C Memoire

[pointers 2](#_Toc73047988)

[C-strings 3](#_Toc73047989)

[errorno 4](#_Toc73047990)

[<stdio.h> 5](#_Toc73047991)

[file operations 5](#_Toc73047992)

[file access 5](#_Toc73047993)

[formatted I/O 5](#_Toc73047994)

[character I/O 5](#_Toc73047995)

[direct I/O 5](#_Toc73047996)

[file positioning 5](#_Toc73047997)

[error-handling 6](#_Toc73047998)

[macros 6](#_Toc73047999)

[types 6](#_Toc73048000)

[stream properties 6](#_Toc73048001)

[formatting 6](#_Toc73048002)

[<stdlib.h> 9](#_Toc73048003)

[string conversion 9](#_Toc73048004)

[random 9](#_Toc73048005)

[memory management 9](#_Toc73048006)

[environment 10](#_Toc73048007)

[searching and sorting 11](#_Toc73048008)

[compar function 11](#_Toc73048009)

[integer functions 11](#_Toc73048010)

[multibyte characters 11](#_Toc73048011)

[multibyte strings 12](#_Toc73048012)

[types and constants 12](#_Toc73048013)

[Get result of system() command 13](#_Toc73048014)

[array\_size macro 13](#_Toc73048015)

[get environment 13](#_Toc73048016)

[C vs C++ 16](#_Toc73048017)

# 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;

# C-strings

# errorno

# [<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.

## 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*.

freopen()

setbuf()

setvbuf()

\_IOFBF

\_IOLBF

\_IONBF

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

## formatted I/O

fprintf()

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

printf()

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

snprintf()

sprint()

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

vfprintf()

vfscanf()

vprintf()

vscanf()

vsnprintf()

vsprintf()

## character I/O

fgetc()

fgets()

fputc()

fputs()

getc()

getchar()

gets()

putc()

putchar()

puts()

ungetc()

## direct I/O

fread()

fwrite()

## file positioning

fgetpos()

fseek()

fsetpos()

ftell()

rewind()

SEEK\_CUR

SEEK\_END

SEEK\_SET

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

## error-handling

clearerr()

feof()

ferror()

perror()

## macros

BUFSIZE

EOF

FILENAME\_MAX

FOPEN\_MAX

L\_tmpnam

NULL

TMP\_MAX

## types

FILE

fpos\_T

size\_t

## 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.

# [<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.

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.

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.

## 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\_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 type int, long-int, and long-long-int

NULL

Null pointer constant. Evaluates to zero.

## Get result 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 ~";

//char \*cmd\_str = "cat /dev/random | head -c 1000";

// 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);

}

// Ongoing: 2021-05-27T00:11:21AEST If our goal is 'all output in single string', do we a loop with fgets()

// 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);

//trim(cmd\_out\_line); // Definition/use of trim() ommited

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

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

}

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

## copying

memcpy()

memmove()

strcpy()

strncpy()

## concatenation

strcat()

strncat()

## comparison

memcmp()

strcmp()

strcoll()

strncmp()

strxfrm()

## searching

memchr()

strchr()

strcspn()

strpbrk()

strrchr()

strspn()

strstr()

strtok()

## other

memset()

strerror()

strlen()

# <math.h>

# 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>++