C Memoire

[pointers 4](#_Toc74075778)

[arrays are not pointers 4](#_Toc74075779)

[Preprocessors 5](#_Toc74075780)

[C-strings 6](#_Toc74075781)

[errno 7](#_Toc74075782)

[<stdio.h> 8](#_Toc74075783)

[file operations 8](#_Toc74075784)

[file access 8](#_Toc74075785)

[formatted I/O 9](#_Toc74075786)

[character I/O 10](#_Toc74075787)

[direct I/O 11](#_Toc74075788)

[file positioning 11](#_Toc74075789)

[error-handling 11](#_Toc74075790)

[pipe to/from process 12](#_Toc74075791)

[macros 12](#_Toc74075792)

[types 12](#_Toc74075793)

[stream properties 13](#_Toc74075794)

[formatting 13](#_Toc74075795)

[wide characters 14](#_Toc74075796)

[read multiple lines 14](#_Toc74075797)

[<stdlib.h> 15](#_Toc74075798)

[string conversion 15](#_Toc74075799)

[random 15](#_Toc74075800)

[memory management 15](#_Toc74075801)

[environment 16](#_Toc74075802)

[searching and sorting 17](#_Toc74075803)

[compar function 17](#_Toc74075804)

[integer functions 17](#_Toc74075805)

[multibyte characters 17](#_Toc74075806)

[multibyte strings 18](#_Toc74075807)

[types and constants 18](#_Toc74075808)

[Get output of system() command 19](#_Toc74075809)

[array\_size macro 19](#_Toc74075810)

[get environment 19](#_Toc74075811)

[qsort() usage example 20](#_Toc74075812)

[random number range 20](#_Toc74075813)

[<string.h> 21](#_Toc74075814)

[copying 21](#_Toc74075815)

[concatenation 21](#_Toc74075816)

[comparison 21](#_Toc74075817)

[searching 22](#_Toc74075818)

[other 22](#_Toc74075819)

[strtok() usage 23](#_Toc74075820)

[strsep() usage 23](#_Toc74075821)

[<wchar.h> 24](#_Toc74075822)

[I/O 24](#_Toc74075823)

[utils 24](#_Toc74075824)

[conversions 24](#_Toc74075825)

[strings 24](#_Toc74075826)

[time 25](#_Toc74075827)

[types and constants 25](#_Toc74075828)

[<wctype.h> 26](#_Toc74075829)

[classification 26](#_Toc74075830)

[conversion 26](#_Toc74075831)

[extensible classification/conversion 26](#_Toc74075832)

[types and constants 26](#_Toc74075833)

[<math.h> 27](#_Toc74075834)

[trigonometric functions 27](#_Toc74075835)

[hyperbolic functions 27](#_Toc74075836)

[expodential/logarithmic functions 27](#_Toc74075837)

[power functions 27](#_Toc74075838)

[error/gamma functions 27](#_Toc74075839)

[rounding/remainder functions 27](#_Toc74075840)

[floating-point manipulation functions 28](#_Toc74075841)

[max/min/difference functions 28](#_Toc74075842)

[other functions 28](#_Toc74075843)

[classification macros 28](#_Toc74075844)

[comparison macros 28](#_Toc74075845)

[macro constants 28](#_Toc74075846)

[types 28](#_Toc74075847)

[<complex.h> 29](#_Toc74075848)

[<ctype.h> 30](#_Toc74075849)

[classification 30](#_Toc74075850)

[conversion 30](#_Toc74075851)

[<time.h> 31](#_Toc74075852)

[terminology 31](#_Toc74075853)

[time manipulations 31](#_Toc74075854)

[conversion 31](#_Toc74075855)

[<other> 32](#_Toc74075856)

[types and constants 32](#_Toc74075857)

[strftime() format specifiers 32](#_Toc74075858)

[<assert.h> 34](#_Toc74075859)

[<signal.h> 34](#_Toc74075860)

[functions 34](#_Toc74075861)

[types 34](#_Toc74075862)

[<setjmp.h> 34](#_Toc74075863)

[<limits.h> 35](#_Toc74075864)

[<unistd.h> 36](#_Toc74075865)

[key items 36](#_Toc74075866)

[<getopt.h> 37](#_Toc74075867)

[C vs C++ 38](#_Toc74075868)

[gcc built-in functions 39](#_Toc74075869)

# pointers

Whitespace around the asterisk doesn’t make a difference to the compiler. That, is, the following are equivalent:

int\* foo

int \*foo

int \* foo

int \* foo

However, this may cause ambiguity. Consider the following (equivalent) statements – declaring a char pointer and a char – the latter makes this fact more obvious. It may be good practice not to declare pointer and non-pointer variables in the same statement.

char\* str1, str2; // str1 is a char\*, str2 is a char

char \*str1, str2;

## arrays are not pointers

sizeof(pointer)

Is always the same, regardless of number, or type, of elements.

sizeof(array)

Depends on both the size of the array, and the type <product of both?>.

Cannot do pointer arithmetic with arrays.

Arrays cannot have zero length.

The misconception that arrays and pointers are the same stems from the fact that the following are the same:

void foo(char\* x);

void foo(char x[]);

(However), In both examples, *x* is a pointer, not an array.

Using a pointer to an array, as an array:

# Preprocessors

The C pre-processor is a text substitution tool, which instructs the compiler to do the required pre-processing before actual compilation.

#define <name> <value>

Substitute a preprocessor macro – replace instances of name with value

#include <lib.h>

If *‘lib.h’* is in chevrons, add text from corresponding system library. If it is in double quotes, add corresponding library from local directory.

#undef

Undefine a preprocessor macro

#ifdef

Begin conditional block, return true if *name* is already defined.

#ifndef <name>

Begin conditional block, return true if *name* is not already defined.

#if

Begin conditional block.

#else

Alternative condition for *‘if’*.

#elif

*‘else’* and *‘if’* in one statement.

#endif

End conditional block.

#error

Print error message on *stderr*

#pragma

Issue special command to compiler

\_\_DATE\_\_

The current date as character literal in *‘MMM DD YYYY’* format

\_\_TIME\_\_

The current time as character literal in *‘HH:MM:SS’* format

\_\_FILE\_\_

The current filename as string literal

\_\_LINE\_\_

The current line number as a decimal constant

\_\_STDC\_\_

Defined as *1* when the compiler complies with the ANSI standard

# C-strings

char \*a = “abc”;

while allowable, should actually be:

const char \*a = “abc”;

(?)

# buffer overflow

Functions to avoid, and their alternatives:

strcpy() strncpy()

strcat() strncat()

sprint() snprintf()

vsprintf()

gets() fgets()

Do not use *strlen()* unless argument (C string) is known to contain a null terminator.

# errno

# [<stdio.h>](https://www.cplusplus.com/reference/cstdio/)

## file operations

int remove(*const char\* filename*)

Delete the file whose name is specified in C-string *filename*. Returns zero if successful, otherwise return non-zero and set *errno*.

int rename(*const char\* oldname, const char\* newname*)

Change name of file or directory specified by C-string *oldname* to C-string *newname*. If *oldname* and *newname* specify different paths, move the file. Return zero if successful, otherwise non-zero and set *errno*.

FILE\* tmpfile(*void*)

Create a temporary binary file, open for update (*‘wb’* mode), with filename guaranteed to be different from any other existing file. This file is deleted when the stream is closed, *fclose()*, or when program terminates. Returns stream pointer to temporary file, or null on failure.

char\* tmpnam(*char\* str*)

Return a (pointer to a) C-string containing a filename different from the name of any existing file, and thus suitable to safely create a temporary file. If *str* is a null pointer, the resulting path is stored in an internal buffer, otherwise *str* is returned if successful. Returns null on failure.

int fileno(*FILE\* stream*)

Return the integer file-descriptor associated with the stream pointed to by *stream*.

## file access

int fclose(*FILE \*stream*)

Close the file associated with file pointer *stream* and disassociate it. All internal buffers associated with stream are flushed, the contents of any unwritten output buffer is written, and the content of any unread input buffer is discarded. Returns zero if successful, otherwise *EOF*.

int fflush(*FILE \*stream*)

If the given file pointer *stream* was open for writing (or the last operation was an output), any unwritten data in buffer is written to file. If *stream* is a null pointer, flush all streams. The stream remains open after call. Returns zero if successful, otherwise return *EOF* and set *stream* indicator (which is checked with *ferror()*).

FILE\* fopen(*const char\* filename, const char\* mode*)

Opens the file whose name is specified by *filename*, and return file pointer to stream. The stream is fully buffered by default (set with *setbuf()*). Closed with *fclose()*, or *freopen()*, or upon normal program termination. Up to *FOPEN\_MAX* files may be open simultaneously <by: system/program?>  
Access *mode* values: *“r”* Read, *“w”* Write, *“a”* Append, *“r+*” Read/Update, *“w+”* Write/Update, *“a+*” Append/Update

Include *“b”* (*ie: “wb”, “rb”*) for binary mode. (C2011, include *“x”* to fail if file exists).

Returns null pointer on failure, and set *errno*.

FILE\* freopen(*cvonst char\* filename, const char\* mode, FILE\* stream*)

Closes any file associated with file pointer *stream*, then open file associated with C-string *filename* as file pointer *stream*, with specified *mode* (same options as *fopen()*). If *filename* is null, re-open file already associated with *stream* in specified *mode*. If successful, returns *stream*, otherwise return null, and set *errno*. This function is useful for redirecting predefined streams, such as *stdin*, *stdout*, and *stderr*.

void setbuf(*FILE\* stream, char\* buffer*)

Use *setvbuf()* instead. Specifies the *buffer* to be used by *stream* for I/O operations, which becomes fully-buffered stream. Or, if *buffer* is null, *stream* becomes an unbuffered stream. Function should be used after *stream* has been associated with an open file, but before performing I/O. Buffer is assumed to be at least *BUFSIZ* bytes (default 1024). All files are opened with a default allocated buffer (fully-buffered) if they are known to not refer to an interactive device. Otherwise they may either be line-buffered of unbuffered.

int setvbuf(*FILE\* stream, char\* buffer, int mode, size\_t size*)

Specifies a *buffer* for *stream*. Also specify *mode*, and *size* of the buffer (in bytes). If *buffer* is null, automatically allocate *size* bytes. Function should be used after *stream* has been associated with an open file, but before performing I/O. All files are opened with a default allocated buffer (fully-buffered) if they are known to not refer to an interactive device. Otherwise they may either be line-buffered or unbuffered.

