M

main — Definition

main is the name of the function that is called when a program begins execution under a hosted environment. A program must have one function named **main.** This function is special not only because it marks the beginning of program execution, but because it is the only function that may be called with either zero arguments or two arguments:

```
int main(void) {  }
```

or

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

Let's C allows **main** to take three arguments. Programs that use more than two arguments to **main**, however, do not conform strictly to the Standard.

The two **standard** arguments to **main** are called **argc** and **argv**. These names are used by convention; a programmer may use any names he wishes.

argv points to an array of pointers to strings. These strings can modify the operation of the program; thus, they are called *program parameters*. **argc** gives the number of strings in the array to which **argv** points.

The third variable to **main**, which is specific to **Let's C**, is **envp**. This variable points to an array of pointers to environmental variables.

If **main** calls **return**, it is equivalent to its calling **exit** with the same parameter. For example, the statement

```
return(EXIT_SUCCESS);
```

in **main** is equivalent to the call

```
exit(EXIT_SUCCESS);
```

If **main** returns without returning a value to the host environment, the value that is returned to the host environment is undefined.

Cross-references

Standard, §2.1.2.2
The C Programming Language, ed. 2, pp. 6, 164

See Also

argc, argv, Environment, envp

main — Technical information

Introduce program's main function

A C program consists of a set of functions, one of which must be called **main**. This function is called from the runtime startup routine after the runtime environment has been initialized.

Programs can terminate in one of two ways. The easiest is simply to have the **main** routine **return**. Control returns to the runtime startup; it closes all open file streams and otherwise cleans up, and then returns control to the operating system, passing it the value returned by **main** as exit status.

In some situations (errors, for example), it may be necessary to stop a program, and you may not want to return to **main**. Here, you can use **exit**; it cleans up the debris left by the broken program and returns control directly to the operating system.

A second exit routine, called **_exit**, quickly returns control to the operating system without performing any cleanup. This routine should be used with care, because bypassing the cleanup will leave files open and buffers of data in memory.

Programs compiled by **Let's C** return to the program that called them; if they return from **main** with a value or call **exit** with a value, that value is returned to their caller. Programs that invoke other programs through the **system** or **execall** functions check the returned value to see if these secondary programs terminated successfully.

See Also

argc, argv, envp, exit, _exit, runtime startup

make — Command

Program building discipline
make [*option …***] [***argument …***] [***target …***]**

make helps you build programs that consist of more than one file of source code.

Complex programs often consist of several *object modules*, each of which is the product of compiling a *source file*. A source file may refer to one or more **include** files, which can also be changed. Some programs may be generated from specifications given to program generators, such as **yacc**. Recompiling and relinking complicated programs can be difficult and tedious.

make regenerates programs automatically. It follows a specification of the structure of the program that you write into a file called **makefile**. **make** also checks the date and time that MS-DOS has recorded for each source file and its corresponding object module. To avoid unnecessary recompilation, **make** will recompile a source file only if it has been altered since its object module was last compiled.

The makefile

A **makefile** consists of three types of instructions: *macro definitions*, *dependency definitions*, and *commands*.

A macro definition simply defines a macro for use throughout the **makefile**. For example, the macro definition

```
FILES=file1.obj file2.obj file3.obj
```

The use of the equal sign '='.

A dependency definition names the object modules used to build the target program, and source files used to build each object module . It consists of the *target name*, or name of the program to be created, followed by a colon ':' and the names of the object modules that build it. For example, the statement

```
example:  $(FILES)
```

uses the macro **FILES** to name the object modules used to build the program **example**. Likewise, the dependency definition

```
file1.obj:  file1.c macros.h
```

defines the object module **file1.obj** as consisting of the source file **file1.c** and the header file **macros.h**.

Finally, a command line details an action that **make** must perform to build the target program. Each command line must begin with a space or tab character. For example, the command line

```
        cc -o example $(FILES)
```

gives the **cc** command needed to build the program **example**. The **cc** command lists the *object modules* to be used, *not* the source files.

Finally, you can embed comments within a **makefile**. **make** recognizes any line that begins with a pound sign '#' as being a comment, and ignores it.

make searches for **makefile** first in directories named in the environmental variable **PATH**, and then in the current directory.

Dependencies

The **makefile** specifies which files depend upon other files, and how to recreate the dependent files. For example, if the target file **test** depends upon the object module **test.obj**, the dependency is as follows:

```
test: test.obj
      cc -o test test.obj
```

make knows about common dependencies, e.g., that **.obj** files depend upon **.c** files with the same base name. The target **.SUFFIXES** contains the suffixes that **make** recognizes.

make also has a set of rules to regenerate dependent files. For example, for a source file with suffix **.c** and a dependent file with the suffix **.obj**, the target **.c.obj** gives the regeneration rule:

```
.c.obj:
      cc -c $<
```

The **-c** option to the **cc** commands tells **cc** not to link or erase the compiled object module. **$<** is a macro that **make** defines. It stands for the name of the file that causes the current action. The default suffixes and rules are kept in the files **mmacros** and **mactions**. The dependencies can be changed by editing these files.

Both of these must be kept in a directory named by the environmental variable **LIBPATH**. You can set this variable with the **set** command. For example, placing the command

```
set LIBPATH=\bin;\lib
```

into **autoexec.bat** sets **LIBPATH** to **\bin** and **\lib**. If **LIBPATH** is not set, the default directory is **\lib**.

