C Standard Library

[Last modified : 2003.07.09 by **Ross L. Richardson**, School of Information Systems, University of Tasmania, Australia]

<assert.h>

`void assert(int` *expression*`);`

Macro used for internal error detection. (Ignored if `NDEBUG` is defined where `<assert.h>` is included.) If *expression* equals zero, message printed on stderr and abort called to terminate execution. Source filename and line number in message are from preprocessor macros `__FILE__` and `__LINE__`.

<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. In ASCII, control characters are `0x00` (`NUL`) to `0x1F` (`US`), and `0x7F` (`DEL`)

`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). In ASCII, printing characters are `0x20` (`' '`) to `0x7E` (`'~'`)

`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

<errno.h>

`errno`
: object to which **certain** library functions assign specific positive values on error

`EDOM`
: code used for domain errors

`ERANGE`
: code used for range errors

Notes:

- other implementation-defined error values are permitted
- to determine the value (if any) assigned to errno by a library function, a program should assign zero to errno immediately prior to the function call

<float.h>

`FLT_RADIX`
: radix of floating-point representations

`FLT_ROUNDS`
: floating-point rounding mode

Where the prefix "`FLT`" pertains to type `float`, "`DBL`" to type `double`, and "`LDBL`" to type `long double`:

`FLT_DIG`
`DBL_DIG`
`LDBL_DIG`
: precision (in decimal digits)

`FLT_EPSILON`
`DBL_EPSILON`
`LDBL_EPSILON`
: smallest number *x* such that `1.0 + x != 1.0`

`FLT_MANT_DIG`
`DBL_MANT_DIG`
`LDBL_MANT_DIG`
: number of digits, base FLT_RADIX, in mantissa

`FLT_MAX`
`DBL_MAX`
`LDBL_MAX`
: maximum number

`FLT_MAX_EXP`
`DBL_MAX_EXP`
`LDBL_MAX_EXP`
: largest positive integer exponent to which FLT_RADIX can be raised and remain representable

`FLT_MIN`
`DBL_MIN`
`LDBL_MIN`
: minimum normalised number

`FLT_MIN_EXP`
`DBL_MIN_EXP`
`LDBL_MIN_EXP`
: smallest negative integer exponent to which FLT_RADIX can be raised and remain representable

<limits.h>

`CHAR_BIT`
: number of bits in a `char`

`CHAR_MAX`
: maximum value of type `char`

`CHAR_MIN`
: minimum value of type `char`

`SCHAR_MAX`
: maximum value of type `signed char`

`SCHAR_MIN`
: minimum value of type `signed char`

`UCHAR_MAX`
: maximum value of type `unsigned char`

`SHRT_MAX`
: maximum value of type `short`

`SHRT_MIN`
: minimum value of type `short`

`USHRT_MAX`
: maximum value of type `unsigned short`

`INT_MAX`
: maximum value of type `int`

`INT_MIN`
: minimum value of type `int`

`UINT_MAX`
: maximum value of type `unsigned int`

`LONG_MAX`
: maximum value of type `long`

`LONG_MIN`
: minimum value of type `long`

`ULONG_MAX`
: maximum value of type `unsigned long`

<locale.h>

`struct lconv`
: Describes formatting of monetary and other numeric values:

  `char* decimal_point;`
  : decimal point for non-monetary values

  `char* grouping;`
  : sizes of digit groups for non-monetary values

  `char* thousands_sep;`
  : separator for digit groups for non-monetary values (left of "decimal point")

  `char* currency_symbol;`
  : currency symbol

  `char* int_curr_symbol;`
  : international currency symbol

  `char* mon_decimal_point;`
  : decimal point for monetary values

  `char* mon_grouping;`
  : sizes of digit groups for monetary values

  `char* mon_thousands_sep;`
  : separator for digit groups for monetary values (left of "decimal point")