\_IOFBF

\_IOLBF

\_IONBF

*setvbuf()* modes, in order: Full buffering, Line buffering, and No buffering

## formatted I/O

int fprintf(*FILE\* stream, const char\* format, …*)

Write the C-string pointed by *format* to *stream*, with any *specifiers* (*%*) replaced with additional arguments following *format*. Function expects at least as many additional arguments as *format* contains specifiers. If successful, return number of characters written. Returns negative and sets relevant *stream* indicator (check with *ferror()*) if writing error encountered. If encoding error encountered writing wide-characters, set *errno* to *EILSEQ*.

int fscanf(*FILE \*restrict stream, const char \*restrict format, …*)

Reads data from *stream*, and store values according to the parameter *format* into the locations pointed to by the additional arguments (which should be pointers to initialized storage of correct type). There should be at least as many additional arguments as *format* contains specifiers. Returns int as many items as are filled according to *format*. Returns *EOF* and sets relevant *stream* indicator (check with *feof()* or *ferror()*) if reading error or *EOF* encountered while reading. If encoding error encountered interpreting wide-characters, set *errno* to *EILSEQ*.

int printf(*const char\* format, …*)

As per *fprintf()*, writing to *stdout*. If successful, return number of bytes written. If writing error occurs, set r elevant *stream* indicator (check with *ferror()*) and return negative value. If encoding error encountered interpreting wide-characters, set *errno* to *EILSEQ*.

int scanf(*const char \*restrict format, …*)

Read data from *stdin*, and store according to parameters in *format* into the locations pointed to by the additional arguments. There should be at least as many additional arguments as *format* contains specifiers. Returns int as many items as are filled according to *format*. Returns *EOF* and sets relevant *stream* indicator (check with *feof()* or *ferror()*) if reading error or *EOF* encountered while reading. If encoding error encountered interpreting wide-characters, set *errno* to *EILSEQ*.

int snprintf(*char\* s, size\_t n, const char\* format, …*)

Compose a string with the same text that would be printed by *printf()*, and store as C-string in buffer pointed to by *s* (up to *n* bytes, including null terminator, that is, *n-1* bytes are read). There should be at least as many additional arguments as *format* contains specifiers. Buffer *s* should be at least *n* bytes. Returns number of characters that would have been written had *n* been sufficiently large, excluding null terminator. Returns negative for encoding error.

int sprint(*char \*str, const char\* format, …*)

As per *snprintf()*, without length checking.

int sscanf(*const char \*restrict buffer, const char \*restrict format, …*)

Reads data from C-string *s*, and store according to parameters in *format* into the locations pointed to by the additional arguments. There should be at least as many additional arguments as *format* contains specifiers. Returns int as many items as are filled according to *format*, or *EOF* in-case of input failure.

int vfprintf(*FILE\* stream, const char\* format, va\_list arg*)

Write the C-string pointed to by *format* to *stream*, replacing any specifiers as per *printf()*, but using elements in *va\_list arg* instead of additional arguments. Returns int as many items as are filled according to *format*. If writing error occurs, return negative, and set *stream* error indicator (check with *ferror()*). If multibyte character encoding encountered, set *errno* to *ELISEQ* and return negative.

int vfscanf(*FILE\* stream, const char\* format, va\_list arg*)

Reads data from *stream*, and store according to parameters in *format* into locations identified by *arg*. Returns int as many items as are filled according to *format*. If reading error occurs or *EOF* encounter while reading, set *stream* error indicator (check with *feof()* or *ferror()*), and return *EOF* if function failed to read data. If encoding error happens interpreting wide characters, set *errno* to *EILSEQ*.

int vprintf(*const char\* format, va\_list arg*)

As per *printf()*, but using *arg* instead of additional arguments.

int vsnprintf(*char\* s, size\_t n, const char\* format, va\_list arg*)

As per *snprintf()*, but using *arg* instead of additional arguments.

int vsprintf(*char\* s, const char\* format, va\_list arg*)

As per *sprint()*, but using arg instead of additional arguments.

int vsscanf(*const char\* s, const char\* format, va\_list arg*)

As per *sscanf()*, but using *arg* instead of additional arguments.

## character I/O

int fgetc(*FILE\* stream*)

Returns (as type int) the character currently pointed to by the internal position indicator of *stream*, and advance indicator to next character. Return *EOF* and set indicator for stream (check with *feof()*) if *EOF* encountered when called. If other error encountered, return *EOF* and set indicator (check with *ferror()*). Equivalent to *getc()*

char\* fgets(*char\* str, int num, FILE\* stream*)

Reads characters from *stream* and stores them as C-string in *str* until *num-1* (final character is the null terminator) characters are read, or *EOF* or newline have been read (newline will be read into *str*). Then append null terminator to *str*. On success, return *str*. On *EOF* while attempting to read, set *stream* indicator (check with *feof()*). Return null on error preventing reading of values. Set *stream* indicator (check with *ferror()*) on read error.

int fputc(*int character, FILE\* stream*)

Write *character* to *stream*, and advance position indicator. On success, return *character*. On error, return *EOF*, and set *stream* indicator (check with *ferror()*).

int fputs(*const char\* str, FILE\* stream*)

Write the C-string *str* to *stream*. Copies from address specified *str* until reaching null terminator (which is not written). Returns non-negative value on success. On error, return *EOF*, and set *stream* indicator (check with *ferror()*). Unlike *puts()*, does not append newlines.

int getc(*FILE\* stream*)

Return (as type int) the character currently pointed to by the internal position indicator of *stream*, and advance indicator to next character. Return *EOF* and set indicator for stream (check with *feof()*) if *EOF* encountered when called. If other error encountered, return *EOF* and set indicator (check with *ferror()*).

int getchar(*void*)

Equivalent to *getc(stdin)*.

char\* gets(*char\* str*)

Deprecated/removed in newer versions of C/C++. Reads characters from *stdin* and stores them as C-strings in *str* until newline or *EOF* reached. Does not specify buffer maximum size. Newline is not copied. On success, returns *str*. If *EOF* while reading file, set *stream* indicator (check with *feof()*). If read error occurs, set *stream* indicator (check with *ferror()*). If unable to read values, returns null.

int putc(*int character, FILE\* stream*)

Write *character* to *stream*, and advance position indicator. On success, return *character*. If writing error occurs, return *EOF*, and set *stream* indicator (check with *ferror()*).

int putchar(*int character*)

Equivalent to *putc(character, stdout)*.

int puts(*const char\* str*)

Write the C-string *str* to *stdout*, and append a newline. Copies from *str* until null terminating character is encountered. Null terminator is not written. Unlike *fputs()*, does append newline.

int ungetc(*int character, FILE\* stream*)

A character is virtually put back into an input *stream*, decreasing its internal file position as if a previous *getc()* operation was undone. Note: this only affects further input operations on *stream*, and not content of physical file associated with it, which is not modified by function. If successful, clears *stream EOF* indicator (if it was set), and decrements internal file position indicator (if in binary mode – in text mode, position indicator is undefined until all characters put back with *ungetc()* have been read/discarded. Call to any of *fseek()*, *fsetpos()*, or *rewind()* on *stream* will discard characters previous put back with function. Fails if given *EOF* as *character*. On success, return *character* that has been put back. On failure, return *EOF*.

## direct I/O

size\_t fread(*void\* ptr, size\_t size, size\_t count, FILE\* stream*)

Read an array of *count* elements, each one with a size of *size* bytes, from *stream*, and store in block of memory pointed to by *ptr*. Position indicator of stream is advanced by number of bytes read (*size\*count*). If successful, return number of elements successfully read. (If this number differs from *count* parameter, either reading error occurred, or *EOF* was reached. In either case, respective *stream* indicators are set (check with *ferror()*), (check with *feof()*).

size\_t fwrite(*const void\* ptr, size\_t size, size\_t count, FILE\* stream*)

Write an array of *count* elements, each with size of *size* bytes, from block of memory pointed to by *ptr*, to the current position in the stream. The position indicator is advanced by as many bytes as are written. Internally, is equivalent to calling *fputc()* for each byte of *ptr* for (*size\*count*) elements. If successful, return number of elements written (equal to *count* if successful). If writing error prevents function from completing set *stream* indicator (check with *ferror()*). Returns zero without error if *size* or *count* are zero.

## file positioning

int fgetpos(*FILE \*stream, fpos\_t\* pos*)

Retrieves the current position in *stream*. Fills *fpos\_t* (which should be initialized) with information needed from *stream* position indicator to restore the stream to its current position (and multibyte state, if wide-oriented) with a call to *fsetpos()*. On success, returns zero, in case of error, return negative, and set *errno*. Alternatively, *ftell()* retrieves *stream* position as integer.

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

Sets the position indicator associated with *stream* to new position. In binary mode, position is defined by adding *offset* as per *origin* (which must be one of: *SEEK\_SET, SEEK\_CUR, SEEK\_END*). In text mode, *offset* must be either zero or value previously returned by *ftell()*, and *origin* shall be *SEEK\_SET*. Unsets *EOF* indicator, if set, and undoes effects of *ungetc*(). On *stream* open for update (*“r+w”*), a call to function allows switch between reading and writing. If successful, returns zero, otherwise return non-zero. If error occurs, set *stream* indicator (check with *ferror()*).

int fsetpos(*FILE\* stream, const fpos\_t\* pos*)

Restores the current position in *stream* to *pos*, which is a pointer to an *fpos\_t* objects whose value has been previously obtained from *fgetpos()*. Unset *EOF* indicator, if set, and undoes effects of *ungetc()*. If successful, return zero, upon failure, return non-zero, and set *errno*.

long int ftell(*FILE\* stream*)

Return current value of the position indicator of *stream*. For binary streams, this is the number of bytes from the beginning of the file. The value may not be meaningful when used with text streams (but it can be used with *fseek()*).

void rewind(*FILE\* stream*)

Set the position indicator associated with *stream* to beginning of the file. Unset *EOF* indicator, if set, and undoes effects of *ungetc()*. On *stream* open for update (*“r+w”*), a call to function allows switch between reading and writing.

SEEK\_CUR

SEEK\_END

SEEK\_SET

*fseek()* constants, in order: Current position of the file pointer, End of file, Beginning of file

## error-handling

void clearerr(*FILE\* stream*)

Reset both the *error* and the *eof* indicators of *stream*.

int feof(*FILE\* stream*)

Check whether *EOF* indicator associated with stream is set. Returns non-zero if indicator is set, zero otherwise.

int ferror(*FILE\* stream*)

Check whether *error* indicator associated with stream is set. Returns non-zero if indicator is set, zero otherwise.

void perror(*const char\* str*)

Interprets the value of *errno* as an error message, and prints it to *stderr*, optionally preceding it with custom message *str* (if it is not null). Error messages are platform dependent. Function should be used right after error was produced, lest it said error be overwritten by subsequent error.

## pipe to/from process

FILE\* popen(*const char\* command, const char\* mode*)

Open a bidirectional pipe, fork, and invoke the shell. Any streams opened by previous *popen()* calls are closed. The *command* argument is.a C-string containing a shell command to be executed. Returns a standard IO stream (which must be closed with *pclose()*, not *fclose()*), or null if call to fork or pipe fails.

int pclose(*FILE\* stream*)

Wait for process associated with stream to terminate, then close stream. Returns exit status of command, or negative one if stream is not associated with command, is already closed, or otherwise unable to be closed.

## macros

BUFSIZE

Size of buffer used by *setbuf()* function.

EOF

Negative int (typically -1), denoting end-of-file, or an invalid character.

FILENAME\_MAX

Size needed for an array of *char* elements, to hold the longest file name C-string allowed, or recommended value (1024) if there is no such limit.

FOPEN\_MAX

Number of files that can be opened simultaneously <by: system/application?>

L\_tmpnam

Size need for array of *char* elements, to hold the longest filename C-string generated by *tmpnam()*

NULL

Null Pointer constant, zero. Indicates pointer is not pointing to an object.

TMP\_MAX

Minimum number of unique temporary filenames that are guaranteed to be possible to generate using *tmpnam()*.

## types

FILE

Object type that identifies a stream, and contains information need to control it, including pointer to its buffer, its position indicator, and all its state indicators. Contents of stream is generally not meant to be accessed outside of using functions provided by *stdio.h* and *wchar.h*. Created with call to either *fopen()* or *tmpfile()*, and closed with *fclose()*. Three such streams are created on import of *stdio.h*: *stdin*, *stdout*, and *stderr*.

fpos\_T

Object type that specifies a position within a file. Filled by *fgetpos()*, and used by *fsetpos()*.

size\_t

Alias for unsigned-int. Used to represent size of a given object in bytes.

## stream properties

Streams are an abstraction to interact with input/output devices. Handled as pointers to *FILE* objects. Standard streams are *stdin*, *stdout*, and *stderr*.

Read/Write access

Whether stream has read and/or write access to media they are associated with

Text/Binary

Text streams are thought to represent a set of text lines, each one ending with a newline. Some character translation may take place depending on environment. A binary stream is a sequence of characters (bytes) read/written with no translation.

Buffer

A buffer is a block of memory where data is accumulated before being physically read/written to associated file/device. Streams can be fully buffered (data written when buffer filled), line-buffered (data written when newline encountered), or unbuffered (written as soon as possible).

