E

ecvt() — Extended function (libc)

Convert floating-point numbers to strings
char *ecvt(double *d*, **int** *prec*, **int** **dp*, **int** ***signp);**

ecvt converts *d* into a null-terminated ASCII string of numerals with the precision of *prec*. Its operation resembles that of **printf**'s **%e** operator. **ecvt** rounds the last digit and returns a pointer to the result. On return, **ecvt** sets *dp* to point to an integer that indicates the location of the decimal point relative to the beginning of the string, to the right if positive, to the left if negative. It sets *signp* to point to an integer that indicates the sign of *d*, zero if positive and nonzero if negative.

Example

The following program demonstrates **ecvt**, **fcvt**, and **gcvt**.

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

/* prototypes for extended functions */
extern char *ecvt(double d, int prec, int *dp, int *signp);
extern char *fcvt(double d, int w, int *dp, int *signp);
extern char *gcvt(double d, int prec, char *buffer);

main(void)
{
      char buf[64];
      double d;
      int i, j;
      char *s;

      d = 1234.56789;
      s = ecvt(d, 5, &i, &j);
      /* prints ecvt="12346" i=4 j=0 */
      printf("ecvt=\"%s\" i=%d j=%d\n", s, i, j);

      strcpy(s, fcvt(d, 5, &i, &j));
      /* prints fcvt="123456789" i=4 j=0 */
      printf("fcvt=\"%s\" i=%d j=%d\n", s, i, j);

      s = gcvt(d, 5, buf);
      /* prints gcvt="1234.56789" */
      printf("gcvt=\"%s\"\n", s);

      return EXIT_SUCCESS;
}
```

See Also

extended miscellaneous, fcvt, frexp, gcvt, ldexp, modf, printf

Notes

ecvt performs conversions within static string buffers that are overwritten by each execution.

egrep — Command

Extended pattern search
egrep [*option …***] [***pattern***] [***file …***]**

The command **egrep** searches each *file* for occurrences of *pattern* (also called a regular expression). If no *file* is specified, it searches what is typed into the standard input. Normally, it prints each line

matching the *pattern*.

Wildcards

The simplest *patterns* accepted by **egrep** are ordinary alphanumeric strings. **egrep** can also process *patterns* that include the following wildcard characters:

^ Match beginning of line, unless it appears immediately after '[' (see below).

$ Match end of line.

***** Match zero or more repetitions of preceding character.

. Match any character except newline.

[*chars***]**
Match any one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

[^*chars***]**
Match any character *except* one of the enclosed *chars*. Ranges of letters or digits may be indicated using '-'.

*****c Disregard special meaning of character *c*.

| Match the preceding pattern *or* the following pattern. For example, the pattern **cat | dog** matches either **cat** or **dog**. A newline within the *pattern* has the same meaning as '|'. Under MS-DOS, the '|' has special meaning, and must be enclosed within apostrophes.

+ Match one or more occurrences of the immediately preceding pattern element; it works like '*', except it matches at least one occurrence instead of zero or more occurrences.

? Match zero or one occurrence of the preceding element of the pattern.

(...) Parentheses may be used to group patterns. For example, **(Ivan)+** matches a sequence of one or more occurrences of the four letters 'I' 'v' 'a' or 'n'.

Because the metacharacters '*' and '?', are also special to MS-DOS, patterns that contain those literal characters must be quoted by enclosing *pattern* within double quotation marks.

Options

The following lists the available options:

-b With each output line, print the block number in which the line started (used to search file systems).

-c Print how many lines match, rather than the lines themselves.

-e The next argument is *pattern* (useful if the pattern starts with '-').

-f The next argument is a file that contains a list of patterns separated by newlines; there is no *pattern* argument.

-h When more than one *file* is specified, output lines are normally accompanied by the file name; **-h** suppresses this.

-l Print the name of each file that contains the string, rather than the lines themselves. This is useful when you are constructing a batch file.

-n When a line is printed, also print its number within the file.

-s Suppress all output, just return exit status.

LEXICON

-v Print a line only if the pattern is *not* found in the line.

-y Lower-case letters in the pattern match lower-case *and* upper-case letters on the input lines. A letter escaped with '\' in the pattern must be matched in exactly that case.

Diagnostics

egrep returns an exit status of zero for success, one for no matches, and two for error.

See Also

commands

Notes

egrep uses a deterministic finite automaton (DFA) for the search. It builds the DFA dynamically, so it begins doing useful work immediately. This means that **egrep** is considerably faster than earlier pattern-searching commands, on almost any length of file.

else — C keyword

Conditionally execute a statement
else *statement*;

else is the flip side of **if**: if the condition described in the **if** statement equals zero, then the statement introduced by **else** is executed. If, however, the condition described in the **if** statement is nonzero, then the statement it introduces is executed and the statement introduced by **else** is ignored.

An **else** statement is associated with the first preceding **else**-less **if** statement that is within the same block, but not within an enclosed block. For example,

