C Operator Precedence and Associativity

This page lists all C operators in order of their precedence (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity

`()`
`[]`
`.`
`->`
	Parentheses (grouping)
	Brackets (array subscript)
	Member selection via object name
	Member selection via pointer
		left-to-right

`++ --`
`+ -`
`! ~`
`(type)`
`*`
`&`
`sizeof`
	Unary preincrement/predecrement
	Unary plus/minus
	Unary logical negation/bitwise complement
	Unary cast (change *type*)
	Dereference
	Address
	Determine size in bytes
		right-to-left

`* / %`	Multiplication/division/modulus	left-to-right
`+ -`	Addition/subtraction	left-to-right
`<< >>`	Bitwise shift left, Bitwise shift right	left-to-right
`< <=` `> >=`	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
`== !=`	Relational is equal to/is not equal to	left-to-right
`&`	Bitwise AND	left-to-right
`^`	Bitwise exclusive OR	left-to-right
`|`	Bitwise inclusive OR	left-to-right
`&&`	Logical AND	left-to-right
`||`	Logical OR	left-to-right
`?:`	Ternary conditional	right-to-left

`=`
`+= -=`
`*= /=`
`%= &=`
`^= |=`
`<<= >>=`
	Assignment
	Addition/subtraction assignment
	Multiplication/division assignment
	Modulus/bitwise AND assignment
	Bitwise exclusive/inclusive OR assignment
	Bitwise shift left/right assignment
		right-to-left

`,`	Comma (separate expressions)	left-to-right

Updated: 02.09.2005

Appendix B - Standard Library

This appendix is a summary of the library defined by the ANSI standard. The standard library is not part of the C language proper, but an environment that supports standard C will provide the function declarations and type and macro definitions of this library. We have omitted a few functions that are of limited utility or easily synthesized from others; we have omitted multi-byte characters; and we have omitted discussion of locale issues; that is, properties that depend on local language, nationality, or culture.

The functions, types and macros of the standard library are declared in standard *headers*:

```
<assert.h>  <float.h>   <math.h>    <stdarg.h>  <stdlib.h>
<ctype.h>   <limits.h>  <setjmp.h>  <stddef.h>  <string.h>
<errno.h>   <locale.h>  <signal.h>  <stdio.h>   <time.h>
```

A header can be accessed by

```
#include <header>
```

Headers may be included in any order and any number of times. A header must be included outside of any external declaration or definition and before any use of anything it declares. A header need not be a source file.

External identifiers that begin with an underscore are reserved for use by the library, as are all other identifiers that begin with an underscore and an upper-case letter or another underscore.

B.1 Input and Output: <stdio.h>

The input and output functions, types, and macros defined in `<stdio.h>` represent nearly one third of the library.

A *stream* is a source or destination of data that may be associated with a disk or other peripheral. The library supports text streams and binary streams, although on some systems, notably UNIX, these are identical. A text stream is a sequence of lines; each line has zero or more characters and is terminated by `'\n'`. An environment may need to convert a text stream to or from some other representation (such as mapping `'\n'` to carriage return and linefeed). A binary stream is a sequence of unprocessed bytes that record internal data, with the property that if it is written, then read back on the same system, it will compare equal.

A stream is connected to a file or device by *opening* it; the connection is broken by *closing* the stream. Opening a file returns a pointer to an object of type `FILE`, which records whatever information is necessary to control the stream. We will use ``file pointer'' and ``stream'' interchangeably when there is no ambiguity.

When a program begins execution, the three streams `stdin`, `stdout`, and `stderr` are already open.

B.1.1 File Operations

The following functions deal with operations on files. The type `size_t` is the unsigned integral type produced by the `sizeof` operator.

`FILE *fopen(const char *filename, const char *mode)`
`fopen` opens the named file, and returns a stream, or `NULL` if the attempt fails. Legal values for `mode` include:

`"r"`	open text file for reading
`"w"`	create text file for writing; discard previous contents if any
`"a"`	append; open or create text file for writing at end of file
`"r+"`	open text file for update (i.e., reading and writing)
`"w+"`	create text file for update, discard previous contents if any
`"a+"`	append; open or create text file for update, writing at end

Update mode permits reading and writing the same file; `fflush` or a file-positioning function must be called between a read and a write or vice versa. If the mode includes `b` after the initial letter, as in `"rb"` or `"w+b"`, that indicates a binary file. Filenames are limited to `FILENAME_MAX` characters. At most `FOPEN_MAX` files may be open at once.

