

---

# S

**save calling environment function** *See* `setjmp`.

**scalar** An object having a simple type like `int`, `char`, `float`, or pointer. Nonscalar object types are unions and aggregates (structures and arrays).

**SC22** The abbreviated name for a subcommittee within ISO/IEC JTC 1, that deals with standards for various programming languages and environments.

**SC22/WG14** Working Group 14 (C Language) within ISO/IEC JTC 1 subcommittee SC22. *See also* WG14.

**SC22/WG15** Working Group 15 (POSIX) within ISO/IEC JTC 1 subcommittee SC22. *See also* WG15.

**SC22/WG21** Working Group 21 (C++ Language) within ISO/IEC JTC 1 subcommittee SC22. *See also* WG21.

`scalbln[f|l]C99` A function that computes  $x \times \text{FLT\_RADIX}^n$  efficiently.

```
#include <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);
```

A range error may occur. *See also* `scalbn[f|l]`.

`scalbn[f|l]C99` A function that computes  $x \times \text{FLT\_RADIX}^n$  efficiently.

```
#include <math.h>
double scalbn(double x, int n);
float scalbnf(float x, int n);
long double scalbnl(long double x, int n);
```

A range error may occur. *See also* `scalbln[f|l]`.

**scanf** A function that reads formatted input from `stdin` as specified by `format`.

```
#include <stdio.h>
int scanf(const char * restrict format, ...);
```

Characters in `format` other than white space and those specifying a particular conversion specifier are expected to appear in the input as

written. %% matches a single %. An arbitrary amount of consecutive white space in the format causes white space on input to be skipped. When `scanf` completes its conversion, any white space left in the input buffer immediately following the last converted input field is left there.

The general format of a `scanf` conversion specifier is

`%[*][width][modifier]specifier`

where the values for *modifier* and *specifier* are as follows:

scanf Modifiers	
<i>Symbol</i>	<i>Meaning</i>
<code>h</code>	Pointer to <code>short int</code>
<code>hh</code> <sup>C99</sup>	Pointer to <code>char</code>
<code>j</code> <sup>C99</sup>	Pointer to <code>[u]intmax_t</code>
<code>l</code>	Pointer to <code>long int</code> or <code>double</code>
<code>ll</code> <sup>C99</sup>	Pointer to <code>long long int</code>
<code>L</code>	Pointer to <code>long double</code>
<code>t</code> <sup>C99</sup>	Pointer to <code>ptrdiff_t</code>
<code>z</code> <sup>C99</sup>	Pointer to <code>size_t</code>

scanf Specifiers	
<i>Symbol</i>	<i>Meaning</i>
<code>a</code> <sup>C99</sup>	Floating-point number
<code>c</code>	Character(s)
<code>d</code>	Signed decimal
<code>e</code>	Floating-point number
<code>f</code>	Floating-point number
<code>g</code>	Floating-point number
<code>i</code>	Signed decimal
<code>n</code>	Stores character count in <code>int</code>
<code>o</code>	Octal
<code>p</code>	Pointer to <code>void</code>
<code>s</code>	String
<code>u</code>	Decimal
<code>x</code>	Hexadecimal
<code>[...]</code>	String with pattern match
<code>%</code>	Read <code>%</code> character

`scanf` returns the number of input items assigned. This does not include items skipped using the assignment-suppression character (`*`) or input fields written that corresponded to `n` conversion characters. If an error occurred, EOF is returned.

All arguments must be passed to `scanf` by address.

A call to `scanf` is equivalent to a call to `fscanf` using a stream of `stdin`.  
*See also* future library directions; `wscanf`.

`SCHAR_MAX`<sup>C89</sup> A macro, defined in `limits.h`, that designates the maximum value for an object of type `signed char`. It must be at least 127 (8 bits). This macro expands to an integer constant expression suitable for use with a `#if` directive.

`SCHAR_MIN`<sup>C89</sup> A macro, defined in `limits.h`, that designates the minimum value for an object of type `signed char`. It must be at least  $-127$  (8 bits). This macro expands to an integer constant expression suitable for use with a `#if` directive.

`SCNdN`<sup>C99</sup> A macro, defined in `limits.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `intN_t`. For example

```
int32_t x;
int y;
scanf("%" SCNd32 " %d", &x, &y);
```

`SCNdLEASTN`<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `int_leastN_t`.

`SCNdFASTN`<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `int_fastN_t`.

`SCNdMAX`<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `intmax_t`.

`SCNdPTR`<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `intptr_t`.

`SCNiN`<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `intN_t`. For example

```
int32_t x;
int y;
scanf("%" SCNi32 " %d", &x, &y);
```

SCNiLEAST<sup>C99</sup>*N* A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `int_leastN_t`.

SCNiFAST<sup>C99</sup>*N* A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `int_fastN_t`.

SCNiMAX<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `intmax_t`.

SCNiPTR<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `intptr_t`.

SCNo*N*<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintN_t`. For example

```
uint32_t x;
int y;
scanf("%" SCNo32 " %d", &x, &y);
```

SCNoLEAST<sup>C99</sup>*N* A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uint_leastN_t`.

SCNoFAST<sup>C99</sup>*N* A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uint_fastN_t`.

SCNoMAX<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintmax_t`.

SCNoPTR<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintptr_t`.

SCNu<sup>C99</sup>*N* A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of

the `scanf` and `wscanf` family when converting the type `uintN_t`. For example

```
uint32_t x;
int y;
scanf("%" SCNu32 " %d", &x, &y);
```

**SCNuLEAST*N*<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uint_leastN_t`.

**SCNuFAST*N*<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uint_fastN_t`.

**SCNuMAX<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintmax_t`.

**SCNuPTR<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintptr_t`.

**SCNx*N*<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintN_t`. For example

```
uint32_t x;
int y;
scanf("%" SCNx32 " %d", &x, &y);
```

**SCNxLEAST*N*<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uint_leastN_t`.

**SCNxFAST*N*<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uint_fastN_t`.

**SCNxMAX<sup>C99</sup>** A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintmax_t`.

**SCNxPTR**<sup>C99</sup> A macro, defined in `inttypes.h`, that expands to a character string literal containing a conversion specifier suitable for use with members of the `scanf` and `wscanf` family when converting the type `uintptr_t`.

**scope** That region of a program within which an identifier is declared. The kinds of scope are block, file, function, and function prototype. *See also* linkage; storage duration.

**scope, block** A type of scope in which an identifier is declared inside a block or parameter list of a function definition. This scope ends at the `}` terminating that block (which, in the case of a parameter identifier, is at the end of the function body). All of the identifiers inside the following function `f` have block scope (although `f` itself has file scope):

```
void f(int j)
{
    int i, g(void);
    static int si;

    if (j > 5) {
        double d;
        int i;
    }
}
```

Note that while `g` has block scope, it has external linkage.

**scope, file** A type of scope in which an identifier is declared outside all blocks and parameter lists. This scope ends at the end of that translation unit. All of the identifiers in the following example have file scope:

```
int i;
static int si;
extern double ed;
void f(void);

void g()
{
}
```

**scope, function** The type of scope for user-defined labels. User-defined labels are the only identifiers that have function scope. That is, they are visible from anywhere in the function in which they are defined. As such, label names must be unique within a function.

**scope, prototype**<sup>C89</sup> The type of scope for an identifier declared inside a function prototype. This scope ends at the end of the function declara-

tor. Although identifiers in prototypes are optional, if they exist, they must be unique within that prototype. For example, in

```
int f(int i, double d);
int g(int i, double d);
```

**search functions** The string.h functions `memchr`, `strchr`, `strcspn`, `strpbrk`, `strrchr`, `strspn`, `strstr`, and `strtok` and their wide counterparts.

**SEEK\_CUR** A macro, defined in `stdio.h`, that expands to an integer constant expression that can be used as the third argument to `fseek`. It indicates a position relative to the current position. *See also* `SEEK_END`; `SEEK_SET`.

**SEEK\_END** A macro, defined in `stdio.h`, that expands to an integer constant expression that can be used as the third argument to `fseek`. It indicates a position relative to the end of the file. *See also* `SEEK_CUR`; `SEEK_SET`.

**SEEK\_SET** A macro, defined in `stdio.h`, that expands to an integer constant expression that can be used as the third argument to `fseek`. It indicates a position relative to the start of the file. *See also* `SEEK_CUR`; `SEEK_END`.

**selection statements** *See* `if/else`; `switch`.

**semicolon punctuator** A punctuator used as a statement or declaration terminator. It is also used to separate the three optional expressions in a `for` statement.

The null statement consists solely of a semicolon.

