The ANSI C Standard Library - Contents

<assert.h>

`void assert(int expression);`

Macro used to add diagnostics. If *expression* is false, message printed on `stderr` and abort called to terminate execution. Source file and line number in message come from preprocessor macros `__FILE__` and `__LINE__`. If `NDEBUG` is defined where `<assert.h>` is included, `assert` macro is ignored.

<ctype.h>

`int isalnum(int c);`

`isalpha(c)` or `isdigit(c)`

`int isalpha(int c);`

`isupper(c)` or `islower(c)`

`int iscntrl(int c);`

is control character

`int isdigit(int c);`

is decimal digit

`int isgraph(int c);`

is printing character other than space

`int islower(int c);`

is lower-case letter

`int isprint(int c);`

is printing character (including space)

`int ispunct(int c);`

is printing character other than space, letter, digit

`int isspace(int c);`

is space, formfeed, newline, carriage return, tab, vertical tab

`int isupper(int c);`

is upper-case letter

`int isxdigit(int c);`

is hexadecimal digit

`int tolower(int c);`

return lower-case equivalent

`int toupper(int c);`

return upper-case equivalent

Notes:

- In ASCII (7-bit), printing characters are `0x20 (' ')` to `0x7E ('~')`; control characters are `0x00 (NUL)` to `0x1F (US)` and `0x7F (DEL)`

<errno.h>

`extern int errno;`

An error code value set by some functions. It is generally the responsibility of the programmer to clear `errno` before calling such a function.

<float.h>

`FLT_RADIX`

`FLT_ROUNDS`

`FLT_DIG`

`FLT_EPSILON`

smallest number *x* such that `1.0 + x != 1.0`

`FLT_MANT_DIG`

`FLT_MAX`

maximum floating-point number

`FLT_MAX_EXP`

`FLT_MIN`

minimum normalised floating-point number

`FLT_MIN_EXP`

`DBL_DIG`

`DBL_EPSILON`

`DBL_MANT_DIG`

`DBL_MAX`

maximum `double` floating-point number

`DBL_MAX_EXP`

`DBL_MIN`

minimum normalised `double` floating-point number

`DBL_MIN_EXP`

<limits.h>

`CHAR_BIT`

number of bits in a `char`

`CHAR_MAX`

maximum value of `char`

`CHAR_MIN`

minimum value of `char`

`INT_MAX`

maximum value of `int`

`INT_MIN`

minimum value of `int`

`LONG_MAX`
maximum value of `long`

`LONG_MIN`
minimum value of `long`

`SCHAR_MAX`
maximum value of `signed char`

`SCHAR_MIN`
minimum value of `signed char`

`SHRT_MAX`
maximum value of `short`

`SHRT_MIN`
minimum value of `short`

`UCHAR_MAX`
maximum value of `unsigned char`

`UCHAR_MIN`
minimum value of `unsigned char`

`UINT_MAX`
maximum value of `unsigned int`

`ULONG_MAX`
maximum value of `unsigned long`

`USHRT_MAX`
maximum value of `unsigned short`

<math.h>

```
double sin(double x);
double cos(double x);
double tan(double x);
double asin(double x);
double acos(double x);
double atan(double x);
double atan2(double y, double x);
double sinh(double x);
double cosh(double x);
double tanh(double x);
double exp(double x);
double log(double x);
double log10(double x);
double pow(double x, double y);
```

x raised to power *y*

```
double sqrt(double x);
double ceil(double x);
```

smallest integer not less than *x*

```
double floor(double x);
```

largest integer not greater than *x*

```
double fabs(double x);
double ldexp(double x, int n);
double frexp(double x, int* exp);
double modf(double x, double* ip);
double fmod(double x, double y);
```

<setjmp.h>

`int setjmp(jmp_buf env);`

Save state information in *env*. Zero returned from direct call; non-zero from subsequent call of `longjmp`.

`void longjmp(jmp_buf env, int val);`

Restore state saved by most recent call to `setjmp` using information saved in *env*. Execution resumes as if `setjmp` just executed and returned non-zero value *val*.

<signal.h>

Handling exceptional conditions.

`SIGABRT`

abnormal termination

`SIGFPE`

arithmetic error

`SIGILL`

illegal function image

`SIGINT`

interactive attention

`SIGSEGV`

illegal storage access

`SIGTERM`

termination request sent to program

`void (*signal(int sig, void (*handler)(int)))(int);`

Install handler for subsequent signal *sig*. If *handler* is `SIG_DFL`, implementation-defined default behaviour is used; if *handler* is `SIG_IGN`, signal is ignored; otherwise function pointed to by *handler* is called with argument *sig*. `signal` returns the previous handler or `SIG_ERR` on error. When signal *sig* subsequently occurs, the signal is **restored to its default behaviour** and the handler is called. If the handler returns, execution resumes where signal occurred. Initial state of signals is implementation-defined.