`FILE *freopen(const char *filename, const char *mode, FILE *stream)`
`freopen` opens the file with the specified mode and associates the stream with it. It returns `stream`, or `NULL` if an error occurs. `freopen` is normally used to change the files associated with `stdin`, `stdout`, or `stderr`.

`int fflush(FILE *stream)`
On an output stream, `fflush` causes any buffered but unwritten data to be written; on an input stream, the effect is undefined. It returns `EOF` for a write error, and zero otherwise. `fflush(NULL)` flushes all output streams.

`int fclose(FILE *stream)`
`fclose` flushes any unwritten data for `stream`, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream. It returns `EOF` if any errors occurred, and zero otherwise.

`int remove(const char *filename)`
`remove` removes the named file, so that a subsequent attempt to open it will fail. It returns non-zero if the attempt fails.

`int rename(const char *oldname, const char *newname)`
`rename` changes the name of a file; it returns non-zero if the attempt fails.

`FILE *tmpfile(void)`
`tmpfile` creates a temporary file of mode `"wb+"` that will be automatically removed when closed or when the program terminates normally. `tmpfile` returns a stream, or `NULL` if it could not create the file.

`char *tmpnam(char s[L_tmpnam])`
`tmpnam(NULL)` creates a string that is not the name of an existing file, and returns a pointer to an internal static array. `tmpnam(s)` stores the string in `s` as well as returning it as the function

value; s must have room for at least L_tmpnam characters. tmpnam generates a different name each time it is called; at most TMP_MAX different names are guaranteed during execution of the program. Note that tmpnam creates a name, not a file.

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size)
```

setvbuf controls buffering for the stream; it must be called before reading, writing or any other operation. A mode of _IOFBF causes full buffering, _IOLBF line buffering of text files, and _IONBF no buffering. If buf is not NULL, it will be used as the buffer, otherwise a buffer will be allocated. size determines the buffer size. setvbuf returns non-zero for any error.

```
void setbuf(FILE *stream, char *buf)
```

If buf is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to (void) setvbuf(stream, buf, _IOFBF, BUFSIZ).

B.1.2 Formatted Output

The printf functions provide formatted output conversion.

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

fprintf converts and writes output to stream under the control of format. The return value is the number of characters written, or negative if an error occurred.

The format string contains two types of objects: ordinary characters, which are copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive argument to fprintf. Each conversion specification begins with the character % and ends with a conversion character. Between the % and the conversion character there may be, in order:

- Flags (in any order), which modify the specification:
  - -, which specifies left adjustment of the converted argument in its field.
  - +, which specifies that the number will always be printed with a sign.
  - *space*: if the first character is not a sign, a space will be prefixed.
  - 0: for numeric conversions, specifies padding to the field width with leading zeros.
  - #, which specifies an alternate output form. For o, the first digit will become zero. For x or X, 0x or 0X will be prefixed to a non-zero result. For e, E, f, g, and G, the output will always have a decimal point; for g and G, trailing zeros will not be removed.
- A number specifying a minimum field width. The converted argument will be printed in a field at least this wide, and wider if necessary. If the converted argument has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is 0 if the zero padding flag is present.
- A period, which separates the field width from the precision.
- A number, the precision, that specifies the maximum number of characters to be printed from a string, or the number of digits to be printed after the decimal point for e, E, or f conversions, or the number of significant digits for g or G conversion, or the number of digits to be printed for an integer (leading 0s will be added to make up the necessary width).
- A length modifier h, l (letter ell), or L. "h" indicates that the corresponding argument is to be printed as a short or unsigned short; "l" indicates that the argument is a long or unsigned long, "L" indicates that the argument is a long double.

Width or precision or both may be specified as *, in which case the value is computed by converting the next argument(s), which must be int.

The conversion characters and their meanings are shown in Table B.1. If the character after the % is not a conversion character, the behavior is undefined.

Table B.1 Printf Conversions

Character	Argument type; Printed As
d,i	int; signed decimal notation.
o	int; unsigned octal notation (without a leading zero).
x,X	unsigned int; unsigned hexadecimal notation (without a leading 0x or 0X), using abcdef for 0x or ABCDEF for 0X.
u	int; unsigned decimal notation.
c	int; single character, after conversion to unsigned char
s	char *; characters from the string are printed until a '\0' is reached or until the number of characters indicated by the precision have been printed.
f	double; decimal notation of the form [-]*mmm.ddd*, where the number of *d*'s is given by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
e,E	double; decimal notation of the form [-]*m.ddddde+/-xx* or [-]*m.ddddd*E+/-*xx*, where the number of *d*'s is specified by the precision. The default precision is 6; a precision of 0 suppresses the decimal point.
g,G	double; %e or %E is used if the exponent is less than -4 or greater than or equal to the precision; otherwise %f is used. Trailing zeros and a trailing decimal point are not printed.
p	void *; print as a pointer (implementation-dependent representation).
n	int *; the number of characters written so far by this call to printf is *written into* the argument. No argument is converted.
%	no argument is converted; print a %