**sequence point**<sup>C89</sup> A point in a program at which all side effects of previous evaluations must be complete and no side effects of subsequent evaluations shall have taken place. There is a sequence point at the end of a full expression as well as at the end of a full declarator. The following operators also have sequence points: `&&` and `||`, after the left operand had been evaluated; `?:`, after the first operand has been evaluated; comma operator, after the left operand has been evaluated; and the function call operator, after all arguments and the function designator have been evaluated, but before the function is actually called.

**setbuf** A function that is equivalent to a specific invocation of `setvbuf`. That is, calling `setvbuf` with `mode` equal to `_IOFBF` and `size` equal to `BUFSIZ`, or with `buf` being `NULL` and `mode` being `_IONBF`, is the same as calling `setbuf`.

```
#include <stdio.h>
void setbuf(FILE * restrict stream, char * restrict buf);
```

`setbuf` returns no value. The responsibility is on the programmer to make sure `stream` points to an open file and that `buf` is either `NULL` or a pointer to a sufficiently large buffer.

**setjmp** A macro, defined in `setjmp.h`, that saves a program's current context (or calling environment) in a user-defined object of type `jmp_buf` so the program can be restored to that context by a subsequent call to `longjmp`.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Since `setjmp` can be a macro, its definition is restricted in certain ways. For example, it cannot be called via a pointer to function. See your library manual for details. When the programmer explicitly calls `setjmp`, it returns a zero value. When `setjmp` returns via an unconditional jump from `longjmp`, it returns a user-defined nonzero value.

Note that `setjmp` really saves its own context, not that of its caller. Therefore, when `longjmp` is called to restore a saved context, control is transferred back into `setjmp`, which then returns to its original caller.

**setjmp.h** A header that contains the type `jmp_buf` and declares the functions `setjmp` and `longjmp`, all of which are used to save and restore a program context. Together, these items provide a nonlocal goto facility—the ability to jump out of one function and into the middle of another, provided the destination function is further up the call hierarchy.

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

<i>Name</i>	<i>Purpose</i>
<code>jmp_buf</code>	<code>setjmp</code> save buffer type
<code>longjmp</code>	Restore a saved environment
<code>setjmp</code>	Save a runtime environment

**setlocale**<sup>C89</sup> A function that allows the program to change either a complete locale or a category of the current locale, or to find the “name” of the current locale.

```
#include <locale.h>
char *setlocale(int category, char *locale);
```

`category` must be one of the standard or implementation-defined `LC_*` macros. `locale` is either the standard `"C"` locale or some other implementation-defined locale. At program startup, the locale is set to `"C"` automatically.

If a pointer to a locale string is given to `setlocale` and that locale is available, a locale string corresponding to the given category is returned. This string can be given back to `setlocale` in future calls. If the requested locale is not available or known, `NULL` is returned and the locale

is not changed. A NULL locale pointer causes the string defining the current locale to be returned.

**setvbuf** A function that permits the type of buffering for a newly opened file to be changed. It also permits users to supply their own buffer for that file.

```
#include <stdio.h>
int setvbuf(FILE * restrict stream, char * restrict buf,
            int mode, size_t size);
```

**setvbuf** must be called before any reading or writing takes place on the newly opened stream. **mode** may be one of the following: **\_IOFBF** (fully buffered), **\_IOLBF** (line buffered), or **\_IONBF** (no buffering). If **buf** is NULL, **setvbuf** uses its own internal buffer; otherwise, it uses that pointed to by **buf**, in which case, **size** should be at least as big as the array pointed to by **buf**. If **mode** equals **\_IOFBF** and **size** equals **BUFSIZ**, **setbuf** can be used instead. The same is true if **buf** is NULL and **mode** is **\_IONBF**.

**setvbuf** returns zero on success and nonzero on failure. A failure could result from an invalid value for **mode**.

**shift state**<sup>C89</sup> The context for a multibyte character set that has a state-dependent encoding. The value of a multibyte character may be encoded in a state-dependent way that involves switching between various shift states. A change in shift state is indicated by one or more characters with special values. When a change in shift state is found, subsequent characters are interpreted according to the current shift state until either the shift state changes again, or the sequence of characters ends. *See also* shift state, initial.

**shift state, initial**<sup>C89</sup> The default shift state in which the implementation starts out looking at multibyte characters. *See also* shift sequence.

**shift sequence**<sup>C89</sup> A sequence of one or more bytes that indicate a change in encoding. When a sequence of multibyte characters is being scanned, the detection of a shift sequence causes the characters following to be interpreted differently until the shift state is changed (possibly by restoring to the original state) or the end of the character sequence is reached. All comments, string literals, character constants, and header names are required to begin and end in their initial shift state. In the initial shift state, single-byte characters have their usual meaning; that is, they do not alter the shift state. A redundant shift sequence is one that is followed immediately by another shift sequence or is one that switches to the (already) current mode.

**Shift-JIS** A scheme commonly used to encode Japanese text in multi-byte characters. *See also* EUC; JIS.

**short** A permitted abbreviation for `short int`.

**short int** An integer type. Standard C requires it to be at least 16 bits. A plain `short int` is signed. `short int` expressions traditionally were widened to `int` when used in expressions and as arguments to functions. However, Standard C allows them to be used without widening, provided the same result is obtained. *See also* conversion, function arguments; integer type.

**SHRT\_MAX**<sup>C89</sup> A macro, defined in `limits.h`, that designates the maximum value for an object of type `short int`. It must be at least 32,767 (16 bits). This macro expands to an integer constant expression suitable for use with a `#if` directive.

**SHRT\_MIN**<sup>C89</sup> A macro, defined in `limits.h`, that designates the minimum value for an object of type `short int`. It must be at least  $-32,767$  (16 bits). This macro expands to an integer constant expression suitable for use with a `#if` directive.

**side effect** The act of accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of these things, is known as a side effect. Essentially, a side effect results in a change in the state of the execution environment. Each of the following expressions contains side effects:

```
++i    j--    x = 4    j *= a    f()
```

In the call to function `f`, we assume the function body itself (or some function it calls) has one or more side effects.

Any statement other than `break`, `continue`, and `goto`, that does not directly or indirectly result in a side effect is a vacuous statement. *See also* sequence point.

**SIG\*** Signal-type macros that can be used as the first argument to `signal`. This argument indicates the particular kind of signal the user wishes to process with the `signal` function. Standard C defines the following signal type macros: `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, and `SIGTERM`. It also reserves names of the form `SIG*` (where `*` begins with an uppercase letter) for use as signal number macros. *See also* future library directions.

**SIG\_\*** Function pointer macros, the following three of which are defined by Standard C: `SIG_DFL`, `SIG_IGN`, and `SIG_ERR`. These macros expand to

distinct constant expressions that have a type compatible with the second argument to, and the return value of, the function `signal`. Other, implementation-defined value macros are permitted so Standard C reserves names of the form `SIG_*` for that use, where `*` begins with an uppercase letter. *See also* future library directions.

**SIGABRT** A signal type macro, defined in `signal.h`, that indicates abnormal termination (such as a call to `abort`). *See also* `assert`.

**SIG\_ATOMIC\_MAX**<sup>C99</sup> A macro, defined in `stdint.h`, that is intended to indicate the maximum value which can be stored in an object of type `sig_atomic_t`. If that type is a signed integer, `SIG_ATOMIC_MAX` shall be no less than 127; if the type is an unsigned integer, `SIG_ATOMIC_MAX` shall be no less than 255.

This macro shall expand to a constant expression suitable for use in `#if` preprocessing directives.

**SIG\_ATOMIC\_MIN**<sup>C99</sup> A macro, defined in `stdint.h`, that is intended to indicate the minimum value that can be stored in an object of type `sig_atomic_t`. If that type is a signed integer, `SIG_ATOMIC_MIN` shall be no greater than -127; if the type is an unsigned integer, `SIG_ATOMIC_MIN` shall be 0.

This macro shall expand to a constant expression suitable for use in `#if` preprocessing directives.

**sig\_atomic\_t**<sup>C89</sup> The integer type, which is guaranteed to be accessed as an atomic entity, even in the presence of signals. That is, when a signal occurs, an object of this type cannot be partially updated—it has either been completely updated or not been updated at all. *See also* `SIG_ATOMIC_MAX`; `SIG_ATOMIC_MIN`.

**SIG\_DFL macro** A macro, defined in `signal.h`, that expands to a constant expression suitable for use as the second argument to the `signal` function. Its value must not be equal to that of any declarable function, nor to that of `SIG_ERR` or `SIG_IGN`. It is used to tell `signal` to use the implementation-defined default handler for the given interrupt type.