`char* negative_sign;`
negative sign for monetary values
`char* positive_sign;`
positive sign for monetary values
`char frac_digits;`
number of digits to be displayed to right of "decimal point" for monetary values
`char int_frac_digits;`
number of digits to be displayed to right of "decimal point" for international monetary values
`char n_cs_precedes;`
whether currency symbol precedes (`1`) or follows (`0`) negative monetary values
`char n_sep_by_space;`
whether currency symbol is (`1`) or is not (`0`) separated by space from negative monetary values
`char n_sign_posn;`
format for negative monetary values:
`0` : parentheses surround quantity and currency symbol
`1` : sign precedes quantity and currency symbol
`2` : sign follows quantity and currency symbol
`3` : sign immediately precedes currency symbol
`4` : sign immediately follows currency symbol
`char p_cs_precedes;`
whether currency symbol precedes (`1`) or follows (`0`) positive monetary values
`char p_sep_by_space;`
whether currency symbol is (`1`) or is not (`0`) separated by space from non-negative monetary values
`char p_sign_posn;`
format for non-negative monetary values, with values as for n_sign_posn
Implementations may change field order and include additional fields. *Standard C Library* functions use only decimal_point.

`struct lconv* localeconv(void);`
returns pointer to formatting information for current locale

`char* setlocale(int category, const char* locale);`
Sets components of locale according to specified *category* and *locale*. Returns string describing new locale or null on error. (Implementations are permitted to define values of *category* additional to those describe here.)

`LC_ALL`
category argument for all categories

`LC_NUMERIC`
category for numeric formatting information

`LC_MONETARY`
category for monetary formatting information

`LC_COLLATE`
category for information affecting collating functions

`LC_CTYPE`
category for information affecting character class tests functions

`LC_TIME`
category for information affecting time conversions functions

`NULL`
null pointer constant

<math.h>

On domain error, implementation-defined value returned and errno set to EDOM. On range error, errno set to ERANGE and return value is HUGE_VAL with correct sign for overflow, or zero for underflow. Angles are in radians.

`HUGE_VAL`
: magnitude returned (with correct sign) on overflow error

`double exp(double x);`
: exponential of *x*

`double log(double x);`
: natural logarithm of *x*

`double log10(double x);`
: base-10 logarithm of *x*

`double pow(double x, double y);`
: *x* raised to power *y*

`double sqrt(double x);`
: square root of *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);`
: absolute value of *x*

`double ldexp(double x, int n);`
: *x* times 2 to the power *n*

`double frexp(double x, int* exp);`
: if *x* non-zero, returns value, with absolute value in interval [1/2, 1), and assigns to `*exp` integer such that product of return value and 2 raised to the power `*exp` equals *x*; if *x* zero, both return value and `*exp` are zero

`double modf(double x, double* ip);`
: returns fractional part and assigns to `*ip` integral part of *x*, both with same sign as *x*

`double fmod(double x, double y);`
: if *y* non-zero, floating-point remainder of *x*/*y*, with same sign as *x*; if *y* zero, result is implementation-defined

`double sin(double x);`
: sine of *x*

`double cos(double x);`
: cosine of *x*

`double tan(double x);`
: tangent of *x*

`double asin(double x);`
: arc-sine of *x*

`double acos(double x);`
: arc-cosine of *x*

`double atan(double x);`
: arc-tangent of *x*

`double atan2(double y, double x);`
: arc-tangent of *y*/*x*

`double sinh(double x);`
: hyperbolic sine of *x*

`double cosh(double x);`
: hyperbolic cosine of *x*

`double tanh(double x);`
: hyperbolic tangent of *x*

<setjmp.h>

`jmp_buf`
: type of object holding context information

`int setjmp(jmp_buf env);`
: Saves context information in *env* and returns zero. Subsequent call to longjmp with same *env* returns non-zero.

`void longjmp(jmp_buf env, int val);`
: Restores context saved by most recent call to setjmp with specified *env*. Execution resumes as a second return from setjmp, with returned value *val* if specified value non-zero, or 1 otherwise.

