C

cabs() — Extended function (libm)

Complex absolute value function
#include <xmath.h>
double cabs(struct { double *r*, *i*; } *z*);

cabs computes the absolute value, or modulus, of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle whose sides are given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the squares of the parts.

See Also

extended mathematics, hypot

Notes

To conform to the ANSI Standard, **cabs** has been moved from the header **math.h** to the header **xmath.h**. This may require that some code be altered.

cabs is not described in the ANSI Standard. Any program that uses it does not conform strictly to the Standard, and may not be portable to other compilers or environments.

calloc() — General utility (libc)

Allocate and clear dynamic memory
#include <stdlib.h>
void *calloc(size_t *count*, **size_t** *size*);

calloc allocates a portion of memory large enough to hold *count* items, each of which is *size* bytes long. It then initializes every byte within the portion to zero.

calloc returns a pointer to the portion allocated. The pointer is aligned for any type of object. If it cannot allocate the amount of memory requested, it returns NULL.

Example

For an example of this function, see **stdarg**.

Cross-references

Standard, §4.10.3.1
The C Programming Language, ed. 2, p. 167

See Also

alignment, free, general utilities, malloc, realloc

Notes

If *count* or *size* is equal to zero, then the behavior of **calloc** is implementation defined: **calloc** returns either NULL or a unique pointer. This is a quiet change that may silently break some existing code.

case — C keyword

Mark entry in switch table
case *expression*:

case is a label that introduces an entry within the body of a **switch** statement. The value of the **switch** statement's conditional expression is compared with the value of every **case** label's expression. When the two match, then the program jumps to the point marked by that **case** label and execution continues from there. Execution continues until a **break** statement is encountered.

Each **case** label must mark an expression whose value differs from those of every other **case** label for that **switch** statement. See **switch** for more information.

Example

For an example, see **printf**.

Cross-references

Standard, §3.6.1
The C Programming Language, ed. 2, p. 58

See Also

break, C keywords, default, statements, switch

Notes

Every conforming implementation must be able to accept at least 257 **case** labels within a **switch** statement.

cc — Command

Compiler controller
cc [*options***]** *file . . .*

cc is the program that controls compilation. It guides files of source and object code through each phase of compilation and linking. **cc** has many options to assist in the compilation of C programs; in essence, however, all you need to do to produce an executable file from your C program is type **cc** followed by the name of the file or files that hold your program. It checks whether the file names you give it are reasonable, selects the right phase for each file, and performs other tasks that ease the compilation of your programs.

File Names

cc assumes that each *file* name that ends in **.c** or **.h** is a C program and passes it to the C compiler for compilation.

cc assumes that each *file* argument that ends in **.s** is in Mark Williams assembly language and processes it with the assembler **as**; and that every file with the suffix **.asm** is written in Microsoft macro assembly language, and attempts to assemble it with the macro assembler **MASM**.

cc assumes that all files with the suffix **.m** are assembly-language files that also use C preprocessor instructions. **cc** will pass these files to the C preprocessor **cpp**, and pass its output to the assembler **as**. This hybrid allows you to write assembly-language programs that are model-independent. For an example of a **.m** file, see the Lexicon entry for **as**, the assembler. For more information on building **.m** files, see the entry for **larges.h**.

cc also passes all files with the suffixes **.obj** or **.lib** unchanged to the linker MS-LINK.

How cc Works

cc normally works as follows: First, it compiles or assembles the source files, naming the resulting object files by replacing the **.c**, **.s**, **.m**, or **.asm** suffixes with the suffix **.obj**. Then, it links the object files with the C runtime startup routine and the standard C library, and leaves the result in file *file***.exe**. If only one object file is created during compilation, it is deleted after linking; however, if more than one object file is created, or if an object file of the same name existed before you began to compile, then the object file or files are not deleted.

Arguments and Wildcards

The option **-na** (for "no arguments") tells **Let's C** that a program does not take command-line arguments. This option may be used with or without the **-ns** option, which suppresses the linking

of STDIO into your program.

The option **-w** (for "wildcard") tells **Let's C** to include code in your program that will allow it to expand the wildcard characters '?' and '*' in command-line arguments. For example, if program **example.exe** is *not* built with the **-w** option, the command

```
example *.c
```

results in an argument count of two, and an argument list that contains two non-NULL members. If **example.exe** is built *with* the **-w** option, **Let's C** will include code so that your program will automatically expand the wildcard argument ***.c**. The argument count and argument list are altered to reflect the number of such files and their names, respectively.

If a program defines a global array **char _cmdname[]** that gives the name of the command, the **-w** option fills in **argv[0]** with the command name and looks for environmental variables of the form *<name>***HEAD** and *<name>***TAIL**. If found, these are added to **argv[]** before and after command-line arguments, respectively. This option limits your program to 256 arguments at any one time. If you happen to need to use more than 256 arguments, use the program **msdoscvt**, which is presented as an example in the entry for **exargs**.

For example, the **wc** command is built with the **-w** option and defines

```
_cmdname = "wc";
```

If the current directory contains files **a.c** and **b.c**, and environmental variable **WCHEAD** is set to **-l**, the command

```
wc *.c
```

has the same effect as the command

```
wc -l a.c b.c
```

that is, it counts the lines in **a.c** and **b.c**.

The arguments to **main** are defined as

```
main(argc, argv)
int argc; char *argv[];
```

On some systems, a third argument is available:

```
main(argc, argv, envp)
int argc; char *argv[], *envp[];
```

The **envp** argument is a NULL-terminated array of pointers to environmental variables, each of the form *var=value*. If a program is compiled without the **-w** option, **Let's C** passes an empty list as **envp**. If a program is compiled with the **-w** option, **Let's C** passes an **envp** that contains MS-DOS environmental variables.