**SIG\_ERR macro** A macro, defined in `signal.h`, that expands to a constant expression suitable for use as the second argument to the `signal` function. Its value must not be equal to that of any declarable function, nor to that of `SIG_DFL` or `SIG_IGN`. It is returned by `signal` to indicate an error occurred when an attempt was made to specify handling for a particular interrupt type.

**SIGFPE** A signal type macro, defined in `signal.h`, that indicates an erroneous arithmetic operation, such as zero-divide or an operation resulting in overflow. (Its name comes from “Floating-Point Exception.”)

**SIG\_IGN macro** A macro, defined in `signal.h`, that expands to a constant expression suitable for use as the second argument to the `signal` function. Its value must not be equal to that of any declarable function, nor to that of `SIG_DFL` or `SIG_ERR`. It is used to tell `signal` to ignore the given interrupt type. In some implementations, while you can request certain signals to be ignored, they still will be trapped. For example, if a nonprivileged program requests to ignore all privileged attempts to kill it and this were to be permitted, a security hole would exist.

**SIGILL** A signal type macro, defined in `signal.h`, that indicates an invalid function image; possibly an illegal instruction was detected.

**SIGINT** A signal type macro, defined in `signal.h`, that indicates receipt of an interactive attention signal (such as `CTRL/C` or `CTRL/D`).

**signal** An asynchronous or synchronous event that interrupts a program. Some signals can be trapped or ignored. This is done using the machinery provided in `signal.h`. Standard C defines six specific signal types (named `SIG*`): abnormal termination; an erroneous arithmetic operation, such as zero-divide or an operation resulting in overflow; invalid function image (possibly an illegal instruction was detected); receipt of an interactive attention signal; an invalid access to storage; and a termination request sent to the program.

**signal** A function that is used to indicate the type of action to be taken when a signal of the specified type is encountered. A signal may be ignored, handled by the system in a default manner, or processed by a user-supplied handler.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

`sig` is the signal type to be processed and is typically a macro name of the form `SIG*` (such as `SIGINT`). `func` is the signal-handling method to be used and is typically a macro name of the form `SIG_*` (such as `SIG_IGN`), or the address of a user-written signal handler function.

`signal` returns `SIG_ERR` if it cannot perform the requested operation. Otherwise, it returns the value given to `signal` (as its second argument) in the previous call for that signal number. That is, you can save the current signal-handling context, change it temporarily, and then restore it again using this return value.

The initial state of signal handling at program startup is implementation defined. *See also* `raise`.

**signal handler** A function that is given control when its associated signal type is detected. A signal-handling function must take one argument (of

type `int`) and have `void` return type. A signal handler is registered via the `signal` function. Once a given signal type has been detected, the library behaves as if it calls `signal` with a second argument of `SIG_DFL` for that signal type, before it passes control to the user-written handler. That is, registering a signal handler lasts for only one signal detection. Each time the handler runs, it must reregister itself if you intend to catch further interrupts of that type.

**signal.h** A header that declares the type `sig_atomic_t` and declares several functions (`signal` and `raise`) and macros (`SIG*` and `SIG_*`) that are useful in handling signals. (Signals are often called exceptions or interrupts.)

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

<i>Name</i>	<i>Purpose</i>
<code>raise</code>	Generate a signal synchronously
<code>sig_atomic_t</code>	Atomic operation type
<code>SIG_DFL</code>	Use default handler
<code>SIG_ERR</code>	<code>signal</code> return error value
<code>SIG_IGN</code>	Ignore a given signal type
<code>SIGABRT</code>	Abnormal termination
<code>SIGFPE</code>	Erroneous arithmetic operation
<code>SIGILL</code>	Invalid function image
<code>SIGINT</code>	Interactive attention signal
<code>signal</code>	Establish a signal handler
<code>SIGSEGV</code>	Invalid access to storage
<code>SIGTERM</code>	Termination request

*See also* future library directions.

**signed**<sup>C89</sup> A keyword used with integer data types to indicate they are signed. It may be applied to `char`, `short int`, `int`, `long int`, and `long long int`. It allows signed arithmetic to be performed. When used on its own, it implies signed `int`. The use of `signed` with `short`, `int`, `long`, and `long long` is redundant because these types are signed anyway. C89 invented this modifier to permit explicitly signed versions of `char` and `int` bit-fields. Prior to that, you had unsigned and plain `chars` only, and it was implementation defined as to the signedness of a plain `char` and a plain `int` bit-field. *See also* integer type.

**signed char** A `char` type that is explicitly signed. A plain `char` written without the modifier `signed` or `unsigned` might be signed or unsigned—that is implementation defined. *See also* `signed`.

**signed char type conversion** *See* conversion, integer type; unsigned preserving rule; value preserving rule.

**signed integer types** The types `signed char`, `signed short`, `short`, `int`, `signed int`, `long`, `signed long`, `long long`, and `signed long long`.  
*See also* bit-field, plain int; char, plain.

**significand part, floating constant** That part of a floating constant preceding the optional exponent and suffix parts.

**SIGSEGV** A signal type macro, defined in `signal.h`, that indicates an invalid access to storage (segment violation).

**SIGTERM** A signal type macro, defined in `signal.h`, that indicates that a termination request was sent to the program.

**simple assignment operator** *See* assignment operator, simple.

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

```
#include <math.h>
double sin(double x);
float sinf(float x);
long double sinl(long double x);
```

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

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

**single quote escape sequence** A single quote character `'` can be included in a character constant only in its escape sequence form `\'`. It can occur in a string literal in either form, however. That is, either as `"'"` or `"\'"`.

**sinh[f|l]** A function that computes the hyperbolic sine of its argument `x`.

```
#include <math.h>
double sinh(double x);
float sinhf(float x);
long double sinhl(long double x);
```

If the magnitude of the argument is too large, a range error occurs.

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

**SIZE\_MAX<sup>C99</sup>** A macro, defined in `stdint.h`, that indicates the maximum value of the type `size_t`. It expands to an integer constant expression suitable for use with a `#if` directive.

**sizeof** A keyword used to represent an operator. It returns the size in bytes of its operand. It can be used with expressions or types, other than function and `void` type. It cannot be used with bit-fields. It does not evaluate its operand except to determine its type. The type of the result is `size_t`. `sizeof` can be used in either of the following ways:

```
sizeof( type )
sizeof expression
```

In the second case, the expression is often inside (redundant) grouping parentheses. Prior to C99, a `sizeof` expression was evaluated translation-time. However, with the addition of variable-length arrays in C99, `sizeof(vla_designator)` is actually computed at run time.

**size\_t**<sup>C89</sup> The type of the result obtained from the `sizeof` operator. This type is an unsigned integer type. Many standard library functions (`strlen` and `calloc`, for example) expect arguments and/or return values of this type. Prior to C99, assuming an object's size could be represented in an `unsigned long`, the value of an expression of type `sizeof` could be displayed as follows:

```
printf("%lu", (unsigned long)sizeof(int));
```

however, with the addition of the `long long int` type in C99, as well as the possibility of implementations' supporting even larger extended integer types, C99 added the conversion specifier `%z` to the `printf` function family to allow values of type `size_t` to be printed in a portable fashion. `size_t` is defined in each of the following headers: `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, `time.h`, and `wchar.h`. *See also* `SIZE_MAX`.

**snprintf**<sup>C99</sup> A function that behaves like `fprintf`, except that it writes its output to an array instead of a stream.

```
#include <stdio.h>
int snprintf(char * restrict s, size_t n,
             const char * restrict format, ...);
```

If `n` is zero, no output is produced and `s` can be null. Otherwise, the first `n - 1` characters are written to the array pointed to by `s`, and a null character is appended. The behavior is undefined if the source and destination arrays overlap.

**sort function** *See* `qsort`.

**source file inclusion** *See* `#include`.

**Spirit of C** A guiding principal used by X3J11 in determining just what changes and additions it should consider in producing C89. According to the Standard C Rationale document:

“There are many facets of the spirit of C, but the essence is a community sentiment of the underlying principles upon which the C language is based. Some of the facets of the spirit of C can be summarized in phrases like:

- Trust the programmer.
- Don’t prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Provide only one way to do an operation.

“One of the goals of the Committee was to avoid interfering with the ability of translators to generate compact, efficient code. In several cases, the Committee has introduced features to improve the possible efficiency of the generated code; for instance, floating point operations may be performed in single precision if both operands are `float` rather than `double`.”