Macros

To simplify the writing of complex dependencies, **make** provides a *macro* facility. To define a macro, write

NAME = string

The *string* is terminated by the end-of-line character, so it can contain blanks. To refer to the value of the macro, use a dollar sign '$' followed by the macro name enclosed in parentheses:

$(**NAME**)

If the macro name is one character, parentheses are not necessary. **make** uses macros in the definition of default rules:

```
.c.obj:
      $(CC) $(CFLAGS) -c $<
```

where the macros are defined as

```
CC=cc
CFLAGS=-V
```

The other built-in macros are:

$*	Target name, minus suffix
$@	Full target name
$<	List of referred files
$?	Referred files newer than target

Each command line *argument* should be a macro definition of the form

```
OBJECT=a.obj b.obj
```

You can override any built-in macro by resetting its value in the environment. For example, setting the following environmental variable

```
set CFLAGS=-VLARGE, -VCSD
```

ensures that **make** will always interpret the macro **CFLAGS** as meaning **-VLARGE,** regardless of how it is otherwise set in any file.

Options

The following lists the options that can be passed to **make** on its command line.

-d (Debug) Give verbose printout of all decisions and information going into decisions.

-f *file* *file* contains the **make** specification. If this option does not appear, **make** uses the file **makefile**, which is sought first in the directories named in the **PATH** environmental variable, and then in the current directory.

-i Ignore all errors from commands, and continue processing. Normally, **make** exits if a command returns an error.

-n Test only; suppresses actual execution of commands.

-p Print all macro definitions and target descriptions.

-q Return a zero exit status if the targets are up to date. Do not execute any commands.

-r Do not use the built-in rules that describe dependencies.

-s Do not print command lines when executing them. Commands preceded by '@' are not printed, except under the **-n** option.

-t (Touch option) Force the dates of targets to be the current time, and bypass actual regeneration.

See Also

as, cc, commands, ld

Diagnostics

make reports its exit status if it is interrupted or if an executed command returns error status. It replies "Target *name* not defined" or "Don't know how to make target *name*" if it cannot find appropriate rules.

Notes

The order of items in **mmacros\.SUFFIXES** is significant. The consequent of a default rule (e.g., **.obj**) must *precede* the antecedent (e.g., **.c**) in the entry **.SUFFIXES**. Otherwise, **make** will not work properly.

malloc() — General utility (libc)

Allocate dynamic memory
#include <stdlib.h>
void *malloc(size_t *size***);**

malloc allocates a block of memory *size* bytes long.

malloc uses a circular, first-fit algorithm to select an unused block of at least *size* bytes, marks the portion it uses, and returns a pointer to it. The function **free** returns allocated memory to the free memory pool.

Each area allocated by **malloc** is rounded up to the nearest even number and preceded by an **unsigned int** that contains the true length. Thus, if you ask for one byte, you will get four, and the **unsigned** that precedes the newly allocated area will be set to four.

When an area is freed, its low order bit is turned on. Consolidation occurs when **malloc** passes over an area as it searches for space. The end of each arena contains a block with a length of zero, followed by a pointer to the next arena. Arenas point in a circle.

The most common problem with **malloc** occurs when a program modifies more space than it allocates with **malloc**. This can cause later **malloc**s to go into a loop.

malloc returns a pointer to the block of memory it has allocated. The pointer is aligned for any type of object. If it could not allocate the amount of memory requested, it returns NULL.

Example

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

Cross-references

Standard, §4.10.3.3
The C Programming Language, ed. 2, p. 167

See Also

__end, alignment, arena, calloc, free, general utilities, lmalloc, realloc

Notes

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

manifest constant — Definition

A *manifest constant* is a value that has been given a name.

The following demonstrates the definition of a manifest constant:

```
#define MAXFILES 9
```

Here, the constant **MAXFILES** is defined as having the value of nine. During the preprocessing phase of translation, the translator will substitute the character ‘9’ for **MAXFILES** wherever it appears — or behave as if it had made such a substitution.

These constants serve two purposes within a C program: First, a constant can be changed throughout the program simply by changing its definition. Second, a programmer who reads the program will find it easier to understand the meaning of a well-named manifest constant than to understand its numeric analogue; for example, it is easy to grasp that **MAXFILES** represents the maximum number of files, but it is not nearly as easy to understand what **9** means.

Manifest constants have file scope, unless undefined with an **#undef** directive.

Cross-reference

The C Programming Language, ed. 2, p. 230

See Also

Definitions, macro, scope

Notes

The C Programming Language calls these constants *symbolic constants.*

math.h — Header

Header for mathematics functions
#include <math.h>

math.h is the header file that declares and defines mathematical functions and macros.

The Standard describes three manifest constants to be included in **math.h**, as follows:

EDOM	Domain error
ERANGE	Range error
HUGE_VAL	Unrepresentable object

The first two are used to set the global variable **errno** to an appropriate value when, respectively, a domain error or a range error occurs. **HUGE_VAL** is returned when any mathematics function attempts to calculate a number that is too large to be encoded into a **double**.

Let's C also includes 27 mathematics functions. For a listing of them, see **mathematics**.

Cross-references

Standard, §4.5
The C Programming Language, ed. 2, p. 250

See Also

header, mathematics

mathematics — Overview

The Standard describes 22 mathematics functions that are to be included with every conforming implementation of C, as follows:

60u
Exponent-log functions

exp	Compute exponential
frexp	Fracture floating-point number
ldexp	Load floating-point number
log	Compute natural logarithm
log10	Compute common logarithm
modf	Separate floating-point number

Hyperbolic functions

cosh	Calculate hyperbolic cosine
sinh	Calculate hyperbolic sine
tanh	Calculate hyperbolic tangent

Integer, value, remainder

ceil	Set integral ceiling of a number
fabs	Compute absolute value
floor	Set integral floor of a number
fmod	Calculate modulus for floating-point number

Power functions

pow	Raise one number to the power of another
sqrt	Calculate the square root of a number

Trigonometric functions

acos	Calculate inverse cosine
asin	Calculate inverse sine
atan	Calculate inverse tangent
atan2	Calculate inverse tangent
cos	Calculate cosine
sin	Calculate sine
tan	Calculate tangent

The Standard reserves all names that match those in this section and have a suffix of **f** or **l**, e.g., **ftan** or **ltan**. A future version of the Standard may provide additional library support for functions that manipulate **float**s or **long double**s.