Options

The following lists all of **cc**'s command-line options. **cc** passes some options through to the linker MS-LINK unchanged, and correctly interprets to it the options **-o**, **-u**, **-y/**, **-yf**, **-ym**, **-yn**, **-ys**, and **-yu**.

A number of the options are esoteric and normally are not used when compiling a C program. The following are the most commonly used options:

LEXICON

-A	invoke MicroEMACS when errors occur
-f	include floating-point **printf**
-l*name*	pass library **lib***name***.lib** to linker
-o *name*	call executable file *name*
-V	print details of compiler's actions
-VASM	generate assembly-language output

-A MicroEMACS option. If an error occurs during compilation, **cc** automatically invokes the MicroEMACS screen editor. The error or errors are displayed in one window and the source code file in the other, with the cursor set to the line number indicated by the first error message. Typing **<ctrl-X>>** moves to the next error, **<ctrl-X><** moves to the previous error. To recompile, close the edited file with **<ctrl-Z>**. Compilation will continue either until the program compiles without error, or until you exit from the editor by typing **<ctrl-U>** followed by **<ctrl-X><ctrl-C>**.

-c Compile option. Suppress linking and the removal of the object files.

-cc2l Use a LARGE-model version of the code generator **cc2**. This allows the creation of extremely large programs, but runs more slowly than the default **cc2**, which is in SMALL model.

*-D***name**[=**value**]
Define *name* to the preprocessor, as if set by a **#define** directive. If *value* is present, it is used to initialize the definition.

-E Expand option. Run the C preprocessor **cpp** and write its output onto the standard output.

-f Floating point option. Include library routines that perform floating-point arithmetic. Because the floating-point routines require approximately five kilobytes of memory, the standard C library does not include them; the **-f** option tells the compiler to include them. If a program is compiled without the **-f** option but attempts to print a floating point number during execution by using the **e**, **f**, or **g** format specifications to **printf**, the message

```
You must compile with -f option for floating point
```

will be printed and the program will exit.

-I*directory*
Include option. Specify the directory the preprocessor should search for files given in **#include** directives, using the following criteria: If the **#include** statement reads

```
#include "file.h"
```

cc searches for **file.h** first in the source directory, then in the directory named in the **-I***directory* option, and finally in the system's default directories. If the **#include** statement reads

```
#include <file.h>
```

cc searches for **file.h** first in the directory named in the **-I***directory* option, and then in the system's default directories. Multiple **-I***directory* options are executed in their order of appearance.

-K Keep option. Do not erase the intermediate files generated during compilation. Temporary files will be written into the current directory. The **-K** option takes precedence over the **-xt** option: when **-K** is set, the temporary files are always written into the directory in which the source code is kept.

-l *name*
library option. Pass the name of a library to the linker. **cc** expands **-l***name* into **lib***name***.a**.

-m Mini-**make** option: Compile file of source code only if it has been changed since its identically changed object file was last compiled.

-na No arguments option. The compiled program does not use **argc** or **argv**. See *Arguments and wildcards*, above, for more information.

-ns Do not link in **stdio**. If the standard I/O library is not needed, the **-ns** option produces much smaller object modules.

-o *name*
Output option. Rename the executable file from the default *file***.exe** to *name*.

-U *name*
Undefine symbol *name*. Use this option to undefine symbols that the preprocessor defines implicitly, such as the name of the native system or machine.

-V Verbose option. **cc** prints onto the standard output a step-by-step description of each action it takes.

V*string*
Variant option. *Toggle* (i.e., turn on or off) the variant *string* during the compilation. Options marked **Strict:** generate messages that warn of the conditions in question. If you name an option once in the **CCHEAD** environmental variable and again on the **cc** command line, these two togglings will cancel each other out. **cc** recognizes the following variants:

-V80186 Output code that uses the instructions native to the Intel i80186 and i80286 microprocessors. This switch also works with the assembler **as**: assembly-language programs that contain i80186/286 instructions will be assembled correctly when the assembler is invoked using this option. Programs compiled with this option cannot be run on an IBM PC or strictly compatible machines, but will take full advantage of the instruction set of the IBM AT and its compatibles. The code will also execute correctly on the NEC V20 and V30 processors. Default is **off**.

-VALIEN Enable the **alien** keyword. Under **Let's C**, the **alien** keyword allows direct calls of PL/M, Pascal, and FORTRAN functions and procedures. These differ from C functions in the following ways: (1) C pushes arguments from right to left; the other languages push from left to right. (2) C arguments are popped by the calling function, whereas under the other languages arguments are popped by the called function. (3) **Let's C** appends an underbar character to the end of every function name, whereas the other languages do not. Default is **off**.

-VASM Output assembly-language code. It can be used with the **-VLINES** option, described below, to generate a line-numbered file of assembly language. Default is **off**.

-VCSD Generate debugging information for **csd**, the Mark Williams C Source Debugger.

-VFLOAT Include floating point **printf** routines. Same as **-f** option, above.

-VLARGE LARGE-model output. Default is **off**.

-VLINES Generate line number information. Can be used with the option **-S**, described above to generate assembly language output that uses line numbers. Default is **off**.

LEXICON

-VNDP Generate i8087 floating-point code. The code generated with this option will run only on machines that have an i8087 mathematics co-processor. If this option is *not* used, **Let's C** automatically uses libraries that sense the presence of the i8087: if an i8087 is present, floating point routines will be run on it; but if one is not present, they will be emulated in software. Default is **off**.

-VOPT Turn on optimization. Default is **off**.

-VPSTR Put strings into the shared segment, if possible. Used to generate ROMable code. Default is **off**.