Orientation:

Streams have no orientation on opening. As soon as I/O is performed, they become either byte-oriented, or wide-oriented, depending on operation performed (generally *stdio* functions are byte-orientated, *cwchar* functions are wide-oriented).

Indicators:

Streams have internal indicators that specify their current state and which affect the behaviour of some input and output performed on them.

Error indicator – set when error occurs related to stream. Check with *ferror()*, reset with *clearer(), freopen(),* or *rewind().*

End-of-file indicator – indicates that last reading/writing operation performed reached EOF. Check with *feof()*, reset with call to *clearer()*, *freopen()*, *rewind()*, *fseek()*, *fsetpos()*

Position indicator – internal pointer, points to the next character to be read or written in the next I/O operation. Get value with *ftell()*, *fgetpos()*, and changed with *rewind()*, *fseek()*, *fsetpos()*

## formatting

%[flags][width][.precision][length]specifier

Format specifier prototype

|  |  |  |
| --- | --- | --- |
| ***specifier*** | **Output** | **Example** |
| d *or* i | Signed decimal integer | 392 |
| u | Unsigned decimal integer | 7235 |
| o | Unsigned octal | 610 |
| x | Unsigned hexadecimal integer | 7fa |
| X | Unsigned hexadecimal integer (uppercase) | 7FA |
| f | Decimal floating point, lowercase | 392.65 |
| F | Decimal floating point, uppercase | 392.65 |
| e | Scientific notation (mantissa/exponent), lowercase | 3.9265e+2 |
| E | Scientific notation (mantissa/exponent), uppercase | 3.9265E+2 |
| g | Use the shortest representation: %e or %f | 392.65 |
| G | Use the shortest representation: %E or %F | 392.65 |
| a | Hexadecimal floating point, lowercase | -0xc.90fep-2 |
| A | Hexadecimal floating point, uppercase | -0XC.90FEP-2 |
| c | Character | a |
| s | String of characters | sample |
| p | Pointer address | b8000000 |
| n | Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location. |  |
| % | A % followed by another % character will write a single % to the stream. | % |

The type of the argument is given by *specifier*.

|  |  |
| --- | --- |
| ***flags*** | **description** |
| - | Left-justify within the given field width; Right justification is the default (see *width* sub-specifier). |
| + | Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign. |
| *(space)* | If no sign is going to be written, a blank space is inserted before the value. |
| # | Used with o, x or X specifiers the value is preceeded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written. |
| 0 | Left-pads the number with zeroes (0) instead of spaces when padding is specified (see *width* sub-specifier). |

|  |  |
| --- | --- |
| ***width*** | **description** |
| *(number)* | Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger. |
| \* | The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted. |

|  |  |
| --- | --- |
| ***.precision*** | **description** |
| .*number* | For integer specifiers (d, i, o, u, x, X): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, 0 is assumed. |
| .\* | The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted. |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **specifiers** | | | | | | |
| ***length*** | **d i** | **u o x X** | **f F e E g G a A** | **c** | **s** | **p** | **n** |
| *(none)* | int | unsigned int | double | int | char\* | void\* | int\* |
| hh | signed char | unsigned char |  |  |  |  | signed char\* |
| h | short int | unsigned short int |  |  |  |  | short int\* |
| l | long int | unsigned long int |  | [wint\_t](https://www.cplusplus.com/wint_t) | wchar\_t\* |  | long int\* |
| ll | long long int | unsigned long long int |  |  |  |  | long long int\* |
| j | [intmax\_t](https://www.cplusplus.com/intmax_t) | [uintmax\_t](https://www.cplusplus.com/uintmax_t) |  |  |  |  | [intmax\_t](https://www.cplusplus.com/intmax_t)\* |
| z | [size\_t](https://www.cplusplus.com/size_t) | [size\_t](https://www.cplusplus.com/size_t) |  |  |  |  | [size\_t](https://www.cplusplus.com/size_t)\* |
| t | [ptrdiff\_t](https://www.cplusplus.com/ptrdiff_t) | [ptrdiff\_t](https://www.cplusplus.com/ptrdiff_t) |  |  |  |  | [ptrdiff\_t](https://www.cplusplus.com/ptrdiff_t)\* |
| L |  |  | long double |  |  |  |  |

The *length* sub-specifier modifies the length of the datatype.

## wide characters

## read multiple lines

Use *fgetc()* instead of *fscanf()* in order to read multiple lines of input.

char \_str[STRLEN] = "";

char input\_char;

int chars\_remaining = STRLEN - 1;

while (chars\_remaining > 0 && (input\_char = fgetc(stdin)) != EOF) {

strcat(\_str, &input\_char);

chars\_remaining -= strlen(&input\_char);

}

# <stdio\_ext.h>

Available for Linux, not for macOS.

# [<stdlib.h>](https://www.cplusplus.com/reference/cstdlib/)

## string conversion

double atof(*const char\* str*)

Parse C-string *str* as floating pointer number, and return result as double. Zero is returned if *str* is not valid for conversion. Result is undefined if converted value is outside range of float.

int atoi(*const char\* str*)

Parse C-string as integer, and return result as int. Zero is returned if *str* is not valid for conversion. Result is undefined if converted value is outside range of int.

long int atol(*const char\* str*)

Parse C-string as integer, and return result as long int. Zero is returned if *str* is not valid for conversion. Result is undefined if converted value is outside range of long int.

long long int atoll(*const char\* str*)

Parse C-string as integer, and return result as long long int. Zero is returned if *str* is not valid for conversion. Result is undefined if converted value is outside range of long long int.

double strtod(*const char\* str, char\*\* endptr*)

Parse C-string *str* as float, and return result as double. If *endptr* is not null, set it to point to the first character after the number. Zero is returned if *str* is not valid for conversion. Returns *HUGE\_VAL* if converted value is outside range of double, and sets *errno* to *ERANGE*.

long int strtol(*const char\* str, char\*\* endptr, int base*)

Parse C-string *str* as integer of specified base, and return as long int. If *endptr* is not null, set it to point to the first character after the number. Zero is returned if *str* is not valid for conversion. Returns *LONG\_MAX* or *LONG\_MIN* if converted value is outside range of long int, and set *errno* to *ERANGE*.

long double strtold(*const char\* str, char\*\* endptr*)

Parse C-string *str* as float, and return as long double. If *endptr* is not null, set it to point to the first character after the number. Zero is returned if *str* is not valid for conversion. Returns positive/negative *HUGE\_VAL* if converted value is outside range of long double, and set *errno* to *ERANGE*.

long long int strtoll(*const char\* str, char\*\* endptr, int base*)

Parse C-string *str* as integer of specified base, and return as long long int. If *endptr* is not null, set it to point to the first character after the number. Zero is returned if *str* is not valid for conversion. Returns *LLONG\_MAX* or *LLONG\_MIN* if converted value is outside range of long int, and set *errno* to *ERANGE*.

unsigned long int strtoul(*const char\* str, char\*\* endptr, int base*)

Parse C-string *str* as integer of specified base, and return as unsigned long int. If *endptr* is not null, set it to point to the first character after the number. Zero is returned if *str* is not valid for conversion. Returns *ULONG\_MAX* if converted value is outside range of unsigned long int, and set *errno* to *ERANGE*.

unsigned long long int strtoull(*const char\* str, char\*\* endptr, int base*)

Parse C-string *str* as integer, and return as unsigned long long int. If *endptr* is not null, set it to point to the first character after the number. Zero is returned if *str* is not valid for conversion. Returns *ULLONG\_MAX* if converted value is outside range of unsigned long long int, and set *errno* to *ERANGE*.

## random

int rand(*void*)

Return a pseudo-random integer in the range *[0, RAND\_MAX]*

void srand(*unsigned int seed*)

Initialize RNG with given seed.

## memory management

void\* calloc(*size\_t num, size\_t size*)

Allocate a block of memory for an array of *num* elements, each of them *size* bytes long, and initialize all bytes to zero. Returns pointer to allocated memory, or null pointer upon failure. Must be released with *free()*.

void free(*void\* ptr*)

Deallocate a block of memory previously allocated by *malloc(), calloc(),* or *realloc()*. Behaviour is undefined if pointer does not point to memory allocated with said functions. Does nothing if given a null pointer.

void\* malloc(*size\_t size*)

Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block, or a null pointer upon failure. Contents of memory is indeterminate. Must be released with *free()*.

void\* realloc(*void\* ptr, size\_t size*)

Changes the size of the memory block pointed to by *ptr*. The contents of the memory block is preserved up to the lesser of the new and old sizes, even if the block is moved to a new location. If the new size is larger, the value of the newly allocated portion is indeterminate. Returns a pointer to the beginning of the new block. Behaves like *malloc()* if *ptr* is null. Must be released with *free()*.

## environment

void abort(*void*)

Abort the current process, producing an abnormal program termination. Raises *SIGABRT* signal. Program is terminated without destroying any objects, and without calling any of the functions passed to *atexit()* or *at\_quick\_exit()*.

int atexit(*void (\*func) (void)*)

The function pointed by *func* is automatically called without arguments when the program terminates normally. If multiple calls specifying functions are passed to *atexit()*, they are executed in reverse order. Return zero if function successfully registered.

int at\_quick\_exit(*void (\*func) (void)*)

The function pointed by *func* is automatically called without arguments when *quick\_exit()* is called. If multiple calls specifying functions are passed to *at\_quick\_exit()*, they are executed in reverse order. Returns zero if function successfully registered.

void exit(*int status*)

Terminates the process normally, performing the regular clean-up for terminating programs. If *status* is zero or *EXIT\_SUCCESS,* a successful termination status is returned to host environment. If *status* is EXIT\_FAILURE, an unsuccessful termination status is returned to the host environment.

char\* getenv(*const char\* name*)

Retrieves a C-string containing the value of the environment variable whose *name* is specified as argument. Returns a null pointer if variable *name* is not part of environment.

\_Noreturn void quick\_exit(*int status*)

Terminates the process normally by returning control to the host environment after calling all functions registered using *at\_quick\_exit()*. If *status* is zero or *EXIT\_SUCCESS,* a successful termination status is returned to host environment. If *status* is EXIT\_FAILURE, an unsuccessful termination status is returned to the host environment.

int system(*const char\* command*)

Invoke the command processor to execute a command. If the command is a null pointer, check whether command process is available without invoking any commands (returns non-zero if available). Returns status code of called command. Use *popen()* to execute command and capture output.

void \_Exit(*int status*)

Terminate the process by returning control to the host environment, without performing regular clean-up tasks. If *status* is zero or *EXIT\_SUCCESS,* a successful termination status is returned to host environment. If *status* is EXIT\_FAILURE, an unsuccessful termination status is returned to the host environment.

EXIT\_SUCCESS

macro, integer value, when used with *exit()*, indicates application was successful. Typically zero.

EXIT\_FAILURE

macro, integer value, when used with exit, indicates application failed. Typically one.

## searching and sorting

void\* bsearch(*const void\* key, const void\* base, size\_t num, size\_t size, int*

*(\*compar)(const void\*, const void\*)*)

Searches the given *key* in the array pointed to by *base* (which is formed by *num* elements each of *size* bytes), and returns a *void\** pointer to a matching element, if found. *compar* is a pointer to a function that compares two elements.

void qsort(*void\* base, size\_t num, size\_t size, int (\*compar)(const void\*, const void\*)*)

Sorts the *num* elements of the array pointed to by *base*, each element *size* bytes long, using the *compar* function to determine the order. Sort is not stable. Function does not return any value, but modified the content of the array pointed to by *base*.