Some existing implemetations may, on detection of domain or range errors, or other exceptional conditions, allow the function in question to call a user-specified exception handler, **matherr**. UNIX implementations have traditionally behaved this way. The Standard, in trying to accommodate a wide range of floating-point implementations, does not allow this behavior.

Cross-references

Standard, §4.5
The C Programming Language, ed. 2, p. 250

See Also

domain error, range error, Library, math.h

Notes

The Standard excludes the functions **ecvt**, **fcvt**, and **gcvt**, on the grounds that everything they do can be done more easily by the function **sprintf**.

maxmem — External data

extern unsigned int maxmem;
maxmem is an external variable that sets the maximum size of the program's data area. You can set **maxmem** in your program to protect a portion of memory from the memory allocation routine **sbrk**. Otherwise, **maxmem** is set to the end of physical memory by the C runtime startup routine.

See Also

__end, Environment, malloc, sbrk

mblen() — General utility (libc)

Return length of a string of multibyte characters
#include <stdlib.h>
int mblen(const char **address*, **size_t** *number***);**

The function **mblen** checks to see if the *number* or fewer bytes of storage pointed to by *address* form a legitimate multibyte character. If they do, it returns the number of bytes that comprise that character. This function is equivalent to the call

```
	mbtowc((wchar_t *)0, address, number);
```

If *address* is equivalent to NULL, then **mblen** returns zero if the current multibyte character set does not have state-dependent encodings and nonzero if it does. If *address* is not NULL, then **mblen** returns the following: (1) If *address* points to a null character, then **mblen** returns zero. (2)

If the *number* or fewer bytes pointed to by *address* forms a legitimate multibyte character, then **mblen** returns the number of bytes that comprise the character. (3) Finally, if the *number* bytes pointed to by *address* do not form a legitimate multibyte character, **mblen** returns -1. In no instance is the value returned by **mblen** greater than *number* or the value of the macro **MB_CUR_MAX**, whichever is less.

Cross-reference

Standard, §4.10.7.1

See Also

general utility, MB_CUR_MAX, mbtowc, wchar_t, wctomb

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

mbstowcs() — General utility (libc)

Convert sequence of multibyte characters to wide characters
#include <stdlib.h>
size_t mbstowcs(wchar_t ****widechar*, const char ****multibyte***, size_t** *number***);**

The function **mbstowcs** converts a sequence of multibyte characters to their corresponding wide characters. It is the same as a series of calls of the type:

```
mbtowc(widechar, multibyte, MB_LEN_MAX);
```

except that the call to **mbstowcs** does not affect the internal state of **mbtowc**.

multibyte points to the base of the sequence of multibyte characters to be converted to wide characters. *widechars* points to the area where the converted characters are written, and *number* is the number of characters to convert. **mbstowcs** converts characters until either it reads a null character, or until it has converted *number* characters. In the latter case, then, no null character is written onto the end of the sequence of wide characters.

mbstowcs returns -1 cast to **size_t** if it encounters an invalid multibyte character before it has converted *number* multibyte characters. Otherwise, it returns the number of multibyte characters it converted to wide characters, excluding the null character that ends the sequence.

Cross-reference

Standard, §4.10.7.2

See Also

general utilities, wcstombs

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

mbtowc() — General utility (libc)

Convert a multibyte character to a wide character
#include <stdlib.h>
int mbtowc(wchar_t ****charptr*, const char ****address***, size_t** *number***);**

The function **mbtowc** converts *number* or fewer bytes at *address* from a multibyte character to a wide character and stores the result in the area pointed to by *charptr*.

The behavior of **mbtowc** varies depending upon the values of *address* and *charptr*, as follows:

1. If *address* and *charptr* each point to a value other than NULL, then **mbtowc** reads the area pointed to by *address* and checks to see if *number* or fewer bytes comprise a legitimate multibyte character.

   If they do, then **mbtowc** stores the wide character that corresponds to that multibyte character in the area pointed to by *charptr* and returns the number of bytes that form the multibyte character.

   If *address* does not point to the beginning of a legitimate multibyte character, then **mbtowc** returns -1.

   Finally, if *address* points to a null character, **mbtowc** returns zero.

   In no instance does the value returned by **mbtowc** exceed *number* or value of the macro **MB_CUR_MAX**, whichever is less.
2. If *charptr* is set to NULL and *address* is set to a value other than NULL, then **mbtowc** behaves exactly like the function **mblen**: it examines the area pointed to by *address* but does not convert the multibyte character to a wide character.
3. If *address* is set to NULL, or both *address* and *charptr* are set to NULL, then **mbtowc** checks to see if the current multibyte character set have state-dependent encodings. **mbtowc** returns zero if the set does not have state-dependent encodings, and a number greater than zero if it does. It does not store anything in the area pointed to by *charptr*.

Cross-reference

Standard, §4.10.7.3

See Also

general utilities, MB_CUR_MAX, mblen, wchar_t, wctomb

Notes

The operation of this function is affected by the program's locale, as set by the function **setlocale**. See **localization** for more information.

me — Command

MicroEMACS screen editor
me [-e] [*file ...***]**

me is the command for MicroEMACS, the screen editor used by **Let's C**. With MicroEMACS, you can insert text, delete text, move text, search for a string and replace it, and perform many other editing tasks. MicroEMACS reads text from files and writes edited text to files. It can edit several files simultaneously, while displaying the contents of each file in its own screen window.

Screen Layout

If the command **me** is used without arguments, MicroEMACS opens an empty buffer. If used with one or more file name arguments, MicroEMACS will open each of the files named, and display its contents in a window. If a file cannot be found, MicroEMACS will assume that you are creating it for the first time, and create an appropriately named buffer and file descriptor for it.