-VQUIET Suppress all messages. Default is **off**.

-VSBOOK Strict: note deviations from *The C Programming Language*, ed. 2. Default is **off**.

-VSLCON Strict: **int** constant promoted to **long** because value is too big. Default is **on**.

-VSMALL Generate SMALL-model output. Default is **on**.

-VSMEMB Strict: check use of structure/union members for adherence to standard rules of C. Default is **on**.

-VSNREG Strict: register declaration reduced to auto. Default is **on**.

-VSPVAL Strict: pointer value truncated. Default is **off**.

-VSRTVC Strict: risky types in truth contexts. Default is **off**.

-VSTAT Give statistics on optimization.

-VSTRICT Turn on all strict checking. Default is **on**.

-VSUREG Strict: note unused registers. Default is **off**.

-VSUVAR Strict: note unused variables. Default is **on**.

-V3GRAPH
Translate ANSI trigraphs. Default is **off**.

-w Wildcards option: the compiled program can take wildcards in its command line. See *Arguments and wildcards*, above, for more information.

-x<*key***><***directory***>**
Use the given *directory* as the location for one of the following: for compiler files if *key* is **c**; libraries if *key* is **l**; output files if *key* is **o**; or temporary files if *key* is **t**.

-y/*switch*
Pass *switch* directly to MS-LINK.

-yf Force MS-LINK to create a linker command file. For more information on what a linker command file is, see the Lexicon entry for **MS-LINK**.

-ym Force MS-LINK to create a map file. For more information on what a map file is, see the Lexicon entry for **MS-LINK**.

-yn Reset the MS-LINK segments to 1,024, using the form **/segments=1024** required by MS-LINK versions 3.02 and later.

-ys*number*
Force MS-LINK to set the stack size to *number*, where *number* gives the number of bytes of stack required, in decimal figures.

LEXICON

-yu*name*
Undefine *name* for MS-LINK.

-Z Pause between passes and prompt for disk change. Used with the compiler using single-sided disks.

See Also

as, cc0, cc1, cc2, cc3, commands, cpp, ld

cc0 — Definition

cc0 is the **Let's C** *preprocessor* and *parser*. It performs all preprocessing tasks, and parses C programs using the method of recursive descent. It then translates the program into a logical-tree format.

See Also

cc, cc1, cc2, cc3, cpp, Definitions, preprocessing

cc1 — Definition

cc1 is the **Let's C** code generator. This phase generates code from the trees created by the parser, **cc0**. Code generation is table driven, with entries for each operator and addressing mode.

See Also

cc, cc0, cc2, cc3, cpp, Definitions

cc2 — Definition

cc2 is the optimizer/object generator phase of **Let's C**. It optimizes the code generated by **cc1**, and writes the object code.

Let's C uses multiple optimization algorithms. One optimizes jump sequences: it eliminates common code, optimizes span-dependent jumps, and removes jumps to jumps. The other function scans the generated code repeatedly to eliminate unnecessary instructions.

The **cc** option **-cc2l** uses a LARGE-model version of **cc2**. This allows you to create extremely large programs, but runs more slowly than the default version of **cc2**, which is in SMALL model.

See Also

cc, cc0, cc1, cc3, cpp, Definitions

cc3 — Definition

cc3 is the output phase of **Let's C** that writes a file of assembly language rather than a relocatable object module. This phase is optional. It allows you to examine the code generated by the compiler. To produce an assembly-language output of a C program, use the **-VASM** option on the **cc** command line. For example,

```
cc -VASM foo.c
```

tells **cc** to produce a file of assembly language called **foo.s**, instead of an object module.

See Also

cc, cc0, cc1, cc2, cpp, Definitions

CCTAIL — Environmental variable

Variables at end of compilation command

CCTAIL is an environmental variable that is read by the **cc** command. When you issue a **cc** command, **cc** reads **CCTAIL** and appends it to the end of the list of arguments you have given **cc**.

You should set **CCTAIL** in **autoexec.bat** to add options routinely to your **cc** commands. For example, adding the command

```
set CCTAIL=-lm
```

to **autoexec.bat** ensures that the mathematics library **libm.lib** is always linked into your C programs. Thus, typing the command

```
cc foo.c
```

will have the same effect as typing

```
cc foo.c -lm
```

See Also

cc, CCHEAD, environmental variable

ceil() — Mathematics (libm)

Integral ceiling
#include <math.h>
double ceil(double *z***);**

The function **ceil** returns the "ceiling" of a function, or the smallest integer less than *z*. For example, the ceiling of 23.2 is 23, and the ceiling of -23.2 is -23.

ceil returns the value expressed as a **double**.

Cross-references

Standard, §4.5.6.1
The C Programming Language, ed. 2, p. 251

See Also

fabs, floor, fmod, mathematics

char — C keyword

The data type **char** is the smallest addressable unit of data. It consists of one byte of storage, and it can encode all of the characters that can be used to write a C program. **sizeof(char)** returns one by definition, with all other data types defined as multiples thereof.

A **char** may be either signed or unsigned; this is up to the implementation. **Let's C** uses a signed **char** by default. If a **char** holds any of the characters that make up the C character set, then it is positive. ANSI C allows the corresponding types **signed char** and **unsigned char**. Programmers can create signed and unsigned versions of **char** where needed.

The range of values that can be encoded within a **char** are set by the macros **CHAR_MIN** and **CHAR_MAX**. These are defined in the header **limits.h**. The minimum values of these macros depend upon whether the implementation sign-extends a **char** when it is used in an expression. If the implementation does sign extend, then **CHAR_MIN** is equal to **SCHAR_MIN** (at least -127) and **CHAR_MAX** is equal to **SCHAR_MAX** (at least +127). If it does not sign extend, however, **CHAR_MIN** is equal to zero and **CHAR_MAX** is equal to **UCHAR_MAX** (at least +255).