`int raise(int sig);`

Send signal *sig* to the program. Non-zero returned if unsuccessful.

<stdarg.h>

Facilities for stepping through a list of function arguments of unknown number and type.

`void va_start(va_list ap, lastarg);`

Initialisation macro to be called once before any unnamed argument is accessed. *ap* must be declared as a local variable, and *lastarg* is the last named parameter of the function.

`type va_arg(va_list ap, type);`

Produce a value of the type (*type*) and value of the next unnamed argument. Modifies *ap*.

`void va_end(va_list ap);`

Must be called once after arguments processed and before function exit.

<stdio.h>

`FILE`

Type which records information necessary to control a stream.

`stdin`

Standard input stream. Automatically opened when a program begins execution.

`stdout`
: Standard output stream. Automatically opened when a program begins execution.

`stderr`
: Standard error stream. Automatically opened when a program begins execution.

FILENAME_MAX
: Maximum permissible length of a file name

FOPEN_MAX
: Maximum number of files which may be open simultaneously.

TMP_MAX
: Maximum number of temporary files during program execution.

`FILE* fopen(const char* filename, const char* mode);`
: Opens file *filename* and returns a stream, or `NULL` on failure. *mode* may be (combinations of):

    `"r"`
    : text reading

    `"w"`
    : text writing; discard previous content

    `"a"`
    : text append; writing at end

    `"r+"`
    : text update

    `"w+"`
    : text update; discard previous content

    `"a+"`
    : text append; writing at end

`FILE* freopen(const char* filename, const char* mode, FILE* stream);`
: Opens file *filename* with the specified mode and associates with it the specified stream. Returns *stream* or `NULL` on error. Usually used to change files associated with `stdin`, `stdout`, `stderr`.

`int fflush(FILE* stream);`
: Flushes stream *stream*. Effect undefined for input stream. Returns `EOF` for write error, zero otherwise. `fflush(NULL)` flushes all output streams.

`int fclose(FILE* stream);`
: Closes stream *stream* (after flushing, if output stream). Returns `EOF` on error, zero otherwise.

`int remove(const char* filename);`
: Removes file *filename*. Returns non-zero on failure.

`int rename(const char* oldname, const char* newname);`
: Changes name of file *oldname* to *newname*. Returns non-zero on failure.

`FILE* tmpfile();`
: Creates temporary file (mode `"wb+"`) which will be removed when closed or on normal program termination. Returns stream or `NULL` on failure.

`char* tmpname(char s[L_tmpnam]);`
: Assigns to *s* and returns unique name for temporary file.

`int setvbuf(FILE* stream, char* buf, int mode, size_t size);`
: Controls buffering for stream *stream*.

`void setbuf(FILE* stream, char* buf);`
: Controls buffering for stream *stream*.

`int fprintf(FILE* stream, const char* format, ...);`
: Converts (with format *format*) and writes output to stream *stream*. Number of characters written [negative on error] is returned. Between % and format conversion character:

    - Flags:

        -
        : left adjust

        +
        : always sign

        space

space if no sign

`0`

zero pad

`#`

Alternate form: for conversion character `o`, first digit will be zero, for [`xX`], prefix `0x` or `0X` to non-zero, for [`eEfgG`], always decimal point, for [`gG`] trailing zeros not removed.

- Width:
- Period:
- Precision: for conversion character `s`, maximum characters to be printed from the string, for [`eEf`], digits after decimal point, for [`gG`], significant digits, for an integer, minimum number of digits to be printed.
- Length modifier:

h

short or unsigned short

l

long or unsigned long

L

long double

Conversions:

`d, i`

int; signed decimal notation

`o`

int; unsigned octal notation

`x,X`

int; unsigned hexadecimal notation

`u`

int; unsigned decimal notation

`c`

int; single character

`s`

char*;

`f`

double; [-]mmm.ddd

`e,E`

double; [-]m.dddddde(+|-)xx

`g,G`

double

`p`

void*; print as pointer

`n`

int*; number of chars written into arg

`%`

print %

`int printf(const char*` *`format`*`, ...);`

`printf(f, ...)` is equivalent to `fprintf(stdout, f, ...)`

`int sprintf(char*` *`s`*`, const char*` *`format`*`, ...);`

Like fprintf, but output written into string *s*, **which must be large enough to hold the output**, rather than to a stream. Output is `NUL`-terminated. Return length does not include the `NUL`.