The last line of the screen is used to print messages and inquiries. The rest of the screen is portioned into one or more *windows* in which MicroEMACS displays text. The last line of each window shows whether the text has been changed, the name of the buffer, and the name of the file associated with the window.

MicroEMACS notes its *current position*. It is important to remember that the current position is always to the *left* of the cursor, and lies *between* two letters, rather than at one letter or another.

For example, if the cursor is positioned at the letter 'k' of the phrase "Mark Williams", then the current position lies *between* the letters 'r' and 'k'.

Commands and Text

The printable ASCII characters, from **<space>** to '~', can be inserted at the current position. Control characters and escape sequences are recognized as *commands*, described below. A control character can be inserted into the text by prefixing it with **<ctrl-Q>** (that is, hold down the **<control>** key and type the letter 'Q').

There are two types of commands to remove text. *Delete* commands remove text and throw it away, whereas *kill* commands remove text but save it in the *kill buffer*. Successive kill commands append text to the previous kill buffer. Moving the cursor before you kill a line will empty the kill buffer, and write the line just killed into it.

Search commands prompt for a search string terminated by **<return>** and then search for it. Case sensitivity for searching can be toggled with the command **<esc>@**. Typing **<return>** instead of a search string tells MicroEMACS to use the previous search string.

Some commands manipulate words rather than characters. MicroEMACS defines a word as consisting of all alphabetic characters, plus '_' and '$'. Usually, a character command is a control character and the corresponding word command is an escape sequence. For example, **<ctrl-F>** moves forward one character and **<esc>F** moves forward one word. The MicroEMACS commands are not case sensitive. For example, **<ctrl-F>** and **<ctrl-f>** are identical.

Text can also be handled in blocks. MicroEMACS defines a block of text as all the text that lies between the *mark* and the current position of the cursor. For example, typing **<ctrl-W>** kills all text from the mark to the current position of the cursor. This is useful when moving text from one file to another. When you invoke MicroEMACS, the mark is set at the beginning of the file. You can reset the mark to the cursor's current position by typing **<ctrl-@>**.

Using MicroEMACS With the Compiler

MicroEMACS can be invoked automatically by the compiler command **cc** to help you repair all errors that occur during compilation. The **-A** option to **cc** causes MicroEMACS to be invoked automatically when an error occurs. The compiler error messages are displayed in one window, the source code in the other, and the cursor is at the line on which the first error occurred. When the text is altered, exiting from MicroEMACS automatically recompiles the file.

This cycle will continue either until the file compiles without error, or until you break the cycle by typing **<ctrl-U> <ctrl-X> <ctrl-C>**.

The option **-e** to the **me** command allows you to invoke the error buffer by hand.

The MicroEMACS Help Facility

MicroEMACS has a built-in help facility. With it, you can ask for information either for a word that you type in, or for a word over which the cursor is positioned. The MicroEMACS help file contains the bindings for all library functions and macros included with **Let's C**.

For example, consider that you are preparing a C program and want more information about the function **fopen**. Type **<ctrl-X>?**. At the bottom of the screen will appear the prompt

```
	Topic:
```

Type **fopen**. MicroEMACS will search its help file, find its entry for **fopen**, then open a window and print the following:

```
	Open a stream for standard I/O
	#include <stdio.h>
	FILE *fopen (name, type) char *name, *type;
```

If you wish, you can kill the information in the help window and copy it into your program, to ensure that you prepare the function call correctly.

Consider, however, that you are checking a program written earlier, and you wish to check the call for a call to **fopen**. Simply move the cursor until it is positioned over one of the letters in **fopen**, then type **<esc>?**. MicroEMACS will open its help window, and show the same information it did above.

To erase the help window, type **<esc>2**.

Options

The following list gives the MicroEMACS commands. They are grouped by function, e.g., *Moving the cursor*. Some commands can take an *argument*, which specifies how often the command is to be executed. The default argument is 1. The command **<ctrl-U>** introduces an argument. By default, it sets the argument to four. Typing **<ctrl-U>** followed by a number sets the argument to that number. Typing **<ctrl-U>** followed by one or more **<ctrl-U>**s multiplies the argument by four.

Moving the Cursor

<ctrl-A>	Move to start of line.
<ctrl-B>	(Back) Move backward by characters.
<esc>B	Move backward by words.
<ctrl-E>	(End) Move to end of line.
<ctrl-F>	(Forward) Move forward by characters.
<esc>F	(Forward) Move forward by words.
<esc>G	Go to an absolute line number in a file. Same as **<ctrl-X>G**.
<ctrl-N>	(Next) Move to next line.
<ctrl-P>	(Previous) Move to previous line.
<ctrl-V>	Move forward by pages.
<esc>V	Move backward by pages.
<ctrl-X>=	Print the current position.
<ctrl-X>G	Go to an absolute line number in a file. Can be used with an argument. Otherwise, it will prompt for a line number. Same as **<esc>G**.
<esc>!	Move the current line to the line within the window given by *argument*. The position is in lines from the top if positive, in lines from the bottom if negative, and the center of the window if zero.
<esc><	Move to the beginning of the current buffer.
<esc>>	Move to the end of the current buffer.

Killing and Deleting

<ctrl-D>	(Delete) Delete next character.
<esc>D	Kill the next word.

<ctrl-H> If no argument, delete previous character. Otherwise, kill *argument* previous characters.

<ctrl-K> (Kill) With no argument, kill from current position to end of line; if at the end, kill the newline. With argument set to one, kill from beginning of line to current position. Otherwise, kill *argument* lines forward (if positive) or backward (if negative).

<ctrl-W> Kill text from current position to mark.

<ctrl-X><ctrl-O>
Kill blank lines at current position.

<ctrl-Y> (Yank back) Copy the kill buffer into text at the current position. Set current position to the end of the new text.