Cross-references

Standard, §3.1.2.5
The C Programming Language, ed. 2, p. 211

See Also

signed char, types, unsigned char

character constant — Definition

A *character constant* is a constant that encodes a character or escape sequence. A character constant consists of one or more characters or escape sequences that are enclosed within apostrophes ´. To include a literal apostrophe within a character constant, use the escape sequence **\'**.

A character is regarded as having type **char** as it is read, and it yields an object with type **int**. If a character constant contains one character or escape sequence, then the numeric value of that character is written into an **int**-length object. For example, under an implementation that uses ASCII, the character constant **'a'** yields an **int**-length object with the value of 0x61. If a character constant contains more than one character or escape sequence, the result is implementation-defined.

Because the constant being read is regarded as having type **char**, the value of a character constant can change from implementation to implementation, depending upon whether the implementation uses a signed or unsigned **char** by default. For example, in an environment in which a **char** has eight bits and uses two's-complement arithmetic, the character constant **'\xFF'** will yield an **int** with a value of either -1 or +255, depending upon whether a **char** is, respectively, signed or unsigned by default. **Let's C** uses signed **char**s by default.

A *wide-character constant* is a character constant that is formed of a wide character instead of an ordinary, one-byte character. It is marked by the prefix 'L'. For example, in the following

```
L'm';
```

stores the numeric value of 'm' in the form of a wide character.

Example

For an example of using character constants in a program, see **putchar**.

Cross-references

Standard, §3.1.3.4
The C Programming Language, ed. 2, p. 193

See Also

constants, escape sequences

Notes

Although octal escape sequences are limited to three octal digits, hexadecimal escape sequences can be arbitrary length. However, when the value of a hexadecimal escape sequence exceeds that which can be represented in an **int**, behavior is defined by the implementation.

character display semantics — Definition

The Standard describes the semantics by which characters are displayed on an output device. The *active position* is where the output device will print the next character produced by the function **fputc**. On a video terminal, it usually is marked by a cursor. The locale defines the direction of printing, whether from left to right, from right to left, or from top to bottom.

The following escape sequences can be embedded within a string literal or character constant to

affect the behavior of an output device:

\a Generate an alert signal. The alert may take the form of ringing a bell or printing a visual signal on a screen.

\b Backspace: move the active position back one position. If the active position is already at the beginning of the line, the behavior is undefined.

\f Form feed: move the active position to the beginning of the next page. On a hard-copy printer, it feeds a fresh sheet of paper. On a video terminal, it may take the form of clearing the screen and moving the cursor to the "home" position.

\n Newline: move the active position to the beginning of the next line.

\r Return: move the active position to the beginning of the current line.

\t Horizontal tab: move the active position to the beginning of the next horizontal tabulation field. If the active position is already at or past the last horizontal tabulation field on the current line, the behavior is undefined.

\v Vertical tab: move the active position to the beginning of the next vertical tabulation field. If the active position is already at or past the last vertical tabulation field, the behavior is undefined.

Every implementation must define each of these escape sequences as being a unique value that can be stored in one **char** object.

Cross-reference

Standard, §2.2.2

See Also

Environment, escape sequence, trigraph sequences

character handling — Overview

#include <ctype.h>

The Standard's repertoire of library functions includes 13 that test or alter individual characters, as follows:

Character testing

isalnum	Check if a character is a numeral or letter
isalpha	Check if a character is a letter
iscntrl	Check if a character is a control character
isdigit	Check if a character is a numeral
isgraph	Check if a character is printable
islower	Check if a character is a lower-case letter
isprint	Check if a character is printable
ispunct	Check if a character is a punctuation mark
isspace	Check if a character is white space
isupper	Check if a character is a upper-case letter
isxdigit	Check if a character is a hexadecimal numeral

Case mapping

tolower	Convert character to lower case
toupper	Convert character to upper case

All are declared in the header **ctype.h**.

The operation of all character-handling functions (with the exception of **isdigit** and **isxdigit**) is modified by the program's locale, as set by the function **setlocale**. This allows these function to test and modify characters using a locale-specific character set. The calls

```
setlocale(LC_CTYPE, locale);
```

or

```
setlocale(LC_ALL, locale);
```

force these functions to use the locale-specific character set. See **localization** for more information.

Cross-references

Standard, §4.3
The C Programming Language, ed. 2, p. 248

See Also

character, ctype.h, extended character handling, Library

Notes

Although these functions are described as "character handling," they are defined as taking an argument of type **int** to allow them to accept the special value of **EOF** and locale-specific character sets.

clearerr() — STDIO (stdio.h)

Clear a stream's error indicator
#include <stdio.h>
void clearerr(FILE *_fp_**);**

When a file is manipulated, a condition may occur that would cause trouble should the program continue. This could be an error (e.g., a read error), or the program may have read to the end of the file. Most environments use two indicators to signal that such a condition has occurred: the *error indicator* and the *end-of-file* indicator.

When an error occurs, the error indicator is set to a value that indicates what error occurred. The end-of-file indicator is set when the end of a file is read. By checking these indicators, a program can see if all is going well. A file may not be manipulated further until both indicators have been reset to their normal values.

clearerr resets to normal the error indicator and the end-of-file indicator for the stream pointed to by *fp*.

Cross-references

Standard, §4.9.10.1
The C Programming Language, ed. 2, p. 248

See Also

feof, ferror, perror, STDIO

Notes

The indicators are cleared when a file is opened or when the file-position indicator is reset by the function **rewind**. Successful calls to **fseek**, **fsetpos**, or **ungetc** clear the end-of-file indicator.