### compar function

int compar(*const void \*p1, const void\* p2*)

Format of function required by *bsearch()* / *qsort()*. Compares two elements, returning :  
*p1* < p2 <0

*p1 == p2 0*

*p1 > p2 >0*

## integer functions

int abs(*int n*)

Returns absolute value of *n*

div\_t div(*int number, int denom*)

Returns the integer quotient and remainder of division of *number / denom* as *div\_t* structure (which has int members *quot, rem*)

long int labs(*long int n*)

Returns absolute value of *n*

ldiv\_t ldiv(*long int number, long int denom*)

Returns the integer quotient and remainder of division of *number / denom* as *ldiv\_t* structure (which has long int members *quot, rem*)

long long int llabs(*long long int n*)

Returns absolute value of n

lldiv\_t lldiv(*long long int number, long long int denom*)

Returns the integer quotient and remainder of division of *number / denom* as *lldiv\_t* structure (which has long long int members *quot, rem*)

## multibyte characters

int mblen(*const char\* pmb, size\_t max*)

Return the size of the multibyte character pointed by *pmb*, examining at most *max* bytes. Returns zero if given terminating null character, or *-1* for other values that is not a multibyte character.

int mbtowc(*wchar\_t\* pwc, const char\* pmb, size\_t max*)

Convert multibyte C-string to wide-char. Multibyte character pointed to by *pmb* is converted to value of type *wchar\_t*, and stored at location pointed to by *pwc*, examining at most *max* bytes. Function returns length in bytes of multibyte character, zero for the terminating null character, or *-1* for value that is not a multibyte character. Function has internal shift-state, which is reset by call with *pmb* as null pointer.

int wctomb(*char\* pmb, wchar\_t wc*)

Convert Wide-character to multibyte C-string, Wide character *wc* is translated to multibyte equivalent and stored in array pointed to by *pmb*. Returns the length in bytes of the equivalent multibyte sequence pointed to by *pmb* after the call, zero for the terminating null character, or *-1* for value that is not a multibyte character.

MB\_CUR\_MAX

Maximum number of bytes in a multibyte character with the current locale

## multibyte strings

size\_t mbstowcs(*wchar\_t\* dest, const char\* src, size\_t max*)

Convert Multibyte C-String to wide-characters. Translate the multibyte sequence pointed to by *src* to the equivalent sequence of wide-characters (which is stored in the array pointed to by *dest*), up until either *max* wide characters have been translated, or a null terminator is encountered in the multibyte sequence *src*. The resulting string is stored in *dest*, and is null terminated only if less than *max* bytes have been copied. Returns number of wide characters written to *dest*.

size\_t wcstombs(*char\* dest, const wchar\_t\* src, size\_t max*)

Convert wide-characters to multibyte C-string. Translates wide-characters from the sequence pointed to by *src* to the multibyte or equivalent sequence (in array pointed to by *dest*), until either *max* bytes have been translated, or a wide character translates into a null character. Result stored in *dest*, and is null terminated only if less than *max* bytes have been copied. Returns the number of bytes written to *dest*, not including the null terminator.

## types and constants

size\_t

Alias for unsigned-int. Used to represent size of a given object in bytes.

SIZE\_MAX

Maximum possible value of *size\_t*

div\_t

ldiv\_t

lldiv\_t

Structure to represent result of integer division. Contains *quot* and *rem*, of types int, long-int, and long-long-int respectively

NULL

Null pointer constant. Evaluates to zero.

## Get output of system() command

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <ctype.h>

int main()

{

long CMD\_ALL\_MAX = 1024 \* 1024; // 1 MB

char cmd\_out\_all[CMD\_ALL\_MAX];

long CMD\_LINE\_MAX = 10 \* 1024; // 10 kB

char cmd\_out\_line[CMD\_LINE\_MAX];

FILE \*fp\_system\_cmd;

int status;

// command to use: list contents of $HOME

char \*cmd\_str = "ls ~";

// Execute cmd\_str with popen(), and open resulting stream

fp\_system\_cmd = popen(cmd\_str, "r");

if (fp\_system\_cmd == NULL) {

perror("fp\_system\_cmd=popen() error\n");

exit(2);

}

// Read cmd\_str results line-by-line, appending to cmd\_out\_all

long \_strncat\_remaining;

while (fgets(cmd\_out\_line, CMD\_LINE\_MAX, fp\_system\_cmd) != NULL) {

\_strncat\_remaining = CMD\_ALL\_MAX - strlen(cmd\_out\_all);

strncat(cmd\_out\_all, cmd\_out\_line, \_strncat\_remaining);

printf("cmd\_out\_line=(%s)\n", cmd\_out\_line);

}

printf("\n");

status = pclose(fp\_system\_cmd);

if (status == -1) {

perror("pclose error\n");

exit(2);

}

printf("cmd\_out\_all=(%s)\n", cmd\_out\_all);

printf("strlen(cmd\_out\_all)=(%lu)\n", strlen(cmd\_out\_all));

return 0;

}

## array\_size macro

// Works only for static arrays

#if !defined(ARRAY\_SIZE)

#define ARRAY\_SIZE(x) (sizeof((x)) / sizeof((x)[0]))

#endif

## get environment

Either using \*\*environ global variable:

extern char \*\*environ;

void print\_env()

{

for (char \*\*env = environ; \*env; ++env)

printf("env=(%s)\n", \*env);

}

Or as *\*\*envp* argument to *main()*:

int main(int argc, char \*\*argv, char \*\*envp)

{

for (char \*\*env = envp; \*env != 0; env++)

{

char \*thisEnv = \*env;

printf("thisEnv=(%s)\n", thisEnv);

}

printf("\n");

return 0;

}

## qsort() usage example

#include <stdio.h> /\* printf \*/

#include <stdlib.h> /\* qsort \*/

int values[] = { 40, 10, 100, 90, 20, 25 };

int compare (const void \* a, const void \* b)

{

const int \*A = a, \*B = b;

return (\*A > \*B) - (\*A < \*B);

}

int main ()

{

int n;

qsort (values, 6, sizeof(int), compare);

for (n=0; n<6; n++)

printf ("%d ",values[n]);

return 0;

}

## random number range

// Seed RNG and generate number in range [range\_min, range\_max)

srand(time(NULL))

value = rand() % (range\_max + 1 - range\_min) + range\_min;

# [<string.h>](https://www.cplusplus.com/reference/cstring/)

## copying

void\* memcpy(*void\* destination, const void\* source, size\_t num*)

Copies the value of *num* bytes from the location pointed to by *source*, to memory block pointed to by *destination*. Type is not relevant – function copies binary data. Does not check for null terminating character. It is required that arrays pointed to by *source*, and *destination* are at least *num* bytes, and do not overlap (use *memmove()* for overlapping memory). Returns *destination*.

void\* memmove(*void\* destinatinon, const void\* source, size\_t num*)

Copies the value of *num* bytes from the location pointed to by *source* to memory block pointed to by *destination*, as if an intermediate buffer were used, allowing *destination* and *source* to overlap. Type is not relevant – function copies binary data. Does not check for null terminating character. It is required that arrays pointed to by *source*, and *destination* are at least *num* bytes. Returns *destination*.

char\* strcpy(*char\* destination, const char\* source*)

Copies the C-string pointed by *source* into array pointed to by *destination*, up-to and including the null terminator. It is required that array pointed to by *destination* is large enough to contain C-string from *source*, including null terminator, and that it does not overlap with *source*. Returns *destination*.

char\* strncpy(*char\* destination, const char\* source, size\_t num*)

Copies the first *num* characters of C-string *source* to *destination*. If this string is shorter than *num*, *destination* is padded with zeros until *num* characters have been written. Null terminator is not implicitly added, and will be omitted if length of *source* including the terminator is longer than *num* (the resulting string, missing the terminator, cannot safely be used as a string). *source* and *destination* must not overlap. Returns *destination*.

char\* strdup(*const char \*s*)

Returns a pointer to a new string, which is a duplicate of C-string *s*. This pointer must be released with *free()*.

## concatenation

char\* strcat(*char\* destination, const char\* source*)

Appends a copy of the *source* C-string to the destination *C-string*. The null terminator in *destination* is overwritten by the first character of *source*, and a null terminator is included at the end of the new string in *destination*. *destination* must be large enough to store resulting string, and must not overlap with *source*. Returns *destination*.

char\* strncat(*char\* destination, const char\* source, size\_t num*)

Append the first *num* characters of *source* to *destination* (or until null terminator is reached), plus a null terminator. *Destination* must be large enough to contain the concatenated result, including the null terminator. Returns *destination*.

## comparison

int memcmp(*const void\* ptr1, const void\* ptr2, size\_t num*)

Compares the first *num* bytes of array pointed to by *ptr1* with first *num* bytes of array pointed to by *ptr2*, and determine if they match, returning zero if so, *<0* if *ptr1 < ptr2*, and *>0* if *ptr1 > ptr2* (evaluating as unsigned char values).

int strcmp(*const char\* str1, const char\* str2*)

Compares the C-strings *str1*, and *str2* (performing a binary comparison of characters). Returns zero if strings are equal, *<0* if *ptr1 < ptr2*, and *>0* if *ptr1 > ptr2*. Use *strcoll()* for locale-specific rules comparison.

int strcoll(*const char\* str1, const char\* str2*)

Compare the C-strings *str1*, and *str2*, both interpreted according to *LC\_COLLATE* C-locale. Returns zero if strings are equal, *<0* if *ptr1 < ptr2*, and *>0* if *ptr1 > ptr2*.

int strncmp(*const char\* str1, const char\* str2, size\_t num*)

Compare C-strings *str1*, and *str2*, up to *num* characters. Returns zero if strings are equal, *<0* if *ptr1 < ptr2*, and *>0* if *ptr1 > ptr2*.

size\_t strxfrm(*char\* destination, const char\* source, size\_t num*)

Transform C-string *source* according to the current locale, *LC\_COLLATE*, and copies the first *num* characters of transformed string to *destination*, returning its length (not including null terminator). Returns length without copying if *destination* is null and *num* is zero. *destination* and *source* must not overlap.

## searching

void\* memchr(*const void\* ptr, int value, size\_t num*)

Search within the first *num* bytes of the block of memory pointed to by *ptr*, and return pointer to it. Both *value*, and each byte of *ptr* are interpreted as *unsigned char*. If *value* is not found, return null pointer.

char\* strchr(*const char\* str, int character*)

Return a pointer to the first occurrence of *character* in the C-string *str*. The null terminator is considered part of the C-string. *character* is converted to *char* for comparison. Returns null pointer if *character* is not found.

size\_t strcspn(*const char\* str1, const char\* str2*)

Scan C-string *str1* for the first occurrence of any of the characters that are part of *str2*, returning the number of characters of *str1* read before first occurrence. The search includes the null terminator. Returns length of *str1* if none of the characters of *str2* are found in *str1*.

char\* strpbrk(*const char\* str1, const char\* str2*)

Returns a pointer to the first occurrence in *str1* of any of the characters that are part of *str2*, or a null pointer if there are no matches. The search does not include the null terminator.

char\* strrchr(*const char\* str, int character*)

Returns a pointer to the last occurrence of *character* in the C-string *str*. *character* is converted to *char* for comparison. The null terminator is considered part of the C-string. Returns null pointer if *character* is not found.

size\_t strspn(*const char\* str1, const char\* str2*)

Returns the length of the initial portion of *str1* which consists only of characters that are part of *str2*. The search does not include the null terminator. Returns length of *str1* if all characters of *str1* are in *str2*, or zero if first character of *str1* is not in *str2*.

char\* strstr(*const char\* str1, const char\* str2*)

Return a pointer to the first occurrence of *str2* in *str1*, or a null pointer if *str2* is not part of *str1*. The search does not include null terminator.

char\* strtok(*char\* str, const char\* delimiters*)

A sequence of calls to function splits C-string *str* (which must be mutable – not a string literal, make copy with *strdup()* if needed) (note: *str* will not be preserved) into tokens, which are sequences of contiguous characters separated by any of the characters that are part of *delimiters*. On first call, function expects a C-string as argument for *str*, which is scanned from the beginning to identify the first token. Then end of this token is replaced with a null terminator, and the pointer to the beginning is returned, or null is returned if the null terminator of *str* is reached first. Subsequent calls, with a null pointer for *str*, will advance to each next token and return it in the same way until the end is reached. (See: *strtok()* usage).

char\* strsep(*char \*\*stringp, const char \*delim*)

Splits *stringp* (which must be mutable – not a string literal, make copy with *strdup()* if needed) (note: *str* will not be preserved) into ‘tokens’ found when is split by the characters in *delim* on subsequent calls, replacing end of tokens with null terminators and returns a pointer to the start token, or null if *stringp* is null, and sets *stringp* to point past current token, or to null if the current token is the last. (See: *strsep()* usage).

