *25.5*

wcstoumax *Convert Wide String to Unsigned Greatest-Width Integer (C99)* `<inttypes.h>`

```
uintmax_t wcstoumax(const wchar_t * restrict nptr,
                    wchar_t ** restrict endptr,
                    int base);
```

Wide-character version of `strtoumax`. *27.2*

wcsxfrm *Transform Wide String (C99)* `<wchar.h>`

```
size_t wcsxfrm(wchar_t * restrict s1,
               const wchar_t * restrict s2, size_t n);
```

Wide-character version of `strxfrm`. *25.5*

wctob *Convert Wide Character to Byte (C99)* `<wchar.h>`

```
int wctob(wint_t c);
```

Returns Single-byte representation of `c` as an `unsigned char` converted to `int`. Returns `EOF` if `c` doesn't correspond to one multibyte character in the initial shift state. *25.5*

wctomb *Convert Wide Character to Multibyte Character* `<stdlib.h>`

```
int wctomb(char *s, wchar_t wc);
```

Converts the wide character stored in `wc` into a multibyte character. If `s` isn't a null pointer, stores the result in the array that `s` points to.

Returns If `s` is a null pointer, returns a nonzero or zero value, depending on whether or not multibyte characters have state-dependent encodings. Otherwise, returns the number of bytes in the multibyte character that corresponds to `wc`; returns –1 if `wc` doesn't correspond to a valid multibyte character. *25.2*

wctrans *Define Wide-Character Mapping (C99)* `<wctype.h>`

```
wctrans_t wctrans(const char *property);
```

Returns If `property` identifies a valid mapping of wide characters according to the `LC_CTYPE` category of the current locale, returns a nonzero value that can be used as the second argument to the `towctrans` function; otherwise, returns zero. *25.6*

wctype *Define Wide-Character Class (C99)* `<wctype.h>`

```
wctype_t wctype(const char *property);
```

Returns If `property` identifies a valid class of wide characters according to the `LC_CTYPE` category of the current locale, returns a nonzero value that can be used as the second argument to the `iswctype` function; otherwise, returns zero. *25.6*

wmemchr *Search Wide-Character Memory Block for Character (C99)* `<wchar.h>`

```
wchar_t *wmemchr(const wchar_t *s, wchar_t c,
                 size_t n);
```

Wide-character version of `memchr`. *25.5*

wmemcmp *Compare Wide-Character Memory Blocks (C99)* <wchar.h>

```
int wmemcmp(const wchar_t * s1, const wchar_t * s2,
            size_t n);
```

Wide-character version of memcmp. *25.5*

wmemcpy *Copy Wide-Character Memory Block (C99)* <wchar.h>

```
wchar_t *wmemcpy(wchar_t * restrict s1,
                 const wchar_t * restrict s2,
                 size_t n);
```

Wide-character version of memcpy. *25.5*

wmemmove *Copy Wide-Character Memory Block (C99)* <wchar.h>

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2,
                  size_t n);
```

Wide-character version of memmove. *25.5*

wmemset *Initialize Wide-Character Memory Block (C99)* <wchar.h>

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

Wide-character version of memset. *25.5*

wprintf *Wide-Character Formatted Write (C99)* <wchar.h>

```
int wprintf(const wchar_t * restrict format, ...);
```

Wide-character version of printf. *25.5*

wscanf *Wide-Character Formatted Read (C99)* <wchar.h>