CLK_TCK — Manifest constant

#include <time.h>
CLK_TCK is a manifest constant that is defined in the header **time.h**. It represents the number of "ticks" in a second. A "tick" is the unit of time measured by the function **clock**.

clock returns the type **clock_t**. To determine how many seconds a program required to run to the given point, divide the value returned by **clock** by the value of **CLK_TCK**.

Example

For an example of using this macro in a program, see **clock**.

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

clock, clock_t, date and time

clock() —

Get processor time used
#include <time.h>
clock_t clock(void);

clock calculates and returns the amount of processor time a program has taken to execute to the current point. Execution time is calculated from the time the program was invoked. This, in turn, is set as a point from the beginning of an era that is defined by the implementation. For example, under the COHERENT operating system, time is recorded as the number of milliseconds since January 1, 1970, 0h00m00s UTC.

The value **clock** returns is of type **clock_t**. This type is defined in the header **time.h**. The Standard defines it merely as being an arithmetic type capable of representing time. If **clock** cannot determine execution time, it returns -1 cast to **clock_t**.

To calculate the execution time in seconds, divide the value returned by **clock** by the value of the macro **CLK_TCK**, which is defined in the header **time.h**.

Example

This example measures the number of times a **for** loop can run in one second on your system. This is approximate because **CLK_TCK** can be a real number, and because the program probably will not start at an exact tick boundary.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      clock_t finish;
      long i;

      /* finish = about 1 second from now */
      finish = clock() + CLK_TCK;
      for(i = 0; finish > clock(); i++)
            ;
```

```
      printf("The for() loop ran %ld times in one second.\n", i);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.1
The C Programming Language, ed. 2, p. 255

See Also

CLK_TCK, clock_t, date and time, difftime, mktime

clock_t — Type

System time
#include <time.h>

clock_t is a data type that is defined in the header **time.h**. It is an arithmetic type, and is the type returned by the function **clock**.

The unit that **clock_t** holds is implementation-defined. The manifest constant **CLK_TCK** expands to a number that expresses how of many of these units constitute one second of real time.

Example

For an example of using this type in a program, see **clock**.

Cross-references

Standard, §4.12.1
The C Programming Language, ed. 2, p. 255

See Also

CLK_TCK, clock, date and time, time_t

close() — Extended function (libc)

Close a file
short close(short *fd***);**

close closes the file identified by the file descriptor *fd*, which was returned by **creat**, **dup**, or **open**. **close** frees the associated file descriptor.

close returns -1 if an error occurs, such as its being handed a bad file descriptor. Otherwise, it returns zero.

Because each program can have only a limited number of files open at any given time, programs that process many files should **close** files whenever possible.

Example

For an example of this function, see the entry for **open**.

See Also

creat, extended miscellaneous, open

Notes

When a program exits, **Let's C** automatically closes all files that had been opened via the STDIO function **fopen**. However, you must explicitly call **close** to close all files that had been opened with **open**, or the unclosed file will be truncated to zero bytes when the program exits.

LEXICON

cmp — Command

Compare bytes of two files
cmp [-ls] *file1 file2* **[***skip1 skip2***]**

cmp is a command that is included with **Let's C**. It compares *file1* and *file2* character by character, for equality. If *file1* is '**-**', **cmp** reads the standard input.

Normally, **cmp** notes the first difference and prints the line and character position, relative to any skips. If it encounters EOF on one file but not on the other, it prints the message "EOF on file*n*". The following are the options that can be used with **cmp**:

-l Each differing byte by printing the positions and octal values of the bytes of each file.

-s Print nothing, but return the exit status.

If the skip counts are present, **cmp** reads *skip1* bytes on *file1* and *skip2* bytes on *file2* before it begins to compare the two files.

The exit status is zero for identical files, one for non-identical files, and two for errors, e.g., bad command usage or inaccessible file.

See Also

commands

commands — Overview

Let's C includes a number of commands. They are listed below, with the command given on the left and a description on the right.

Commands included with **Let's C**:

cc	The compiler driver
cmp	Compare two files
cpp	The C preprocessor
egrep	String search utility
exetocom	Convert **.exe** files to **.com** files
fixobj	Edit object modules to allow cross-linking
make	Programming discipline
me	Microemacs screen editor
mwlib	Librarian for libraries in MS-DOS format
MWS	Mark Williams shell
nm	Print symbol tables
pr	Paginate text for printing
size	Print size of a file
strip	Remove debug tables from some executables
tail	Print the end of a file
wc	Count words/lines in ASCII files

Additional commands included with **Let's C Utilities**:

diff	Compare two files
ed	Line editor
m4	Macro processor
sort	Sort ASCII files
uniq	List/destroy duplicate lines

For more information on any of these commands, see its entry within the Lexicon.

See Also

DOS-specific features, MWS

comment — Definition

A *comment* is text that is embedded with a program but is ignored by the translator. It is intended to guide the reader of the code.

A comment is introduced by the characters **/***. The only exceptions are when these characters appear within a string literal or a character constant. In these instances, the characters **/*** have no special significance. When **/*** is read, all text is ignored until the characters ***/** are read. Once a comment is opened, the translator does nothing with the text except scan it for multi-byte characters and for the characters ***/** that close the comment.

The translator replaces a comment with a single white-space character; this is done during phase 3 of translation.