<esc><ctrl-H>
Kill the previous word.

<esc><DEL>
Kill the previous word.

<DEL> If no argument, delete the previous character. Otherwise, kill *argument* previous characters.

Windows

<ctrl-X>1 Display only the current window.

<ctrl-X>2 Split the current window into two windows. This command is usually followed by **<ctrl-X>B** or **<ctrl-X><ctrl-V>**.

<ctrl-X>N (Next) Move to next window.

<ctrl-X>P (Previous) Move to previous window.

<ctrl-X>Z Enlarge the current window by *argument* lines.

<ctrl-X><ctrl-N>
Move text in current window down by *argument* lines.

<ctrl-X><ctrl-P>
Move text in current window up by *argument* lines.

<ctrl-X><ctrl-Z>
Shrink current window by *argument* lines.

Buffers

<ctrl-X>B (Buffer) Prompt for a buffer name, and display the buffer in the current window.

<ctrl-X>K (Kill) Prompt for a buffer name and delete it.

<ctrl-X><ctrl-B>
Display a window showing the change flag, size, buffer name, and file name of each buffer.

<ctrl-X><ctrl-F>
(File name) Prompt for a file name for current buffer.

<ctrl-X><ctrl-R>
(Read) Prompt for a file name, delete current buffer, and read the file.

<ctrl-X><ctrl-V>
(Visit) Prompt for a file name and display the file in the current window.

Saving Text and Exiting

<ctrl-X><ctrl-C>
Exit without saving text.

<ctrl-X><ctrl-S>
(Save) Save current buffer to the associated file.

<ctrl-X><ctrl-W>
(Write) Prompt for a file name and write the current buffer to it.

<ctrl-Z> Save current buffer to associated file and exit.

Compilation Error Handling

<ctrl-X>> Move to next error.

<ctrl-X>< Move to previous error.

Search and Replace

<ctrl-R> (Reverse) Incremental search backward. A pattern is sought as each character is typed.

<esc>R (Reverse) Search toward the beginning of the file. Waits for entire pattern before search begins.

<ctrl-S> (Search) Incremental search forward. A pattern is sought as each character is typed.

<esc>S (Search) Search toward the end of the file. Waits for entire pattern before search begins.

<esc>% Search and replace. Prompt for two strings, then search for the first string and replace it with the second.

<esc>/ Search for next occurrence of a string entered with the **<esc>S** or **<esc>R** commands. This remembers whether the previous search had been forward or backward.

<esc>@ Toggle case sensitivity for searches. By default, searches are case insensitive.

Keyboard Macros

<ctrl-X>(Begin a macro definition. MicroEMACS collects everything typed until the next **<ctrl-X>)** for subsequent repeated execution. **<ctrl-G>** breaks the definition.

<ctrl-X>) End a macro definition.

<ctrl-X>E (Execute) Execute the keyboard macro.

Change Case of Text

<esc>C (Capitalize) Capitalize the next word.

<ctrl-X><ctrl-L>
(Lower) Convert all text from current position to mark into lower case.

<esc>L (Lower) Convert the next word to lower case.

<ctrl-X><ctrl-U>
(Upper) Convert all text from current position to mark into upper case.

<esc>U (Upper) Convert the next word to upper case.

White Space

<ctrl-I> Insert a tab.

<ctrl-J> Insert a new line and indent to current level. This is often used in C programs to preserve the current level of indentation.

<ctrl-M> (Return) If the following line is not empty, insert a new line. If empty, move to next line.

<ctrl-O> Open a blank line; that is, insert newline after the current position.

<tab> With argument, set tab fields at every *argument* characters. An argument of zero restores the default of eight characters. Setting the tab to any character other than eight causes space characters to be set in your file instead of tab characters.

Send Commands to Operating System

<ctrl-C> Suspend MicroEMACS and pass commands to MS-DOS. Typing **exit** returns you to MicroEMACS and allows you to resume editing.

<ctrl-X>! Prompt for an MS-DOS command and execute it.

Setting the Mark

<ctrl-@> Set mark at current position.

<esc>. Set mark at current position.

<ctrl><space>
Set mark at current position.

Help Window

<ctrl-X>? Prompt for word for which information is needed.

<esc>? Search for word over which cursor is positioned.

<esc>2 Erase help window.

Miscellaneous

<ctrl-G> Abort a command.

<ctrl-L> Redraw the screen.

<ctrl-Q> (Quote) Insert the next character into text; used to insert control characters.

<esc>Q (Quote) Insert the next control character into the text. Same as **<ctrl-Q>**.

<ctrl-T> Transpose the characters before and after the current position.

<ctrl-U> Specify a numeric argument, as described above.

<ctrl-U><ctrl-X><ctrl-C>
Abort editing and re-compilation. Use this command to abort editing and return to MS-DOS when you are using the **-A** option to the **cc** command.

<ctrl-X>F Set word wrap to *argument* column. If argument is one, set word wrap to cursor's current position.

<ctrl-X><ctrl-X>
Mark the current position, then jump to the previous setting of the mark. This is useful when moving text from one place in a file to another.

MicroEMACS prints error messages on the bottom line of the screen. It prints informational messages (enclosed in square brackets '[' and ']' to distinguish them from error messages) in the same place.

MicroEMACS manipulates text in memory rather than in a file. The file on disk is not changed until you save the edited text. MicroEMACS prints a warning and prompts you whenever a command would cause it to lose changed text.

See Also

commands

Notes

Because MicroEMACS keeps text in memory, it does not work for extremely large files. It prints an error message if a file is too large to edit. If this happens when you first invoke a file, you should exit from the editor immediately. Otherwise, your file on disk will be truncated. If this happens in the middle of an editing session, however, delete text until the message disappears, then save your file and exit. Due to the way MicroEMACS works, saving a file after this error message has appeared will take more time than usual.