## other

void\* memset(*void\* ptr, int value, size\_t num*)

Sets the first *num* bytes of the block of memory pointed to by *ptr* to the specified *value* (interpreted as *unsigned char*). Returns *ptr*.

char\* strerror(*int errnum*)

Interprets value of *errno*, returning a string with a message that describes the error associated with *errno* value. These strings may be specific to system.

size\_t strlen(*const char\* str*)

Returns the length of C-string *str*, or the number of characters <bytes?> between the beginning of the string and the null terminator, not including the terminator.

## strtok() usage

char \*str = strdup("abc,6,23,7,9,4");

char \*token;

char \*ptr = str;

char \*delim = ",";

while ((token = strtok(ptr, delim)) != NULL)

{

printf("token=(%s)\n", token);

ptr = NULL;

}

## strsep() usage

char \*r = strdup("abc,123,456,789");

char \*stringp = r;

char \*token;

char \*delim = ",";

while ((token = strsep(&stringp, delim)) != NULL)

{

printf("token=(%s)\n", token);

}

# <wchar.h>

Functions to handle wide C-strings.

## I/O

fgetwc()

fgetws()

fputwc()

fputws()

fwide()

fwprintf()

fwscanf()

getwc()

getwchar()

putwc()

putwchar()

swprintf()

swscanf()

ungetwc()

vfwprintf()

vfwscanf()

vswprintf()

vswscanf()

vwprintf()

wprintf()

wscanf()

## utils

wcstod()

wcstof()

wcstol()

wcstold()

wcstoll()

wcstoul()

wcstoull()

## conversions

btowc()

mbrlen()

mbrtowc()

mbsinit()

mbsrtowcs()

wcrtomb()

wctob()

wcsrtombs()

## strings

wcscat()

wcschr()

wcscmp()

wcscoll()

wcscspn()

wcslen()

wcsncpy()

wcspbrk()

wcsrchr()

wcsspn()

wcsstr()

wcstok()

wcsxfrm()

wmemchr()

wmemcpy()

wmemmove

wmemset()

## time

scsftime()

## types and constants

mbstate\_t

size\_t

struct tm

wchar\_t

win\_t

NULL

WCHAR\_MAX

WCHAR\_MIN

WEOF

# <wctype.h>

Functions to classify/transform wide characters. (Mostly) wide versions of *<string.h>* functions.

## classification

iswalnum()

iswalpha()

iswblank()

iswdigit()

iswgraph()

iswlower()

iswprint()

iswpunct()

iswspace()

iswupper()

iswxdigit()

## conversion

towlower()

towupper()

## extensible classification/conversion

iswctype()

towctrans()

wctrans()

wctype()

## types and constants

wctrans\_t

wctype\_t

wint\_t

WEOF

# [<math.h>](https://www.cplusplus.com/reference/cmath/)

## trigonometric functions

cos()

sin()

tan()

acos()

asin()

atan()

atan2()

## hyperbolic functions

cosh()

sinh()

tanh()

acosh()

asinh()

atanh()

## expodential/logarithmic functions

exp()

frexp()

ldexp()

log()

log10()

modf()

exp2()

expm1()

ilogb()

log1p()

log2()

logb()

scalbn()

scalbln()

## power functions

pow()

sqrt()

cbrt()

hypot()

## error/gamma functions

erf()

erfc()

tgamma()

lgamma()

## rounding/remainder functions

ceil()

floor()

fmod()

trunc()

round()

lround()

llround()

rint()

lrint()

llrint()

nearbyint()

remainder()

remquo()

## floating-point manipulation functions

copysign()

nan()

nextafter()

nexttoward()

## max/min/difference functions

fdim()

fmax()

fmin()

## other functions

fabs()

abs()

fma()

## classification macros

fpclassify()

isfinite()

isinf()

isnan()

isnormal()

signbit()

## comparison macros

isgreater()

isgreaterequal()

isless()

islessequal()

islessgreater()

isunordered()

## macro constants

math\_errhandling

INFINITY

NAN

HUGE\_VAL

HUGE\_VALF

HUGE\_VALL

## types

double\_t

float\_t

# <complex.h>

# [<ctype.h>](https://www.cplusplus.com/reference/cctype/)

## classification

isalnum()

isalpha()

isblank()

iscntrl()

isdigit()

isgraph()

islower()

isprint()

ispunct()

isspace()

isupper()

isxdigit()

## conversion

tolower()

toupper()

|  |
| --- |
|  |
| **ASCII values** | **characters** | [**iscntrl**](https://www.cplusplus.com/iscntrl) | [**isblank**](https://www.cplusplus.com/isblank) | [**isspace**](https://www.cplusplus.com/isspace) | [**isupper**](https://www.cplusplus.com/isupper) | [**islower**](https://www.cplusplus.com/islower) | [**isalpha**](https://www.cplusplus.com/isalpha) | [**isdigit**](https://www.cplusplus.com/isdigit) | [**isxdigit**](https://www.cplusplus.com/isxdigit) | [**isalnum**](https://www.cplusplus.com/isalnum) | [**ispunct**](https://www.cplusplus.com/ispunct) | [**isgraph**](https://www.cplusplus.com/isgraph) | [**isprint**](https://www.cplusplus.com/isprint) |
| 0x00 .. 0x08 | NUL, (other control codes) | x |  |  |  |  |  |  |  |  |  |  |  |
| 0x09 | tab ('\t') | x | x | x |  |  |  |  |  |  |  |  |  |
| 0x0A .. 0x0D | (white-space control codes: '\f','\v','\n','\r') | x |  | x |  |  |  |  |  |  |  |  |  |
| 0x0E .. 0x1F | (other control codes) | x |  |  |  |  |  |  |  |  |  |  |  |
| 0x20 | space (' ') |  | x | x |  |  |  |  |  |  |  |  | x |
| 0x21 .. 0x2F | !"#$%&'()\*+,-./ |  |  |  |  |  |  |  |  |  | x | x | x |
| 0x30 .. 0x39 | 0123456789 |  |  |  |  |  |  | x | x | x |  | x | x |
| 0x3a .. 0x40 | :;<=>?@ |  |  |  |  |  |  |  |  |  | x | x | x |
| 0x41 .. 0x46 | ABCDEF |  |  |  | x |  | x |  | x | x |  | x | x |
| 0x47 .. 0x5A | GHIJKLMNOPQRSTUVWXYZ |  |  |  | x |  | x |  |  | x |  | x | x |
| 0x5B .. 0x60 | [\]^\_` |  |  |  |  |  |  |  |  |  | x | x | x |
| 0x61 .. 0x66 | abcdef |  |  |  |  | x | x |  | x | x |  | x | x |
| 0x67 .. 0x7A | ghijklmnopqrstuvwxyz |  |  |  |  | x | x |  |  | x |  | x | x |
| 0x7B .. 0x7E | {|}~ |  |  |  |  |  |  |  |  |  | x | x | x |
| 0x7F | (DEL) | x |  |  |  |  |  |  |  |  |  |  |  |

# [<time.h>](https://www.cplusplus.com/reference/ctime/)

## terminology

A *‘calendar time’* is a point in the continuum of time.

## time manipulations

clock\_t clock(*void*)

Returns the processor time consumed by the program – expressed in clock-ticks, a constant, but system specific length (related by *CLOCKS\_PER\_SEC*). Processing time of a program is given by differences in values returned by *clock()*. Returns *-1* on failure.

double difftime(*time\_t end, time\_t beginning*)

Calculate the difference in seconds between *beginning* and *end*. Value will be negative if *beginning* is before *end*.

time\_t mktime(*struct tm \*timeptr*)

Return a *time\_t* representing the local time described by the *tm* structure pointed to by *timeptr*, or *-1* if the calendar time cannot be represented – that-is, the reverse translation of *localtime()*. To determine date, *tm\_wday*, and *tm\_yday*, are ignored, and others are interpreted even if out of valid ranges (see *struct tm*). (For example), values of *tm\_mday* above *31* are interpreted as days that follow the last day of the selected month. Values of *timeptr* are adjusted if they are off-range or in the case of *tm\_wday*,and *tm\_yday*,have values that do not match the date described the other members.

time\_t time(*time\_t\* timer*)

Return the current calendar time as type *time\_t*. If *timer* is not null, the same result is also stored there.

int clock\_gettime(*clockid\_t clk\_id, struct timespec \*tp*)

Retrieve the time of the according to specified clock *clk\_id*, and store it in *tp*. Returns zero if successful, otherwise return negative, and set *errno*.

int clock\_settime(*clockid\_t clk\_id, const struct timespec \*tp*)

Set the time of the specified clock *clk\_id*, using value of *tp*. Returns zero if successful, otherwise return negative and set *errno*. Setting system clock requires superuser privileges.

int clock\_getres(*clockid\_t clockid, struct timespec\* res*)

Find the resolution (precision) of specified clock *clockid*, and (if it is non-null) stores it in *res*.

## conversion

char\* asctime(*const struct tm\* timeptr*)

Interpret content of *timeptr* as calendar time, and convert it C-string, of the format: *“Www Mmm dd hh:mm:ss yyyy”*, followed by newline, and null terminator.

char\* ctime(*const time\_t\* timer*)

Interpret value pointed to by *timer*, and convert it to C-string, of the format: *“Www Mmm dd hh:mm:ss yyyy”*, followed by newline, and null terminator.

struct tm\* gmtime gmtime(*const time\_t\* timer*)

Use the value pointed to by *timer* to fill a *tm* structure with the values that represent the corresponding time, expressed as a UTC time. Returns pointer to *tm* structure.

struct tm\* localtime(*const time\_t\* timer*)

Use the value pointed to by *timer* to fill a *tm* structure with values that represent the corresponding time, expressed in local timezone. Returns pointer to *tm* structure.

size\_t strftime(*char\* ptr, size\_t maxsize, const char\* format, const struct tm\* timeptr*)

Copies into *ptr* the contents of *format*, expanding its format specifiers into corresponding values that represent the time described in *timeptr*, with a limit of *maxsize* characters (in local timezone). If the length of the resulting C-string (including the null terminator) does not exceed *maxsize*, returns the total number of characters copied to *ptr* (excluding null terminator). Otherwise, return zero (in which case, value of *ptr* is indeterminate).

## <other>

int nanosleep(*const struct timespec \*req, struct timespec \*rem*)

Suspends execution of calling thread until either at least time specified in *req* has elapsed, or delivery of signal that triggers invocation of a handler in the calling thread or that terminates the process. Returns zero if successful, otherwise minus one, set *errno*, and write remaining time to *rem* (if it is not null). If the interval is not an exact multiple of the clock resolution, it is rounded up to the next multiple.

int clock\_nanosleep(*clockid\_t clock\_id, int flags, const struct timespec\* request, struct timespec\* remain*)

Like *nanosleep()*, but allows the clock used to measure interval to be measured.

## types and constants

CLOCK\_REALTIME

CLOCK\_MONOTONIC

CLOCKS\_PER\_SEC

Relation between clock-ticks and seconds elapsed. Typically 1,000,000.