Cross-references

Standard, §3.1.9
The C Programming Language, ed. 2, p. 192

See Also

***/, /*, lexical elements, translation phases**

Notes

The Standard's definition of a comment does not allow comments to "nest." That is, you cannot have a comment within a comment. This may require that some code be revised. If you wish to exclude some code from translation temporarily, a sounder practice is to use the preprocessing directives **#ifdef** and **#endif**. For example,

```
#ifdef DEBUG
   . . .
#endif
```

will include code only if **DEBUG** has been defined as being a macro.

It is possible to open a comment inadvertently. For example, the code

```
int *intptr, int1, int2;
   . . .
int2 = int1/*intptr;
```

inadvertently creates a comment symbol out of the division operator '/' and the pointer-dereference operator '*'. *Caveat utilitor.*

compatible types — Definition

To judge whether two types are compatible, several factors must be considered.

Scalar types

First, the base types must be identical. Second, all specifiers must match, except for signedness (i.e., it does not matter whether either or both are signed or unsigned). Third, all type qualifiers must match. There are special semantics to determine whether qualified objects are compatible to ensure that qualified types are not "hidden". See the entry **type qualifiers** for more information.

Structures

For two structures to be compatible, they must have the same "tagged type". For example, the structures

```
FILE struct1;
FILE struct2;
```

are compatible, because the tagged type of each is **FILE**. On the other hand, in the following code

```
struct s1 { int s1_i } s1;
struct s2 { int s2_i } s2;
```

the structures **s1** and **s2** are not compatible.

Pointers

For two pointers to be compatible, they must point to the same type of object. Other pointers may be compatible if they are suitably cast.

Cross-reference

Standard, §3.1.2.6, §3.5.2-4

See Also

type specifier, types

compile —

To *compile* a program means to translate it with a compiler. A compiler is a translator that takes a set of high-level source instructions (i.e., C code) and produces a set of machine instructions that implement the behavior that the source instructions describe.

See Also

Definitions, interpret, link

compliance — Definition

Compliance refers to the degree to which a program and an implementation conform to the Standard's descriptions of the C language.

A *strictly conforming program* is one that uses only the features of the language and the library routines that are described within the Standard. It does not produce any behavior that is implementation defined, unspecified, or undefined. It does not exceed any minimum maximum set by the Standard. A strictly conforming program should be maximally portable to any environment for which a conforming implementation exists.

A *conforming program* is any program that can be translated by a conforming implementation. It may use library functions other than those described in the Standard, it may evoke non-Standard behavior, and it may use extensions to the language that are recognized by the implementation.

There are two varieties of *conforming implementation*: *conforming hosted implementation* and *conforming freestanding implementation*. A conforming hosted implementation is one that can translate any strictly conforming program. A conforming freestanding implementation is one that can translate any strictly conforming program whose use of macros and functions is restricted to those defined in the headers **float.h**, **limits.h**, **stdarg.h**, and **stddef.h**.

Cross-reference

Standard, §1.7

See Also

behavior, Definitions, limits

con — Operating system device

Logical device for the console

MS-DOS gives names to its logical devices. **Let's C** uses these names to allow its STDIO library routines to access these devices via MS-DOS. **con** is the logical device for the console.

Example

The following example demonstrates how to open the console device.

```
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      FILE *fp, *fopen();
      if ((fp = fopen("con", "w")) != NULL)
            fprintf(fp, "con enabled.\n");
      else printf("con: cannot open.\n");
      return EXIT_SUCCESS;
}
```

See Also

aux, com1, lpt1, nul, operating system devices

const — C keyword

Qualify an identifier as not modifiable

The type qualifier **const** marks an object as being unmodifiable. An object declared as being **const** cannot be used on the left side of an assignment, or have its value modified in any way. Because of these restrictions, an implementation may place objects declared to be **const** into a read-only region of storage.

Judicious use of **const** allows the translator to optimize more thoroughly, for it does not have to include code to check whether the object has been modified.

Most of the prototypes for library functions use **const** to mark identifiers that are not modified by the function.

Cross-references

Standard, §3.5.3
The C Programming Language, ed. 2, p. 40

See Also

type qualifier, volatile

constant expressions — Definition

A *constant expression* is one that represents a constant. Constant expressions are required in a variety of situations: when the value of an enumeration constant is set; when the size of an array is declared; as a constant to be used in a **case** statement; or as the size of a bit-field declaration.

Every constant expression must return a value that is within the range representable by its type. No constant expression can contain assignment operators, increment or decrement operators, function calls, or the comma operator. The only exception is when it used as the operand to the operator **sizeof**.

LEXICON

The Standard describes the following varieties of constant expressions:

Address constant expression
This type of constant is an expression that points to an object or a function. The operators **[]**, *****, **&**, **.**, and **->** may be used to create an address constant, as may a pointer cast.

Arithmetic constant expression
This type of constant has an arithmetic type, and is one the following:

- character constant
- enumeration constant
- floating constant
- integer constant
- **sizeof** expression

  An arithmetic constant expression can be cast only to another arithmetic type, except when it is an operand to **sizeof**.

Integral constant expression
This type of constant has integral type, and is one of the following:

- character constant
- enumeration constant
- a floating constant that is the immediate operand of a cast.
- integer constant
- **sizeof** constant

When a constant expression is used to initialize a static variable, it must resolve, when translated, into one of the following:

- An address constant.
- An address constant for an object type, plus or minus an integral constant expression.
- An arithmetic constant expression.

Initializers on local variables that are not declared **static** are not so restrictive.

Cross-references

Standard, §3.4
The C Programming Language, ed. 2, p. 38

See Also

constants, expressions, initializers, Language, void expression

Notes

Constant expressions can be combined when translated. The precision and accuracy of such translation-time evaluation must be at least those of the execution environment. This requirement was designed with cross-compilers in mind, where the execution environment might differ from translation environment.