```
if(conditional1) {
        if(conditional2)
                statement1
} else
        statement2
```

the **else** is associated with the **if** statement that uses *conditional1*, not the one that uses *conditional2*. On the other hand, in the code

```
if(conditional1)
        if(conditional2)
                statement1
else
        statement2
```

which does not use braces, the **else** is associated with the **if** statement that uses *conditional2*, not the one that uses *conditional1*.

Example

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

Cross-references

Standard, §4.6.4.1
The C Programming Language, ed. 2, pp. 55*ff*

See Also

if, statements, switch

enum — C keyword

Enumerated data type
enum *identifier* **{** *enumerations* **}**

An **enum** is a data type whose possible values are limited to a set of constants.

For example,

```
enum opinion { yes, no, maybe };
```

declares type **opinion** to have one of three constant values; these are identified by the members **yes**, **no**, and **maybe**.

The translator assigns values to the identifiers from left to right, beginning with zero and increasing by one for each successive term. In the above example, the values of **yes**, **no**, and **maybe** are set, respectively, to zero, one, and two. Thus, the following example

```
enum opinion guess;
   . . .
guess = no;
```

sets the value of **guess** to one.

All enumerated identifiers must be distinct from all other identifiers in the program. The identifiers act as constants and are used wherever constants are appropriate.

If a member of an enumeration is followed by an equal sign and an integer, the identifier is assigned the given value and subsequent values increase by one from that value. For example,

```
enum opinion {yes, no=50, maybe};
```

sets the values of the members **yes**, **no**, and **maybe** to zero, 50, and 51, respectively. More than one enumerator can have the same value. For example:

```
enum opinion {yes, no=50, nah=50, nope=50, maybe};
```

assigns duplicate values to the members **no**, **nah**, and **nope**.

An enumeration constant always has type **int**.

Cross-references

Standard, §3.5.2.2
The C Programming Language, ed. 2, p. 39

See Also

type specifier

Notes

Prior to the introduction of enumerated data types in C, programmers would create lists of manifest constants whose values took the values that enumerated constants now take.

Unlike more strongly typed languages, in which enumerated constants are checked to ensure that they are part of the specified set of values, **enum**s in C are only required to be of type **int**. No additional checking is performed on enumeration constants.

enumeration constant — Definition

An *enumeration constant* is a member of an enumeration. This constant has type **int**.

For example, in the enumeration