`int vfprintf(FILE*` *`stream`*`, const char*` *`format`*`, va_list` *`arg`*`);`

Equivalent to fprintf except that the variable argument list is replaced by *arg*, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See

`int vprintf(const char*` *`format`*`, va_list` *`arg`*`);`

Equivalent to printf except that the variable argument list is replaced by *arg*, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `int vsprintf(char* s, const char* format, va_list arg);`

Equivalent to sprintf except that the variable argument list is replaced by *arg*, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See

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

Performs formatted input conversion, reading from stream *stream* according to format *format*. The function returns when *format* is fully processed. Returns `EOF` if end-of-file or error occurs before any conversion; otherwise, the number of items converted and assigned. Each of the arguments following *format* **must be a pointer**. Format string may contain

- *Blanks*, *Tabs* : ignored
- *ordinary characters* : expected to match next non-white-space
- `%` : Conversion specification, consisting of `%`, optional assignment suppression character `*`, optional number indicating maximum field width, optional [`hlL`] indicating width of target, conversion character.

Conversion characters:

`d`
decimal integer; `int*` parameter required

`i`
integer; `int*` parameter required; decimal, octal or hex

`o`
octal integer; `int*` parameter required

`u`
unsigned decimal integer; `unsigned int*` parameter required

`x`
hexadecimal integer; `int*` parameter required

`c`
characters; `char*` parameter required; up to width; no `'\0'` added; no skip

`s`
string of non-white-space; `char*` parameter required; `'\0'` added

`e,f,g`
floating-point number; `float*` parameter required

`p`
pointer value; `void*` parameter required

`n`
chars read so far; `int*` parameter required

[...]
longest non-empty string from set; `char*` parameter required; '\0'

[^...]
longest non-empty string not from set; `char*` parameter required; '\0'

`%`
literal `%`; no assignment

`int scanf(const char* format, ...);`

`scanf(f, ...)` is equivalent to `fscanf(stdin, f, ...)`

`int sscanf(char* s, const char* format, ...);`

Like fscanf, but input read from string *s*.

`int fgetc(FILE* stream);`

Returns next character from stream *stream* as an `unsigned char`, or `EOF` on end-of-file or error.

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

Reads at most the next *n*-1 characters from stream *stream* into *s*, stopping if a newline is encountered (after copying the newline to *s*). *s* is `NUL`-terminated. Returs *s*, or `EOF` on end-of-file or error.

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

Writes *c*, converted to `unsigned char`, to stream *stream*. Returns the character written, or `EOF` on error.

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

Writes *s*, which need not contain `'\n'` on stream stream. Returns non-negative on success, `EOF` on error.

`int getc(FILE* stream);`

Equivalent to `fgetc` except that it may be a macro.

`int getchar();`

Equivalent to `getc(stdin)`.

`char* gets(char* s);`

Reads next line from stdin into *s*. Replaces terminating newline with `'\0'`. Returns *s*, or `NULL` on end-of-file or error.

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

Equivalent to `fputc` except that it may be a macro.

`int putchar(int c);`

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

`int puts(const char* s);`

Writes s and a newline to stdout. Returns non-negative on success, `EOF` on error.

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

Pushes *c* (which must not be `EOF`), converted to unsigned char, onto stream *stream* such that it will be returned by the next read. Only one character of pushback is guaranteed for a stream. Returns *c*, or `EOF` on error.

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

Reads at most *nobj* objects of size *size* from stream *stream* into *ptr*. Returns the number of objects read. `feof` and `ferror` must be used to determine status.

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

Writes to stream *stream*, *nobj* objects of size *size* from array *ptr*. Returns the number of objects written (which will be less than *nobj* on error).

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

Sets file position for stream *stream*. For a binary file, 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`). Returns non-zero on error.

`long ftell(FILE* stream);`

Returns current file position for stream *stream*, or `-1L` on error.

`void rewind(FILE* stream);`

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

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

Assigns current position in stream *stream* to `*ptr`. Type `fpos_t` is suitable for recording such values. Returns non-zero on error.

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

Sets current position of stream *stream* to `*ptr`. Returns non-zero on error.

`void clearerr(FILE* stream);`

Clears the end-of-file and error indicators for stream *stream*.

`int feof(FILE* stream);`

Returns non-zero if end-of-file indicator for stream *stream* is set.

`int ferror(FILE* stream);`

Returns non-zero if error indicator for stream *stream* is set.