```
int printf(const char *format, ...)
```
printf(...) is equivalent to fprintf(stdout, ...).

```
int sprintf(char *s, const char *format, ...)
```
sprintf is the same as printf except that the output is written into the string s, terminated with '\0'. s must be big enough to hold the result. The return count does not include the '\0'.

```
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *stream, const char *format, va_list arg)
int vsprintf(char *s, const char *format, va_list arg)
```
The functions vprintf, vfprintf, and vsprintf are equivalent to the corresponding printf functions, except that the variable argument list is replaced by arg, which has been initialized by the va_start macro and perhaps va_arg calls. See the discussion of <stdarg.h> in Section B.7.

B.1.3 Formatted Input

The scanf function deals with formatted input conversion.

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

`fscanf` reads from `stream` under control of `format`, and assigns converted values through subsequent arguments, *each of which must be a pointer*. It returns when `format` is exhausted. `fscanf` returns `EOF` if end of file or an error occurs before any conversion; otherwise it returns the number of input items converted and assigned.

The format string usually contains conversion specifications, which are used to direct interpretation of input. The format string may contain:

- Blanks or tabs, which are not ignored.
- Ordinary characters (not %), which are expected to match the next non-white space character of the input stream.
- Conversion specifications, consisting of a `%`, an optional assignment suppression character `*`, an optional number specifying a maximum field width, an optional `h`, `l`, or `L` indicating the width of the target, and a conversion character.

A conversion specification determines the conversion of the next input field. Normally the result is placed in the variable pointed to by the corresponding argument. If assignment suppression is indicated by `*`, as in `%*s`, however, the input field is simply skipped; no assignment is made. An input field is defined as a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that `scanf` will read across line boundaries to find its input, since newlines are white space. (White space characters are blank, tab, newline, carriage return, vertical tab, and formfeed.)

The conversion character indicates the interpretation of the input field. The corresponding argument must be a pointer. The legal conversion characters are shown in Table B.2.

The conversion characters `d`, `i`, `n`, `o`, `u`, and `x` may be preceded by `h` if the argument is a pointer to `short` rather than int, or by `l` (letter ell) if the argument is a pointer to `long`. The conversion characters `e`, `f`, and `g` may be preceded by `l` if a pointer to `double` rather than `float` is in the argument list, and by `L` if a pointer to a `long double`.

Table B.2 Scanf Conversions

Character	Input Data; Argument type
`d`	decimal integer; `int*`
`i`	integer; `int*`. The integer may be in octal (leading `0`) or hexadecimal (leading `0x` or `0X`).
`o`	octal integer (with or without leading zero); `int *`.
`u`	unsigned decimal integer; `unsigned int *`.
`x`	hexadecimal integer (with or without leading `0x` or `0X`); `int*`.
`c`	characters; `char*`. The next input characters are placed in the indicated array, up to the number given by the width field; the default is 1. No `'\0'` is added. The normal skip over white space characters is suppressed in this case; to read the next non-white space character, use `%1s`.
`s`	string of non-white space characters (not quoted); `char *`, pointing to an array of characters large enough to hold the string and a terminating `'\0'` that will be added.
`e,f,g`	floating-point number; `float *`. The input format for `float`'s is an optional sign, a string of numbers possibly containing a decimal point, and an optional exponent field containing an `E` or `e` followed by a possibly signed integer.
`p`	pointer value as printed by `printf("%p");`, `void *`.

n	writes into the argument the number of characters read so far by this call; int *. No input is read. The converted item count is not incremented.
[...]	matches the longest non-empty string of input characters from the set between brackets; char *. A '\0' is added. []...] includes] in the set.
[^...]	matches the longest non-empty string of input characters *not* from the set between brackets; char *. A '\0' is added. [^]...] includes] in the set.