This version of MicroEMACS does not include many facilities available in the original EMACS display editor, which was written by Richard Stallman at M.I.T. In particular, it does not include user-defined commands or pattern search commands.

The current version of MicroEMACS, including source code, is proprietary to Mark Williams Company. The code may be altered or otherwise changed for your personal use, but it may *not* be used for commercial purposes, and it may not be distributed without prior written consent by Mark Williams Company.

MicroEMACS is based upon the public domain editor by David G. Conroy.

member — Definition

A *member* names an element within a structure or a **union**. It can be accessed via the member-selection operators '.' or '->'. For example, consider the following:

```
struct example {
      int member1;
      long member2;
      example *member3;
};

struct example object;
struct example *pointer = &object;
```

To read the contents of **member1** within **object**, use the '.', as follows:

```
object.member1
```

On the other hand, to read the contents of **member1** via **pointer**, use the '->' operator:

```
pointer->member1
```

The same is true for a **union**, but with the following restriction: if a value is stored in one member of a **union**, then attempting to read another member of the **union** generates implementation-defined behavior. This restriction has one exception. If the **union** consists of several structures that have a common initial sequence, then that common sequence can be read when a value is written into any of the structures.

Cross-references

Standard, §3.1.2.6, §3.3.2.3
The C Programming Language, ed. 2, p. 128

See Also

->, ., name space, struct, union

memchr() — String handling (libc)

Search a region of memory for a character
#include <string.h>
void *memchr(const void ***region*, int** *character***, size_t** *n***);**

memchr searches the first *n* characters in *region* for *character*. It returns a pointer to *character* if it is found, or NULL if it is not.

Unlike the string-search function **strchr**, **memchr** searches a region of memory. Therefore, it does not stop when it encounters a null character.

Example

The following example deals a random hand of cards from a standard deck of 52. The command line takes one argument, which indicates the size of the hand you want dealt. It uses an algorithm published by Bob Floyd in the September 1987 *Communications of the ACM.*

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define DECK 52

main(int argc, char *argv[])
{
      char deck[DECK], *fp;
      int  deckp, n, j, t;

      if(argc != 2 ||
         52 < (n = atoi(argv[1])) ||
         1 > n) {
               printf("usage: memchr n # where 0 < n < 53\n");
            exit(EXIT_FAILURE);
      }

      /* exercise rand() to make it more random */
      srand((unsigned int)time(NULL));
      for(j = 0; j < 100; j++)
            rand();

      deckp = 0;
      /* Bob Floyd's algorithm */
      for(j = DECK - n; j < DECK; j++) {
            t = rand() % (j + 1);
            if((fp = memchr(deck, t, deckp)) != NULL)
                  *fp = (char)j;
            deck[deckp++] = (char)t;
      }
```

```
	for(t = j = 0; j < deckp; j++) {
		div_t card;

		card = div(deck[j], 13);
		t += printf("%c%c  ",
			/* note useful string addressing */
			"A23456789TJQK"[card.rem],
			"HCDS"[card.quot]);

		if(t > 50) {
			t = 0;
			putchar('\n');
		}
	}

	putchar('\n');
	return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.5.1
The C Programming Language, ed. 2, p. 250

See Also

strchr, strcspn, string handling, strpbrk, strrchr, strspn, strstr, strtok

memcmp() — String handling (libc)

Compare two regions
#include <string.h>
int memcmp(const void **region1*, **const void** **region2*, **size_t** *count***);**

memcmp compares *region1* with *region2* character by character for *count* characters.

If every character in *region1* is identical to its corresponding character in *region2*, then **memcmp** returns zero. If it finds that a character in *region1* has a numeric value greater than that of the corresponding character in *region2*, then it returns a number greater than zero. If it finds that a character in *region1* has a numeric value less than less that of the corresponding character in *region2*, then it returns a number less than zero.

For example, consider the following code:

```
	char region1[13], region2[13];
	strcpy(region1, "Hello, world");
	strcpy(region2, "Hello, World");
	memcmp(region1, region2, 12);
```

memcmp scans through the two regions of memory, comparing **region1[0]** with **region2[0]**, and so on, until it finds two corresponding "slots" in the arrays whose contents differ. In the above example, this will occur when it compares **region1[7]** (which contains 'w') with **region2[7]** (which contains 'W'). It then compares the two letters to see which stands first in the character table used in this implementation, and returns the appropriate value.

Cross-references

Standard, §4.11.4.1
The C Programming Language, ed. 2, p. 250

See Also

strcmp, strcoll, string handling, strncmp, strxfrm

Notes

memcmp differs from the string comparison routine **strcmp** in the following ways:

First, **memcmp** compares regions of memory rather than strings; therefore, it does not stop when it encounters a null character.

Second, **memcmp** takes two pointers to **void**, whereas **strcmp** takes two pointers to **char**. The following code illustrates how this difference affects these functions:

```
      char carray[10];
      int iarray[10];
      char *s = "hi";
          . . .
      strcmp(carray, s)           /* RIGHT */
      memcmp(carray, s, 3)        /* RIGHT */
      strcmp(iarray, s)           /* WRONG, 1st arg not char * */
      memcmp(iarray, s, 3)        /* RIGHT, args converted to void * */
```

It is wrong to use **strcmp** to compare an **int** array with a **char** array because this function compares strings. Using **memcmp** to compare an **int** array with a **char** array is permissible because **memcmp** simply compares areas of data.

memcpy() — String handling (libc)

Copy one region of memory into another
#include <string.h>
void *memcpy(void ****region1*, const void ****region2***, size_t** *n***);**

memcpy copies *n* characters from *region2* into *region1*. Unlike the routines **strcpy** and **strncpy**, **memcpy** copies from one region to another. Therefore, it will not halt automatically when it encounters a null character.

memcpy returns *region1*.

Example

The following example copies a structure and displays it.