**sprintf** A function that writes formatted output to the string pointed to by `s` in a format specified by `format`.

```
#include <stdio.h>
int sprintf(char * restrict s,
            const char * restrict format, ...);
```

The value returned is the number of characters written to the string. It does not include the null character that is automatically appended. On error, a negative value is returned just as for `scanf` and `fprintf`. For details of `format` see `printf`. See also `wsprintf`.

**sqrt[f|l]** A function that computes the nonnegative square root of its argument `x`.

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
long double sqrtl(long double x);
```

If the argument is negative, a domain error occurs. Note, however, that some implementations (for example, those based on the IEEE-754 standard) support signed zeros including a negative floating zero, in which case `sqrt(-0)` may be required to return `-0` with no domain error being produced. Standard C requires a domain error to be generated, but

permits an implementation-defined value to be returned (thus permitting a result of -0).

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

**srand** A function that uses its argument `seed` as a seed for a new sequence of pseudo random numbers to be returned by subsequent calls to `rand`.

```
#include <stdlib.h>
void srand(unsigned int seed);
```

If `srand` has never been called, `rand` behaves as if `srand` had been called with a seed of 1. Identical seeds generate identical pseudo random number sequences.

**sscanf** A function that reads formatted input from the string specified by `s` using a format specified by `format`.

```
#include <stdio.h>
int sscanf(const char * restrict s,
           const char * restrict format, ...);
```

`sscanf` returns the number of input items assigned. If an error occurred, EOF is returned.

All arguments must be passed to `sscanf` by address. For a discussion of `format` see `scanf`. See also `wsscanf`.

**stack** An area of memory in which automatic objects and function argument lists often are stored depending on the machine's architecture. Depending on the amount of space required by these (and the existence of recursion), on some systems, you may need to specify a stack size when compiling or linking. It can be very difficult to estimate the amount of stack needed to run a program since that depends on how much is used for automatic variables, function call overhead, the depth of any recursive calls, and that used by library functions.

**Standard C** A generic term for the current formal definition of the C language, preprocessor, and runtime library. Although the original ANSI C standard preceded the first ISO C standard, once an ISO standard is adopted, the U.S. typically replaces its current ANSI standard with that from ISO. Because these two standards are expected to continue to be technically equivalent, the term "Standard C" is preferable to avoid the implication that these standards may differ. See also C89; C90; C95; C99; C9X.

**standard header** See header, standard.

**standard streams** *See* `stderr`; `stdin`; `stdout`.

**state-dependent encoding**<sup>C89</sup> *See* multibyte character; shift state.

**statement** A C statement consists of either one of the constructs defined by the language (such as `if/else`, `for`, and `while`), an expression statement, a block, or a null statement.

**statement, compound** *See* block.

**static** A storage class keyword used in the declaration of an object to designate static storage duration. A static object may have either no linkage or internal linkage. This keyword also can be used with functions. A static function is callable only from functions defined in the same source code file or in any headers included in that file.

C99 added the ability to use the keyword `static` inside any dimension of an array parameter (possibly along with `const` and/or `restrict`); for example,