%	literal %; no assignment is made.

`int scanf(const char *format, ...)`
`scanf(...)` is identical to `fscanf(stdin, ...)`.

`int sscanf(const char *s, const char *format, ...)`
`sscanf(s, ...)` is equivalent to `scanf(...)` except that the input characters are taken from the string `s`.

B.1.4 Character Input and Output Functions

`int fgetc(FILE *stream)`
`fgetc` returns the next character of `stream` as an `unsigned char` (converted to an `int`), or `EOF` if end of file or error occurs.

`char *fgets(char *s, int n, FILE *stream)`
`fgets` reads at most the next `n-1` characters into the array `s`, stopping if a newline is encountered; the newline is included in the array, which is terminated by `'\0'`. `fgets` returns `s`, or `NULL` if end of file or error occurs.

`int fputc(int c, FILE *stream)`
`fputc` writes the character `c` (converted to an `unsigend char`) on `stream`. It returns the character written, or `EOF` for error.

`int fputs(const char *s, FILE *stream)`
`fputs` writes the string `s` (which need not contain '\n) on `stream`; it returns non-negative, or `EOF` for an error.

`int getc(FILE *stream)`
`getc` is equivalent to `fgetc` except that if it is a macro, it may evaluate `stream` more than once.

`int getchar(void)`
`getchar` is equivalent to `getc(stdin)`.

`char *gets(char *s)`
`gets` reads the next input line into the array `s`; it replaces the terminating newline with `'\0'`. It returns `s`, or `NULL` if end of file or error occurs.

`int putc(int c, FILE *stream)`
`putc` is equivalent to `fputc` except that if it is a macro, it may evaluate `stream` more than once.

`int putchar(int c)`
`putchar(c)` is equivalent to `putc(c,stdout)`.

`int puts(const char *s)`

puts writes the string s and a newline to stdout. It returns EOF if an error occurs, non-negative otherwise.

`int ungetc(int c, FILE *stream)`

ungetc pushes c (converted to an unsigned char) back onto stream, where it will be returned on the next read. Only one character of pushback per stream is guaranteed. EOF may not be pushed back. ungetc returns the character pushed back, or EOF for error.

B.1.5 Direct Input and Output Functions

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`

fread reads from stream into the array ptr at most nobj objects of size size. fread returns the number of objects read; this may be less than the number requested. feof and ferror must be used to determine status.

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)`

fwrite writes, from the array ptr, nobj objects of size size on stream. It returns the number of objects written, which is less than nobj on error.

B.1.6 File Positioning Functions

`int fseek(FILE *stream, long offset, int origin)`

fseek sets the file position for stream; a subsequent read or write will access data beginning at the new position. For a binary file, the position is set to offset characters from origin, which may be SEEK_SET (beginning), SEEK_CUR (current position), or SEEK_END (end of file). For a text stream, offset must be zero, or a value returned by ftell (in which case origin must be SEEK_SET). fseek returns non-zero on error.

`long ftell(FILE *stream)`

ftell returns the current file position for stream, or -1 on error.

`void rewind(FILE *stream)`

rewind(fp) is equivalent to fseek(fp, 0L, SEEK_SET); clearerr(fp).

`int fgetpos(FILE *stream, fpos_t *ptr)`

fgetpos records the current position in stream in *ptr, for subsequent use by fsetpos. The type fpos_t is suitable for recording such values. fgetpos returns non-zero on error.

`int fsetpos(FILE *stream, const fpos_t *ptr)`

fsetpos positions stream at the position recorded by fgetpos in *ptr. fsetpos returns non-zero on error.

B.1.7 Error Functions

Many of the functions in the library set status indicators when error or end of file occur. These indicators may be set and tested explicitly. In addition, the integer expression errno (declared in <errno.h>) may contain an error number that gives further information about the most recent error.

`void clearerr(FILE *stream)`

clearerr clears the end of file and error indicators for stream.

`int feof(FILE *stream)`
`feof` returns non-zero if the end of file indicator for `stream` is set.

`int ferror(FILE *stream)`
`ferror` returns non-zero if the error indicator for `stream` is set.

`void perror(const char *s)`
`perror(s)` prints `s` and an implementation-defined error message corresponding to the integer in `errno`, as if by