<signal.h>

`SIGABRT`
: abnormal termination

`SIGFPE`
: arithmetic error

`SIGILL`
: invalid execution

`SIGINT`
: (asynchronous) interactive attention

`SIGSEGV`
: illegal storage access

`SIGTERM`
: (asynchronous) termination request

`SIG_DFL`
: specifies default signal handling

`SIG_ERR`
: signal return value indicating error

`SIG_IGN`
: specifies that signal should be ignored

`void (*signal(int sig, void (*handler)(int)))(int);`
: Install handler for subsequent signal *sig*. If *handler* is SIG_DFL, implementation-defined default behaviour will be used; if SIG_IGN, signal will be ignored; otherwise function pointed to by *handler* will be invoked with argument *sig*. In the last case, handling is **restored to default behaviour** before *handler* is called. If *handler* returns, execution resumes where signal occurred. signal returns the previous handler or `SIG_ERR` on error. Initial state is implementation-defined. Implementations may may define signals additional to those listed here.

`int raise(int sig);`
: Sends signal *sig*. Returns zero on success.

<stdarg.h>

`va_list`
: type of object holding context information

`void va_start(va_list ap, lastarg);`
: Initialisation macro which must be called once before any unnamed argument is accessed. Stores context information in *ap*. *lastarg* is the last named parameter of the function.

type `va_arg(va_list ap, type);`
: Yields value of the type (*type*) and value of the next unnamed argument.

`void va_end(va_list ap);`
: Termination macro which must be called once after argument processing and before exit from function.

<stddef.h>

`NULL`
: Null pointer constant.

`offsetof(`*stype*`, `*m*`)`
: Offset (in bytes) of member *m* from start of structure type *stype*.

`ptrdiff_t`
: Type for objects declared to store result of subtracting pointers.

`size_t`
: Type for objects declared to store result of `sizeof` operator.

<stdio.h>

`BUFSIZ`
: Size of buffer used by setbuf.

`EOF`
: Value used to indicate end-of-stream or to report an error.

`FILENAME_MAX`
: Maximum length required for array of characters to hold a filename.

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

`L_tmpnam`
: Number of characters required for temporary filename generated by tmpnam.

`NULL`
: Null pointer constant.

`SEEK_CUR`
: Value for *origin* argument to fseek specifying current file position.

`SEEK_END`
: Value for *origin* argument to fseek specifying end of file.

`SEEK_SET`
: Value for *origin* argument to fseek specifying beginning of file.

`TMP_MAX`
: Minimum number of unique filenames generated by calls to tmpnam.

`_IOFBF`
: Value for *mode* argument to setvbuf specifying full buffering.

`_IOFBF`
: Value for *mode* argument to setvbuf specifying line buffering.

`_IOFBF`
: Value for *mode* argument to setvbuf specifying no buffering.

`stdin`
: File pointer for standard input stream. Automatically opened when program execution begins.

`stdout`
: File pointer for standard output stream. Automatically opened when program execution begins.

`stderr`
: File pointer for standard error stream. Automatically opened when program execution begins.

`FILE`
: Type of object holding information necessary to control a stream.

`fpos_t`
: Type for objects declared to store file position information.

`size_t`
: Type for objects declared to store result of `sizeof` operator.

FILE* fopen(const char* *filename*, const char* *mode*);
Opens file named *filename* and returns a stream, or NULL on failure.
mode may be one of the following for text files:
"r" : text reading
"w" : text writing
"a" : text append
"r+" : text update (reading and writing)
"w+" : text update, discarding previous content (if any)
"a+" : text append, reading, and writing at end
or one of those strings with b included (after the first character), for binary files.