A constant expression may be resolved into a constant by the translator. Therefore, it can be used in any circumstance that calls for a constant. For this reason, the Standard forbids the use in an

#if statement in a constant expression that queries the run-time environment. A program that does include a **#if** statement that queries the environment will not run the same when translated by an ANSI-compatible translator.

constants — Overview

A *constant* is a lexical element that represents a set numerical value. The four categories of constants are as follows:

character constants	A character constant or wide-character constant
enumeration constants	A constant used in an **enum**
floating constants	A floating-point number
integer constants	An integer

Each type is determined by the form of the token. For example,

```
5L
```

defines a constant of type **long**, and

```
5.03
```

is a floating-point constant.

Cross-references

Standard, §3.1.3
The C Programming Language, ed. 2, pp. 192*ff*

See Also

constant expressions, lexical elements

continue — C keyword

Force next iteration of a loop
continue;

continue forces the next iteration of a **for**, **while**, or **do** loop. It works only upon the smallest enclosing loop.

continue forces a loop to iterate by jumping to the end of the loop, which is where iteration evaluation is made. For example, the code

```
while(statement) {
        . . .
      if (statement)
            continue;
        . . .
}
```

is equivalent to:

```
while(statement) {
        . . .
      if (statement)
            goto end;
        . . .
      end: ;
}
```

Example

For an example of this statement, see **mktime**.

LEXICON

Cross-references

Standard, §3.6.6.2
The C Programming Language, ed. 2, p. 64

See Also

break, C keywords, goto, return, statements

conversions — Definition

The term *conversion* means to change the type of an object, function, or constant, either explicitly or implicitly. Explicit conversion occurs when an object or function is cast to another type by a cast operator. Implicit conversion occurs when the type of the object or function is changed by an operator without a cast operator being used.

When an object or function is converted into a compatible type, its value does not change.

The following paragraphs summarize conversion for different types of objects.

Enumeration constants
These constants are always converted implicitly to **int**s.

Floating types
When a floating type is converted to an integral type, the fractional portion is thrown away. If the value of the integral part cannot be represented by the new type, behavior is undefined.

When a **float** is promoted to **double** or **long double**, its value is unchanged. Likewise, when a **double** is promoted to a **long double**, its value is unchanged.

A floating type may be converted to a smaller floating type. If its value cannot be represented by the new type, behavior is undefined. If its value lies within the range of values that can be represented by the smaller type but cannot be represented precisely, then its value is rounded to the next highest or next lowest value, depending upon the implementation.

Integral types
A **char**, a **short int**, an enumerated type, or a bit-field, whether signed or unsigned, may be used in any situation that calls for an **int**. The type to be promoted is converted to an **int** if an **int** can hold all of its possible values. If an **int** cannot hold all of its possible values, then it is converted to an **unsigned int**. This rule is called *integral promotion*. This conversion retains the value of the type to be promoted, including its sign. Thus, it is called a *value-preserving* promotion.

Some current implementations of C use a scheme for promotion that is called *unsigned preserving*. Under this scheme, an **unsigned char** or **unsigned short** is always promoted to **unsigned int**. Under certain circumstances, a program that depends upon unsigned-preserving promotion will behave differently when subjected to value-preserving promotion, and probably without warning. This is a quiet change that may break some existing code.

An integral type may be converted to a floating type. If its value lies within the range of values that can be represented by the floating type, but it cannot be represented precisely, then its value is rounded to the next highest or next lowest value, depending upon the implementation.

Signed and unsigned integers
The following rules apply when a signed or an unsigned integer is converted to another integral type:

- When a positive, signed integer is promoted to an unsigned integer of the same or larger type, its value is unchanged.
- When a negative integer is promoted to an unsigned integer of the same or larger type, it is first promoted to the signed equivalent of the unsigned type. It is then converted to unsigned by incrementing its value by one plus the maximum value that can be held by the unsigned type. On two's complement machines, the bit pattern of the promoted object does not changed. The only exception is that the sign bit is copied to fill any extra bits of new type, should it be larger than the old type.
- When a signed or unsigned integer is demoted to a smaller, unsigned type, its value is the non-negative remainder that occurs when the value of the original type is divided by one plus the maximum value that can be held by the smaller type.
- When a signed or unsigned integer is demoted to a smaller, signed type, if its value cannot be represented by the new type, the result is implementation-defined.
- When an unsigned integer is converted to a signed type of the same size, if its value cannot be represented by the new type, the result is implementation-defined.

Usual arithmetic conversions

Many binary operators convert their operands and yield a result of a type common to both. The rules that govern such conversions are called the *usual arithmetic conversions*. The following lists the usual arithmetic conversions. If two conflict, the rule *higher* in the list applies:

- If either operand has type **long double**, the other operand is converted to **long double**.
- If either operand has type **double**, the other operand is converted to **double**.
- If either operand has type **float**, the other operand is converted to **float**.
- If either operand has type **unsigned long int**, then the other operand is converted to **unsigned long int**.
- If one operand has the type **long int** and the other operand has type **unsigned int**, the other operand is converted to **long int** if that type can hold all of the values of an **unsigned int**. Otherwise, both operands are promoted to **unsigned long int**.
- If either operand has type **long int**, the other operand is converted to **long int**.
- If either operand has type **unsigned int**, the other operand is converted to **unsigned int**.
- If none of the above rules apply, then both operands have type **int**.

Cross-references

Standard, §3.2
The C Programming Language, ed. 2, pp. 197*ff*

See Also

explicit conversion, function designator, implicit conversion, integral promotions, Language, lvalue, null pointer constant, value preserving, void expression

Notes

The "as if" rule gives implementors some leeway in applying the rules for usual arithmetic conversions. For example, the conversion rules specify that operands of type **char** must first be widened to type **int** before the operation is performed; however, if the same result would be produced by performing the operation on **char** operands, then the operands need not be widened.

