Salford C++ Help

Salford C++ Functions

!

Empty

D

difftime	_dos_getdate	_dos_setdate
div	_dos_gettime	_dos_settime

H

Empty

J

Empty

L

labs	localeconv	_lrotl
ldexp	localtime	_lrotr
ldiv	log10	lsearch
lfind	log	_lsearch
_lfind	longjmp	ltoa

N

Empty

O

_OFFSET	outport	outportb
_outp	outport32	_outpw

P

perror	putch	putchar
pow	_putch	puts
printf	putc	

W

Empty

Y

Empty

Z

Empty

Help on using SCC Help

1. The help system is hierarchical. Use the browse **<<** and **>>** buttons to obtain other topics on the same level. Use the **Up** button to jump to the corresponding branch point on the level above.
2. Use the **Copy** button to copy a topic to the clipboard.
3. Some topics have an illustrative example that can be viewed by clicking on an **Example** button at the head of the topic (or by pressing ENTER). If you want to copy the example to the clipboard, then click on the **Copy** button that appears in the pop-up "Example" window (or press ENTER).

Salford C++ command line options

The compilation/loading process

A C/C++ program must be converted to binary form before it can be executed. The process of producing an executable program takes place in two phases.

- Compilation: where the C/C++ program is checked for syntactic and semantic correctness and re-locatable binary code is output to an intermediate file, <filename>.OBJ, where <filename>.C is the name of the source file.
- Loading: using the Salford C++ Linker, LINK77, where the re-locatable binary code is loaded together with any other re locatable binary code files, library files that might have been produced by previous compilation(s) with Salford C++ (or other compatible compilers, such as FTN77 or Sheffield Pascal) and routines from the C/C++ and other system libraries.

Note that .OBJ files produced by Salford C++ cannot be loaded using the MS-DOS linker, which is only suitable for 16-bit code. Likewise, LINK77 cannot be used to load real mode object files.

Salford C++ allows the user to combine these two phases by means of the "load-and-go" facility in which no permanent object file is produced.

Compiler source input

The compiler reads programs from MS-DOS ASCII files which have been created by any of the various PC text editors.

The source file should be specified using the first parameter to the SCC command as follows:

```
SCC <pathname>
```

The compiler searches for the file <pathname>.CPP and if it finds it, the file is compiled. Otherwise, the compiler outputs an error message, as in this example:

```
SCC SOCRATES

SOCRATES.CPP File not found
```

Source files must have .CPP as a suffix, or be specified with an explicit suffix. Any filename acceptable to MS-DOS can be used. In order to compile the program in file MYPROG.CPP, the following command should be used:

```
SCC MYPROG
```

In this case MYPROG.CPP is a local file in the current directory. An example of <pathname> specification might be:

```
SCC C:\SCC\PROJECT\MYPROG
```

In this case the compiler searches for the file

```
C:\SCC\PROJECT\MYPROG.CPP
```

Only one source file name may be specified unless wild cards are used. For example:

```
SCC *.CPP
```

would compile all of the .CPP files in the current directory. An explicit extension (like .CPP) is essential. The object code can be output to a single file by using the /BINARY option.

The command line

Compiler options are specified as part of the SCC command line, for example:

```
SCC MYPROG /LIST
```

causes SCC to compile a program held in a source file called MYPROG.CPP. The suffix .CPP is assumed by default. However, if you specify a file with an explicit suffix, then the file with that suffix will be used. Furthermore, the assumed .CPP suffix can be changed by configuring the options .

Note that:

- SCC can be operated in ANSI C mode (using the /ANSI_C option or a .C suffix), when it conforms to the standard with a few extensions (see also /AUTO_SUFFIX), or in C++ mode by default.
- the option /LIST tells SCC to generate a compilation listing of the program source in a file MYPROG.LIS.

Note that options may be abbreviated, but care should be exercised to ensure that the abbreviated form is unique.

The compiler option defaults described in this help system are those provided when Salford C++ is distributed.

Configuring the SCC command

The SCC command has many options and it is often convenient to alter the default settings, or to create alternative versions of the command for specific purposes. For example, you might wish to create a command which would run a program with /ANSI_C, /LGO and /CHECK using a default file suffix of .C. To configure your compiler type:

```
SCC /CONFIG
```

You will be presented with a master configuration menu of four options thus:

a) The default options

b) The compiler messages

c) The source file suffix (the default is .CPP)

d) Save configuration information and exit

In general you will select one or more of items (a)-(c), followed by (d) to save the results. When you specify option (d) you will be prompted to save the results in the existing compiler file (e.g.SCC.EXE) or to create a new one. In general we recommend that you do not alter the supplied compiler file (SCC.EXE). If at any time you decide to abandon configuration process, press the ESC key.

- Selecting option (a) presents a screen of compiler command line options. An option may be selected using the cursor keys and may be toggled **on** or **off** by pressing the spacebar. An arrow next to the option indicates that it is currently configured **on** by default. A brief explanation of the currently selected option is available at the bottom of the screen. Press ENTER to return to the main configuration menu.
- Selecting option (b) from the main configuration menu presents a scrollable list of messages generated by the compiler at compile time (i.e. not link or run time diagnostics).

  The compilation diagnostic messages are split into three groups:

  - "comments" that are provided for information only,
  - "warnings" that are designed to show possible areas which may lead to misleading results,
  - "errors" that are caused by incorrect program code.

  Both warnings and comments may be configured to be on, off or to be treated as errors. A particular type of error (designated as "fatal") causes immediate termination of the compilation regardless of the number of errors have have already occurred.

  After selecting a message you may alter it as follows:

  - Pressing E will convert the message into an error. This may be useful if you wish to enforce certain programming standards. For example, you could configure a compiler which would not compile programs with variables which were declared but never used.

    - Pressing W will convert the message to a warning.
    - Pressing C will convert the message to a comment.
    - Pressing S or the spacebar will suppress the message entirely.
    - Press ENTER to return to the main menu.

  Messages defined to be errors in SCC cannot be downgraded, as this could result in unpredictable results at link/run time, however, error messages created by a previous configuration can be changed.

- Selecting option (c) from the main menu will present the current default source file suffix and enable you to modify it. The new suffix must contain between one and three characters. Press ENTER to return to the main menu.
- Use option (d) from the main menu to save the configuration information. The configuration information is actually stored in the compiler .EXE file itself. You will be presented with the full path name of the compiler. You may alter the current file by simply pressing ENTER, or modify the file name as required.

Notes:

- The pathname you choose must have the .EXE suffix.
- If at any time you receive a fresh version of Salford C++ you must recreate any configured versions of the SCC command.
- If at any point you decide to abandon the configuration process press ESC.

Controlling compile-time messages

SCC can issue a large variety of error, warning, and comment messages. In general it is undesirable to allow the compiler to generate large numbers of warnings or comments, as this will make it difficult to recognise those messages that are significant. The following facilities are available to control these messages.

- All warnings and comments may be suppressed using the /SILENT option. This option should only be used if you are sure that these messages are of no interest to you.
- The compiler can be configured to suppress particular warnings or comments, or to upgrade such messages to errors, which will cause the compilation to fail (see Configuring the SCC command).
- By including the line

  ```
  #pragma suppress <n>
  ```

  in a program you can suppress message number <n>. The number of the message you want to suppress can be obtained by including the /ERROR_NUMBERS option in an earlier compilation. You can include as many suppress pragmas as you require, and by incorporating them in conditional compilation blocks it is even possible to adjust the message suppression automatically.