```
fprintf(stderr, "%s: %s\n", s, "error message");
```

See `strerror` in Section B.3.

B.2 Character Class Tests: <ctype.h>

The header `<ctype.h>` declares functions for testing characters. For each function, the argument list is an `int`, whose value must be `EOF` or representable as an `unsigned char`, and the return value is an `int`. The functions return non-zero (true) if the argument `c` satisfies the condition described, and zero if not.

`isalnum(c)`	`isalpha(c)` or `isdigit(c)` is true
`isalpha(c)`	`isupper(c)` or `islower(c)` is true
`iscntrl(c)`	control character
`isdigit(c)`	decimal digit
`isgraph(c)`	printing character except space
`islower(c)`	lower-case letter
`isprint(c)`	printing character including space
`ispunct(c)`	printing character except space or letter or digit
`isspace(c)`	space, formfeed, newline, carriage return, tab, vertical tab
`isupper(c)`	upper-case letter
`isxdigit(c)`	hexadecimal digit

In the seven-bit ASCII character set, the printing characters are `0x20` (`' '`) to `0x7E` (`'~'`); the control characters are 0 `NUL` to `0x1F` (US), and `0x7F` (DEL).

In addition, there are two functions that convert the case of letters:

`int tolower(c)`	convert `c` to lower case
`int toupper(c)`	convert `c` to upper case

If `c` is an upper-case letter, `tolower(c)` returns the corresponding lower-case letter, `toupper(c)` returns the corresponding upper-case letter; otherwise it returns `c`.

B.3 String Functions: <string.h>

There are two groups of string functions defined in the header `<string.h>`. The first have names beginning with `str`; the second have names beginning with `mem`. Except for `memmove`, the behavior is undefined if copying takes place between overlapping objects. Comparison functions treat arguments

as unsigned char arrays.

In the following table, variables s and t are of type char *; cs and ct are of type const char *; n is of type size_t; and c is an int converted to char.

char *strcpy (s,ct)	copy string ct to string s, including '\0'; return s.
char *strncpy (s,ct,n)	copy at most n characters of string ct to s; return s. Pad with '\0's if ct has fewer than n characters.
char *strcat (s,ct)	concatenate string ct to end of string s; return s.
char *strncat (s,ct,n)	concatenate at most n characters of string ct to string s, terminate s with '\0'; return s.
int strcmp(cs,ct)	compare string cs to string ct, return <0 if cs<ct, 0 if cs==ct, or >0 if cs>ct.
int strncmp (cs,ct,n)	compare at most n characters of string cs to string ct; return <0 if cs<ct, 0 if cs==ct, or >0 if cs>ct.
char *strchr (cs,c)	return pointer to first occurrence of c in cs or NULL if not present.
char *strrchr (cs,c)	return pointer to last occurrence of c in cs or NULL if not present.
size_t strspn (cs,ct)	return length of prefix of cs consisting of characters in ct.
size_t strcspn (cs,ct)	return length of prefix of cs consisting of characters *not* in ct.
char *strpbrk (cs,ct)	return pointer to first occurrence in string cs of any character string ct, or NULL if not present.
char *strstr (cs,ct)	return pointer to first occurrence of string ct in cs, or NULL if not present.
size_t strlen(cs)	return length of cs.
char *strerror(n)	return pointer to implementation-defined string corresponding to error n.
char *strtok (s,ct)	strtok searches s for tokens delimited by characters from ct; see below.

A sequence of calls of strtok(s,ct) splits s into tokens, each delimited by a character from ct. The first call in a sequence has a non-NULL s, it finds the first token in s consisting of characters not in ct; it terminates that by overwriting the next character of s with '\0' and returns a pointer to the token. Each subsequent call, indicated by a NULL value of s, returns the next such token, searching from just past the end of the previous one. strtok returns NULL when no further token is found. The string ct may be different on each call.

The mem... functions are meant for manipulating objects as character arrays; the intent is an interface to efficient routines. In the following table, s and t are of type void *; cs and ct are of type const void *; n is of type size_t; and c is an int converted to an unsigned char.

void *memcpy (s,ct,n)	copy n characters from ct to s, and return s.
void *memmove (s,ct,n)	same as memcpy except that it works even if the objects overlap.
int memcmp (cs,ct,n)	compare the first n characters of cs with ct; return as with strcmp.
void *memchr	return pointer to first occurrence of character c in cs, or NULL if not present

`(cs,c,n)`	among the first `n` characters.
`void *memset (s,c,n)`	place character `c` into first `n` characters of `s`, return `s`.

B.4 Mathematical Functions: <math.h>

The header `<math.h>` declares mathematical functions and macros.