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

struct stuff {
      int a, b, c;
} x, y;

main(void)
{
      x.a = 1;
      /* this would stop strcpy or strncpy. */
      x.b = 0;
      x.c = 3;

      /* y = x; would do the same */
      memcpy(&y, &x, sizeof(y));
      printf("a =%d, b =%d, c =%d\n", y.a, y.b, y.c);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.2.1
The C Programming Language, ed. 2, p. 250

See Also

memmove, strcpy, string handling, strncpy

Notes

If *region1* and *region2* overlap, the behavior of **memcpy** is undefined. *region1* should point to enough reserved memory to hold *n* bytes of data; otherwise, code or data will be overwritten.

memmove() — String handling (libc)

Copy region of memory into area it overlaps
#include <string.h>
void *memmove(void ****region1*, const void ****region2***, size_t** *count***);**

memmove copies *count* characters from *region2* into *region1*. Unlike **memcpy**, **memmove** correctly copies the region pointed to by *region2* into that pointed by *region1* even if they overlap. To "correctly copy" means that the overlap does not propagate, not that the moved data stay intact. Unlike the string-copying routines **strcpy** and **strncpy**, **memmove** continues to copy even if it encounters a null character.

memmove returns *region1*.

Example

The following example rotates a block of memory by one byte.

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

char *
rotate_left(char *region, size_t len)
{
      char sav;

      sav = *region;
      /* with memcpy this might propagate the last char */
      memmove(region, region + 1, --len);
      region[len] = sav;
      return(region);
}

char nums[] = "0123456789";
main(void)
{
      printf(rotate_left(nums, strlen(nums)));
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.2.2 *The C Programming Language,* ed. 2, p. 250

See Also

memcpy, strcpy, string handling, strncpy

Notes

region1 should point to enough reserved memory to hold the contents of *region2*. Otherwise, code or data will be overwritten.

memset() — String handling (libc)

Fill an area with a character
#include <string.h>
void *memset(void ****buffer*, int** *character***, size_t** *n***);**

memset fills the first *n* bytes of the area pointed to by *buffer* with copies of *character*. It casts *character* to an **unsigned char** before filling *buffer* with copies of it.

memset returns the pointer *buffer*.

Example

The following example fills an area with 'X', and prints the result.

```
#include <stdio.h>
#include <string.h>
#define BUFSIZ 20

main(void)
{
      char buffer[BUFSIZ];

      /* fill buffer with 'X' */
      memset(buffer, 'X', BUFSIZ);

      /* append null to end of buffer */
      buffer[BUFSIZ-1] = '\0';

      /* print the result */
      printf("%s\n", buffer);
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.11.6.1
The C Programming Language, ed. 2, p. 250

See Also

memchr, memcmp, memcpy, memmove, string handling

mktemp() — Extended function (libc)

Generate a temporary file name
char *mktemp(char ****pattern*);**

mktemp generates a unique file name. It can be used, for example, to name intermediate data files.

The *pattern* argument consists of a string that includes a capital '**X**'. **mktemp** replaces this **X** with '**A**' through '**Z**' to create up to 26 unique file names. It is normal practice to place temporary files in the directory **\tmp**. The start of the file name identifies the program that uses the file; for example, **\tmp\sortX** creates a temporary file to be used by **sort**. **mktemp** returns *pattern*.

The functions **tmpnam** and **tempnam** each assemble a temporary file name and then call **mktemp**. These routines ease the difficulty in creating a proper name for a temporary file.

See Also

extended miscellaneous, tempnam, tmpnam

mktime() — Time function (libc)

Turn broken-down time into calendar time
#include <time.h>
time_t mktime(struct tm ***timeptr*);**

mktime reads broken-down time from the structure pointed to by *timeptr* and converts it into calendar time of the type **time_t**. It does the opposite of the functions **localtime** and **gmtime**, which turn calendar time into broken-down time.

mktime manipulates the structure **tm** as follows:

1. It reads the contents of the structure, but ignores the fields **tm_wday** and **tm_yday**.
2. The original values of the other fields within the **tm** structure need not be restricted to the values described in the article for **tm**. This allows you, for example, to increment the member **tm_hour** to discover the calendar time one hour hence, even if that forces the value of **tm_hour** to be greater than 23, its normal limit.
3. When calculation is completed, the values of the fields within the **tm** structure are reset to within their normal limits to conform to the newly calculated calendar time. The value of **tm_mday** is not set until after the values of **tm_mon** and **tm_year**.
4. The calendar time is returned.

If the calendar time cannot be calculated, **mktime** returns -1 cast to **time_t**.

Example

This example gets the date from the user and writes it into a **tm** structure.

```
#include <math.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define BAD_TIME ((time_t)-1)

/* ask for a number and return it. */
int
askint(char * msg)
{
      char buf[20];

      printf("Enter %s ", msg);
      fflush(stdout);

      if(gets(buf) == NULL)
            exit(EXIT_SUCCESS);
      return(atoi(buf));
}

main(void)
{
      struct tm t;

      for(;;) {
            t.tm_mon  = askint("month");
            t.tm_mday = askint("day");
            t.tm_year = askint("year");
            t.tm_hour = t.tm_min = t.tm_sec = 1;
            if(BAD_TIME == mktime(&t)) {
                  printf("Invalid date\n");
                  continue;
            }