FILE* freopen(const char* *filename*, const char* *mode*, FILE* *stream*);
Closes file associated with *stream*, then opens file *filename* with specified mode and associates it with *stream*. Returns *stream* or NULL on error.

int fflush(FILE* *stream*);
Flushes stream *stream* and returns zero on success or EOF on error. Effect undefined for input stream. 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 specified file. 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* (if *s* non-null) and returns unique name for a temporary file. Unique name is returned for each of the first TMP_MAX invocations.

int setvbuf(FILE* *stream*, char* *buf*, int *mode*, size_t *size*);
Controls buffering for stream *stream*. *mode* is _IOFBF for full buffering, _IOLBF for line buffering, _IONBF for no buffering. Non-null *buf* specifies buffer of size *size* to be used; otherwise, a buffer is allocated. Returns non-zero on error. Call must be before any other operation on stream.

void setbuf(FILE* *stream*, char* *buf*);
Controls buffering for stream *stream*. For null *buf*, turns off buffering, otherwise equivalent to (void)setvbuf(*stream*, *buf*, _IOFBF, BUFSIZ).

int fprintf(FILE* *stream*, const char* *format*, ...);
Converts (according to format *format*) and writes output to stream *stream*. Number of characters written, or negative value on error, is returned. Conversion specifications consist of:

- %
- (optional) flag:
  - : 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 value, for [eEfgG], always decimal point, for [gG] trailing zeros not removed.
- (optional) minimum width: if specified as *, value taken from next argument (which must be int).
- (optional) . (separating width from precision):

- (optional) 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. If specified as `*`, value taken from next argument (which must be `int`).
- (optional) length modifier:
  `h` : `short` or `unsigned short`
  `l` : `long` or `unsigned long`
  `L` : `long double`
- conversion character:
  `d,i` : `int` argument, printed in signed decimal notation
  `o` : `int` argument, printed in unsigned octal notation
  `x,X` : `int` argument, printed in unsigned hexadecimal notation
  `u` : `int` argument, printed in unsigned decimal notation
  `c` : `int` argument, printed as single character
  `s` : `char*` argument
  `f` : `double` argument, printed with format [-]*mmm.ddd*
  `e,E` : `double` argument, printed with format [-]*m.ddddd*(`e`|`E`)(+|-)*xx*
  `g,G` : `double` argument
  `p` : `void*` argument, printed as pointer
  `n` : `int*` argument : the number of characters written to this point is written *into* argument
  `%` : no argument; prints %

`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. Returns length (excluding the terminating `NUL`).

`int vfprintf(FILE* stream, const char* format, va_list arg);`
Equivalent to fprintf with variable argument list replaced by *arg*, which must have been initialised by the va_start macro (and may have been used in calls to va_arg).

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

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

`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 number of items converted and assigned, or EOF if end-of-file or error occurs before any conversion. Each of the arguments following *format* **must be a pointer**. Format string may contain:

- blanks and tabs, which are ignored
- ordinary characters, which are expected to match next non-white-space of input
- conversion specifications, consisting of:
  - `%`
  - (optional) assignment suppression character "`*`"
  - (optional) maximum field width
  - (optional) target width indicator:
    `h` : argument is pointer to `short` rather than `int`
    `l` : argument is pointer to `long` rather than `int`, or `double` rather than `float`
    `L` : argument is pointer to `long double` rather than `float`

- conversion character:
  - `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; white-space is not skipped, and `NUL`-termination is not performed
  - `s` : string of non-white-space; `char*` parameter required; string is `NUL`-terminated
  - `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 specified set; `char*` parameter required; string is `NUL`-terminated
  - `[^...]` : longest non-empty string not from specified set; `char*` parameter required; string is `NUL`-terminated
  - `%` : 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 (input) stream *stream*, or EOF on end-of-file or error.

`char* fgets(char* s, int n, FILE* stream);`
: Copies characters from (input) stream *stream* to *s*, stopping when *n*-1 characters copied, newline copied, end-of-file reached or error occurs. If no error, *s* is `NUL`-terminated. Returns NULL on end-of-file or error, *s* otherwise.

`int fputc(int c, FILE* stream);`
: Writes *c*, to stream *stream*. Returns *c*, or EOF on error.