LEXICON

Because the Standard now allows single-precision floating-point arithmetic on **float** operands, some round-off error could occur. Casts will force the operands in question to be promoted, and the operation to be carried out with the wider type.

cos() — Mathematics (libm)

Calculate cosine
#include <math.h>
double cos(double *radian***);**

cos calculates and returns the cosine of its argument *radian*, which must be expressed in radians.

Example

For an example of this function, see **sin**.

Cross-references

Standard, §4.5.2.5
The C Programming Language, ed. 2, p. 251

See Also

acos, asin, atan, atan2, mathematics, sin, tan

cosh() — Mathematics (libm)

Calculate hyperbolic cosine
#include <math.h>
double cosh(double *value***);**

cosh calculates and returns the hyperbolic cosine of *value*. A range error will occur if the argument is too large.

Cross-references

Standard, §4.5.3.1
The C Programming Language, ed. 2, p. 251

See Also

mathematics, sinh, tanh

cpp — Command

C preprocessor
cpp [*option…***] [***file…***]**

The command **cpp** calls the preprocessor/parser **cc0** to perform C preprocessing on C programs. It performs the operations described in section 3.8 of the Standard; these include file inclusion, conditional code selection, constant definition, and macro definition. See the entry on **preprocessing** for a full description of the C's preprocessing language.

cpp reads each input *file*, or **stdin** if no file is specified; processes directives, and writes its product on **stdout**. If the option **-E** is not used, **cpp** also writes into its output statements of the form **#***n filename*, so that the parser will be able to connect its error messages and debugger output with the original line numbers in your source files.

Options

The following summarizes **cpp**'s options:

-D*VARIABLE*
Define *VARIABLE* for the preprocessor at compilation time. For example, the command

```
cc -DLIMIT=20 foo.c
```

tells the preprocessor to define the variable **LIMIT** to be 20. The compiled program acts as though the directive **#define LIMIT 20** were included before its first line.

-E Strip all comments and line numbers from the source code. This option is used to preprocess assembly-language files or other sources, and should not be used with the other compiler phases.

-I *directory*
C allows two types of **#include** directives in a C program, i.e., **#include "file.h"** and **#include <file.h>**. The **-I** option tells **cpp** to search a specific directory for the files you have named in your **#include** directives, in addition to the directories that it searches by default. You can have more than one **-I** option on your **cc** command line.

-o *file*
Write output into *file*. If this option is missing, **cpp** writes its output onto **stdout**, which may be redirected.

-U*VARIABLE*
Undefine *VARIABLE*, as if an **#undef** directive were included in the source program. This is used to undefine the variables that **cpp** defines by default.

See Also

cc, preprocessing

creat() — Extended function (libc)

Create/truncate a file
short creat(char **file***, short** *mode***);**

creat creates a new *file* or truncates an existing *file*. It returns a file descriptor that identifies *file* for subsequent system calls. If *file* already exists, its contents are erased.

creat ignores its *mode* argument. This argument exists for compatibility with implementations of **creat** under UNIX and related operating systems.

If the call is successful, **creat** returns a file descriptor. It returns **-1** if it could not create the file, typically because of insufficient system resources, or nonexistent path.

Example

For an example of this routine, see the entry for **open**.

See Also

extended miscellaneous, fopen, fdopen

csreg() — i8086 support (libc)

Get value from CS register
#include <dos.h>
unsigned csreg(void)

csreg returns the value from the i8086 CS register, which points to the base of the code segment.

Example

The following example uses the functions **csreg**, **dsreg**, **esreg**, and **ssreg** to print the contents of the segment registers.

LEXICON

```
#include <dos.h>
#include <stdio.h>
#include <stdlib.h>

main(void)
{
      printf("csreg=%04x\n", csreg());
      printf("dsreg=%04x\n", dsreg());
      printf("esreg=%04x\n", esreg());
      printf("ssreg=%04x\n", ssreg());
      return EXIT_SUCCESS;
}
```

See Also

dsreg, esreg, i8086 support, ssreg

ctime() — Time function (libc)

Convert calendar time to text
#include <time.h>
char *ctime(const time_t ****timeptr*);**

The function **ctime** reads the calendar time pointed to by *timeptr*, and converts it into a string of the form

```
Tue Dec 10 14:14:55 1987\n\0
```

ctime is equivalent to:

```
asctime(localtime(timeptr));
```

timeptr points to type **time_t**, which is defined in the header **time.h**.

Example

This example displays the current time.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main(void)
{
      time_t t;
      time(&t);

      printf(ctime(&t));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.3.2
The C Programming Language, ed. 2, p. 256

See Also

asctime, date and time, gmtime, localtime, strftime, time_t

ctype.h — Header

Header for character-handling functions
#include <ctype.h>

ctype.h is the header file that declares the functions used to handle characters. These are as follows:

isalnum	Check if a character is a numeral or letter
isalpha	Check if a character is a letter
iscntrl	Check if a character is a control character
isdigit	Check if a character is a numeral
isgraph	Check if a character is printable
islower	Check if a character is a lower-case letter
isprint	Check if a character is printable
ispunct	Check if a character is a punctuation mark
isspace	Check if a character is white space
isupper	Check if a character is a upper-case letter
isxdigit	Check if a character is a hexadecimal numeral
tolower	Convert character to lower case
toupper	Convert character to upper case

Cross-references

Standard, §4.3
The C Programming Language, ed. 2, p. 248

See Also

character handling, header, xctype.h