```
void f(int table[static 5]);
```

This declaration specifies that the argument corresponding to `table` in any call to function `f` must be a nonnull pointer to the first of at least five `int`.

**static storage duration** *See* storage duration, static.

**static\_cast** A C++ keyword that is not part of Standard C. If you think you might wish to move C code to a C++ environment in the future, you should refrain from using `static_cast` as an identifier in new C code you write.

**stdarg.h**<sup>C89</sup> A header that provides a way to access variable argument lists portably, such as those passed to the `printf` and `scanf` library families. It defines the type `va_list` and the macros `va_start`, `va_arg`, and `va_end`. The header `stdarg.h` was modeled closely on the UNIX `varargs.h` capability.

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

<i>Name</i>	<i>Purpose</i>
<code>va_arg</code>	Get argument from list
<code>va_copy</code> <sup>C99</sup>	Copies a <code>va_list</code>
<code>va_end</code>	Terminate argument list processing
<code>va_list</code>	Argument list manipulation type
<code>va_start</code>	Prepare for argument list processing

**stdbool.h**<sup>C99</sup> A header that contains the following macro definitions pertaining to boolean type support:

<i>Name</i>	<i>Purpose</i>
<code>bool</code>	A synonym for the type <code>_Bool</code>
<code>__bool_true_false_are_defined</code>	Allow a test for boolean support
<code>false</code>	A value representing false
<code>true</code>	A value representing true

**STDC**<sup>C99</sup> A preprocessing token that immediately follows `#pragma` in a pragma directive to indicate the use of a standard pragma. *See also* `#pragma STDC`.

`__STDC__` **prefix**<sup>C99</sup> *See* future language directions.

`__STDC__`<sup>C89</sup> A predefined macro that is set to 1 for standard-conforming implementations.

`__STDC_CONSTANT_MACROS`<sup>C99</sup> The header `stdint.h` defines a family of function-like macros that describe minimum-width integer and greatest-width integer constants. These macros should only be defined by a C++ implementation if the macro `__STDC_CONSTANT_MACROS` is defined before that header is included.

`__STDC_FORMAT_MACROS`<sup>C99</sup> The header `inttypes.h` defines a family of object-like macros that expand to format specifiers. These macros should only be defined by a C++ implementation if the macro `__STDC_FORMAT_MACROS` is defined before that header is included.

`__STDC_HOSTED__`<sup>C99</sup> A predefined macro that expands to the integer constant 1 if the implementation is running in a hosted environment; otherwise it expands to the integer constant 0.

`__STDC_IEC_559__`<sup>C99</sup> A macro that is conditionally predefined to the integer constant 1 if the implementation conforms to the floating-point standard IEC 60559.

`__STDC_IEC_559_COMPLEX__`<sup>C99</sup> A macro that is conditionally predefined to the integer constant 1 if the implementation conforms to the floating-point standard IEC 60559-compatible complex arithmetic.

`__STDC_ISO_10646__`<sup>C99</sup> A macro that is conditionally predefined to an integer constant of the form `yyyymmL` if the implementation uses the ISO/IEC 10646 character set to represent values of type `wchar_t`. The date `yyyymm` indicates the version of that standard.

**\_\_STDC\_LIMIT\_MACROS**<sup>C99</sup> The header `stdint.h` defines a family of object-like macros that describe specified-width integer constants. These macros should only be defined by a C++ implementation if the macro `__STDC_LIMIT_MACROS` is defined before that header is included.

**\_\_STDC\_VERSION\_\_**<sup>C95</sup> Although not defined in C89, in C95, this macro expands to the integer constant `199409L` while in C99, it expands to `199901L`. It is the committee's intent to update this value with each new revision of the standard.

**stddef.h**<sup>C89</sup> A header that contains several miscellaneous macro definitions and types. The macros are `NULL` and `offsetof`, and the types are `size_t`, `ptrdiff_t`, and `wchar_t`.

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

<i>Name</i>	<i>Purpose</i>
<code>NULL</code>	Null pointer constant
<code>offsetof</code>	Structure offset macro
<code>ptrdiff_t</code>	Pointer difference type
<code>size_t</code>	Size/count type
<code>wchar_t</code>	Wide character type

**stderr** A macro, defined in `stdio.h`, that expands to an expression of type `FILE *` that points to a file object corresponding to the standard error device. It is not necessarily a translation-time constant.

**stdin** A macro, defined in `stdio.h`, that expands to an expression of type `FILE *` that points to a file object corresponding to the standard input device. It is not necessarily a translation-time constant.

**stdint.h**<sup>C99</sup> A header that declares sets of integer types having specified widths, and defines corresponding sets of macros. It also defines macros that specify limits of integer types corresponding to types defined in other standard headers. The identifiers defined in this header are as follows:

<i>Name</i>	<i>Purpose</i>
INT_FASTN_MAX	Maximum value of type <code>int_fastN_t</code>
INT_FASTN_MIN	Minimum value of type <code>int_fastN_t</code>
<code>int_fastN_t</code>	Fastest minimum-width integer type
INT_LEASTN_MAX	Maximum value of type <code>int_leastN_t</code>
INT_LEASTN_MIN	Minimum value of type <code>int_leastN_t</code>
<code>int_leastN_t</code>	Minimum-width integer type
INTMAX_C	Create a constant of type <code>intmax_t</code>
INTMAX_MAX	Maximum value of type <code>intmax_t</code>
INTMAX_MIN	Minimum value of type <code>intmax_t</code>
<code>intmax_t</code>	Largest signed integer type
INTN_C	Create a constant of type <code>int_leastN_t</code>
INTN_MAX	Maximum value of type <code>intN_t</code>
INTN_MIN	Minimum value of type <code>intN_t</code>
<code>intN_t</code>	Exact-width integer type
INTPTR_MAX	Maximum value of type <code>intptr_t</code>
INTPTR_MIN	Minimum value of type <code>intptr_t</code>
<code>intptr_t</code>	void pointer container
PTRDIFF_MAX	Maximum value of type <code>ptrdiff_t</code>
PTRDIFF_MIN	Minimum value of type <code>ptrdiff_t</code>
SIG_ATOMIC_MAX	Maximum value of type <code>sig_atomic_t</code>
SIG_ATOMIC_MIN	Minimum value of type <code>sig_atomic_t</code>
SIZE_MAX	Maximum value of type <code>size_t</code>
UINT_FASTN_MAX	Maximum value of type <code>uint_fastN_t</code>
<code>uint_fastN_t</code>	Fastest minimum-width integer type
UINT_LEASTN_MAX	Maximum value of type <code>uint_leastN_t</code>
<code>uint_leastN_t</code>	Minimum-width integer type
UINTMAX_C	Create a constant of type <code>uintmax_t</code>
UINTMAX_MAX	Maximum value of type <code>uintmax_t</code>
<code>uintmax_t</code>	Largest unsigned integer type
UINTN_C	Create a constant of type <code>uint_leastN_t</code>
UINTN_MAX	Maximum value of type <code>uintN_t</code>
<code>uintN_t</code>	Exact-width integer type
UINTPTR_MAX	Maximum value of type <code>uintptr_t</code>
<code>uintptr_t</code>	void pointer container
WCHAR_MAX	Maximum value of type <code>wchar_t</code>
WCHAR_MIN	Minimum value of type <code>wchar_t</code>
WINT_MAX	Maximum value of type <code>wint_t</code>
WINT_MIN	Minimum value of type <code>wint_t</code>

In the names above,  $N$  represents an unsigned decimal integer with no leading zeros.

See future library directions.

**stdio.h** A header that defines several types and macros and declares numerous functions useful for performing file and terminal I/O. `stdio.h` contains definitions or declarations for the identifiers in the following table:

<i>Name</i>	<i>Purpose</i>
<code>BUFSIZ</code>	<code>setbuf</code> buffer size
<code>clearerr</code>	Clear error and end-of-file flags
<code>EOF</code>	End-of-file indicator
<code>fclose</code>	Close file
<code>feof</code>	Check for end-of-file
<code>ferror</code>	Check for file error
<code>fflush</code>	Force write to file
<code>fgetc</code>	Read character from file
<code>fgetpos</code> <sup>C89</sup>	Get file position
<code>fgets</code>	Read string from file
<code>FILE</code>	File context block type
<code>FILENAME_MAX</code>	Maximum length of a filename
<code>FOPEN_MAX</code>	Maximum number of open files
<code>fopen</code>	Open file
<code>fpos_t</code>	File position type
<code>fprintf</code>	Formatted write to file
<code>fputc</code>	Write character to file
<code>fputs</code>	Write string to file
<code>fread</code>	Binary read from file
<code>freopen</code>	Recycle <code>FILE</code> pointer
<code>fscanf</code>	Formatted read from file
<code>fseek</code>	Randomly position in file
<code>fsetpos</code> <sup>C89</sup>	Randomly position in file
<code>ftell</code>	Get file position
<code>fwrite</code>	Binary write to file
<code>getc</code>	Read character from <code>stdin</code>
<code>getchar</code>	Read character from <code>stdin</code>
<code>gets</code>	Read string from <code>stdin</code>
<code>_IOFBF</code>	Type of <code>setvbuf</code> buffering
<code>_IOLBF</code>	Type of <code>setvbuf</code> buffering
<code>_IONBF</code>	Type of <code>setvbuf</code> buffering
<code>L_tmpnam</code>	Maximum length of temporary file name
<code>NULL</code>	Null pointer constant
<code>perror</code>	Produce error message
<code>printf</code>	Formatted write to <code>stdout</code>
<code>putc</code>	Write character to <code>stdout</code>
<code>putchar</code>	Write character to <code>stdout</code>
<code>puts</code>	Write string to <code>stdout</code>
<code>remove</code>	Remove or delete file

<i>Name</i>	<i>Purpose</i>
rename	Rename file
rewind	Position to start of file
scanf	Formatted read from <code>stdin</code>
SEEK_CUR	<code>fseek</code> position argument
SEEK_END	<code>fseek</code> position argument
SEEK_SET	<code>fseek</code> position argument
setbuf	Set file buffer characteristics
setvbuf	Set file buffer characteristics
size_t	Size/count type
snprintf <sup>C99</sup>	Formatted write to an array
sprintf	Formatted write to string
sscanf	Formatted read from string
stderr	Pointer to standard error FILE
stdin	Pointer to standard input FILE
stdout	Pointer to standard output FILE
TMP_MAX	Minimum number of unique temporary files
tmpfile	Open scratch file
tmpnam	Create unique file name
ungetc	Pushback character to <code>stdin</code>
vfprintf	Formatted write to a file
vscanf <sup>C99</sup>	Formatted read from a file
vprintf	Formatted write to <code>stdout</code>
vscanf <sup>C99</sup>	Formatted read from <code>stdin</code>
vsnprintf <sup>C99</sup>	Formatted read from an array
vsprintf	Formatted Write to a string
vsscanf <sup>C99</sup>	Formatted read from a string

See also future library directions.

**stdlib.h**<sup>C89</sup> A header that contains definitions or declarations for the identifiers in the following table:

<i>Name</i>	<i>Purpose</i>
abort	Force abnormal termination
abs	Compute absolute value
atexit	Register exit processing function
atof	Convert string to floating
atoi	Convert string to integer
atol	Convert string to integer
atoll <sup>C99</sup>	Convert string to integer
bsearch	Do binary search
calloc	Allocate and initialize memory
div	Perform division
div_t	Type returned from <code>div</code>

<i>Name</i>	<i>Purpose</i>
<code>exit</code>	Force normal termination
<code>_Exit</code> <sup>C99</sup>	Force normal termination
<code>EXIT_FAILURE</code>	Failure value for <code>exit</code>
<code>EXIT_SUCCESS</code>	Success value for <code>exit</code>
<code>free</code>	Release allocated memory
<code>getenv</code>	Get environment variable
<code>labs</code>	Compute absolute value
<code>ldiv</code>	Perform division
<code>ldiv_t</code>	Type returned from <code>ldiv</code>
<code>llabs</code> <sup>C99</sup>	Compute absolute value
<code>lldiv</code> <sup>C99</sup>	Perform division
<code>lldiv_t</code> <sup>C99</sup>	Type returned from <code>lldiv</code>
<code>malloc</code>	Allocate memory
<code>MB_CUR_MAX</code>	Maximum size of a multibyte character
<code>mblen</code>	Length of multibyte character
<code>mbstowcs</code>	Convert multibyte string
<code>mbtowc</code>	Convert multibyte character
<code>NULL</code>	Null pointer constant
<code>qsort</code>	Quicksort
<code>rand</code>	Generate random number
<code>RAND_MAX</code>	Maximum value returned by <code>rand</code>
<code>realloc</code>	Expand/contract allocated memory
<code>size_t</code>	Size/count type
<code>srand</code>	Set random number seed
<code>strtod</code>	Convert string to floating
<code>strtof</code> <sup>C99</sup>	Convert string to floating
<code>strtol</code>	Convert string to integer
<code>strtold</code> <sup>C99</sup>	Convert string to floating
<code>strtoll</code> <sup>C99</sup>	Convert string to integer
<code>strtoull</code> <sup>C99</sup>	Convert string to integer
<code>strtol</code>	Convert string to integer
<code>system</code>	Call out to system
<code>wchar_t</code>	Wide character type
<code>wcstombs</code>	Convert wide character string
<code>wctomb</code>	Convert wide character

*See also* future library directions.

**stdout** A macro, defined in `stdio.h`, that expands to an expression of type `FILE *` which points to a file object corresponding to the standard output device. It is not necessarily a translation-time constant.

**storage class** The characteristic of an object or function that indicates its scope and linkage. For objects, storage class also indicates their lifetime. Storage class is specified in a declaration by one of the keywords

**auto**, **static**, **extern**, or **register**. Inside a function definition, an object declaration containing no class keyword has storage class **auto**. The particular storage class keyword used and the location of the declaration in a source file (relative to being inside or outside a function definition) dictate the identifier's storage duration and linkage. Technically, **typedef** is also a storage class keyword; however, it implies neither storage duration nor linkage. The storage class of a variable or function is not part of its type. A declaration cannot contain more than one storage class keyword. And because **typedef** is, technically, a storage class keyword, you cannot include a storage class keyword in a **typedef**. That is,