  A given message will be suppressed for the remainder of the code or until a corresponding

  ```
  #pragma enable <n>
  ```

  is encountered.
- Sometimes you may wish to suppress an unwanted warning message in a particular context. The "silent" pragma is useful in these cases. For example:

  ```
  #pragma silent
      while(*p++ = *q++)
  ```

  Here the pragma will suppress any warnings or comments (but not errors) relating to the following line (and this line only). This means that in the above example a warning about the use of "=" rather than the expected "= =" inside the **while** statement would not be issued (unless the compiler had been configured to convert this message into an error).

Index of compiler options

ANNOTATE_CLASSES	DELETE_OBJ_ON_ERROR	NO_GUESSES
ANSI_C	ERROR_NUMBERS	NO_LINE
ASMBREAK	EXPLIST	NO_OFFSETS
AUTO_SUFFIX	FORTRAN_CALLS	NO_ZEROISE
BINARY	FULL_UNDEF	ONLY_PREPROCESS
BREAK	HARDFAIL	OPTIMISE
BRIEF	INCLUDE	OPTIONS
CHECK	LGO	PARAMS
CONFIG	LIBRARY	PROFILE
C++	LINK	PROTOTYPES
CRX1	LINK_MAP	SHOW_PREPROCESS
CRX2	LIST	SILENT
CRX3	LIST_INSERT_FILES	STATISTICS
CRX4	NESTED_COMMENTS	UNDEF
CRX5	NEVER_INLINE	UNDERFLOW
DBREAK	NO_ACCESS_CONTROLS	UNLIMITED_ERROS
DEBUG	NO_CLASS_CONSISTENCY	WINDOWS
DEFINE	NO_CODE	

Program listing options

The options within this group specify the kind of program listing that the compiler is to provide.

LIST	EXPLIST	ONLY_PREPROCESS
LIST_INSERT_FILES	NO_OFFSET	SHOW_PREPROCESS

/LIST <pathname> *or* /LIST

During compilation and loading, a listing can be output to a file. This file is specified by the /LIST compiler option. If <pathname> is omitted, then the default name for the listing file is <filename>.LIS.

When a compilation listing is produced, it always contains the following information:

- Date and time of compilation
- Source file pathname
- Compiler version number
- Compiler options in use
- Source statement listing

Ÿ Error, warning and comment messages

Depending on the options chosen, assembly language instructions corresponding to statements in the program can be output to the listing file.

The relative address of each statement is printed in hexadecimal at the right of the line (unless the /NO_OFFSETS option is used to suppress the printing of these addresses). Relative addresses allow the user to locate the source of run-time errors which occur in parts of the program where no checks have been specified. The relative address is the byte address of the first machine instruction corresponding to the statement, relative to the start of the current function. The relative address is incremented for each statement for which the compiler generates code. Code generation ceases for the remainder of the source file when a compilation error is found.

/LIST_INSERT_FILES

By default the contents of included files (e.g. compiler header files) are not included in the listing file, however the option /LIST_INSERT_FILES when included with /LIST can be used to override the default. When this option is used, line numbers in the listing file relate to the current include level (which may be nested). The numbers are preceded by the include level in the form *include level/line number*.

/EXPLIST

This option causes each source statement to be followed by the 32-bit Intel assembler statements into which it was compiled. The assembly language listing is fully symbolic and even includes some comments. Information on 32-bit Intel assembler can be found in the Intel 80386 Programmer's Reference Manual. If /EXPLIST is omitted, no assembly language listing is produced.

/SHOW_PREPROCESS and /ONLY_PREPROCESS

Both of these options send the output from the preprocessor to a file. The file name is generated from the source file name by appending .PRE to the root of the file name. The /SHOW_PREPROCESS option continues to compile the program as well as generate the preprocessor output file. /ONLY_PREPROCESS will only generate the preprocessor file. The latter option allows the preprocessor to be used on files other than C++ source files.

/NO_OFFSETS

Suppresses the address information in the listing file.

Run-time checking options

One of the most powerful features of Salford C++ lies in its ability to detect runtime errors such as illegal references to memory and references through NULL pointers. Compiler options which enable runtime checking are listed here for reference. Further information is given in chapter 5 of the manual.

DEBUG	FULL_UNDEF	NO_ZERIOSE
CHECK	NEVER_INLINE	PROFILE
UNDEF		

/DEBUG

This option causes SCC to activate the symbolic debugger (when fatal errors occur). Since /DEBUG is implicit in the /CHECK and /UNDEF options, it is used in the absence of these options in order to allow the debugger to be used on "dirty" code.

/CHECK

The /CHECK option (which implies /DEBUG) causes the compiler to produce code which checks at run time for misuse of pointers and certain other errors. The use of /CHECK is fully described in chapter 5 of the manual.

/UNDEF and /FULL_UNDEF

The /UNDEF option (which implies /CHECK) includes checking for the use of undefined variables.   /UNDEF checks for undefined arithmetic and pointer variables whilst /FULL_UNDEF works with character variables as well.   If this option is used in conjunction with /NO_ZEROISE then static and external arithmetic variables are also checked.

/NEVER_INLINE

This forces the compiler to call static versions of inline functions rather than merging the contents of the function into the code each time it is used. One use for this option is in conjunction with the /DEBUG option to enable the contents of inline functions to be debugged. Note that this option is implied in CHECK mode.

/NO_ZEROISE

Standard C sets static and external arithmetic variables to be zero and pointer variables to be **NULL**. The /NO_ZEROISE option generates a bit pattern which represents an uninitialised state and stores this in the appropriate locations. By using the /UNDEF option, you can trap the use of this type of variable when it has not been assigned a value.

/PROFILE

It is often useful to know how many times each statement in a program has been executed. Such information may reveal logical errors and can often help in tracing the execution path in the event of a run-time failure. It will also indicate which parts of a program are most heavily used so that those parts can be examined and recoded to improve execution speed should this be considered worthwhile. The /PROFILE option activates this facility. The /PROFILE option has a further use in ensuring that test data exercises all parts of a program.

To obtain a profile listing you should compile your program with the /PROFILE option and execute it using /BREAK (to enter the symbolic debugger, see section 7.1 of the manual), run the program - either to a breakpoint or to completion - and issue the command

```
PROFILE <pathname>
```

from the source window. The profile information will then be written to <pathname> (which should be a different name from the listing file name) in the form of an annotated source listing. If you omit the pathname the profile information will be overlaid on the source listing and directed to the screen (this can also be achieved by pressing the F9 function key).

Note that the PROFILE facility cannot be applied simultaneously to more than one source file.

Load and go

Salford C++ provides a load-and-go facility so that even quite large and complex programs which require libraries can automatically and quickly be linked, loaded and executed. No permanent object file is produced (although there must be enough disk space to accommodate a temporary object file). These features make this facility invaluable for teaching, testing and development where repeated compilations and test runs are the norm.

Note: If a program is required to be linked (that is, a .EXE file produced) but not executed immediately, the /LINK option may be used.