```
enum example { blue, green, yellow };
```

blue is an enumeration constant.

Cross-references

Standard, §3.1.3.3
The C Programming Language, ed. 2, pp. 39, 194

See Also

constants, enum

environmental variable — Overview

An *environmental variable* is a variable that is set through the operating system, and which a program can read at run time. These variables are most commonly used to change the way a program behaves.

Let's C uses the following environmental variables in its operation:

CCHEAD	Variables at head of compilation command
CCTAIL	Variables at tail of compilation command
INCDIR	Directory that holds include files
LIBPATH	Directories that hold libraries
PATH	Directories that hold executable files
TIMEZONE	Time zone information
TMPDIR	Directory that holds temporary files

Because of the limited environment space available under many version of MS-DOS, the variables **INCDIR**, **LIBPATH**, and **TMPDIR** often are not set. Instead, their information is placed into the file **ccargs**, which is built automatically when you install **Let's C**. You need to set the variable **TIMEZONE** only if you are writing programs that need exact time zone information.

See Also

Environment

envp — Definition

Argument passed to main
char *envp[];

envp is an abbreviation for **env**ironmental **p**arameter. It is the traditional name for a pointer to an array of string pointers passed to a C program's **main** function, and is by convention the third argument passed to **main**.

The MS-DOS runtime startup routines always set **envp** to NULL, i.e., no **envp** is passed. **Let's C** calls **main(***argc***,** *argv***, NULL)**; however, **envp** is significant under some other operating systems, including TOS, UNIX, and COHERENT.

See Also

argc, argv, Environment, main

EOF — Manifest constant

Indicate end of a file
#include <stdio.h>

EOF is an indicator that is returned by several STDIO functions to indicate that the current file position is the end of the file. Its value is defined by the implementation, but a common value is -1 on many systems, which is also a common error return.

The actual bytes used to delineate the end of a file may vary between implementations.

Many STDIO functions, when they read **EOF**, set the end-of-file indicator that is associated with the stream being read. Before more data can be read from the stream, its end-of-file indicator must be cleared. Resetting the file-position indicator with the functions **fseek**, **fsetpos**, or **ftell** will clear the indicator, as will returning a character to the stream with the function **ungetc**.

Example

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

Cross-references

Standard, §4.3, §4.9.1; Rationale, §4.3
The C Programming Language, ed. 2, p. 151

See Also

file, stream, STDIO, stdio.h

errno — Macro

External integer that holds error status
#include <errno.h>

errno is a macro that is defined in the header **errno.h**. It expands to a global integer of type **volatile int**.

When a program begins to execute, **errno** is initialized to zero. Thereafter, whenever a mathematics function or other library function wishes to return information about any error that occurs during its operation, it writes the appropriate error number into **errno**, where it can be read either by the environment or by another function.

The functions **perror** and **strerror** can be used to translate the contents of **errno** into a text message.

Example

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

Cross-references

Standard, §4.1.3
The C Programming Language, ed. 2, p. 248

See Also

EDOM, ERANGE, errno.h, errors, mathematics

Notes

Only certain library functions set **errno**, and then only if certain error conditions occur. Remember that it is your responsibility to clear **errno** before the function in question is called. Other functions may also set **errno**.

LEXICON

Although it is widely believed that a program that checks the value of **errno** after each function is more portable than one that does not, this is not necessarily true. Some implementations use in-line expansion of library function to speed execution, and so forego the use of **errno**. The cautious programmer is best advised to check the value of input arguments before calling a library function and, of course, to check its return value before checking **errno**.

errno.h — Header

Define errno and error codes
#include <errno.h>

errno.h is a header that holds information which relates to the reporting of error conditions. It defines the macro **errno**, which expands to global variable of type **volatile int**. If an error condition occurs, a function can write a value into **errno**, to report just what type of error occurred.

For a list of the MS-DOS system errors described in **errno.h**, see **error**codes**.**

Cross-references

Standard, §4.1.3
The C Programming Language, ed. 2, p. 248

See Also

errno, error codes, errors

escape sequences — Definition

An *escape sequence* is a set of characters that, together, represent one character that may have a special significance. The Standard recognizes the following escape sequences:

\'	Literal apostrophe
\"	Literal quotation mark
\?	Literal question mark
****	Literal backslash
\a	Alert; ring the bell or print visual alert
\b	Horizontal backspace
\f	Form feed; force output device to begin a new page
\n	Newline; move to next line
\r	Carriage return; move to beginning of line
\t	Horizontal tabulation; move to next tabulation mark
\v	Vertical tabulation; move to next tabulation mark
*****NNN*	Octal number
\x*NN*	Hexadecimal number

An escape sequence may be embedded within a character constant or a string literal. In a string literal, the apostrophe may be represented either by itself or by its escape sequence, whereas in a character constant the quotation mark may be represented by itself or by its escape sequence.

Two question marks together may introduce a trigraph, which is interpreted even within a string literal. If you want to print two literal question marks, use the escape sequence **\?\?**. For more information, see **trigraph sequences**.

The escape sequences **\a** through **\v** let you use characters that control the output device.

A backslash followed by one, two, or three octal digits encodes an octal number. For example, in ASCII implementations of C, the escape sequence **'\141'** encodes the octal value 141 into an **int**-length object. When interpreted under an environment that uses ASCII, this prints the letter 'a'. Likewise, the escape sequence **\x** followed by an arbitrary number of hexadecimal digits encodes a

hexadecimal number.

Example

The following example demonstrates the use of the escape sequence **\b**, which prints a backspace character. It prints a message, backspaces over it, and then prints another message.