`char* fputs(const char* s, FILE* stream);`
: Writes *s*, to (output) stream stream. Returns non-negative on success or EOF on error.

`int getc(FILE* stream);`
: Equivalent to fgetc except that it may be a macro.

`int getchar(void);`
: Equivalent to `getc(stdin)`.

`char* gets(char* s);`
: Copies characters from stdin into *s* until newline encountered, end-of-file reached, or error occurs. Does not copy newline. `NUL`-terminates *s*. Returns *s*, or NULL on end-of-file or error. **Should not be used because of the potential for buffer overflow.**

`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* (excluding terminating `NUL`) and a newline to stdout. Returns non-negative on success, EOF on error.

`int ungetc(int c, FILE* stream);`
: Pushes *c* (which must not be EOF), onto (input) stream *stream* such that it will be returned by the next read. Only one character of pushback is guaranteed (for each 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* and returns number of objects read. (feof and ferror can be used to check 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 number of objects written.

`int fseek(FILE* stream, long offset, int origin);`
: Sets file position for stream *stream* and clears end-of-file indicator. For a binary stream, file position is set to *offset* bytes from the position indicated by *origin*: beginning of file for SEEK_SET, current position for SEEK_CUR, or end of file for SEEK_END. Behaviour is similar for a text stream, but *offset* must be zero or, for SEEK_SET only, a value returned by ftell. Returns non-zero on error.

`long ftell(FILE* stream);`
: Returns current file position for stream *stream*, or `-1` on error.

`void rewind(FILE* stream);`
: Equivalent to `fseek(stream, 0L, SEEK_SET); clearerr(stream).`

`int fgetpos(FILE* stream, fpos_t* ptr);`
: Stores current file position for stream *stream* in **ptr*. 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 end-of-file and error indicators for stream *stream*.

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

`int ferror(FILE* stream);`
: Returns non-zero if error indicator is set for stream *stream*.

`void perror(const char* s);`
: Prints *s* (if non-null) and strerror(errno) to standard error as would:
`fprintf(stderr, "%s: %s\n", (s != NULL ? s : ""), strerror(errno))`

<stdlib.h>

`EXIT_FAILURE`
: Value for *status* argument to exit indicating failure.

`EXIT_SUCCESS`
: Value for *status* argument to exit indicating success.

`RAND_MAX`
: Maximum value returned by rand().

`NULL`
: Null pointer constant.

`div_t`
: Return type of div(). Structure having members:
`int quot;`
quotient
`int rem;`
remainder

`ldiv_t`
: Return type of ldiv(). Structure having members:
`long quot;`
quotient
`long rem;`
remainder

`size_t`
: Type for objects declared to store result of `sizeof` operator.

`int abs(int n);`
`long labs(long n);`
: Returns absolute value of *n*.

`div_t div(int num, int denom);`
`ldiv_t ldiv(long num, long denom);`
: Returns quotient and remainder of *num*/*denom*.

`double atof(const char* s);`
: Equivalent to strtod(*s*, (char**)NULL) except that errno is not necessarily set on conversion error.

`int atoi(const char* s);`
: Equivalent to (int)strtol(*s*, (char**)NULL, 10) except that errno is not necessarily set on conversion error.

`long atol(const char* s);`
: Equivalent to strtol(*s*, (char**)NULL, 10) except that errno is not necessarily set on conversion error.

`double strtod(const char* s, char** endp);`
: Converts initial characters (ignoring leading white space) of *s* to type `double`. If *endp* non-null, stores pointer to unconverted suffix in `*endp`. On overflow, sets errno to ERANGE and returns HUGE_VAL with the appropriate sign; on underflow, sets errno to ERANGE and returns zero; otherwise returns converted value.

`long strtol(const char* s, char** endp, int base);`
: Converts initial characters (ignoring leading white space) of *s* to type `long`. If *endp* non-nu ll, stores pointer to unconverted suffix in `*endp`. If *base* between 2 and 36, that base used for conversion; if zero, leading (after any sign) `0X` or `0x` implies hexadecimal, leading `0` (after any sign) implies octal, otherwise decimal assumed. Leading `0X` or `0x` permitted for base hexadecimal. On overflow, sets errno to ERANGE and returns LONG_MAX or LONG_MIN (as appropriate for sign); otherwise returns converted value.