The macros `EDOM` and `ERANGE` (found in `<errno.h>`) are non-zero integral constants that are used to signal domain and range errors for the functions; `HUGE_VAL` is a positive `double` value. A *domain error* occurs if an argument is outside the domain over which the function is defined. On a domain error, `errno` is set to `EDOM`; the return value is implementation-defined. A *range error* occurs if the result of the function cannot be represented as a `double`. If the result overflows, the function returns `HUGE_VAL` with the right sign, and `errno` is set to `ERANGE`. If the result underflows, the function returns zero; whether `errno` is set to `ERANGE` is implementation-defined.

In the following table, `x` and `y` are of type `double`, `n` is an `int`, and all functions return `double`. Angles for trigonometric functions are expressed in radians.

`sin(x)`	sine of x
`cos(x)`	cosine of x
`tan(x)`	tangent of x
`asin(x)`	$\sin^{-1}(x)$ in range [-pi/2,pi/2], x in [-1,1].
`acos(x)`	$\cos^{-1}(x)$ in range [0,pi], x in [-1,1].
`atan(x)`	$\tan^{-1}(x)$ in range [-pi/2,pi/2].
`atan2(y,x)`	$\tan^{-1}(y/x)$ in range [-pi,pi].
`sinh(x)`	hyperbolic sine of x
`cosh(x)`	hyperbolic cosine of x
`tanh(x)`	hyperbolic tangent of x
`exp(x)`	exponential function e^x
`log(x)`	natural logarithm ln(x), $x>0$.
`log10(x)`	base 10 logarithm $\log_{10}(x)$, $x>0$.
`pow(x,y)`	x^y. A domain error occurs if $x=0$ and $y<=0$, or if $x<0$ and y is not an integer.
`sqrt(x)`	sqare root of x, x>=0.
`ceil(x)`	smallest integer not less than `x`, as a `double`.
`floor(x)`	largest integer not greater than `x`, as a `double`.
`fabs(x)`	absolute value |x|
`ldexp(x,n)`	$x*2^n$
`frexp(x, int *ip)`	splits x into a normalized fraction in the interval [1/2,1) which is returned, and a power of 2, which is stored in `*exp`. If x is zero, both parts of the result are zero.
`modf(x, double *ip)`	splits x into integral and fractional parts, each with the same sign as x. It stores the integral part in `*ip`, and returns the fractional part.
`fmod(x,y)`	floating-point remainder of x/y, with the same sign as x. If y is zero, the result is implementation-defined.

B.5 Utility Functions: <stdlib.h>

The header <stdlib.h> declares functions for number conversion, storage allocation, and similar tasks.

`double atof(const char *s)`
atof converts s to double; it is equivalent to strtod(s, (char**)NULL).

`int atoi(const char *s)`
converts s to int; it is equivalent to (int)strtol(s, (char**)NULL, 10).

`long atol(const char *s)`
converts s to long; it is equivalent to strtol(s, (char**)NULL, 10).

`double strtod(const char *s, char **endp)`
strtod converts the prefix of s to double, ignoring leading white space; it stores a pointer to any unconverted suffix in *endp unless endp is NULL. If the answer would overflow, HUGE_VAL is returned with the proper sign; if the answer would underflow, zero is returned. In either case errno is set to ERANGE.

`long strtol(const char *s, char **endp, int base)`
strtol converts the prefix of s to long, ignoring leading white space; it stores a pointer to any unconverted suffix in *endp unless endp is NULL. If base is between 2 and 36, conversion is done assuming that the input is written in that base. If base is zero, the base is 8, 10, or 16; leading 0 implies octal and leading 0x or 0X hexadecimal. Letters in either case represent digits from 10 to base-1; a leading 0x or 0X is permitted in base 16. If the answer would overflow, LONG_MAX or LONG_MIN is returned, depending on the sign of the result, and errno is set to ERANGE.

`unsigned long strtoul(const char *s, char **endp, int base)`
strtoul is the same as strtol except that the result is unsigned long and the error value is ULONG_MAX.

`int rand(void)`
rand returns a pseudo-random integer in the range 0 to RAND_MAX, which is at least 32767.

`void srand(unsigned int seed)`
srand uses seed as the seed for a new sequence of pseudo-random numbers. The initial seed is 1.

`void *calloc(size_t nobj, size_t size)`
calloc returns a pointer to space for an array of nobj objects, each of size size, or NULL if the request cannot be satisfied. The space is initialized to zero bytes.

`void *malloc(size_t size)`
malloc returns a pointer to space for an object of size size, or NULL if the request cannot be satisfied. The space is uninitialized.

`void *realloc(void *p, size_t size)`
realloc changes the size of the object pointed to by p to size. The contents will be unchanged up to the minimum of the old and new sizes. If the new size is larger, the new space is uninitialized. realloc returns a pointer to the new space, or NULL if the request cannot be satisfied, in which case *p is unchanged.

`void free(void *p)`

free deallocates the space pointed to by p; it does nothing if p is NULL. p must be a pointer to space previously allocated by calloc, malloc, or realloc.