All the standard compiler options are available together with a number of extra options which allow the following:

- Specification of library and relocatable binary files
- Underflow trapping
- Interactive debugging

LGO	ASMBREAK	HARDFAIL
LIBRARY	BREAK	PARAMS
	DBREAK	UNDERFLOW

LGO option

The load-and-go facility is invoked by the /LGO option. For example:

```
SCC MYPROG  /CHECK  /LGO
```

would compile, load and execute the program held in the source file MYPROG.CPP. The order of specifiers on the command line is immaterial, except when an option requires a name, in which case the name must follow the option.

The options /BREAK and /DBREAK both imply /LGO. These options also imply either /DEBUG or /CHECK. These four options are summarised in the following table for easy reference.

Option	**Debug code planted**	**Check code planted**	**Immediate entry to debugger**	**/LGO implied**
/DEBUG	ü			
/CHECK	ü	ü		
/BREAK	ü	ü	ü	ü
/DBREAK	ü		ü	ü

/ASMBREAK

Implies /CHECK and /LGO and is used to enter the machine code debugger at first assembler code instruction.

/BREAK

This option implies /CHECK and /LGO and causes program execution to be suspended at the first instruction in the main program with entry to the debugger.

/DBREAK

/DBREAK is like /BREAK but does not include the planting of code to check for runtime errors (i.e. it implies /DEBUG but otherwise not /CHECK) and causes program execution to be suspended at the first instruction in the main program with entry to the debugger.

Using /LIBRARY with /LGO

The use of the /LGO option is not restricted to programs that require only the Salford C++ library. Other system libraries, user libraries and relocatable binary input files can be specified. These files can be specified in any combination of the following:

- By using the /LIBRARY option. For example:

```
        SCC MYPROG  /LGO  /LIBRARY  GKSLIB
```

- By using a **#pragma** library directive (see chapter 8 of the manual).

If a library filename (GKSLIB in the above command line) does not include path information, the current directory is searched, followed by the directory containing the Salford C++ compiler.

Notes:

- The compiler will automatically search first for a library or relocatable binary file with the name suffixed by .OBJ (even though it does not appear with suffix in the command line), and then for the unsuffixed filename .
- Dynamic link libraries can be listed in a directory called LIBRARIES.DIR, see section 13.7.2 of the manual for further details.

The /HARDFAIL option

If a program is compiled using the /LGO option and an error occurs, the program is suspended and the debugger is entered. If this is considered to be undesirable, the use of the /HARDFAIL option causes run time errors to produce a machine level message and a return to MSDOS, rather than entering the symbolic debugger. This is sometimes useful if the program contains assembler code.

The /PARAMS option

The /PARAMS option is provided for use with /LGO, /BREAK and /DBREAK so that the user can introduce command line options and filenames that relate to his own object program rather than to the compiler. This option prevents the compiler from scanning the remainder of the SCC command line before the user's program is executed. For example:

```
#include <stdio.h>
int main(int argc,char *argv[])
{ FILE *source,*dest;
  char ch;
  source=fopen(argv[1],"r");
  dest=fopen(argv[2],"w");
  while(!feof(source))
  {
    ch=fgetc(source);
    fputc(ch,dest);
  }
  fclose(source);
  fclose(dest);
  return 0;
}
```

The command line:

```
   SCC FILECOPY /LGO /PARAMS DATA1 DATA2
```

will load and execute the program in the file FILECOPY.CPP, which copies the file DATA1 to the file DATA2.

The /UNDERFLOW option

In /LGO mode, the compiler normally generates zero values when floating point underflow occurs. Use of the /UNDERFLOW option ensures that the first occurrence of underflow in an arithmetical computation is treated as a failure and is not ignored. A large number of occurrences of underflow during execution can result in long execution times because of the way in which the underflow condition is treated. If an underflow is trapped, the message

```
ERROR: Floating point arithmetic underflow
```

is output and the interactive debugger is entered (see chapter 7 of the manual). If underflows occur during program execution and the /UNDERFLOW option is not used, a message is output specifying the number of underflows that have occurred.

Error message options

All error, warning and comment messages are output to the screen and to the compilation listing file if one is specified or implied. Messages fall into three categories namely: error, warning and comment messages. Warning and comment messages can be suppressed.

The following list includes options for disabling certain messages and options for specifying what constitutes an error.

ANSI_C	C++	NO_GUESSES
AUTO_SUFFIX	ERROR_NUMBERS	PROTOTYPES
ANNOTATE_CLASSES	NESTED_COMMENTS	SILENT
BRIEF	NO_CLASS_CONSISTENCY	UNLIMITED_ERRORS

/ANSI_C

When the /ANSI_C option is used, the compiler uses ANSI C rules where these conflict with C++. In particular, the use of this option means that C++ keywords are treated as ordinary tokens. Note, however, that the following two C++ extensions are also available in ANSI C mode (i.e. when /ANSI_C is used):

- the use of "//" for comments is permitted,
- when calling a function, parameters may be called by reference.

If ANSI C is set as the default mode using the /CONFIG option, then C++ mode can be reselected by using /C++.

/AUTO_SUFFIX

By default, files that have a .C extension are compiled in ANSI C mode. Unless /ANSI_C is used (or SCC is reconfigured to make /ANSI_C a default option), any other file will be compiled in C++ mode. This is called AUTO_SUFFIX mode and is selected by default in the /CONFIG menu when the compiler is shipped. The /AUTO_SUFFIX option is only needed when AUTO_SUFFIX mode has been switched **off** in the /CONFIG menu and the user wishes to temporarily switch AUTO_SUFFIX mode back **on**.

/ANNOTATE_CLASSES

Using this option causes the compiler to output comments which identify constructors, destructors, and certain other special member functions of classes.

/BRIEF

Causes all errors, warnings and comments to be output in a form which is compatible with the BRIEF text editor.   Programs can then be compiled and then edited whilst still within BRIEF (see chapter 15 of the manual).

/ERROR_NUMBERS

This option causes the compiler to include the message number with each message (warning, error, or comment) that it generates.   This is useful in order to obtain numbers for the **#pragma** “suppress” feature (see section 8.2 of the manual).

/NESTED_COMMENTS

allows the use of nested comments within a program.   For example in

```
/*
   int i;
   char *ptr; /* pointer to character */
*/
```

the initial “/*” will be matched with the final rather than the first “*/”.

/NO_CLASS_CONSISTENCY

By default, the compiler informs the linker about every **class**, **struct**, or **union** used in a program. The size of the object, together with a hash code derived from the entire structure of the class, is passed to the linker. The linker will report warnings if a named **class**, **struct**, or **union** is used inconsistently in separately compiled modules. Often this is a consequence of a change of a definition in an include file, and is an indication that some files have not been recompiled to incorporate the change. The /NO_CLASS_CONSISTENCY option suppresses this information. (See also section 8.2 of the manual for a mechanism to suppress this feature for specific classes.)

/NO_GUESSES

When reporting an error the compiler will often offer a guess as to what was intended. For example, if it found “Int” in a context where a keyword was required, it would suggest that “int” was intended. The /NO_GUESSES option suppresses this feature.