NULL

Null pointer constant. Evaluates to zero.

clock\_t

Type used to represent clock ticks elapsed

size\_t

Alias for unsigned-int. Used to represent size of a given object in bytes.

time\_t

Type used to represent unix time (in seconds), (as returned by *time()*). Generally either *4* or *8* byte integer.

struct tm

Structure containing datetime, as nine *ints*. *tm\_sec* is generally *[0, 59]*, the extra range is to accommodate leap seconds. *tm\_isdst* is zero if DST is not in effect, greater than zero if DST is in effect, and less than zero if information is not available.

|  |
| --- |
|  |
| **Member** | **Type** | **Meaning** | **Range** |
| tm\_sec | int | seconds after the minute | 0-60/61\* |
| tm\_min | int | minutes after the hour | 0-59 |
| tm\_hour | int | hours since midnight | 0-23 |
| tm\_mday | int | day of the month | 1-31 |
| tm\_mon | int | months since January | 0-11 |
| tm\_year | int | years since 1900 |  |
| tm\_wday | int | days since Sunday | 0-6 |
| tm\_yday | int | days since January 1 | 0-365 |
| tm\_isdst | int | Daylight Saving Time flag |  |

struct timespec

Structure holding an interval broken down into seconds and nanoseconds, comprised of: seconds *time\_t tv\_sec*, and nanoseconds *long tv\_nsec*.

## strftime() format specifiers

|  |  |  |
| --- | --- | --- |
| **specifier** | **Replaced by** | **Example** |
| %a | Abbreviated weekday name \* | Thu |
| %A | Full weekday name \* | Thursday |
| %b | Abbreviated month name \* | Aug |
| %B | Full month name \* | August |
| %c | Date and time representation \* | Thu Aug 23 14:55:02 2001 |
| %C | Year divided by 100 and truncated to integer (00-99) | 20 |
| %d | Day of the month, zero-padded (01-31) | 23 |
| %D | Short MM/DD/YY date, equivalent to %m/%d/%y | 08/23/01 |
| %e | Day of the month, space-padded ( 1-31) | 23 |
| %F | Short YYYY-MM-DD date, equivalent to %Y-%m-%d | 2001-08-23 |
| %g | Week-based year, last two digits (00-99) | 01 |
| %G | Week-based year | 2001 |
| %h | Abbreviated month name \* (same as %b) | Aug |
| %H | Hour in 24h format (00-23) | 14 |
| %I | Hour in 12h format (01-12) | 02 |
| %j | Day of the year (001-366) | 235 |
| %m | Month as a decimal number (01-12) | 08 |
| %M | Minute (00-59) | 55 |
| %n | New-line character ('\n') |  |
| %p | AM or PM designation | PM |
| %r | 12-hour clock time \* | 02:55:02 pm |
| %R | 24-hour HH:MM time, equivalent to %H:%M | 14:55 |
| %S | Second (00-61) | 02 |
| %t | Horizontal-tab character ('\t') |  |
| %T | ISO 8601 time format (HH:MM:SS), equivalent to %H:%M:%S | 14:55:02 |
| %u | ISO 8601 weekday as number with Monday as 1 (1-7) | 4 |
| %U | Week number with the first Sunday as the first day of week one (00-53) | 33 |
| %V | ISO 8601 week number (01-53) | 34 |
| %w | Weekday as a decimal number with Sunday as 0 (0-6) | 4 |
| %W | Week number with the first Monday as the first day of week one (00-53) | 34 |
| %x | Date representation \* | 08/23/01 |
| %X | Time representation \* | 14:55:02 |
| %y | Year, last two digits (00-99) | 01 |
| %Y | Year | 2001 |
| %z | ISO 8601 offset from UTC in timezone (1 minute=1, 1 hour=100) If timezone cannot be determined, no characters | +100 |
| %Z | Timezone name or abbreviation \* If timezone cannot be determined, no characters | CDT |
| %% | A % sign | % |

Specifiers marked with (\*) are locale-dependent.

Yellow rows indicate specifiers and sub-specifiers introduced in C99.

Two locale-specific modifiers can also be inserted between the percentage sign (%) and the specier proper to request an alternative format, where applicable.

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Meaning** | **Applies to** |
| E | Uses the locale's alternative representation | %Ec %EC %Ex %EX %Ey %EY |
| O | Uses the locale's alternative numeric symbols | %Od %Oe %OH %OI %Om %OM %OS %Ou %OU %OV %Ow %OW %Oy |

# <errno.h>

Must be imported to have access to *errno* variable.

# <assert.h>

void assert(int expression)

If argument *expression* compares equal to zero (that is, *expression* is false), a message is written to *stderr*, and *abort()* is called. Message includes *expression*, name of source file, and line number. Is abled if macro *NDEBUG* has been defined before import of *<assert.h>*.

# <signal.h>

## functions

void (\*signal(*int sig, void\* func(int)*))(*int*)

Specifies a way to handle the signal with number specified by *sig* (see table below). Parameter *func* specifies how to handle signal, an can be either: Default *SIG\_DEL*, Ignore *SIG\_IGN*, or a specified function, *void handler\_function(int parameter)*. Return type is same type as that of *func*. If successful, return pointer to handler function that was in-charge of handling given signal before the call, if any. If unsuccessful at registering new signal handling procedure, return *SIG\_ERR*, and set *errno*.

int raise(*int sig*)

Send signal *sig* (see table below) to the current executing program, to be handled as specified by *signal()*. Return zero if successful, otherwise non-zero.

## types

sig\_atomic\_t

Integer type that can be accessed as atomic entity, even in the presence of asynchronous signals.

|  |  |  |
| --- | --- | --- |
| **type** | **macro** | **signal** |
| int (signals) | SIGABRT | (Signal Abort) Abnormal termination, such as is initiated by the [abort](https://www.cplusplus.com/abort) function. |
| SIGFPE | (Signal Floating-Point Exception) Erroneous arithmetic operation, such as zero divide or an operation resulting in overflow (not necessarily with a floating-point operation). |
| SIGILL | (Signal Illegal Instruction) Invalid function image, such as an illegal instruction. This is generally due to a corruption in the code or to an attempt to execute data. |
| SIGINT | (Signal Interrupt) Interactive attention signal. Generally generated by the application user. |
| SIGSEGV | (Signal Segmentation Violation) Invalid access to storage: When a program tries to read or write outside the memory it has allocated. |
| SIGTERM | (Signal Terminate) Termination request sent to program. |
| *functions* (handlers) | SIG\_DFL | Default handling: The signal is handled by the default action for that particular signal. |
| SIG\_IGN | Ignore Signal: The signal is ignored. |
| SIG\_ERR | Special return value indicating failure. |

# <setjmp.h>

# <limits.h>

|  |  |  |
| --- | --- | --- |
| **name** | **expresses** | **possible value\*** |
| CHAR\_BIT | Number of bits in a char object (byte) | 8 or greater\* |
| SCHAR\_MIN | Minimum value for an object of type signed char | -127 (-27+1) or less\* |
| SCHAR\_MAX | Maximum value for an object of type signed char | 127 (27-1) or greater\* |
| UCHAR\_MAX | Maximum value for an object of type unsigned char | 255 (28-1) or greater\* |
| CHAR\_MIN | Minimum value for an object of type char | either SCHAR\_MIN or 0 |
| CHAR\_MAX | Maximum value for an object of type char | either SCHAR\_MAX or UCHAR\_MAX |
| MB\_LEN\_MAX | Maximum number of bytes in a multibyte character, for any locale | 1 or greater\* |
| SHRT\_MIN | Minimum value for an object of type short int | -32767 (-215+1) or less\* |
| SHRT\_MAX | Maximum value for an object of type short int | 32767 (215-1) or greater\* |
| USHRT\_MAX | Maximum value for an object of type unsigned short int | 65535 (216-1) or greater\* |
| INT\_MIN | Minimum value for an object of type int | -32767 (-215+1) or less\* |
| INT\_MAX | Maximum value for an object of type int | 32767 (215-1) or greater\* |
| UINT\_MAX | Maximum value for an object of type unsigned int | 65535 (216-1) or greater\* |
| LONG\_MIN | Minimum value for an object of type long int | -2147483647 (-231+1) or less\* |
| LONG\_MAX | Maximum value for an object of type long int | 2147483647 (231-1) or greater\* |
| ULONG\_MAX | Maximum value for an object of type unsigned long int | 4294967295 (232-1) or greater\* |
| LLONG\_MIN | Minimum value for an object of type long long int | -9223372036854775807 (-263+1) or less\* |
| LLONG\_MAX | Maximum value for an object of type long long int | 9223372036854775807 (263-1) or greater\* |
| ULLONG\_MAX | Maximum value for an object of type unsigned long long int | 18446744073709551615 (264-1) or greater\* |

\* the actual value depends on the particular system and library implementation, but shall reflect the limits of these types in the target platform.

# [<unistd.h>](https://pubs.opengroup.org/onlinepubs/007908799/xsh/unistd.h.html)

Not a default C library, provides Unix system calls.

**<Arguably the most importantly library – finish functions list below>**

## key items

unsigned int sleep(unsigned int seconds)

Make the calling thread sleep until *seconds* have elapsed, or a signal arrives which is not ignored. Returns zero at conclusion of specified sleep, or the number of seconds remanning if interrupted by signal handler. Shouldn’t be <mixed> with calls to *alarm()*. Using *longjmp()* from signal handler, or modifying *SIGALRM* while sleeping can cause undefined behaviour.

int usleep(*useconds\_t usec*)

Make the calling thread sleep until *usec* microseconds have elapsed, or a signal arrives that is not ignored.

unsigned int alarm(*unsigned int seconds*)

Arranges for a *SIGALRM* signal to be delievered to the calling process in *seconds*, unless it is zero, in which case, any pending alarm is cancelled (pending alarms are also cancelled on scheduling of new alarm). Returns number of seconds remaining until previous alarm was scheduled, or zero if there was no previously scheduled alarm. Preserved across call to *execve()*, not inherited by children created by *fork()*.

pid\_t fork(*void*)

Creates a new process by duplicating the calling process. The new process is referred to as the child, the calling process as the parent. Returns PID in the parent, and zero in the child, if successful, otherwise return minus one (in parent) and set *errno*.

int execve(*const char\* pathname, char\* const argv[], char\* const envp[]*)

Execute the program referred to by *pathname*, replacing the current process, with new stack, heap, and data segments. *pathname* must be a binary executable, or a script beginning with a shebang. Does not return on success.

## filesystem

## I/O

ssize\_t read(*int fd, void \*buf, size\_t count*)

int access(*const char \*pathname, int mode*)

int faccessat(*int dirfd, const char \*pathname, int mode, int flags*)

int faccessat2(*int dirfd, const char \*pathname, int mode, int flags*)

int stat(*const char \*restrict pathname, struct stat \*restrict statbuf*);

int fstat(*int fd, struct stat \*statbuf*);

int lstat(*const char \*restrict pathname, struct stat \*restrict statbuf*);

int isatty(*int fd*)

Test whether *fd* is an open file-descriptor referring to a terminal, return one if so, othersize zero, and set *errno*. Get the file-descriptor of a stream with *fileno()*.

char\* ttyname(*int fd*)

int ttyname\_r(*int fd, char \*buf, size\_t buflen*)

Get the name of a terminal. *ttyname()* returns a pointer to a null-terminated string containing the pathname of the terminal device open on *fd*, or null if *fd* is not a terminal, or upon error, and set *errno*. *ttyname\_r()* returns 0 on success, and an error number otherwise, if successful, it also writes result to buffer *buf* up to size *buflen*.