```
#include <stdio.h>
main()
{
      printf("BLINK!\b\b\b\b\b\bhello, world\n");
}
```

Cross-references

Standard, §2.2.2, §3.1.3.4
The C Programming Language, ed. 2, p. 193

See Also

character constant, constants, string literal, trigraph sequences

Notes

Previous releases of **Let's C** defined the escape sequences **\a** and **\x** differently.

Some implementations of C permit the digit '8' to be used with an octal number. For example, the character constant **'\078'** is regarded by these implementations as being equivalent to octal 100. Under ANSI C, **'\078'** will be interpreted as representing octal 7 plus the character constant **'8'**. This, too, is a quiet change that may break some existing code.

The escape sequence '\0' is used by many existing implementations to represent the null character.

esreg() — i8086 support (libc)

Get value from ES segment register
#include <dos.h>
unsigned esreg(void)

esreg returns the value from from the i8086 ES register, which points to the base of the "extra" segment. In SMALL model, this register always holds the same value as the DS register.

Example

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

See Also

csreg, dsreg, i8086 support, ssreg " ENVIRONMENTS: LC

exargs() — Extended miscellaneous (libc)

Get and parse a command line
int exargs(char **name*, **int** *argc*, **char** **argv*[], **char** **xargv*[], **int** *maxarg*);

exargs provides a uniform mechanism by which programs that are run under MS-DOS can read and parse command lines. It cooperates with the C runtime startup to be as transparent as possible to the user.

The parameters *argc* and *argv* are the usual parameters to **main**. They are parsed from the MS-DOS command tail by **_main** in the C runtime startup routine. **exargs** simply takes all of a command's arguments from *argv*.

exargs parses command lines by breaking them into a list of arguments separated by white space (i.e., a space or tab character). It expands wildcard arguments, writes pointers to the arguments

into the array *xargv*, and returns the number of arguments. **exargs** then puts a **NULL** pointer at the end of the list, so *xargv* looks much like the *argv* parameter to **main**. *maxarg* is the maximum number of arguments that a command can take, that is, the maximum number that will fit into the array *xargv*.

exargs interprets a command line of the form **@***name* as a file reference: it opens the file *name* and reads command lines from it. Such files can also contain references to yet other files.

exargs uses **getenv** to search the environment for the strings *name***HEAD** and *name***TAIL**. If found, it adds the value of *name***HEAD** at the beginning of the argument list and the value of *name***TAIL** at the end. For example, the **cc** command uses **exargs** with a *name* argument of **cc**; accordingly, it looks for **CCHEAD** and **CCTAIL** in the environment to provide command-specific information.

exargs returns the size of its argument list, which is suitable for assignment to **argc**.

Example

The following function converts UNIX and COHERENT utilities to MS-DOS utilities by changing **argc** and **argv** via **exargs**.

```
#define MAXARGS 1023
#include <stdio.h>
#include <stdlib.h>

void
msdoscvt(int *argc, char *name, ***argv)
{
      char **xargv;

      if(NULL == (xargv = malloc((MAXARGS + 1)
            * sizeof(char *))))
            abort();
      *argc = exargs(name, *argc, *argv, &xargv[1], MAXARGS) + 1;
      xargv[0] = name;
      *argv = realloc(xargv, (*argc + 1) * sizeof(char *));
}

/*
 * Expand argument list and display it.
 */
#ifdef TEST
main(int argc, char **argv)
{
      int i;