```
void abort(void)
```
abort causes the program to terminate abnormally, as if by raise(SIGABRT).

```
void exit(int status)
```
exit causes normal program termination. atexit functions are called in reverse order of registration, open files are flushed, open streams are closed, and control is returned to the environment. How status is returned to the environment is implementation-dependent, but zero is taken as successful termination. The values EXIT_SUCCESS and EXIT_FAILURE may also be used.

```
int atexit(void (*fcn)(void))
```
atexit registers the function fcn to be called when the program terminates normally; it returns non-zero if the registration cannot be made.

```
int system(const char *s)
```
system passes the string s to the environment for execution. If s is NULL, system returns non-zero if there is a command processor. If s is not NULL, the return value is implementation-dependent.

```
char *getenv(const char *name)
```
getenv returns the environment string associated with name, or NULL if no string exists. Details are implementation-dependent.

```
void *bsearch(const void *key, const void *base,
              size_t n, size_t size,
              int (*cmp)(const void *keyval, const void *datum))
```

bsearch searches base[0]...base[n-1] for an item that matches *key. The function cmp must return negative if its first argument (the search key) is less than its second (a table entry), zero if equal, and positive if greater. Items in the array base must be in ascending order. bsearch returns a pointer to a matching item, or NULL if none exists.

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *))
```

qsort sorts into ascending order an array base[0]...base[n-1] of objects of size size. The comparison function cmp is as in bsearch.

```
int abs(int n)
```
abs returns the absolute value of its int argument.

```
long labs(long n)
```
labs returns the absolute value of its long argument.

```
div_t div(int num, int denom)
```
div computes the quotient and remainder of num/denom. The results are stored in the int members quot and rem of a structure of type div_t.

```
ldiv_t ldiv(long num, long denom)
```
ldiv computes the quotient and remainder of num/denom. The results are stored in the long members quot and rem of a structure of type ldiv_t.