/PROTOTYPES

This option is used when operating in ANSI C mode to demand that a function be prototyped before it is used. This option should **always** be used when new C code is being developed in ANSI C mode, as mismatches between assumed function prototypes and actual functions can cause serious problems in C. This option is not required in C++ mode, where functions must always be prototyped.

/SILENT

This option suppresses the printing of warning and comment messages. Unless the /SILENT option is in force, the message that is output on the screen at the end of compilation includes the numbers of warning and comment messages as in the following example:

```
NO ERRORS,3 WARNINGS,2 COMMENTS [Salford C++/x86]
```

/UNLIMITED_ERRORS

Normally compilation stops after issuing 12 error messages. This option allows compilation to continue until the entire source file has been processed.

Properties of the code

This group consistutes a miscellany of options that control various properties of the object code.

DEFINE	INCLUDE	NO_LINE
FORTRAN_CALLS	NO_ACCESS_CONTROLS	WINDOWS

/DEFINE

This allows a preprocessor macro to be defined from the command line. The macro cannot be a function like macro, it can only take a simple value. If a value for the definition is not specified, a default value of 1 is assigned. For example:

```
SCC MYPROG /DEFINE ARRAY_SIZE=34
```

will define a macro **ARRAY_SIZE** to have a value of 34, whilst

```
SCC MYPROG /DEFINE DEBUG
```

will simply define a macro **DEBUG** which may be used with **#ifdef**.

/FORTRAN_CALLS

This option causes SCC to load the ESI register before calling a function. This instruction is only required if the called function is compiled with FTN77. By default, SCC only loads ESI if the target name contains the @ symbol, or the function has been declared extern "FORTRAN". In general it is better to declare functions extern "FORTRAN" and avoid unnecessary code for other calls to C functions.

/INCLUDE <pathname>

This option extends the search path sequence. Normally the sequence consists of the current directory followed by the system include directory. The use of the option changes this to the current directory followed by *pathname* (enclosed in quotation marks " ") and then the system include directory.

/NO_ACCESS_CONTROLS

This option causes the C++ access control for classes to be ignored. This removes the protection of private and protected data members of a class. This may be useful when debugging to enable a private member of a class can be printed out.

/NO_LINE

This option instructs the compiler to ignore **#line** directives. When **#line** directives are used, the resulting executable code will use line numbers and filenames generated by the directive instead of those relating directly to the C program. /NO_LINE cancels this effect.

/WINDOWS

The /WINDOWS option is used for programs which are designed to run under Microsoft Windows 3.x. With this option the program will start by generating a call to a function which initialises the windows environment (see sections 13.2.1, 8.2 and 8.7 of the manual, and the ClearWin Reference Manual for further details).

Optimisation of code

/OPTIMISE

This causes the compiler to generate optimised code. This option is not effective when used in conjuction with either /CHECK or /DEBUG.

Linker control

The following options are provided to invoke the linker or to pass information to it.

LINK	LINK_MAP	LIBRARY

See also /LGO.

/LINK <filename> *or* <directory_name>

After compilation is complete, the linker will be invoked to generate an executable file. If <filename> is not specified, the name of the executable file is generated from the source file name, otherwise <filename> is the name of the resulting executable file. If <directory_name> is not specified, then the executable file will be placed in the same directory as the source file, otherwise it will be placed in the specified directory.

/LINK_MAP

This option can be used in conjunction with the /LINK or /LGO options in order to provide a linker map. It may be followed by the name of a file into which a linker map is to placed, otherwise the default file name for the map has the form <filename>.MAP.

/LIBRARY <filename>

This causes the file <filename> to be searched for any functions that are found to be missing in a subsequent linking process. <filename> must include an explicit .OBJ or .LIB extension. The option is particularly useful when used in conjunction with /LINK or /LGO (see also **#pragma** <filename> in section 8.2 of the manual).

/STATISTICS

This causes the compiler to output a message on the screen stating the number of lines compiled (including all header files) and the compilation time.

Destination of the code

By default the compiler produces relocatable binary code even when errors are encountered in the source file. If this code is error free it can either be loaded automatically using the /LGO option or be made available to a file for loading with LINK77. When /LGO is not used, by default a binary file is created which has a name based on the source file name but with the .OBJ extension. The following options provide alternatives to the default response.

BNARY	NO_CODE	DELETE_OBJ_ON_ERROR

/BINARY <filename>

This option specifies <filename> as the name of the resulting object file, overriding the default name. This is particularly useful when used in conjunction with wild cards where the effect is to output all of the code into one file (see Compiler source input).

/NO_CODE

causes the creation of a binary file to be suppressed.

/DELETE_OBJ_ON_ERROR

causes the binary file to be deleted if the code is not error free.

Optional syntax restrictions

One problem with the C and C++ languages, especially for beginners, is the ease with which a mistake can appear to the compiler as valid syntax with a totally different meaning. The following options may be used to restrict the syntax which the compiler accepts in order to control these problems. It is expected that these options will be used by teachers, who will configure the compiler to turn these options on, or by programmers who wish to impose a “house style”. Other restrictions can be imposed by configuring certain warnings as errors.

Because the meaning of these restrictions is subtle, no attempt has been made to give them meaningful names. The following options are available:

CRX1	CRX3	CRX5
CRX2	CRX4	

/CRX1

This restriction prevents a named structure, class, or union being defined and used to declare data items (outside the class) on the same line. Thus the following will generate an error if this option is used:

```
class test{
int alpha;
}
foo(int j);
```

Presumably **foo** is supposed to be an integer function and the programmer has missed the semicolon at the end of the class definition. Unnamed aggregate definitions can still define data, as otherwise they would be useless!

/CRX2

Forces the declaration or definition of a function to include a return type (rather than taking the default int type). If this option is used, even the **main** function must be defined with a return type.

/CRX3

Stops the compiler accepting traditional C style function declarations such as:

```
int foo(a,b)
int a;
float b;
{
}
```

/CRX4

Prevents the compiler from accepting **goto** statements.

/CRX5

This option prevents the declaration of **float** objects (as opposed to **double**), and faults any attempt to perform single precision floating point arithmetic. This can be useful when porting code from a machine with a higher floating point precision. Including this option will ensure that floating point arithmetic is performed to 64-bit precision.

Reading compiler options from a file

/OPTIONS

Compiler options can be read from a file. The contents of the file will be used as a set of options which will be combined with the command line. For example, suppose you had a file called MYOPT containing the following:

```
/ANSI_C /LIST
```

The effect of the above SCC command could be obtained by typing

```
SCC MYPROG /OPTIONS MYOPT
```

You can use multiple and/or nested options files, but such files must not contain options that will be passed to the program to be run using the /PARAMS mechanism.

Functions defined in ctype.h

isalnum	isgraph	isupper
isalpha	islower	isxdigit
iscntrl	isprint	tolower
isextended	ispunct	toupper
isdigit	isspace	

About functions defined in ctype.h

The **<ctype.h>** header provides prototypes for a number of functions which can be used in the analysis and mapping of text data.   These functions enable the programmer to determine whether a character is alphabetic, alphanumeric, etc.. The programmer should ensure that all input values either lie in the range 0..255 or take the value **EOF**.   Other values will produce unpredictable results.   In particular, separate provision should be made for non-ASCII characters such as the UK currency symbol.   Make and break keyboard codes will need to be appropriately masked before input in order to remove unwanted data.

isalnum

Example

Purpose To test for an alphanumeric character.

