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;

## 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:

# C-strings

char \*a = “abc”;

while allowable, should actually be:

const char \*a = “abc”;

(?)

# 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*.

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*)

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), and finish with null terminator. 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*) characters are read, or *EOF* is encountered. Stops reading on (and copies) newline. 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()*).

rewind()

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.

## 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

# [<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 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 ~";

// 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;

}

# [<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>

# <math.h>

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

|  |
| --- |
|  |
| **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/)

## 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*. Function

performs reverse translation of *localtime()*.

time()

## conversion

asctime()

ctime()

gmtime()

localtime()

strftime()

## types and constants

CLOCKS\_PER\_SEC

NULL

clock\_t

size\_t

time\_t

struct tm

# <assert.h>

# <complex.h>

# <signal.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.

# 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