            printf("Day of week is %d\n", t.tm_wday);
            break;
      }
      return(EXIT_SUCCESS);
}
```

Cross-references

Standard, §4.12.2.3
The C Programming Language, ed. 2, p. 256

See Also

clock, date and time, difftime

Notes

The above description may appear to be needlessly complex. However, the Committee intended that **mktime** be used to implement a portable mechanism for determining time and for controlling time-dependent loops. This function is needed because not every environment describes time internally as a multiple of a known time unit.

model — Technical information

In the context of C programming, a **model** is a memory format that can be used on the i8086 microprocessor. Intel Corporation has defined six models for use on the i8086: TINY, SMALL, COMPACT, MEDIUM, LARGE, and HUGE. Mark Williams C compilers currently implement the SMALL and LARGE models, which experience shows can handle practically any programming task that can be executed with reasonable efficiency on the i8086.

In SMALL model, a program has two segments, each no larger than 64 kilobytes. One segment, the *code* segment, contains the code generated by the compiler. The other, the *data* segment, contains all pure and impure data, the stack, and the arena. “Pure” data are user data that have not yet been altered by the program, whereas “impure” data are user data that have been altered. In SMALL model, pointers are two **char**s (16 bits) long, which limits their addressing to 64 kilobytes.

In LARGE model, pointers consist of an offset and a segment. The actual address is calculated by shifting the segment left four and adding the offset. This can address up to one megabyte, although on the IBM PC the practical limit of memory is 640 kilobytes.

Code that is properly written can, in most instances, be ported from SMALL to LARGE model without modification. Routines that have integer-pointer puns, however, will run correctly under SMALL model but might fail under LARGE model.

See Also

LARGE model, pointer, pun, SMALL model, technical information

modf() — Mathematics (libm)

Separate floating-point number
#include <math.h>
double modf(double *real*, **double** **ip*);

modf breaks the floating-point number *real* into its integer and fraction.

modf stores the integer in the location pointed to by *ip*, and returns the fraction *real*. Both the integer and the fraction have the same sign. *f* in the range 0 <= *f* < 1.

Cross-references

Standard, §4.5.4.6
The C Programming Language, ed. 2, p. 251

See Also

exp, frexp, ldexp, log, log10, mathematics, modf

mtype.h — Header

List processor code numbers

The header file **mtype.h** assigns a code number to each of the processors supported by Mark Williams C compilers. These include the Intel i8086, i8088, i80186, and i80286; the Zilog Z8001 and Z8002; the DEC PDP-11 and VAX; the IBM 370; and the Motorola 68000.

See Also

header, portability

multibyte characters — Overview

C was invented at Bell Laboratories as a portable language for implementing the UNIX operating system. Since then, C has grown into a language used throughout the world, for both operating systems and applications.

The character sets of many nations are too large to be encoded within one eight-bit byte. The Japanese Kanji characters form one such set; the ideograms of Mandarin Chinese form another. For the sake of brevity, the following discussion will call such sets *large-character sets*. A character from a large character set will be called a *large character.*

Wide Characters

The Standard describes two ways to encode a large character: by using a *multibyte character* or a *wide character.*

wchar_t is a typedef that is declared in the header **stdlib.h**. It is defined as the integral type that can represent all characters of given national character set.

The following restrictions apply to objects of this type: (1) The null character still has the value of zero. (2) The characters of the standard C character set must have the same value as they would when used in ordinary **char**s. (3) **EOF** must have a value that is distinct from every other character in the set.

wchar_t is a typedef of an integral type, whereas a multibyte character is a bundle of one or more one-byte characters. The format of a multibyte character is defined by the implementation, whereas a **wchar_t** can be used across implementations.

Wide characters are used to store large character sets in a device-independent manner. Multibyte characters are used most often to pass large characters to a terminal device. Most terminal devices can receive only one byte at a time. Thus, passing the pieces of a wide character to a terminal would undoubtedly create problems; the individual characters of a multibyte character, however, can be passed safely. This is also important because the Standard does not describe any function that reads more than one byte from a stream at any time — there is no Standard version of **fgetw** or **fputw**.

Multibyte Characters

The Standard describes multibyte characters as follows:

- A multibyte character may not contain a null character or 0xFF (-1, or **EOF**) as one of its bytes.
- All of the characters in the C character set must be present in any set of multibyte characters.
- An implementation of multibyte characters may use a *shift state* or a special sequence of characters that marks when a sequence of multibyte characters begins and when it ends. Depending upon the shift state, the bytes of a multibyte character may either be read as individual characters or as forming one multibyte character. Note, too, that a shift state may allow *state-dependent coding*, by which one of a number of possible sets of multibyte characters is indicated by the shift state.
- A comment, string literal, or character constant must begin and end in the same shift state. For example, a comment cannot consist of multibyte characters mixed with single-byte characters; it must be all one or all the other. If a comment, string literal, or character constant is built of multibyte characters, each such character must be valid.

Multibyte Character Functions

The support added to the C language for multibyte characters thus far is limited to character constants, string literals, and comments. The Standard describes five functions that handle multibyte characters:

mblen	Compute length of a multibyte character
mbstowcs	Convert sequence of multibyte characters to wide characters
mbtowc	Convert multibyte character to wide character
wcstombs	Convert sequence of wide characters to multibyte characters
wctomb	Convert a wide character to a multibyte character

As mentioned above, a wide character is encoded using type **wchar_t**. The macro **MB_CUR_MAX** holds the largest number of characters of any multibyte character for the current locale. It is never greater than the value of the macro **MB_LEN_MAX**. **wcstombs** and **mbstowcs** convert sequences of characters from one type to the other.

All of the above are defined in the header **stdlib.h**.

Localization

The sets of multibyte characters and wide characters recognized by the above functions are determined by the program's locale, as set by the function **setlocale**.

To load the appropriate sets of multibyte characters and wide characters, use the call

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

or

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

See the entry for **localization** for more information.

Cross-reference

Standard, §2.2.1.2, §4.10.7

See Also

general utilities

Notes

Because compiler vendors are active in Asia, and because there is an active Japanese standards organization, a future version of the Standard may include more extensive support for multibyte characters, such as additional library functions. The support added to the C language for multibyte characters thus far is limited to character constants, string literals, and comments.

At present, all function names that begin with **wcs** are reserved. They should not be used if you wish your code to be maximally portable.