Syntax

```
#include <ctype.h>
int isalnum(int c);
```

Return value **isalnum** returns a non-zero value if *c* is the ASCII value for an alphanumeric character (0..9, A..Z, or a..z); zero if not.

isalpha

Example

Purpose To test for an alphabetic character.

Syntax

```
#include <ctype.h>
int isalpha(int c);
```

Return value **isalpha** returns a non-zero value if *c* is the ASCII value for an alphabetic character (A..Z, or a..z); zero if not.

iscntrl

Example

Purpose To test for a control character.

Syntax

```
#include <ctype.h>
int iscntrl(int c);
```

Return value **iscntrl** returns a non-zero value if *c* is the ASCII value for a “control” character; zero if not.   A control character is defined as not a “printing” character (see **isprint**).

isextended

Example

Purpose To test for a character in the extended set.

Syntax

```
#include <ctype.h>
int isextended(int c);
```

Return value **isextended** returns a non-zero value if *c* is the ASCII value for a character in the extended ASCII set (ASCII values 128..255); zero if not.

isdigit

Example

Purpose To test for a digit character.

Syntax

```
#include <ctype.h>
int isdigit(int c);
```

Return value **isdigit** returns a non-zero value if *c* is the ASCII value for a digit character (0..9); zero if not.

isgraph

Example

Purpose To test for a graphic character.

Syntax

```
#include <ctype.h>
int isgraph(int c);
```

Return value **isgraph** returns a non-zero value if *c* is the ASCII value for a graphics character which is defined as a printing character (see **isprint**) but not a space.

islower

Example

Purpose To test for a lower case alphabetic character.

Syntax

```
#include <ctype.h>
int islower(int c);
```

Return value **islower** returns a non-zero value if *c* is the ASCII value for a lower case letter (a..z); zero if not.

isprint

Example

Purpose To test for a printing character.

Syntax

```
#include <ctype.h>
int isprint(int c);
```

Return value **isprint** returns a non-zero value if *c* is the ASCII value for a “printing” character; zero if not. The definition of a “printing” character may be both implementation and locale dependent.

ispunct

Example

Purpose To test for a punctuation character.

Syntax

```
#include <ctype.h>
int ispunct(int c);
```

Return value **ispunct** returns a non-zero value if *c* is the ASCII value for a “punctuation” character i.e. one of:

```
!"#$%&’()*+,-./:;<=>?@[\]^_‘{|}~
```

A punctuation character is a character in the non-extended ASCII set which is neither a control nor an alphanumeric nor a space character.

isspace

Example

Purpose To test for a white-space character.

Syntax

```
#include <ctype.h>
int isspace(int c);
```

Return value **isspace** returns a non-zero value if *c* is the ASCII value for a white-space character (ASCII values 9,10,11,12,13,32); zero if not.   These normally correspond to tab, line-feed, vertical-tab, form-feed, return and space.

isupper

Example

Purpose To test for an upper case alphabetic character.

Syntax

```
#include <ctype.h>
int isupper(int c);
```

Return value **isupper** returns a non-zero value if *c* is the ASCII value for an upper case letter (A..Z); zero if not.

isxdigit

Example

Purpose To test for a hexadecimal digit character.

Syntax

```
#include <ctype.h>
int isxdigit(int c);
```

Return value **isxdigit** returns a non-zero value if *c* is the ASCII value for a hexadecimal digit (0..9, A..F, or a..f); zero if not.

Example

Copy

```
#include <stdio.h>                      // for printf,EOF
#include <ctype.h>                      // for isalpha etc.
int main()
{
  for(int i=EOF;i<256;i++)
  if(isalnum(i)) printf("%c",i);
  return 0;
}
```

tolower

Example

Purpose To change any upper case alphabetic characters to lower case.

Syntax

```
#include <ctype.h>
int tolower(int c);
```

Return value **tolower** returns the lower case ASCII value in the range a..z when *c* is the ASCII value for one of the letters A..Z.   Other values are left unchanged.

See also **strlwr**.

toupper

Example

Purpose To change any lower case alphabetic characters to upper case.

Syntax

```
#include <ctype.h>
int toupper(int c);
```

Return value **toupper** returns the upper case ASCII value in the range A..Z when *c* is the ASCII value for one of the letters a..z. Other values are left unchanged.

See also **strupr**.

Example

Copy

```
#include <stdio.h>                      // for printf
#include <ctype.h>                      // for tolower
int main()
{
  for (int i=65;i<91;i++) printf("%c %c\n",i,tolower(i));
  return 0;
}
```

Functions defined in math.h

- acos
- asin
- atan
- atan2
- ceil
- cos
- cosh
- exp
- fabs
- floor
- fmod
- frexp
- ldexp
- log
- log10
- modf
- pow
- sin
- sinh
- sqrt
- tan
- tanh

About functions defined in math.h

The **<math.h>** header provides prototypes for various mathematical functions. It also includes the following definitions.

```
#define HUGE_VAL        1e308
#define EDOM_VAL        1e308
#define M_E             2.718281828459045235    // exp(1)
#define M_LOG2E         1.442695040888963407    // 1/ln2
#define M_LOG10E        0.434294481903251828    // 1/ln10
#define M_LN2                 0.693147180559945309    // ln(2)
#define M_LN10          2.302585092994045684    // ln(10)
#define M_PI            3.141592653589793238    // PI
#define M_PI_2          1.570796326794896619    // PI/2
#define M_PI_4          0.785398163397448310    // PI/4
#define M_1_PI          0.318309886183790672    // 1/PI
#define M_2_PI          0.636619772367581343    // 2/PI
#define M_1_SQRTPI      0.564189583547756287    // 1/sqrt(PI)
#define M_2_SQRTPI            1.128379167095512574    // 2/sqrt(PI)
#define M_SQRT2         1.414213562373095049    // sqrt(2)
#define M_SQRT_2        0.707106781186547524    // 1/sqrt(2)
```

A particular function call will lead to a “domain error” if an input argument lies outside the interval over which the mathematical function is defined. In such cases the function returns the value of the macro **EDOM_VAL** and sets **errno** to **EDOM** (see also error.h header). Similarly a “range error” will occur when the result of the function cannot be expressed as a double value. In such cases the function returns the macro **HUGE_VAL** (with an appropriate sign) and sets **errno** to **ERANGE**. If the result of a function underflows then zero is returned and **errno** is not set (this feature is implementation dependent). A closed interval which includes an end point is denoted below by square brackets. An open interval which does not include an end point is denoted by round brackets. e.g. (0,3] includes 3 but not 0.

acos

Purpose To compute the arc cosine of a double value.

Syntax

```
#include <math.h>
double acos(double x);
```

Return value **acos** returns the radian value in the range [0,π] for the inverse cosine of *x*.

Notes Gives a domain error if the magnitude of *x* is greater than 1 (returns **EDOM_VAL** and sets **errno** to **EDOM**).

asin

Purpose To compute the arc sine of a double value.

Syntax

```
#include <math.h>
double asin(double x);
```

Return value **asin** returns the radian value in the range [-π/2,π/2] for the inverse sine of *x*.