      msdoscvt("test", &argc, &argv);
      for(i = 0; i < argc; i++)
            printf("Argument %d -- %s\n", i, argv[i]);
      return EXIT_SUCCESS;
}
#endif
```

See Also

cc (-w option), end, extended miscellaneous, malloc, runtime startup

Diagnostics

exargs prints an appropriate message and aborts if it cannot open or read an indirect file, or if there are too many arguments in a command line.

Notes

This routine is specific to MS-DOS, and cannot be ported to other compilers or operating systems.

The **-w** ("wildcards") option to the **cc** command uses a special runtime startup routine that gives **argv** much of the functionality of **exargs**. See the entry for **cc** for more information.

exception — Definition

An *exception* is said to occur when an expression generates a result that cannot be represented by the hardware or defined mathematically, e.g., division by zero. When an exception occurs, behavior is undefined.

Cross-references

Standard, §3.3
The C Programming Language, ed. 2, p. 255

See Also

expressions

execall() — Extended function (libc)

Execute a subprogram
int execall(char *_command_**, char** *_tail_**);**

execall sends a command and its arguments (the "argument tail") directly to MS-DOS. Unlike its cousin **system**, it does not work through **command.com**; therefore, it cannot execute any MS-DOS built-in commands.

execall looks for the executable file pointed to by *command*, loads it into memory, and executes it with the *tail* that is pointed to by *tail*. If *command* has no suffix, **execall** appends **.exe** onto it. When *command* has finished executing, **execall** returns its exit status code to the program that called it.

execall works only if *command* exits by returning to its caller, rather than by executing the MS-DOS system reset function *warm boot*. **execall** can only call programs that exist as executable files. Therefore, it cannot call the MS-DOS built-in commands, such as **dir**, or commands that rely on MS-DOS to parse the command line into the formatted parameter area. You should use **system** for these programs.

Commands compiled by **Let's C** always exit by returning to their callers and always return a useful exit status. Therefore, you can use **execall** to call any program compiled by **Let's C**.

An exit status code of zero (**EXIT_SUCCESS**) means that *command* executed successfully. An exit status code other than zero (**EXIT_FAILURE**) means that it failed. If *command* cannot be located, opened, or executed, an explanatory message is printed on the console, and **execall** returns 0177 (octal).

Example

The following example consists of two brief programs, one of which calls the other. The first program, called **one.c**, does the calling:

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

main(void)
{
      printf("This is 'one'\n");
      printf("\n'two.exe' exited with value %d.\n",
            execall("two", ""));
      printf("Good-bye.\n");
      return EXIT_SUCCESS;
}
```

LEXICON

The second program, **two.c**, is called by **one.c**, and returns a value to it:

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

main(void)
{
      printf("\nHere is 'two'.\n");
      printf("I'm exiting with value %d\n", EXIT_SUCCESS);
      exit(EXIT_SUCCESS);
}
```

Compile these programs, then run **one.exe**. It will call **two.exe** for execution.

See Also

exargs, extended miscellaneous, system

Notes

execall does not fill in the formatted parameter areas.

executable file — Definition

An **executable file** is one that can be loaded directly by the operating system and executed. Normally, an executable file is one that has both been *compiled*, where it is rendered into machine language, and *linked*, where the compiled program has received all operating system-specific information and library functions.

See Also

Definitions, file

exit() — General utility (libc)

Terminate a program gracefully
#include <stdlib.h>
void exit(int *status***);**

exit terminates a program gracefully. Unlike the function **abort**, **exit** performs all processing that is necessary to ensure that buffers are flushed, files are closed, and allocated memory is returned to the environment.

When it is called, **exit** does the following:

1. It executes all functions registered by the function **atexit**, in reverse order of registration. These functions must execute as if **main** had returned. If any function accesses an **auto**, its behavior is undefined.
2. It flushes all buffers associated with output streams, closes the streams, and removes all files created by the function **tmpfile**.
3. It returns control to the host environment. If *status* is zero or **EXIT_SUCCESS**, then the program indicates to the environment that the program terminated with success. If *status* is set to **EXIT_FAILURE**, then the program indicates that the program terminated with failure.

exit does not return to its caller.

Example

This program exits, and returns the first argument on the command line to MS-DOS as an exit code.

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

main(int argc, char *argv[])
{
      if(argc == 1)
            exit(EXIT_SUCCESS);
      else
            exit(atoi(argv[1]));
}
```

Cross-references

Standard, §4.10.4.3
The C Programming Language, ed. 2, p. 252

See Also

_exit, abort, atexit, general utilities, getenv, system

explicit conversion — Definition

The term *explicit conversion* refers to the deliberate changing of an object's type by means of a cast operation.

For example, one type of pointer can be cast to another, as follows:

```
char *charptr;
int  *intptr;
   . . .
intptr = (int *)charptr;
```

A cast can be used to defeat optimizations performed by the translator. For instance, if an implementation performs single-precision arithmetic on operands of type **float**, an explicit cast will force the operation to be performed in the wider type **double**:

```
float f1, f2, f3;
   . . .
f3 = (double) f1 * f2;
```

Cross-references

Standard, §3.2
The C Programming Language, ed. 2, p. 45

See Also

(), cast operators, conversions, implicit conversion

Notes

A cast is not an lvalue. This renders constructs such as

```
(int *)pointer++; /* WRONG */
```

invalid under ANSI C.