```
int wscanf(const wchar_t * restrict format, ...);
```

Wide-character version of scanf. *25.5*

APPENDIX E

ASCII Character Set

Decimal	Escape Sequence Oct	Hex	Char	Character						
0	\0	\x00		*nul*	32		64	@	96	`
1	\1	\x01		*soh* (^A)	33	!	65	A	97	a
2	\2	\x02		*stx* (^B)	34	"	66	B	98	b
3	\3	\x03		*etx* (^C)	35	#	67	C	99	c
4	\4	\x04		*eot* (^D)	36	$	68	D	100	d
5	\5	\x05		*enq* (^E)	37	%	69	E	101	e
6	\6	\x06		*ack* (^F)	38	&	70	F	102	f
7	\7	\x07	\a	*bel* (^G)	39	'	71	G	103	g
8	\10	\x08	\b	*bs* (^H)	40	(	72	H	104	h
9	\11	\x09	\t	*ht* (^I)	41	)	73	I	105	i
10	\12	\x0a	\n	*lf* (^J)	42	*	74	J	106	j
11	\13	\x0b	\v	*vt* (^K)	43	+	75	K	107	k
12	\14	\x0c	\f	*ff* (^L)	44	,	76	L	108	l
13	\15	\x0d	\r	*cr* (^M)	45	-	77	M	109	m
14	\16	\x0e		*so* (^N)	46	.	78	N	110	n
15	\17	\x0f		*si* (^O)	47	/	79	O	111	o
16	\20	\x10		*dle* (^P)	48	0	80	P	112	p
17	\21	\x11		*dc1* (^Q)	49	1	81	Q	113	q
18	\22	\x12		*dc2* (^R)	50	2	82	R	114	r
19	\23	\x13		*dc3* (^S)	51	3	83	S	115	s
20	\24	\x14		*dc4* (^T)	52	4	84	T	116	t
21	\25	\x15		*nak* (^U)	53	5	85	U	117	u
22	\26	\x16		*syn* (^V)	54	6	86	V	118	v
23	\27	\x17		*etb* (^W)	55	7	87	W	119	w
24	\30	\x18		*can* (^X)	56	8	88	X	120	x
25	\31	\x19		*em* (^Y)	57	9	89	Y	121	y
26	\32	\x1a		*sub* (^Z)	58	:	90	Z	122	z
27	\33	\x1b		*esc*	59	;	91	[	123	{
28	\34	\x1c		*fs*	60	<	92	\	124	|
29	\35	\x1d		*gs*	61	=	93	]	125	}
30	\36	\x1e		*rs*	62	>	94	^	126	~
31	\37	\x1f		*us*	63	?	95	_	127	*del*

BIBLIOGRAPHY

The best book on programming for the layman is "Alice in Wonderland"; but that's because it's the best book on anything for the layman.

C Programming

Feuer, A. R., *The C Puzzle Book*, Revised Printing, Addison-Wesley, Reading, Mass., 1999. Contains numerous "puzzles"—small C programs whose output the reader is asked to predict. The book shows the correct output of each program and provides a detailed explanation of how it works. Good for testing your C knowledge and reviewing the fine points of the language.

Harbison, S. P., III, and G. L. Steele, Jr., *C: A Reference Manual*, Fifth Edition, Prentice-Hall, Upper Saddle River, N.J., 2002. The ultimate C reference—essential reading for the would-be C expert. Covers both C89 and C99 in considerable detail, with frequent discussions of implementation differences found in C compilers. Not a tutorial—assumes that the reader is already well versed in C.

Kernighan, B. W., and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, Englewood Cliffs, N.J., 1988. The original C book, affectionately known as K&R or simply "the White Book." Includes both a tutorial and a complete C reference manual. The second edition reflects the changes made in C89.

Koenig, A., *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass., 1989. An excellent compendium of common (and some not-so-common) C pitfalls. Forewarned is forearmed.

Plauger, P. J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J., 1992. Not only explains all aspects of the C89 standard library, but provides complete source code! There's no better way to learn the library than to study this book. Even if your interest in the library is minimal, the book is worth getting just for the opportunity to study C code written by a master.

Ritchie, D. M., The development of the C programming language, in *History of Programming Languages II,* edited by T. J. Bergin, Jr., and R. G. Gibson, Jr., Addison-Wesley, Reading, Mass., 1996, pages 671–687. A brief history of C written by the language's designer for the Second ACM SIGPLAN History of Programming Languages Conference, which was held in 1993. The article is followed by transcripts of Ritchie's presentation at the conference and the question-and-answer session with the audience.

Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan, UNIX time-sharing system: the C programming language, *Bell System Technical Journal* 57, 6 (July–August 1978), 1991–2019. A famous article that discusses the origins of C and describes the language as it looked in 1978.

Rosler, L., The UNIX system: the evolution of C—past and future, *AT&T Bell Laboratories Technical Journal* 63, 8 (October 1984), 1685–1699. Traces the evolution of C from 1978 to 1984 and beyond.