`void perror(const char* s);`

Prints *s* and implementation-defined error message corresponding to `errno`:
`fprintf(stderr, "%s: %s\n", s, "error message")`
See strerror.

<stdlib.h>

`double atof(const char* s);`

Returns numerical value of *s*. Equivalent to `strtod(s, (char**)NULL)`.

`int atoi(const char* s);`

Returns numerical value of *s*. Equivalent to `(int)strtol(s, (char**)NULL, 10)`.

`long atol(const char* s);`

Returns numerical value of *s*. Equivalent to `strtol(s, (char**)NULL, 10)`.

`double strtod(const char* s, char** endp);`

Converts prefix of *s* to `double`, ignoring leading quite space. Stores a pointer to any unconverted suffix in `*endp` if *endp* non-`NULL`. If answer would overflow, `HUGE_VAL` is returned with the appropriate sign; if underflow, zero returned. In either case, `errno` is set to `ERANGE`.

`long strtol(const char* s, char** endp, int base);`

Converts prefix of *s* to `long`, ignoring leading quite space. Stores a pointer to any unconverted suffix in `*endp` if *endp* non-`NULL`. If *base* between 2 and 36, that base used; if zero, leading `0X` or `0x` implies hexadecimal, leading `0`implies octal, otherwise decimal. Leading `0X` or `0x` permitted for base 16. If answer would overflow, `LONG_MAX` or `LONG_MIN` returned and `errno` is set to `ERANGE`.

`unsigned long strtoul(const char* s, char** endp, int base);`

As for `strtol` except result is `unsigned long` and error value is `ULONG_MAX`.

`int rand();`

Returns pseudo-random number in range `0` to `RAND_MAX`.

`void srand(unsigned int seed);`

Uses *seed* as seed for new sequence of pseudo-random numbers. Initial seed is `1`.

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

Returns pointer to zero-initialised newly-allocated space for an array of *nobj* objects each of size *size*, or `NULL` if request cannot be satisfied.

`void* malloc(size_t size);`

Returns pointer to uninitialised newly-allocated space for an object of size *size*, or `NULL` if request cannot be satisfied.

`void* realloc(void* p, size_t size);`

Changes to *size* the size of the object to which *p* points. Contents unchanged to minimum of old and new sizes. If new size larger, new space is uninitialised. Returns ponter to the new space or, if request cannot be satisfied `NULL` leaving *p* unchanged.

`void free(void* p);`

Deallocats space to which *p* points. *p* must be `NULL`, in which case there is no effect, or a pointer returned by `calloc`, `malloc` or `realloc`.

`void abort();`

Causes program to terminate abnormally, as if by `raise(SIGABRT)`.

`void exit(int status);`

Causes normal program termination. Functions installed using `atexit` are called in reverse order of registration, open files are flushed, open streams are closed and control is returned to environment. *status* is returned to environment in implementation-dependent manner. Zero indicates successful termination and the values `EXIT_SUCCESS` and `EXIT_FAILURE` may be used.

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

Registers *fcn* to be called when program terminates normally. Non-zero returned if registration cannot be made.

`int system(const char* s);`

Passes *s* to environment for execution. If *s* is `NULL`, non-zero returned if command processor exists; return value is implementation-dependent if *s* is non-`NULL`.

`char getenv(const char* name);`

Returns (implementation-dependent) environment string associated with *name*, or `NULL` if no such string exists.

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

Searches `base[0]...base[n-1]` for item matching `*key`. Comparison function *cmp* must return negative if first argument is less than second, zero if equal and positive if greater. The *n* items of *base* must be in ascending order. Returns a pointer to the matching entry or `NULL` if not found.

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

Arranges into ascending order the array `base[0]...base[n-1]` of objects of size *size*. Comparison function *cmp* must return negative if first argument is less than second, zero if equal and positive if greater.

`int abs(int n);`

Returns absolute value of *n*

`long labs(long n);`

Returns absolute value of *n*

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

Returns in fields `quot` and `rem` of structure of type `div_t` the quotient and remainder of *`num/denom`*.

`ldiv_t ldiv(long num, long denom);`

Returns in fields `quot` and `rem` of structure of type `ldiv_t` the quotient and remainder of *`num/denom`*.

<string.h>

`char* strcpy(char* s, const char* ct);`

Copy *ct* to *s* including terminating `NUL`. Return *s*.

`char* strncpy(char* s, const char* ct, int n);`

Copy at most *n* characters of *ct* to *s* Pad with `NUL`s if *ct* is of length less than *n*. Return *s*.