`unsigned long strtoul(const char* s, char** endp, int base);`
: As for strtol except result is `unsigned long` and value on overflow is ULONG_MAX.

`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 on error.

`void* malloc(size_t size);`
: Returns pointer to *uninitialised* newly-allocated space for an object of size *size*, or NULL on error.

`void* realloc(void* p, size_t size);`
: Returns pointer to newly-allocated space for an object of size *size*, initialised, to minimum of old and new sizes, to existing contents of *p* (if non-null), or NULL on error. On success, old object deallocated, otherwise unchanged.

`void free(void* p);`
: If *p* non-null, deallocates space to which it points.

`void abort();`
: Terminates program abnormally, by calling raise(SIGABRT).

`void exit(int status);`
: Terminates program normally. Functions installed using atexit are called (in reverse order to that in which installed), open files are flushed, open streams are closed and control is returned to environment. *status* is returned to environment in implementation-dependent manner. Zero or EXIT_SUCCESS indicates successful termination and EXIT_FAILURE indicates unsuccessful termination. Implementations may define other values.

`int atexit(void (*fcm)(void));`
: Registers *fcn* to be called when program terminates normally (or when `main` returns). Returns non-zero on failure.

`int system(const char* s);`
: If *s* is not NULL, passes *s* to environment for execution, and returns status reported by command processor; if *s* is NULL, non-zero returned if environment has a command processor.

`char* getenv(const char* name);`
: Returns string associated with name *name* from implementation's environment, 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 ordered array *base* (of *n* objects each of size *size*) for item matching *key* according to comparison function *cmp*. *cmp* must return negative value if first argument is less than second, zero if equal and positive if greater. Items of *base* are assumed to be in ascending order (according to *cmp*). Returns a pointer to an item matching *key*, or NULL if none found.

`void qsort(void* base, size_t n, size_t size, int (*cmp)(const void*, const void*));`
Arranges into ascending order array *base* (of *n* objects each of size *size*) according to comparison function *cmp*. *cmp* must return negative value if first argument is less than second, zero if equal and positive if greater.

`int rand(void);`
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`.

<string.h>

`NULL`
Null pointer constant.

`size_t`
Type for objects declared to store result of `sizeof` operator.

`char* strcpy(char* s, const char* ct);`
Copies *ct* to *s* including terminating `NUL` and returns *s*.

`char* strncpy(char* s, const char* ct, size_t n);`
Copies at most *n* characters of *ct* to *s*. Pads with `NUL` characters if *ct* is of length less than *n*. **Note that this may leave *s* without `NUL`-termination.** Return *s*.

`char* strcat(char* s, const char* ct);`
Concatenate *ct* to *s* and return *s*.

`char* strncat(char* s, const char* ct, size_t n);`
Concatenate at most *n* characters of *ct* to *s*. `NUL`-terminates *s* and return it.

`int strcmp(const char* cs, const char* ct);`
Compares *cs* with *ct*, returning negative value if *cs*<*ct*, zero if *cs*==*ct*, positive value if *cs*>*ct*.

`int strncmp(const char* cs, const char* ct, size_t n);`
Compares at most (the first) *n* characters of *cs* and *ct*, returning negative value if *cs*<*ct*, zero if *cs*==*ct*, positive value if *cs*>*ct*.

`int strcoll(const char* cs, const char* ct);`
Compares *cs* with *ct* according to locale, returning negative value if *cs*<*ct*, zero if *cs*==*ct*, positive value if *cs*>*ct*.

`char* strchr(const char* cs, int c);`
Returns pointer to first occurrence of *c* in *cs*, or NULL if not found.