```
typedef static int si;
```

and

```
typedef auto int ai;
```

are both invalid.

**storage-class keyword, position of** The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is obsolescent. For example, the declarations

```
long int static i;  
struct {  
    int i;  
    double d;  
} typedef stru;
```

are both valid; however, the keywords **static** and **typedef** should be placed at the beginning of their respective declarations.

**storage duration** The lifetime of an object. Storage duration refers to the time during which that object is guaranteed to actually exist. (An implementation may make it live longer.) There are three kinds of storage duration: allocated, automatic, and static. *See also* linkage; scope.

**storage duration, allocated** The storage duration of an object created by a call to **calloc**, **malloc**, or **realloc**. Such objects exist until their space is explicitly deallocated.

**storage duration, automatic** The storage duration of an object declared with the storage class keyword **auto** or **register**, or declared inside a function definition and having no storage class keyword. Conceptually,

such objects are created each time their parent block is entered at run-time and are destroyed when that block is exited. Automatic variables typically are maintained on a stack, and their initial value, by default, is undefined.

**storage duration, static** The storage duration of an object declared inside a function definition with the storage class keyword **static**, or outside a function definition, either with or without a storage class keyword. Such objects are created and initialized prior to **main** beginning execution. They retain their values across function calls. Their initial value (if none is provided) is zero, cast to their type. A **static** function also has static storage duration. This means it only can be called by name from within the source file in which it is defined.

**storage unit** The implementation-defined object into which bit-fields are packed. The order in which bit-fields are packed and whether or not they may span storage unit boundaries, is also implementation defined.

**str prefix** *See* future library directions.

**strcat** A function that copies the string pointed to by **s2** to the end of the string pointed to by **s1**. In the process, the trailing null of **s1** is overwritten by the first character of the string pointed to by **s2**. The destination string is nullterminated.

```
#include <string.h>
char *strcat(char * restrict s1, const char * restrict s2);
```

The value of **s1** is returned. If the strings located at **s1** and **s2** overlap, the behavior of **strcat** is undefined. *See also* **wscat**.

**strchr** A function that searches the string **s** for a character **c**.

```
#include <string.h>
char *strchr(const char *s, int c);
```

If **c** is found, **strchr** returns a pointer to that location within **s**; otherwise it returns **NULL**. **c** is converted to **char** before the search begins. The null terminating **s** is included in the search. Therefore, **strchr** can be used to locate the trailing null as well. *See also* **wcschr**.

**strcmp** A function that compares a string at the location pointed to by **s2** to a string at the location pointed to by **s1**.

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

An integer less than, equal to, or greater than zero is returned to indicate whether `s1` is less than, equal to, or greater than `s2`, respectively. *See also* `wscmp`.

`strcoll`<sup>C89</sup> A function that compares a string at the location pointed to by `s2` to a string at the location pointed to by `s1`.

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

An integer less than, equal to, or greater than zero is returned to indicate whether `s1` is less than, equal to, or greater than `s2`, respectively. The comparison is locale specific and, therefore, permits any arbitrary collation sequence to be used, as long as that sequence is provided by the implementation as a locale. The sequence can include non-English letters and characters. In the "C" locale, `strcoll` should give the same result as `strcmp`. *See also* `wscoll`.

`strcpy` A function that copies the string pointed to by `s2` to the location pointed to by `s1`. The destination string is null terminated.

```
#include <string.h>
char *strcpy(char * restrict s1, const char * restrict s2);
```

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

`strcspn` A function that finds the longest string starting at the location pointed to by `s1` such that it does not contain any of the characters in the string pointed to by `s2`. Alternatively, it could be viewed that `strcspn` locates the first character in the string pointed to by `s1` that is present in the string pointed to by `s2`.

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

The return value indicates the length of the nonmatching string found at `s1` (which is the same as the subscript of the first matching character in `s1`). *See also* `wscspn`.

**stream** The logical channel on which I/O is performed. The three standard streams `stdin`, `stdout` and `stderr` refer to standard input, standard output and standard error, respectively. They are opened for you at the start of each program and are closed when the program terminates. A

file is associated with a stream by `fopen`. *See also* `FILE`; `FOPEN_MAX`; stream, binary; stream, text.

**stream, binary** An ordered sequence of characters that can transparently record internal data. Once written, such data can be retrieved exactly as written, by the same implementation. (The differences of byte ordering within words and words within longwords, etc., might prohibit it from being read correctly by other implementations.) There may be some implementation-defined number of null characters appended to the end of a binary stream. When a stream is opened in binary mode (using the `b` mode with `fopen`), no logical translation is performed. For example, if a carriage-return/line-feed pair normally is converted to a new-line on input, in binary mode it would be read as those two separate characters. *See also* stream, text.

**stream, fully buffered** A stream in which it is intended that characters be read or written when a block or buffer is empty, or full, respectively. *See also* `_IOFBF`; `setvbuf`; stream, line buffered; stream, unbuffered.

**stream, line buffered** A stream in which it is intended that characters be read or written when a new-line is encountered. *See also* `_IOLBF`; `setvbuf`; stream, fully buffered; stream, unbuffered.

**stream, standard error** *See* `stderr`.

**stream, standard input** *See* `stdin`.

**stream, standard output** *See* `stdout`.

**stream, text** A stream in which the characters are organized into lines, each of which is terminated by a new-line. When data is written out to and read in from a text stream, there may be some kind of logical-to-physical or physical-to-logical translation. For example, on input, a carriage-return/line-feed pair may be translated to a new-line. An implementation does not have to distinguish between text and binary streams. *See also* the mode argument to `fopen`; stream, binary.

**stream, unbuffered** A stream in which characters should be read or written as soon as practical; that is, they should not be buffered. *See also* `_IONBF`; `setvbuf`; stream, fully buffered; stream, line buffered.

**strerror** A function that returns the address of a string containing a message corresponding to the message code used as the argument.

```
#include <string.h>
char *strerror(int errnum);
```

`strerror` is similar in capability to `perror` in that it is designed for use with `errno`. However, `perror` writes the message directly to `stderr`

along with the user-supplied text, while `strerror` returns a pointer to the message allowing the programmer to process it as desired.

`strptime`<sup>C89</sup> A function that constructs a date and time string by putting characters into the array pointed to by `s` according to the controlling format specified by `format`. No more than `maxsize` characters are written to the array. `timeptr` points to a structure whose contents are used to determine the appropriate characters for the array.

```
#include <time.h>
size_t strptime(char * restrict s, size_t maxsize,
                const char * restrict format,
                const struct tm * restrict timeptr);
```

This function provides a locale-specific way of generating date and time strings. For example,

```
#include <stdio.h>
#include <time.h>

int main()
{
    char string[100];
    time_t cur_time;
    struct tm *ptime;

    cur_time = time(NULL);
    if ((ptime = gmtime(&cur_time)) != NULL) {
        strptime(string, sizeof(string)-1,
                "%Y %B %A %c", ptime);
        printf("%s\n", string);
    }
    else {
        printf("UTC time not available\n");
    }

    return 0;
}
```

```
1991 April Monday Mon Apr 08 15:10:37 1991
```

Refer to your library documentation for specific details of the controlling format conversion specifiers. *See also* `wcsftime`.

**strictly conforming program**<sup>C89</sup> *See* program, strictly conforming.

**string** An array of `char` terminated by a null character. *See also* string literal; string literal, wide; wide string.

**string conversion functions** The `string.h` functions `atof`, `atoi`, `atol`, `strtod`, `strtof`, `strtol`, `strtold`, and `strtoul`.

**string functions** *See* `string.h`; `wchar.h`.

**string literal** A source token having the form `"..."` where `...` can be zero or more characters from the source character set except the double-quote, backslash, or new-line characters. A string literal is stored by the compiler as a `static` array of `char` with a trailing null character appended. It is implementation defined as to whether strings are stored in a writable memory location and whether identical strings are distinct.

**string literal, wide**<sup>C89</sup> A source token having the form `L"..."` where `...` can be zero or more characters from the source character set except the double-quote, backslash, or new-line characters. A wide string literal is stored by the compiler as a `static` array of `wchar_t` with a trailing null wide character appended. It is implementation defined as to whether strings are stored in a writable memory location and whether identical strings are distinct.

**string.h** A header that contains definitions or declarations for the identifiers in the following table:

<i>Name</i>	<i>Purpose</i>
<code>memchr</code>	Search memory for character
<code>memcmp</code>	Compare memory blocks
<code>memcpy</code>	Copy memory blocks
<code>memmove</code>	Safe copy of memory blocks
<code>memset</code>	Initialize memory block
<code>NULL</code>	Null pointer constant
<code>size_t</code>	Size/count type
<code>strcat</code>	Concatenate strings
<code>strchr</code>	Search string for character
<code>strcmp</code>	Compare strings
<code>strcoll</code>	Collate strings
<code>strcpy</code>	Copy a string
<code>strcspn</code>	Compute initial length match
<code>strerror</code>	Produce error message
<code>strlen</code>	Compute string length
<code>strncat</code>	Concatenate strings
<code>strncmp</code>	Compare leading part of strings
<code>strncpy</code>	Copy leading part of string

<i>Name</i>	<i>Purpose</i>
<code>strpbrk</code>	Locate character in string
<code>strrchr</code>	Reverse search for character
<code>strspn</code>	Compute initial length match
<code>strstr</code>	Search string for string
<code>strtok</code>	Break string into tokens
<code>strxfrm</code>	Transform string

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

**stringize operator**<sup>C89</sup> A unary preprocessor operator, `#`, that is defined only for function-like macros. If a parameter in the macro's replacement list is preceded by this operator, both are replaced by a single character string literal token that contains the text of the corresponding argument. If the argument contains " characters, they are escaped with a `\` in the string created. Leading and trailing white-space characters in the argument are ignored, and each occurrence of contiguous white space is replaced by a single space character. For example, given the following macro definition:

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

the macro call:

```
M( 5  "10 \n"  )
```

expands to:

```
f("5 \"10 \\n\"")
```

The precedence and associativity of the `#` and `##` preprocessor operators are unspecified.

**strlen** A function that returns the number of characters in the string (excluding the trailing null) pointed to by `s`.

```
#include <string.h>
size_t strlen(const char *s);
```

**strncat** A function that copies, at most, the first `n` characters from the string pointed to by `s2` to the end of the string pointed to by `s1`. The destination string is null terminated.

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

The value of `s1` is returned. If a null is not found in the first `n` characters of `s2`, `n` characters will be copied followed by a null. If a null is found, it is copied to `s1` and the copy is terminated. If the strings located at `s1` and `s2` overlap, the behavior of `strncat` is undefined. *See also* `wcsncat`.

**strncmp** A function that compares no more than `n` characters from a string at the location pointed to by `s2` to a string at the location pointed to by `s1`. If a null is seen in the first `n` characters, the comparison is terminated.

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

An integer less than, equal to, or greater than zero is returned to indicate whether `s1` is less than, equal to, or greater than `s2`, respectively. *See also* `wcsncmp`.

**strncpy** A function that copies, at most, the first `n` characters from the string pointed to by `s2` to the location pointed to by `s1`. The destination string is null terminated.

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

The value of `s1` is returned. If a null is not found in the first `n` characters of `s2`, the string pointed to by `s1` will not be null terminated. If a null is found, it is copied to `s1`. If there are less than `n` characters in the string pointed to by `s1`, extra null characters are appended to the end of the string pointed to by `s1` such that `n` characters are actually copied there. If the strings located at `s1` and `s2` overlap, the behavior of `strncpy` is undefined. To copy overlapping objects, use `memmove` instead. *See also* `wcsncpy`.

**strpbrk** A function that searches the string pointed to by `s1` for the first of any one of the characters in the string pointed to by `s2`.

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

The return value is a pointer to the character found in `s1` or `NULL` if no match was found. *See also* `wcspbrk`.

**strrchr** A function that searches the string **s** for the last occurrence of the character **c**.

```
#include <string.h>
char *strrchr(const char *s, int c);
```

If **c** is found, **strrchr** returns a pointer to that location within **s**; otherwise it returns **NULL**. **c** is converted to **char** before the search begins. The null-terminating **s** is included in the search. Therefore, **strrchr** can be used to locate the trailing null as well. *See also* **wcsrchr**.

**strspn** A function that finds the longest string starting at the location pointed to by **s1** such that it contains only those characters in the string at the location pointed to by **s2**. Alternatively, it could be viewed that **strspn** locates the first character in the string pointed to by **s1** that is not present in the string pointed to by **s2**.

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

The return value indicates the length of the matching string found at **s1** (which is the same as the subscript of the first nonmatching character in **s1**). *See also* **wcsspn**.

**strstr** A function that searches the string pointed to by **s1** for the substring pointed to by **s2**.

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

The return value represents the location of the substring **s2** within the string **s1**. If the substring is not found, **NULL** is returned. By definition, the null string is always found at the beginning of any string, including a null string. *See also* **wcsstr**.

**strtod** A function that converts the leading part of the string pointed to by **nptr** to a double value.

```
#include <stdlib.h>
double strtod(const char * restrict nptr,
              char ** restrict endptr);
```

Leading white space is ignored. The floating-point constant is terminated by any character not permissible in such a constant (including the null that terminates the input string). A leading sign character is

permitted as is a (possibly signed) exponent using either 'e' or 'E'. A floating-point suffix of 'f' (or 'F') or 'l' (or 'L') is not recognized as such. If no exponent or decimal point appears, a decimal point is assumed to be at the end of the number. The format of the floating-point number is locale specific. If the input contains all white space, is empty, or contains no convertible characters, no conversion occurs, `nptr` is stored in `*endptr` (provided `endptr` is not `NULL`), and a zero value is returned. If the converted value causes overflow, plus or minus `HUGE_VAL` is returned (according to the sign of the value) and `errno` is set to `ERANGE`. On underflow, zero is returned, and `errno` is set to `ERANGE`. `strtod` is a superset of `atof` and is recommended over it.

C99 extended this function to allow it to recognize the following input: hexadecimal floating-point values, infinity, and NaN.

`strtof`<sup>C99</sup> A function that converts the leading part of the string pointed to by `nptr` to a float value.

```
#include <stdlib.h>
float strtof(const char * restrict nptr,
             char ** restrict endptr);
```

Leading white space is ignored. The floating-point constant is terminated by any character not permissible in such a constant (including the null that terminates the input string). A leading sign character is permitted as is a (possibly signed) exponent using either 'e' or 'E'. A floating-point suffix of 'f' (or 'F') or 'l' (or 'L') is not recognized as such. If no exponent or decimal point appears, a decimal point is assumed to be at the end of the number. The format of the floating-point number is locale specific. If the input contains all white space, is empty, or contains no convertible characters, no conversion occurs, `nptr` is stored in `*endptr` (provided `endptr` is not `NULL`), and a zero value is returned. If the converted value causes overflow, plus or minus `HUGE_VALF` is returned (according to the sign of the value) and `errno` is set to `ERANGE`. On underflow, zero is returned, and `errno` is set to `ERANGE`.

The following input is recognized as such: hexadecimal floating-point values, infinity, and NaN.

`strtoumax`<sup>C99</sup> A function that is equivalent to the `strtoul` and `strtoll` functions, except that the initial portion of the string is converted to type `intmax_t`.

```
#include <inttypes.h>
intmax_t strtoumax(const char * restrict nptr,
                  char ** restrict endptr, int base);
```

If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `INTMAX_MAX` or `INTMAX_MIN` is returned (depending on the sign of the value), and `errno` is set to `ERANGE`.

**strtok** A function in which successive calls can be used to break a string, pointed to by `s1`, into a series of null-terminated tokens using the token terminator characters specified by the string pointed to by `s2`.

```
#include <string.h>
char *strtok(char * restrict s1, const char * restrict s2);
```

The value returned is either a pointer to the token found or `NULL` if no token was found. *See also* `wcstok`.

**strtol** A function that converts the leading part of the string pointed to by `nptr` to a long int value.

```
#include <stdlib.h>
long int strtol(const char * restrict nptr,
               char ** restrict endptr, int base);
```

Leading white space is ignored. The integer constant is terminated by any character not permissible in such a constant (including the null that terminates the input string). A leading sign character is permitted, but the suffixes 'u' (or 'U') or 'l' (or 'L') are not. The integer value is interpreted using radix `base`. If `base` is 0, the radix will be determined based on the presence or absence of a leading 0 or 0x (or 0X). Otherwise, `base` may be between 2 and 36. The format of the integer value is locale specific. If the input contains all white space, is empty, or contains no convertible characters, no conversion occurs, `nptr` is stored in `*endptr` (provided `endptr` is not `NULL`), and a zero value is returned. If the converted value causes overflow, either `LONG_MAX` or `LONG_MIN` is returned (depending on the sign of the value) and `errno` is set to `ERANGE`. `strtol` is a superset of `atol` and `atoi` and is recommended over them. *See also* `wcstol`.

**strtold**<sup>C99</sup> A function that converts the leading part of the string pointed to by `nptr` to a long double value.