Summit, S., *C Programming FAQs: Frequently Asked Questions*, Addison-Wesley, Reading, Mass., 1996. An expanded version of the FAQ list that has appeared for years in the Usenet *comp.lang.c* newsgroup.

van der Linden, P., *Expert C Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1994. Written by one of the C wizards at Sun Microsystems, this book manages to entertain and inform in equal amounts. With its profusion of anecdotes and jokes, it makes learning the fine points of C seem almost fun.

UNIX Programming

Rochkind, M. J., *Advanced UNIX Programming,* Second Edition, Addison-Wesley, Boston, Mass., 2004. Covers UNIX system calls in considerable detail. This book, along with the one by Stevens and Rago, is a must-have for C programmers who use the UNIX operating system or one of its variants.

Stevens, W. R., and S. A. Rago, *Advanced Programming in the UNIX Environment,* Second Edition, Addison-Wesley, Upper Saddle River, N.J., 2005. An excellent follow-up to this book for programmers working under the UNIX operating system. Focuses on using UNIX system calls, including standard C library functions as well as functions that are specific to UNIX.

Programming in General

Bentley, J., *Programming Pearls*, Second Edition, Addison-Wesley, Reading, Mass., 2000. This updated version of Bentley's classic programming book emphasizes writing efficient programs, but touches on other topics that are crucial for the professional programmer. The author's light touch makes the book as enjoyable to read as it is informative.

Kernighan, B. W., and R. Pike, *The Practice of Programming,* Addison-Wesley, Reading, Mass., 1999. Read this book for advice on programming style, choosing the right algorithm, testing and debugging, and writing portable programs. Examples are drawn from C, C++, and Java.

McConnell, S., *Code Complete,* Second Edition, Microsoft Press, Redmond, Wash., 2004. Tries to bridge the gap between programming theory and practice by providing down-to-earth coding advice based on proven research. Includes plenty of examples in a variety of programming languages. Highly recommended.

Raymond, E. S., ed., *The New Hacker's Dictionary*, Third Edition, MIT Press, Cambridge, Mass., 1996. Explains much of the jargon that programmers use, and it's great fun to read as well.

Web Resources

ANSI eStandards Store (*webstore.ansi.org*). The C99 standard (ISO/IEC 9899:1999) can be purchased at this site. Each set of corrections to the standard (known as a Technical Corrigendum) can be downloaded for free.

comp.lang.c Frequently Asked Questions (*c-faq.com*). Steve Summit's FAQ list for the *comp.lang.c* newsgroup is a must-read for any C programmer.

Dinkumware (*www.dinkumware.com*). Dinkumware is owned by P. J. Plauger, the acknowledged master of the C and C++ standard libraries. The web site includes a handy C99 library reference, among other things.

Google Groups (*groups.google.com*). One of the best ways to find answers to programming questions is to search the Usenet newsgroups using the Google Groups search engine. If you have a question, it's likely that someone else has already asked the question on a newsgroup and the answer has been posted. Groups of particular interest to C programmers include *alt.comp.lang.learn.c-c++* (for C and C++ beginners), *comp.lang.c* (the primary C language group), and *comp.std.c* (devoted to discussion of the C standard).

International Obfuscated C Code Contest (*www.ioccc.org*). Home of an annual contest in which participants vie to see who can write the most obscure C programs.

ISO/IEC JTC1/SC22/WG14 (*www.open-std.org/jtc1/sc22/wg14/*). The official web site of WG14, the international working group that created the C99 standard and is responsible for updating it. Of particular interest among the many documents available at the site is the rationale for C99, which explains the reasons for the changes made in the standard.

Lysator (*www.lysator.liu.se/c/*). A collection of links to C-related web sites maintained by Lysator, an academic computer society located at Sweden's Linköping University.