Notes Gives a domain error if the magnitude of *x* is greater than 1 (returns **EDOM_VAL** and sets **errno** to **EDOM**).

atan

Purpose To compute the arc tangent of a double value.

Syntax

```
#include <math.h>
double atan(double x);
```

Return value **atan** returns the radian value in the range (-π/2,π/2) for the inverse tangent of *x*.

Example atan(1.0) gives 0.785398... (π/4)

atan2

Purpose To compute the arc tangent for a pair of double values.

Syntax

```
#include <math.h>
double atan2(double y,double x);
```

Return value **atan2** returns the radian value in the range $(-\pi,\pi]$ for the inverse tangent of *y*/*x* , using the signs of both arguments to determine the quadrant of the returned value. This function is more robust than **atan**.

Notes Gives a domain error if *x*=0 and *y*=0.

Example atan2(-1.0,0.0) gives -1.570796...($-\pi/2$).

ceil

Purpose To find the next (ceiling) integer above a given double value.

Syntax

```
#include <math.h>
double ceil(double x);
```

Return value **ceil** returns the value of *x* rounded to the next integer above.

Example ceil(1.1) gives 2.0.

cos

Purpose To compute the cosine of a double value.

Syntax

```
#include <math.h>
double cos(double x);
```

Return value **cos** returns the value in the range [-1,1] for the cosine of *x* which should be supplied in radians.

cosh

Purpose To compute the hyperbolic cosine of a double value.

Syntax

```
#include <math.h>
double cosh(double x);
```

Return value **cosh** returns the value for the hyperbolic cosine of *x*.

Notes A range error occurs if the magnitude of *x* is too large (returns **HUGE_VAL** and sets **errno** to **ERANGE**).

exp

Purpose To compute the standard exponential function for a double value.

Syntax

```
#include <math.h>
double exp(double x);
```

Return value **exp** returns the value for .

Notes A range error occurs if the value of *x* is too large (returns **HUGE_VAL** and sets **errno** to **ERANGE**).

fabs

Purpose To compute the modulus or absolute value of a double value.

Syntax

```
#include <math.h>
double fabs(double x);
```

Return value **fabs**(x) returns x if x $\geq$ 0, -x otherwise.

See also **abs**, **labs**

Example fabs(-1.0) gives 1.0.

floor

Purpose To compute the next (floor) integer below a given double value.

Syntax

```
#include <math.h>
double floor(double x);
```

Return value **floor** returns the value of *x* rounded to the next integer below.

Example floor(1.9) gives 1.0.

fmod

Purpose To compute the floating point remainder of two double values.

Syntax

```
#include <math.h>
double fmod(double x,double y);
```

Return value **fmod** returns the floating point remainder of *x*/*y*. This is the value *x*-*i***y* where the integer *i* is that which causes the result to have the same sign as *x* and the magnitude of the result is less than the magnitude of *y*. The sign of *y* is therefore not significant.

Notes A domain error occurs if *y* = 0. (returns **EDOM_VAL** and sets **errno** to **EDOM**).

Example fmod(4.0,3.5) gives 0.5.
fmod(-4.0,1.5) gives -1.0.

frexp

Purpose To reduce a double value to mantissa and exponent form.

Syntax

```
#include <math.h>
double frexp(double x, int *expon);
```

Description **frexp** splits *x* into a normalised fraction whose magnitude lies in the interval [0.5,1.0) or zero; and an exponent which is an integral power of 2 returned in *expon*.

x=fraction*( ).

Return value **frexp** returns the normalised fraction.

Example frexp(0.3,&*expon*) gives 0.6 with *expon*=(-1).

ldexp

Purpose To load a mantissa and exponent into a double value.

Syntax

```
#include <math.h>
double ldexp(double x,int expon);
```

Return value **ldexp** returns x*( ).

Notes A range error will occur if the combination is too large (returns **HUGE_VAL**, and sets **errno** to **ERANGE**).

Example ldexp(0.6,-1) gives 0.3.

log

Purpose To compute the natural logarithm of a double value.

Syntax

```
#include <math.h>
double log(double x);
```

Return value **log** returns the natural logarithm of *x*.

Notes A domain error occurs if $x < 0$
(returns **EDOM_VAL** and sets **errno** to **EDOM**).
A range error occurs if $x = 0$
(returns -**HUGE_VAL** and sets **errno** to **ERANGE**).

log10

Purpose To compute the logarithm to the base 10 of a double value.

Syntax

```
#include <math.h>
double log10(double x);
```

Return value **log10** returns logarithm to the base 10 of *x*.

Notes A domain error occurs if *x* < 0
(returns **EDOM_VAL** and sets **errno** to **EDOM**).
A range error occurs if *x* = 0
(returns -**HUGE_VAL** and sets **errno** to **ERANGE**).

modf

Purpose To reduce a double value to its integer and fractional parts.

Syntax

```
#include <math.h>
double modf(double x, double *ipart);
```

Description **modf** splits *x* into its integer part which is returned in *ipart* and its fractional part.   Each part has the same sign as *x*.

Return value **modf** returns the signed fractional part.

Example modf(-1.5,&*ipart*) gives -0.5 with *ipart* = (-1.0).

pow

Purpose To compute the power of a double value.

Syntax

```
#include <math.h>
double pow(double x,double y);
```

Return value **pow** returns Example.

Notes A domain error occurs if *x* < 0 and *y* is not an integer. A domain error occurs if *x* = 0 and *y* < 0 (returns **EDOM_VAL** and sets **errno** to **EDOM**). A range error occurs if the combination is too large (returns **HUGE_VAL** and sets **errno** to **ERANGE**).

sin

Purpose To compute the sine of a double value.

Syntax

```
#include <math.h>
double sin(double x);
```

Return value **sin** returns the value in the range [-1,1] for the sine of *x* which should be supplied in radians.

sinh

Purpose To compute the hyperbolic sine of a double value.

Syntax

```
#include <math.h>
double sinh(double x);
```

Return value **sinh** returns the value for the hyperbolic sine of *x*.

Notes A range error occurs if the magnitude of *x* is too large (returns **HUGE_VAL** with the appropriate sign and sets **errno** to **ERANGE**).

sqrt

Purpose To compute the positive square root of a double value.

Syntax

```
#include <math.h>
double sqrt(double x);
```

Return value **sqrt** returns the positive square root of *x*.

Notes A domain error occurs if $x < 0$
(returns **EDOM_VAL** and sets **errno** to **EDOM**).

tan

Purpose To compute the tangent of a double value.

Syntax

```
#include <math.h>
double tan(double x);
```

Return value **tan** returns the value for the tangent of *x* which should be supplied in radians.

tanh

Purpose To compute the hyperbolic tangent of a double value.

Syntax

```
#include <math.h>
double tanh(double x);
```

Return value **tanh** returns the value in the range (-1,1) for the hyperbolic tangent of *x*.