`char* strcat(char* s, const char* ct);`

Concatenate *ct* to *s*. Return *s*.

`char* strncat(char* s, const char* ct, int n);`

Concatenate at most *n* characters of *ct* to *s*. Terminate *s* with `NUL` and return it.

`int strcmp(const char* cs, const char* ct);`

Compare *cs* and *ct*. Return negative if *`cs < ct`*, zero if *`cs == ct`*, positive if *`cs > ct`*.

`int strncmp(const char* cs, const char* ct, int n);`

Compare at most *n* characters of *cs* and *ct*. Return negative if *`cs < ct`*, zero if *`cs == ct`*, positive if *`cs > ct`*.

`char* strchr(const char* cs, int c);`

Return pointer to first occurrence of *c* in *cs*, or `NULL` if not found.

`char* strrchr(const char* cs, int c);`

Return pointer to last occurrence of *c* in *cs*, or `NULL` if not found.

`size_t strspn(const char* cs, const char* ct);`

Return length of prefix of *cs* consisting entirely of characters in *ct*.

`size_t strcspn(const char* cs, const char* ct);`

Return length of prefix of *cs* consisting entirely of characters *not* in *ct*.

`char* strpbrk(const char* cs, const char* ct);`

Return pointer to first occurrence within *cs* of any character of *ct*, or `NULL` if not found.

`char* strstr(const char* cs, const char* ct);`

Return pointer to first occurrence of *ct* in *cs*, or `NULL` if not found.

`size_t strlen(const char* cs);`

Return length of *cs*.

`char* strerror(int n);`

Return pointer to implementation-defined string corresponding with error *n*.

`char* strtok(char* s, const char* t);`

A sequence of calls to `strtok` returns tokens from *s* delimted by a character in *ct*. Non-`NULL` *s* indicates the first call in a sequence. *ct* may differ on each call. Returns `NULL` when no such token found.

`void* memcpy(void* s, const void* ct, int n);`

Copy *n* characters from *ct* to *s*. Return *s*. **Does not work correctly if objects overlap.**

`void* memmove(void* s, const void* ct, int n);`

Copy *n* characters from *ct* to *s*. Return *s*. Works correctly even if objects overlap.

`int memcmp(const void* cs, const void* ct, int n);`

Compare first *n* characters of *cs* with *ct*. Return negative if *`cs < ct`*, zero if *`cs == ct`*, positive if *cs* > *ct*.

`void* strchr(const char* cs, int c, int n);`

Return pointer to first occurrence of *c* in first *n* characters of *cs*, or `NULL` if not found.

`void* strchr(char* s, int c, int n);`

Replace each of the first *n* characters of *s* by *c*. Return *s*.

<time.h>

`clock_t`

An arithmetic type representing time.

`CLOCKS_PER_SEC`

The number of `clock_t` units per second.

`time_t`

An arithmetic type representing time.

`struct tm`

Represents the components of calendar time:

- `int tm_sec;`
  seconds after the minute
- `int tm_min;`
  minutes after the hour
- `int tm_hour;`
  hours since midnight
- `int tm_mday;`
  day of the month
- `int tm_ymon;`
  months **since** January
- `int tm_year;`
  years since 1900
- `int tm_day;`
  days since Sunday
- `int tm_yday;`
  days since January 1
- `int tm_isdst;`
  Daylight Saving Time flag : is positive if DST is in effect, zero if not in effect, negative if information unavailable.

`clock_t clock();`

Returns processor time used by program or -1 if not available.

`time_t time(time_t* tp);`

Returns current calendar time or -1 if not available. If *tp* is non-`NULL`, return value is also assigned to `*tp`.

`double difftime(time_t time2, time_t time1);`

Returns the difference is seconds between *time2* and *time1*.

`time_t mktime(struct tm* tp);`

Returns the local time corresponding to `*tp`, or `-1` if it cannot be represented.

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

Returns the given time as a string of the form:

`Sun Jan 3 14:14:13 1988\n\0`

`char* ctime(const time_t tp);`

Converts the given calendar time to a local time and returns the equivalent string. Equivalent to:

`asctime(localtime(tp))`

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

Returns the given calendar time converted into Coordinated Universal Time, or `NULL` if not available.

`struct tm* localtime(const time_t *tp*);`
: Returns calendar time `*tp` converted into local time.

`size_t strftime(char* *s*, size_t *smax*, const char* *fmt*, const struct tm* *tp*);`
: Formats `*tp` into *s* according to *fmt*.

Notes:

- *Local* time may differ from *calendar* time, for example because of time zone.