```
#include <stdlib.h>
long double strtold(const char * restrict nptr,
                   char ** restrict endptr);
```

Leading white space is ignored. The floating-point constant is terminated by any character not permissible in such a constant (including

the null that terminates the input string). A leading sign character is permitted as is a (possibly signed) exponent using either 'e' or 'E'. A floating-point suffix of 'f' (or 'F') or 'l' (or 'L') is not recognized as such. If no exponent or decimal point appears, a decimal point is assumed to be at the end of the number. The format of the floating-point number is locale specific. If the input contains all white space, is empty, or contains no convertible characters, no conversion occurs, `nptr` is stored in `*endptr` (provided `endptr` is not NULL), and a zero value is returned. If the converted value causes overflow, plus or minus `HUGE_VALL` is returned (according to the sign of the value) and `errno` is set to `ERANGE`. On underflow, zero is returned, and `errno` is set to `ERANGE`.

The following input is recognized as such: hexadecimal floating-point values, infinity, and NaN.

`strtoll`<sup>C99</sup> A function that converts the leading part of the string pointed to by `nptr` to a `long long int` value.

```
#include <stdlib.h>
long long int strtoll(const char * restrict
                    nptr, char ** restrict endptr, int base);
```

Leading white space is ignored. The integer constant is terminated by any character not permissible in such a constant (including the null that terminates the input string). A leading sign character is permitted, but the suffixes 'u' (or 'U') or 'l' (or 'L') are not. The integer value is interpreted using radix `base`. If `base` is 0, the radix will be determined based on the presence or absence of a leading 0 or 0x (or 0X). Otherwise, `base` may be between 2 and 36. The format of the integer value is locale specific. If the input contains all white space, is empty, or contains no convertible characters, no conversion occurs, `nptr` is stored in `*endptr` (provided `endptr` is not NULL), and a zero value is returned. If the converted value causes overflow, either `LLONG_MAX` or `LLONG_MIN` is returned (depending on the sign of the value) and `errno` is set to `ERANGE`. *See also* `wcstoll`.

`strtoul` A function that converts the leading part of the string pointed to by `nptr` to an `unsigned long int` value.

```
#include <stdlib.h>
unsigned long int strtoul(const char * restrict nptr,
                        char ** restrict endptr, int base);
```

Leading white space is ignored. The integer constant is terminated by any character not permissible in such a constant (including the null that

terminates the input string). A leading sign character is permitted, but the suffixes 'u' (or 'U') or 'l' (or 'L') are not. The integer value is interpreted using radix `base`. If `base` is 0, the radix will be determined based on the presence or absence of a leading 0 or 0x (or 0X). Otherwise, `base` may be between 2 and 36. The format of the integer value is locale specific. If the input contains all white space, is empty, or contains no convertible characters, no conversion occurs, `nptr` is stored in `*endptr` (provided `endptr` is not NULL), and a 0 value is returned. If the converted value causes overflow, `ULONG_MAX` is returned, and `errno` is set to `ERANGE`. See also `wcstoul`.

`strtol`<sup>C99</sup> A function that converts the leading part of the string pointed to by `nptr` to an unsigned long long int value.

```
#include <stdlib.h>
unsigned long long int strtoul(const char * restrict nptr,
    char ** restrict endptr, int base);
```

Leading white space is ignored. The integer constant is terminated by any character not permissible in such a constant (including the null that terminates the input string). A leading sign character is permitted, but the suffixes 'u' (or 'U') or 'l' (or 'L') are not. The integer value is interpreted using radix `base`. If `base` is 0, the radix will be determined based on the presence or absence of a leading 0 or 0x (or 0X). Otherwise, `base` may be between 2 and 36. The format of the integer value is locale-specific. If the input contains all white space, is empty, or contains no convertible characters, no conversion occurs, `nptr` is stored in `*endptr` (provided `endptr` is not NULL), and a 0 value is returned. If the converted value causes overflow, `ULLONG_MAX` is returned, and `errno` is set to `ERANGE`. See also `wcstoul`.

`strtoumax`<sup>C99</sup> A function that is equivalent to the `strtol` and `strtol` functions, except that the initial portion of the string is converted to type `uintmax_t`.

```
#include <inttypes.h>
uintmax_t strtoumax(const char * restrict nptr,
    char ** restrict endptr, int base);
```

If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `UINTMAX_MAX` is returned, and `errno` is set to `ERANGE`.

`struct` The keyword used to define a structure type and to declare identifiers of that type.

**structure** An aggregate type that contains one or more members that typically are related. Sufficient storage is allocated for a structure so that at any time, one occurrence of each member can be stored simultaneously. It is declared as follows:

```
struct [ tag ] {
    member-1 ;
    member-2 ;
    ...
    member-n ;
};
```

The size of a structure is at least the sum of the sizes of its members. Holes may exist between adjacent members or after the last member to accommodate hardware alignment requirements. Such holes are included in the structure and in the size produced by `sizeof`.

Although a union is syntactically very similar to a structure, it is subtly different. *See also* arrow operator; bit-field; dot operator.

**structure tag** *See* tag.

**structure/union arrow operator** *See* arrow operator.

**structure/union dot operator** *See* dot operator.

**structure/union member name space** *See* name space.

**strxfrm**<sup>C89</sup> A function that transforms the string pointed to by `s2` into another string pointed to by `s1`. The transformation is locale specific.

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

If two strings are transformed, and the resultant strings are compared using `strcmp`, the same result should be obtained as if the original two strings had been compared using `strcoll`. No more than `n` transformed characters are to be written into `s1`. If the transformed string is bigger than `n`, only `n` characters are copied to `s1`, and the function returns the total number needed to hold the whole transformed string. Normally, `n` is large enough and the return value is the number of characters actually used in `s1` (excluding the null). If `s1` and `s2` designate strings that overlap, the result is undefined. *See also* `wcsxfrm`.

**subnormal** A nonzero floating-point value whose absolute value is smaller than the smallest nonzero normalized value.

**subscript operator** A primary operator, [ ], is used to subscript an array designator or data pointer. One of the two operands must have an object pointer type, and the other must have an integer type. The order of evaluation of the operands is unspecified. This operator is commutative. That is, `a[i]` is equivalent to and completely interchangeable with `i[a]`, although the latter is not recommended as a coding style. This operator always produces an lvalue. It associates left to right. The type of the result is the type of the object to which the pointer operand points.

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

**subtraction operator** A binary operator, `-`, that causes the value of its right operand to be subtracted from the value of its left. The order of evaluation of the two operands is unspecified. Both operands must have arithmetic type, or the right may have integer type if the left is a pointer to an object or both point to elements in the same array. The usual arithmetic conversions are performed on the operands. This operator associates left to right.

**switch** A keyword that introduces a construct allowing control to be passed to one of a given set of labels based on the value of a controlling integer expression. It is used as follows:

```
switch ( expression ) {  
  
    case 1:  
        statement(s)  
  
    case 2:  
        statement(s)  
  
        ...  
  
    case n:  
        statement(s)  
  
    [ default:  
        statement(s) ]  
}
```

If a **case** label with the value of *expression* exists, control is transferred to that label. If no such label exists and a **default** label exists, control transfers to the **default** label. Otherwise, control is transferred to the statement following the **switch**. When control is transferred to a **case** or **default** label, execution continues from that point on, possibly until the end of the **switch**. To interrupt this flow (to make each case mutually exclusive, for example) use something like **break** or **return**. The order in which the **case** and **default** labels are written, can be arbitrary although control can be made to flow through from one case to the next. *expression* is a full expression.

*See also* label, case; label, default; break.

**switch case label** *See* label, case.

**switch default label** *See* label, default.

**symbolic constant** *See* macro, object-like.

**swprintf**<sup>C95</sup> The wide character analog of **printf**.

```
#include <wchar.h>
int swprintf(wchar_t * restrict s, size_t n,
             const wchar_t * restrict format, ...);
```

**swscanf**<sup>C95</sup> The wide character analog of **scanf**.

```
#include <wchar.h>
int swscanf(const wchar_t * restrict s,
            const wchar_t * restrict format, ...);
```

**system** A function that passes off the string pointed to by **string** to the host environment command-line processor.

```
#include <stdlib.h>
int system(const char *string);
```

Standard C does not require that a command-line processor (or its equivalent) exists, in which case an implementation-defined value is returned. To ascertain whether such an environment exists, call **system** with a NULL argument; if a nonzero value is returned, a command-line processor is available. The format of the string passed is implementation defined.