LEXICON

extended character handling — Overview

#include <xctype.h>

In addition to the character-handling functions described in the Standard, **Let's C** includes the following extended character-handling functions and macros:

_tolower	Change a character to lower case
_toupper	Change a character to upper case
isascii	See if a character is in the ASCII character set
toascii	Convert a character to printable ASCII

These functions and macros are declared or defined in the header **xctype.h**. In previous releases of **Let's C**, they had been declared in the header **ctype.h**. This change was made to conform to the Standard, and may require that some code be altered.

A program that uses any of these routines no longer conforms strictly to the Standard, and may not be portable to other compilers or environments.

See Also

character handling, extended mathematics, extended miscellaneous, extended STDIO, extended time, xctype.h

extended time — Overview

#include <xtime.h>

Let's C includes a number of extensions to the ANSI Standard's set of time functions. These are designed to increase the scope and accuracy of the Standard, and to ease calculation of some time elements.

To begin, **Let's C** includes three variables that are used by the function **localtime**. It parses the environmental variable **TIMEZONE** into the following:

timezone	Seconds from UTC to give local time
dstadjust	Seconds to local standard, if any
tzname	Array with names of standard and daylight times

The following functions return information about the calendar:

isleapyear	Is this year AD a leap year?
dayspermonth	How many days in this historical month?

The way **Let's C** models time is based on the method used by the COHERENT operating system. As noted above, the variable **time_t** is defined as the number of seconds since January 1, 1970, 0h00m00s UTC. This moment, in turn, is rendered as day 2,440,587.5 on the Julian calendar. This allows accurate calculation of time as far back as January 1, 4713 B.C.

Conversion to the Gregorian calendar is set to October 1582, when it was first adopted in Rome. The issue of when a nation changed from the Julian to the Gregorian calendar is moot in the United States, Canada (except Quebec), Asia, Africa, Australia, and the Middle East; however, users in Quebec, Latin America, Europe, the Soviet Union, and European-influenced areas of Asia (e.g., India) may wish to to write their own functions to convert historical data properly from the Julian to the Gregorian calendar.

The following functions assist in conversion from Julian to Gregorian time:

time_to_jday	Convert **time_t** to the Julian date
jday_to_time	Convert Julian date to **time_t**
tm_to_jday	Convert **tm** structure to Julian date
jday_to_tm	Convert Julian date to **tm** structure

These functions are not described in the ANSI Standard. A program that uses any of these functions does not conform strictly to the Standard, and may not be portable to other compilers or environments.

See Also

date and time, extended character handling, extended mathematics, extended miscellaneous, extended STDIO, Library, xtime.h

Notes

To conform to the ANSI Standard, all of these functions were moved from the header **time.h** to the header **xtime.h**. This may require that some code be altered.

extern — C keyword

External linkage
extern *type identifier*

The storage-class specifier **extern** declares that *identifier* has external linkage.

Cross-references

Standard, §3.5.1
The C Programming Language, ed. 2, pp. 210, 211

See Also

linkage, storage-class specifiers

external definitions — Overview

A *definition* is a declaration that reserves storage for the thing declared. An *external definition* is a definition whose identifier is defined outside of any function. This makes the object available throughout the file or the program, depending upon whether it has, respectively, internal or external linkage.

If an identifier has external linkage and is used in an expression (except as an operand to the **sizeof** operator), then an external definition must exist for that identifier somewhere in the program.

There are two varieties of external definition: **function definitions** and **object definitions**. See the appropriate entries for more information.

Cross-references

Standard, §3.7
The C Programming Language, ed. 2, p. 226

See Also

declaration, definition, function definition, linkage, object definition

external name — Definition

An *external name* is an identifier that has external linkage. The number and range of characters that may form an external name depends upon the implementation. The minimum maximum for the length of an external name is six characters, and an implementation is not obliged to recognize both upper-case and lower-case characters. An implementation may exceed these limits.

Cross-references

Standard, §3.1.2
The C Programming Language, ed. 2, p. 35

See Also

identifiers, internal name, linkage