## <…>

int access(*const char \*path, int amode*)

Check the file named by *path* for accessibility according to the bit pattern contained in *amode*, using real user/group ID in place of effective user/group ID. The value of *amode* is either the bitwise inclusive OR of the access permissions to be checked (R\_OK, W\_OK, X\_OK) or the existence test, F\_OK. Returns *0* if requested access is permitted, otherwise returns *-1* and set *errno*.

int brk(*void \*addr*)

void \*sbrk(*intptr\_t incr*)

(Legacy). Change the amount of space allocated for the calling process. The change is made by resetting the process' break value and allocating the appropriate amount of space, which increases as the break value increases. Function *brk()* sets the break value to *addr*, returning *0* on success, otherwise *-1* and set *errno*. Function *sbrk()* adds *incr* bytes to the current break value, returning the prior break value if successful, otherwise *(void\*) -1* and set *errno*. Retrieve current value of the program break with *sbrk(0)*.

int chdir(*const char \*path*)

Set the pathname *path* to become the current working directory (the starting point for pathnames not beginning with *'/'*). Returns *0* if successful, otherwise *-1* and set *errno*.

int chroot(*const char \*path*)

(Legacy) Set the pathname *path* to become the root directory (the starting point for pathnames beginning with *'/'*), does not change working directory. Return *0* if successful, otherwise *-1* and set *errno*.

int chown(*const char \*path, uid\_t owner, gid\_t group*)

Change owner and group of file at *path* to *owner* and *group* respectively, and update *st\_ctime* of file. Return *0* if successful, otherwise *-1* (indicating no changes were made) and set *errno*.

int close(*int fildes*)

Deallocate the file descriptor indicated by *fildes*. Any outstanding record locks owned by the process associated with the file descriptor will be removed (unlocked). Returns *0* if successful, otherwise *-1* and set *errno*. Note: files opened with *fopen()* should be closed with *fclose()*.

size\_t confstr(*int name, char \*buf, size\_t len*)

Get configuration defined string values, like *sysconf()*, but with string rather than numeric arguments. *name* is the system variable to be queried, *buf* is a buffer to copy result, and *len* is the length of the buffer. If *name* is invalid, returns *0* and set *errno*. If *name* does not have a value, *0* is returned but *errno* is unchanged. If *name* has a configuration-defined value, returns the size of the buffer that would be needed to hold the entire value (including null terminator). If *len* is less than return value, the value written to *buf* is truncated.

char \*crypt(*const char \*key, const char \*salt*)

Encode (hash) *key* using *salt*, and return pointer to encoded string (the first two characters of which are those of *salt*), or upon error, return null and set *errno*. Values of *salt* are restricted to *[A-Za-z0-9./]*. The value returned is overwritten by each subsequent call.

char \*ctermid(*char \*s*)

Generates a string that, when used as a pathname, referes to the current controlling terminal for the current process. Access to this file is not guaranteed. *s* may be a null pointer (if not using \_POSIX\_THREAD\_SAFE\_FUNCTIONS or \_POSIX\_THREADS), or otherwise a buffer of at least *{L\_ctermid}* bytes. The result is stored in *s* (which is initialized if null), and then returned. If pathname cannot be determined or function is otherwise unsuccessful, an empty string is returned.

char \*cuserid(*char \*s*)

(Legacy) Generate a string of the name associated with the real or effective user ID of the process. *s* may be a null pointer (if not using \_POSIX\_THREAD\_SAFE\_FUNCTIONS or \_POSIX\_THREADS), or otherwise a buffer of at least *{L\_cuserid}* bytes. The result is stored in *s* (which is initialized if null), and returned. If unsuccessful, null is returned if *s* is null, otherwise a null byte *'\0'* is written to *\*s*.

int dup(*int fileds*)

int dup2(*int fildes, int fildes2*)

Duplicate an open file descriptor, alternative to *fcntl()* (using *F\_DUPFD*). If successful, returns a non-negative integer (the file descriptor), otherwise *-1* is returned and *errno* is set.

The following are equivalent:

fid = dup(fildes);

fid = fcntl(fildes, F\_DUPFD, 0);

The following are also equivalent:

fid = dup2(fildes, fildes2);

close(fildes2); fid = fcntl(fildes, F\_DUPFD, fildes2);

Provided: *fildes2* is between *[0, {OPEN\_MAX}]*, *fildes* is a valid file descriptor, not equal to *fildes2*.

void encrypt(*char block[64], int edflag*)

Uses the key generated by *setkey()* to encrypt the *block*, which is an array of length *64* containing only values *0* and *1*. The array *block* is modified in-place. If *edflag* is *0*, argument is encoded, if it is *1* will be decoded if decoding is implemented by system (if it is not, *errno* is set accordingly). Does not return a value.

extern char \*\*environ;

int execl(const char \*path, const char \*arg0, ... /\*, (char \*)0 \*/);

int execv(const char \*path, char \*const argv[]);

int execle(const char \*path, const char \*arg0, ... /\*, (char \*)0, char \*const envp[]\*/);

int execve(const char \*path, char \*const argv[], char \*const envp[]);

int execlp(const char \*file, const char \*arg0, ... /\*, (char \*)0 \*/);

int execvp(const char \*file, char \*const argv[]);

Execute a file - replace the current process image with a new process image.

<...>

There is no return from *exec* when successful, as calling process image is overlaid by new process image. Returns *-1* upon error and set *errno*.

void exit(*int status*)

void \_exit(*int status*)

Terminate a process, after calling all functions registered by *atexit()*. All output streams are flushed, all open streams are closed, and all files created by *tmpfile()* are removed. Control is then returned to the host environment, and the value of *status* (limited to *[0, 255]*) is made available. *status* is normally either *EXIT\_SUCCESS* or *EXIT\_FAILURE*.

<...>

Normally, *exit()* should be used instead of *\_exit()*. The latter does not invoke many of the clean-up handlers that the former does.

int fchown(*int fildes, uid\_t owner, gid\_t group*)

As per *chown()*, except a file descriptor *fildes* is used instead of the file path. Returns *0* if successful, otherwise *-1* and set *errno*.

int fchdir(*int fildes*)

As per *chdir()*, except a file descriptor *fildes* is used instead of the dir path. Returns *0* if successful, otherwise   
*-1* and set *errno*.

int fsync(int fildes)

Forces all currently queued IO operations associated with file descriptor *fildes* to the synchronized IO completion state. Doesn't return until either data is written, or an error occurs. Returns *0* if successful, otherwise returns *-1* and set *errno*.

int fdatasync(*int fildes*)

Like *fsync()*, except the inode block (metadata) is not flushed. Forces all currently queued IO operations associated with file indicated by file descriptor *fildes* to the synchronised IO completion state. Doesn't return until either data is written, or an error occurs. Returns *0* if successful, otherwise returns *-1* and set *errno*.

long int fpathconf(*int fildes, int name*)

long int pathconf(*const char \*path, int name*)

Determine the current value of a configurable limit or option that is associated with a file or directory. *fpathconf()* takes an open file descriptor, *pathconf()* takes a file path. See: pathconf variables. Returns *-1* if *name* is invalid, or upon other error, and set *errno*, otherwise value of specified variable is returned.

int ftruncate(*int fildes, off\_t length*)

int truncate(*const char \*path, off\_t length*)

Sets the size of a given file to *length* bytes. Extra data is discarded if file is larger than this length, while if it is smaller, it is padded to that length with zero bytes. *truncate()* takes a path (process must have permission to write), *ftruncate()* takes a file descriptor (which must be open for writing). *ftruncate()* can also be used to set the size of a shared memory object. Returns *0* if successful, otherwise *-1* and set *errno*.

char \*getcwd(*char \*buf, size\_t size*)

Place the absolute path of the current working directory into array pointed to by *buf* (must not be null), of size *size*, and return it, otherwise return null and set *errno*.

int getdtablesize(*void*)

(Legacy). Get the file descriptor table size and return it. Equivalent to calling *getrlimit()* with option *RLIMIT\_NOFILE*.

gid\_t getegid(*void*)

Get the effective group id of the calling process and return it.

uid\_t geteuid(*void*)

Get the effective user id of the calling process and return it.

gid\_t getgid(*void*)

Get the real group id of the calling process and return it.

int getgroups(*int gidsetsize, gid\_t grouplist[]*)

Fill the array *grouplist*, of length *gidsetsize*, with the current supplementary group ids of the calling process. Returns the number of supplementary group ids stored in *grouplist*. If *gidsetsize* is *0*, returns the number of supplementary group ids associated with calling process, without modifying *grouplist*. The effective group id of the calling process may-or-may-not be included in this list. Returns *-1* upon failure and set *errno*.

long gethostid(*void*)

Get the 32bit identifier for the current host (as set by *sethostid()*) and return it.

char \*getlogin(*void*)

int getlogin\_r(*char \*name, size\_t namesize*)

Get the username (login name) associated with the calling process. *getlogin()* returns a pointer to a string with this value (or null if unsuccessful and set *errno*), while *getlogin\_r()* copies it to array *name* of size *namesize*, returning *0* if successful, otherwise an error number. The maximum size of a login name is given by *{LOGIN\_NAME\_MAX}*.

extern char \*optarg

extern int optind, opterr, optopt

int getopt(*int argc, char \* const argv[], const char \*optstring*)

Used to parse command line arguments. *argc* and *argv* are the same as those passed to *main()*, while *optstring* is a string of recognized character options. Options which take an argument are followed by a colon. *optind* is the index of the next element of *argv* to be processed (initialized to *1* by system, updated by each call to *getopt()*). Returns the next option character from *argv* that matches a character in *optstring*, if there is such a match. If this option takes an argument, *optarg* is set to point to point to it (or an error indicator is returned). If the option is the last character in *argv[optind]*, the argument is determined to be the next element of *argv*, otherwise the option is the string following the option character in *argv*. Returns *'?'* if an option character is not contained in *optstring*. On detecting a missing option-argument *':'* if the first character of *optstring* was a colon, or *'?'* otherwise. In both these cases, *optopt* is set to point to the option that caused the error. If the application has not set *opterr* to *0*, and the first character of *optstring* is not a colon, an error is printed to *stderr*. Returns the next option character specified if successful, or *-1* if all options are parsed. See: *<getopt.h>* example usage.

int getpagesize(*void*)

(Legacy). Get the current page size. Equivalent to *sysconf(\_SC\_PAGE\_SIZE)* and *sysconf(\_SC\_PAGESIZE)* (which should be used instead).

char \*getpass(*const char \*prompt*)

(Legacy). Open the process' controlling terminal, writes the null-terminated string *prompt*, disables echoing, then reads a string of characters up to next newline or *EOF*, restores the terminal state, and closes the terminal. Returns a pointer to string read from terminal (up to *{PASS\_MAX}* bytes), or null upon an error.

pid\_t getpgid(*pid\_t pid*)

Get the process group id of process with process id equal to *pid* and return it. If *pid* is *0*, return the process group id of the calling process. Upon error, returns *(pid\_t) -1* and set *errno*.

pid\_t getpgrp(*void*)

Get the process group id of the calling process and return it.

pid\_t getpid(*void*)

Get the process id of the calling process and return it.

pid\_t getppid(*void*)

Get the parent process id of the calling process and return it.

pid\_t getsid(*pid\_t pid*)

Get the process group id of session leader and return it. *pid* specifies the calling process if it is *(pid\_t) 0*. Returns *(pid\_t) -1* upon error and set *errno*.

uid\_t getuid(*void*)

Get the real user id of the calling process and return it.

char \*getwd(*char \*path\_name*)

(Legacy). Use *getcwd()* instead. Determine absolute path of current working directory of calling process, copy it into array *path\_name*, and return it. If length of path is greater than *({PATH\_MAX}+1)* (including null-terminator), fail and return null pointer.

int lchown(*const char \*path, uid\_t owner, gid\_t group*)

Change the owner and group of a symbolic link. Has the same effect as *chown()*, except if file is a symbolic link, in which case ownership of the symlink is changed, not the file pointed to by symlink.

int link(*const char \*path1, const char \*path2*)

Create new hard link at *path2* for existing file *path1*. Link is created atomically, and link count of file is incremented by *1*. Returns *0* if successful, otherwise return *-1* and set *errno*.