Functions defined in stdio.h

clearerr	freopen	rename
fclose	fscanf	rewind
fcloseall	fseek	scanf
feof	fsetpos	setbuf
ferror	ftell	setvbuf
fflush	fwrite	sprintf
fgetc	getc	sscanf
fgetpos	getchar	tmpfile
fgets	gets	tmpnam
flushall	perror	ungetc
fopen	printf	unlink
fprintf	putc	vfprintf
fputc	putchar	vprintf
fputs	puts	vsprintf
fread	remove	

About functions defined in stdio.h

The **<stdio.h>** header provides prototypes for a number of standard input and standard output functions.   It also defines a number of types and macros.   Those referred to in this section are listed below:

```
typedef unsigned int size_t;
typedef int fpos_t;

#define _IONBF 0
#define _IOLBF 1
#define _IOFBF 2

#define EOF (-1)

#define FOPEN_MAX 20
#define FILENAME_MAX 80

#define L_tmpnam 15
#define TMP_MAX 9999

#define BUFSIZ 512
#define STDIN_OUT_BUFSIZ 80

#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
```

The FILE structure

The stdout, stdin and stderr streams

The FILE structure

Data is normally transmitted to and from files, and to and from the console. A particular input/output stream is referenced by a pointer to an associated **FILE** structure. This structure contains details of the stream and its current state, together with a “buffer” which is a temporary store for the accumulation of data before transmission to the relevant device. The details of the **FILE** structure are implementation dependent and normally are of no interest to the user since the usage is hidden within the coding of the functions. The current definition can be found in the header file.

The stdout_ stdin and stderr streams

The stream **stdout** refers to the standard output stream, normally associated with the console, **stdin** refers to the standard input stream, normally associated with the keyboard, whilst **stderr** refers to the standard error stream, normally associated with the console. Each of these streams is automatically opened when the program is started and closed when it terminates. These streams cannot be opened or closed in any other way. They are line-buffered text streams with a buffer size of **_STDIN_OUT_BUFSIZ** (see setvbuf for details of buffering strategies). **stdout**, **stdin** and **stderr** are coded as pointers to **FILE** structures.

clearerr

Purpose To clear the error and end-of-file flags for a given stream.

Syntax

```
#include <stdio.h>
void clearerr(FILE *stream);
```

Return value None.

Example clearerr(stdin);

fclose

Example

Purpose To close a stream.

Syntax

```
#include <stdio.h>
int fclose(FILE *stream);
```

Description The **fclose** function flushes all buffers associated with the stream and de-allocates them (unless they were assigned by **setbuf** or **setvbuf**). The file is closed and disassociated from the stream.

stdin, **stdout** and **stderr** cannot be closed using **fclose** (an attempt to do so is not flagged as an error).

Return value **fclose** returns zero for success and non-zero for failure.

Notes **fclose** will fail if the stream does not exist or if its buffers cannot be flushed.

See also **fcloseall**

fcloseall

Purpose To close any open streams.

Syntax

```
#include <stdio.h>
int fcloseall(void);
```

Description The function **fcloseall** closes all the streams that have been successfully opened by calls to **fopen** or **freopen** . This does not include the standard streams **stdin**, **stdout** and **strerr**. See **fclose** for further details.

Return value **fcloseall** returns the number of streams that have been closed or **EOF** if one or more of the open streams could not be closed.

Notes An error will occur if, for example, a stream could not be flushed.

feof

Purpose To test if the end-of-file flag is set for a given stream.

Syntax

```
#include <stdio.h>
int  feof(FILE *stream);
```

Return value **feof** returns a non-zero value if the flag is set, otherwise **feof** returns zero.

Example See **fsetpos** and **rewind**.

ferror

Example

Purpose To test if the error flag is set for a given stream.

Syntax

```
#include <stdio.h>
int  ferror(FILE *stream);
```

Return value **ferror** returns a non-zero value if the flag is set, otherwise **ferror** returns zero.

fflush

Example

Purpose To flush a stream.

Syntax

```
#include <stdio.h>
int fflush(FILE *stream);
```

Description The **fflush** function causes any unwritten data in the output buffer associated with a given stream to be delivered to its destination. The stream remains open. **fflush** is not relevant to input streams.

Return value **fflush** returns zero for success and non-zero for failure.

See also **flushall**

fgetc

Example

Purpose To get a character from an input stream.

Syntax

```
#include <stdio.h>
int fgetc(FILE *stream);
```

Description The function **fgetc** gets an unsigned char from the given stream and converts it to an **int**. Where appropriate the buffer position indicator is advanced.

Return value **fgetc** either

- returns the character read or,
- if the buffer is empty, **fgetc** returns **EOF** and sets an end-of-file flag such that feof(*stream*) is true (non-zero).

Notes If a read error occurs, **fgetc** returns **EOF** and sets an error flag such that **ferror**(*stream*) is true (non-zero).

See also **getc**, **getchar**, **ungetc**, **fputc**

Example

```
// fgetc.cpp
#include <stdio.h>                 // for fgetc, etc.
int main()
{
  FILE *infile;
  int c;
  infile=fopen("test.tmp","r");   // file created by fprintf.cpp
  do
  {
    c=fgetc(infile);
    putchar(c);
  } while (c != EOF);
  if (ferror(infile)) printf("ERROR:Read error in fgetc\n");
  fclose(infile);
  return 0;
}
```

fgetpos

Example

Purpose To get the current value of the file position indicator.

Syntax

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

Description The **fgetpos** function stores the current value of the file position indicator (for the given stream) in the object of type **fpos_t** pointed to by *pos*. This value is then available for a later call of the **fsetpos** function. **fgetpos** is related to **fsetpos** in the same way that **ftell** is related to **fseek**.

Return value **fgetpos** returns zero if successful, non-zero otherwise.

fgets

Example

Purpose To get a string from a stream.

Syntax

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Description The function **fgets** reads a string of characters from a stream and puts it into the array pointed to by *s*. *n* is the maximum number of characters to be transmitted including a terminating null. Reading stops when a newline character is encountered if this occurs before *n*-1 characters have been read. In this case the newline character is also transmitted. Reading also stops when end-of-file is encountered.

Return value **fgets** returns *s* if successful.

Notes If end-of-file is encountered before any characters are transmitted then **fgets** returns a **NULL** pointer and the array pointed to by *s* is unchanged.

If a read error occurs, a **NULL** pointer is returned and the array pointed to by *s* will be incomplete.

See also **gets**, **fputs**

Example

```
// fgets.cpp
#include <stdio.h>                 // for fgets, etc.
int main()
{ FILE *infile;
  char str[80];
  infile=fopen("test.tmp","r");  // file created by fprintf.cpp
  while (fgets(str,80,infile)) puts(str);
  if (ferror(infile)) printf("ERROR:Read error in fgets\n");
  fclose(infile);
  return 0;
}
```

flushall

Purpose To flush all open output streams.

Syntax

```
#include <stdio.h>
int flushall(void);
```

Description The function **flushall** flushes all the output streams that are currently open, including **stdout** and **stderr**. See **fflush** for further details.

Return value **flushall** returns a count of the number of streams that are open (including **stdin**, **stdout** and **stderr**). Note, however, that the function has no effect on input streams.

fopen

Example

Purpose To open a file and associate a new stream with it.

Syntax

```
#include <stdio.h>
FILE *fopen(const char *filename,
            const char *mode);
```

Description *filename* points to the name of a file which is opened by **fopen** and which associates the file with a stream. The name refers to the default directory unless a pathname is included. The DOS PATH environment variable is not effective.

mode points to a string consisting of one of the following options:

mode	**Description**
r	Open file for reading only.
w	Create and open a file for writing only or open and clear an existing file for writing.
a	Create and open a file for writing only or open an existing file for writing at end-of-file.
r+	Open file for update (reading and writing).
w+	Create and open a file for update or open and clear an existing file for update.
a+	Create and open a file for update or open an existing file for update, writing at end-of-file.