`char* strrchr(const char* cs, int c);`
Returns pointer to last occurrence of *c* in *cs*, or NULL if not found.

`size_t strspn(const char* cs, const char* ct);`
Returns length of prefix of *cs* which consists of characters which are in *ct*.

`size_t strcspn(const char* cs, const char* ct);`
Returns length of prefix of *cs* which consists of characters which are *not* in *ct*.

`char* strpbrk(const char* cs, const char* ct);`
Returns pointer to first occurrence in *cs* of any character of *ct*, or NULL if none is found.

`char* strstr(const char* cs, const char* ct);`
Returns pointer to first occurrence of *ct* within *cs*, or NULL if none is found.

`size_t strlen(const char* cs);`
Returns length of *cs*.

`char* strerror(int n);`
Returns pointer to implementation-defined message string corresponding with error *n*.
`char* strtok(char* s, const char* t);`
Searches *s* for next token delimited by any character from *ct*. Non-NULL *s* indicates the first call of a sequence. If a token is found, it is `NUL`-terminated and returned, otherwise NULL is returned. *ct* need not be identical for each call in a sequence.
`size_t strxfrm(char* s, const char* ct, size_t n);`
Stores in *s* no more than *n* characters (including terminating `NUL`) of a string produced from *ct* according to a locale-specific transformation. Returns length of *entire* transformed string.
`void* memcpy(void* s, const void* ct, size_t n);`
Copies *n* characters from *ct* to *s* and returns *s*. ***s* may be corrupted if objects overlap.**
`void* memmove(void* s, const void* ct, size_t n);`
Copies *n* characters from *ct* to *s* and returns *s*. ***s* will not be corrupted if objects overlap.**
`int memcmp(const void* cs, const void* ct, size_t n);`
Compares at most (the first) *n* characters of *cs* and *ct*, returning negative value if *cs*<*ct*, zero if *cs*==*ct*, positive value if *cs*>*ct*.
`void* memchr(const void* cs, int c, size_t n);`
Returns pointer to first occurrence of *c* in first *n* characters of *cs*, or NULL if not found.
`void* memset(void* s, int c, size_t n);`
Replaces each of the first *n* characters of *s* by *c* and returns *s*.

<time.h>

`CLOCKS_PER_SEC`
The number of clock_t units per second.
`NULL`
Null pointer constant.
`clock_t`
An arithmetic type elapsed processor representing time.
`time_t`
An arithmetic type representing calendar 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_mon;`
months **since** January
`int tm_year;`
years since 1900
`int tm_wday;`
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 not known.
Implementations may change field order and include additional fields.

clock_t clock(void);
Returns elapsed 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 in seconds between *time2* and *time1*.

time_t mktime(struct tm* *tp*);
If necessary, adjusts fields of **tp* to fall withing normal ranges. Returns the corresponding calendar time, 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 13:08:42 1988\n\0

char* ctime(const time_t* *tp*);
Returns string equivalent to calendar time *tp* converted to local time. Equivalent to:
asctime(localtime(tp))

struct tm* gmtime(const time_t* *tp*);
Returns calendar time **tp* converted to 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*. Places no more than *smax* characters into *s*, and returns number of characters produced (excluding terminating NUL), or 0 if greater than *smax*. Formatting conversions (%*c*) are:
A : name of weekday
a : abbreviated name of weekday
B : name of month
b : abbreviated name of month
c : local date and time representation
d : day of month [01-31]
H : hour (24-hour clock) [00-23]
I : hour (12-hour clock) [01-12]
j : day of year [001-366]
M : minute [00-59]
m : month [01-12]
p : local equivalent of "AM" or "PM"
S : second [00-61]
U : week number of year (Sunday as 1st day of week) [00-53]
W : week number of year (Monday as 1st day of week) [00-53]
w : weekday (Sunday as 0) [0-6]
X : local time representation
x : local date representation
Y : year with century
y : year without century [00-99]
Z : name (if any) of time zone
% : %

Local time may differ from *calendar* time because of time zone.