int lockf(*int fildes, int function, off\_t size*)

Place an advisory-mode lock on a section of a file. The action to be taken is specified by *function*, whose possible values are: *F\_ULOCK* (unlock locked sections), *F\_LOCK* (lock a section for exclusive use, blocking the calling thread until the section to be locked is available), *F\_TLOCK* (test and lock a section for exclusive use, failing if the file is already), and *T\_TEST* (test a section for locks by other processes). The file is specified by file descriptor *fildes*, which must be opened with write or read/write permission. The section to be locked or unlocked starts at the current offset in the file, and extends forwards (or backwards given negative value) *size* bytes, or the largest possible value if *0*. Locks may exist past the end-of-file. Locked sections can be released either wholly, or in part, and unlocking will fail for any sections also locked by another process. A blocking action is interrupted by any signal. Buffered streams should not be locked. If, during an unlock request, the last byte of the requested section is the maximum value for an instance of type *off\_t*, it is treated as *0* and extended to the end of the file. Deadlocking can occur if a process controlling a locked section is in-turn blocked by the locked section of another process. The system <will/can/may?> detect deadlocks, and will respond by causing *lockf()* to fail with the associated error. Return *0* if successful, otherwise return *-1* and set *errno*.

off\_t lseek(*int fildes, off\_t offset, int whence*)

Set the file offset for the open file descriptor *fildes*, as per the value of *whence*. If *whence* is *SEEK\_SET*, the file offset is set to *offset* bytes, if it is *SEEK\_CUR*, the file offset is set to the current location plus *offset*, if it is *SEEK\_END*, the file offset is set to the size of the file <plus?> *offset*. Behaviour for devices where are incapable of seeking is OS dependent. If the offset is set beyond the end of the file, the data will be written to that location with the intermediate space set to *0*, but *lseek()* itself will not extend the size of a file. The results are unspecified if *fildes* referes to shared memory. If successful, returns the resulting offset, measured in bytes from the start of the file, otherwise returns *(off\_t) -1* and set *errno*.

int nice(*int incr*)

Add the value of *incr* to the nice value of the calling process. The nice value of a process is a non-negative number for which a larger value results in less favourable scheduling, with a range *[0, 2\*{NZERO}-1]*. Attempting to set the nice value above or below these limits results in a value of the respectively limit. Nice values have no effect on processes or threads with policy *SCHED\_FIFO* or *SCHED\_RR*. The nice value of a multi-threaded process affects all system scope threads in the process. If successful, return the new nice value minus *{NZERO}*, otherwise return *-1* and set *errno*.

int pause(*void*)

Suspend the calling thread until delivery of a signal whose action is either to execute a signal-catching function (in which case *pause()* will return after the signal-catching function returns), or to terminate the process (in which case *pause()* will not return). There is no successful return value (successful behaviour would be to pause indefinitely, resuming after interrupt signal is considered a failure), return value is *-1*, and set *errno*.

int pipe(*int fildes[2]*)

Create a pipe, placing file descriptors into *fildes[0]* and *fildes[1]* that correspond to the read and write pipe ends respectively. Reading from *fildes[0]* accesses data written to *fildes[1]* on a FIFO basis. Upon success, *st\_atime*, *st\_ctime*, and *st\_mtime* fields of the pipe are marked for update, and *0* is returned. Otherwise, *-1* is returned and *errno* is set.

int pthread\_atfork(*void (\*prepare)(void), void (\*parent)(void), void (\*child)(void)*)

Declare fork handlers to be called before and after *fork()*, in the context of the thread that called *fork()*. *prepare* is called before *fork()* commences. *parent* and *child* are called after *fork()* processing completes in the parent/child process respectively. If null is given as one of these values, calling it is skipped. Multiple functions can be registered by *pthread\_atfork()*, those given for *parent* and *child* are called in the order they were registered, while *prepare* calls are made in the opposite order. If successful, returns *0*, otherwise return an error number.

ssize\_t read(*int fildes, void \*buf, size\_t nbyte*)

Attempt to read *nbyte* bytes (which should be less than *{SSIZE\_MAX}*) from the file descriptor *fildes* (which should not be a shared memory object). For files that support seeking, reading begins at current offset associated with *fildes*, and the file offset is incremented by the number of bytes actually read. For files that do not support seeing (and therefore have an undefined offset), reading begins at current position. No data is read beyond the current end-of-file, and attempting to read with a starting position at or after end-of-file results in *0* being returned. For a special device file, the result of subsequent *read()* requests is OS dependent. Data is only read if it has been previously written to a file, reading portion of file that has not been written yields bytes with value *0*. If the Synchronized Input and Output option is supported: if the *O\_DSYNC* and *O\_RSYNC* bits have been set, read I/O operations on the file descriptor complete as defined by synchronised I/O data integrity completion.

When attempting to read from an empty pipe or FIFO:

* If no process has the pipe open for writing, read() will return 0 to indicate end-of-file.
* If some process has the pipe open for writing and O\_NONBLOCK is set, read() will return -1 and set errno to [EAGAIN].
* If some process has the pipe open for writing and O\_NONBLOCK is clear, read() will block the calling thread until some data is written or the pipe is closed by all processes that had the pipe open for writing.

When attempting to read a file (other than a pipe or FIFO) that supports non-blocking reads and has no data currently available:

* If O\_NONBLOCK is set, read() will return a -1 and set errno to [EAGAIN].
* If O\_NONBLOCK is clear, read() will block the calling thread until some data becomes available.
* The use of the O\_NONBLOCK flag has no effect if there is some data available.

If successful, where *nbyte* is greater than *0*, mark *st\_atime* field of file for update and return number of bytes read. Returns *-1* and set *errno* if interrupted by signal before reading any data, or number of bytes read if interrupted after some reading data. Return *-1* and set *errno* upon error.

A read() from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the *I\_SRDOPT* *ioctl()* request, and can be tested using *I\_GRDOPT* with *ioctl()*.

<STREAMS file modes>

* In byte-stream mode, *read()* retrieves data from the STREAM until as many bytes as were requested are transferred, or there is no more data, or until a zero-byte message block is encountered. Message boundaries are ignored. <If a zero-byte message block is encountered, it is placed back on the STREAM to be read by next *read()*, *readv()*, or *getmsg()* call, unless that message was read as the first message on STREAM>.
* In message-nondiscarding mode, *read()* retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. All remaining data is is left on STREAM can be read by subsequent *read()* calls. <If a zero-byte message block is encountered, *0* is returned, and the message is removed>.
* In message-discarding mode, data is read as in message-nondiscarding mode, but all remaining data is discarded.

ssize\_t pread(*int fildes, void \*buf, size\_t nbyte, off\_t offset*)

As per *read()*, except it reads from position specified by *offset*, and does not change the offset associated with *fildes*. Attempting to call *pread()* on a file that does not allow seeking results in an error.

ssize\_t readv(*int fildes, const struct iovec \*iov, int iovcnt*)

Equivalent to *read()*, but places the input data into the *iovcnt* buffers specified by the members of the iov array: *iov[0]*, *iov[1]*, …, *iov[iovcnt-1]*. The *iovcnt* argument is valid if greater than 0 and less than or equal to *{IOV\_MAX}*. Each *iovec* entry specifies the base address and length of an area in memory where data should be placed. *readv(*) always fills an area completely before proceeding to the next.

int readlink(*const char \*path, char \*buf, size\_t bufsize*)

int rmdir(*const char \*path*)

int setgid(*gid\_t gid*)

pid\_t setpgrp(*void*)

int setregid(*gid\_t rgid, gid\_t egid*)

int setreuid(*uid\_t ruid, uid\_t euid*)

pid\_t setsid(*void*)

int setuid(*uid\_t uid*)

void swab(*const void \*src, void \*dest, ssize\_t nbytes*)

int symlink(*const char \*path1, const char \*path2*)

void sync(*void*)

long int sysconf(*int name*)

pid\_t tcgetpgrp(*int fildes*)

int tcsetpgrp(*int fildes, pid\_t pgid\_id*)

useconds\_t ualarm(*useconds\_t useconds, useconds\_t interval*)

int unlink(*const char \*path*)

int usleep(*useconds\_t useconds*)

pid\_t vfork(*void*)

ssize\_t write(*int fd, const void \*buf, size\_t count*)

ssize\_t pwrite(*int fildes, const void \*buf, size\_t nbyte, off\_t offset*)

ssize\_t writev(*int fildes, const struct iovec \*iov, int iovcnt*)

## types

Defined in <sys/types.h>

size\_t, ssize\_t, uid\_t, guid\_t, off\_t, pid\_t

useconds\_t

Defined in <inttypes.h>

intptr\_t

## environment

extern char \*\*environ

Array of character pointers to the environment strings (each line being of the form *'name=value'*).

## pathconf variables

Variable gives the name of the variable to be queried, and Value-of-name gives the corresponding numerical value to be used with *pathconf()* and *fpathconf()*.

|  |  |  |
| --- | --- | --- |
| **Variable** | **Value-of-*name*** | **Notes** |
| FILESIZEBITS | \_PC\_FILESIZEBITS | 3,4 |
| LINK\_MAX | \_PC\_LINK\_MAX | 1 |
| MAX\_CANON | \_PC\_MAX\_CANON | 2 |
| MAX\_INPUT | \_PC\_MAX\_INPUT | 2 |
| NAME\_MAX | \_PC\_NAME\_MAX | 3,4 |
| PATH\_MAX | \_PC\_PATH\_MAX | 4,5 |
| PIPE\_BUF | \_PC\_PIPE\_BUF | 6 |
| \_POSIX\_CHOWN\_RESTRICTED | \_PC\_CHOWN\_RESTRICTED | 7 |
| \_POSIX\_NO\_TRUNC | \_PC\_NO\_TRUNC | 3,4 |
| \_POSIX\_VDISABLE | \_PC\_VDISABLE | 2 |
| \_POSIX\_ASYNC\_IO | \_PC\_ASYNC\_IO | 8 |
| \_POSIX\_PRIO\_IO | \_PC\_PRIO\_IO | 8 |
| \_POSIX\_SYNC\_IO | \_PC\_SYNC\_IO | 8 |

1. If *path* or *fildes* refers to a directory, the value returned applies to the directory itself.
2. If *path* or *fildes* does not refer to a terminal file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
3. If *path* or *fildes* refers to a directory, the value returned applies to filenames within the directory.
4. If *path* or *fildes* does not refer to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.
5. If *path* or *fildes* refers to a directory, the value returned is the maximum length of a relative pathname when the specified directory is the working directory.
6. If *path* refers to a FIFO, or *fildes* refers to a pipe or FIFO, the value returned applies to the referenced object. If *path* or *fildes* refers to a directory, the value returned applies to any FIFO that exists or can be created within the directory. If *path* or *fildes* refers to any other type of file, it is unspecified whether an implementation supports an association of the variable name with the specified file.
7. If *path* or *fildes* refers to a directory, the value returned applies to any files, other than directories, that exist or can be created within the directory.
8. If *path* or *fildes* refers to a directory, it is unspecified whether an implementation supports an association of the variable name with the specified file.

## constants

### version test

### mandatory symbolic constants

### options and feature groups

### execution-time symbolic constants

### function constants

### filestreams

# <getopt.h>

The C Unix library for parsing command line arguments.

int getopt()

int getopt\_long()

int getopt\_long\_only()

## example usage

# C vs C++

C code will generally work when compiled as C++. However, there are a number of differences in behaviour between them – C++ cannot be considered a superset of C.

<https://www.quora.com/Is-every-valid-C-program-a-valid-C++-program/answer/Sergey-Zubkov-1>

<https://en.wikipedia.org/wiki/Compatibility_of_C_and_C>++

# gcc built-in functions

int \_\_builtin\_constant\_p(*exp*)

Determine if a value *exp* is known to be constant at compile time, returning one if so and zero otherwise