In these forms the file is assumed to be a text file. A binary file is designated by an appended letter b (e.g. "*rb+*" or "*r+b*" etc.). Similarly a letter t designates a text file (the default).

When a file is opened for update, both reading and writing are possible on the associated stream but writing cannot be followed by reading without an intervening **fsetpos, fseek** or **rewind**. Similarly reading cannot be followed by writing without an intervening **fsetpos, fseek**, **rewind** or a read that encounters an end-of-file marker.

fopen allocates space for a buffer of type **_IOFBF** (fully buffered, see **setvbuf** for details) and size **BUFSIZ**. To change either of these values use **setvbuf**.

FOPEN_MAX (or possibly more) files can be opened simultaneously. **FILENAME_MAX** is the maximum length of a filename string.

Return value **fopen** returns a pointer to the new stream or a **NULL** if the function fails.

Notes If the *mode* is invalid or the file cannot be opened then the global variable **errno** will indicate the cause of failure (see **perror**).

See also **freopen**

Example

Example

```
// fopen.cpp
#include <stdio.h>                    // for fopen,perror etc.
int main()
{
  char fname[80],mode[10];
  FILE *tfile;
  while(1)
  {
    printf("Filename:"); gets(fname);
    if(!*fname) break;
    printf("Mode    :"); gets(mode);
    tfile=fopen(fname,mode);
    if (tfile)
    {
      printf("File opened OK\n");
      fclose(tfile);              // note file is not 'removed'
    }
    else perror("Error in fopen");
  }
  return 0;
}
```

fprintf

Example

Purpose To write formatted output to a stream.

Syntax

```
#include <stdio.h>
int fprintf(FILE *stream,
            const char *format,...);
```

Description **fprintf** is the same as **printf** except that the output is directed to a specified stream. See **printf** for further details.

Return value **fprintf** returns the number of characters transmitted.

See also **sprintf**

Example

```
// fprintf.cpp
#include <stdio.h>                    // for fprintf, etc.
int main()
{
  FILE *outfile;
  outfile=fopen("test.tmp","w");
  fprintf(outfile,"Buffer size:%d\n",outfile->buffer_size);
  fprintf(outfile,"Buffer type:%d\n",outfile->buffer_type);
  fprintf(outfile,"File name  :%s\n",outfile->name);
  fprintf(outfile,"File handle:%d\n",outfile->handle);
  fclose(outfile);
  printf("Now list the file test.tmp....");
  return 0;
}
```

fputc

Example

Purpose To write a character to a stream.

Syntax

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

Description The function **fputc** writes *c* as an unsigned char to the given stream at the position given by the file position indicator.   The buffer position indicator is advanced (unless the stream is unbuffered).

Return value **fputc** returns the character *c*.

Notes If a write error occurs, **fputc** returns **EOF** and sets an error flag such that ferror(*stream*) is true (non-zero).

See also **putc**, **putchar**, **fgetc**

Example

Example

```
// fputc.cpp
#include <stdio.h>                // for fputc, etc.
#include <ctype.h>                   // for toupper
int main()
{ FILE *infile,*outfile;
  int c;
  infile =fopen("test.tmp","r"); // file created by fprintf.cpp
  outfile=fopen("test.new","w");
  do
  {
    c=fgetc(infile);
    fputc(toupper(c),outfile);   // change to upper case
  } while (c != EOF);
  if (ferror(outfile)) printf("ERROR:Write error in fputc\n");
  fclose(infile);
  fclose(outfile);
  remove("test.tmp");
  rename("test.new","test.tmp");
  return 0;
}
```

fputs

Example

Purpose To write a string to a stream.

Syntax

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Description **fputs** writes the string pointed to by *s* to the given stream. The terminating null is not copied and (in contrast to **puts**) a newline character is not appended.

Return value **fputs** returns zero or a positive value if successful, otherwise, if a write error occurs, **fputs** returns **EOF**.

See also **fgets**

Example

```
// fputs.cpp
#include <stdio.h>                // for fputs, etc.
#include <string.h>               // for strupr
int main()
{ FILE *infile,*outfile;
  char str[80];
  infile=fopen("test.tmp","r");  // file created by fprintf.cpp
  outfile=fopen("test.new","w");
  while (fgets(str,80,infile))
    fputs(strupr(str),outfile);   // change to upper case
  if (ferror(outfile)) printf("ERROR:Write error in fputs\n");
  fclose(infile);
  fclose(outfile);
  remove("test.tmp");
  rename("test.new","test.tmp");
  return 0;
}
```

fread

Example

Purpose To read data from a stream.

Syntax

```
#include <stdio.h>
size_t fread(void *ptr, size_t s, size_t n,
             FILE *stream);
```

Description The function **fread** reads data from the given stream into the array pointed to by *ptr*. *n* elements of size *s* are read into the array so the number of bytes is (*s*n*). *ptr* is a pointer to an array of any type. The file position indicator is advanced by the number of bytes read.

Return value **fread** returns the number of elements of size *s* that have been successfully read or a value less than *n* if a read error occurs. If either *n* or *s* is zero then there is no change to the array or the stream and **fread** returns a zero.

Notes The cause of failure is given by the value of the global variable **errno**.

See also **fwrite**

Example

```
// fread.cpp
#include <stdio.h>                   // for fread, etc.
#include <string.h>                  // for memset
#include <stdlib.h>                  // for atoi
int main()
{ FILE *tfile;
  char buf[256];
  int n,m;
  tfile=fopen("test.$$$","r");    // file created by fwrite.cpp
  printf("Number of characters in file:48\n");
  printf("Number to read in            :");
  gets(buf); n=atoi(buf);
  memset(buf,0,256);
  m=fread(buf,1,n,tfile);
  printf("Number successfully read    :%d\n",m);
  if (m<n) perror("Error in fread              ");
  printf("The buffer contains         :\n%s\n",buf);
  fclose(tfile);
  return 0;
}
```

freopen

Example

Purpose To open a file and associate an existing stream with it.

Syntax

```
#include <stdio.h>
FILE *freopen(const char *filename,
             const char *mode, FILE *stream);
```

Description The **freopen** function calls upon **fclose** to flush the buffer associated with the stream (but not to de-allocate the space for the **FILE** structure) and then reopens the stream with the new filename by calling upon **fopen**. For example, it may be used to change or allocate a filename in association with **stdin**, **stdout**, or **stderr**.

A failure to **fclose** the stream is ignored. See **fclose** and **fopen** for further details.

Return value **freopen** returns the same value as **fopen**.

Notes See notes for **fopen**.

Example

```
// freopen.cpp
#include <stdio.h>                // for freopen,perror etc.
int main()
{ char fname[80],mode[10];
  FILE *tfile;
  tmpnam(fname);
  tfile=fopen(fname,"w");         // note file is not ’removed’
  while(1)
  {
   printf("Filename:"); gets(fname);
   if(!*fname) break;
   printf("Mode    :"); gets(mode);
   tfile=freopen(fname,mode,tfile);//note file is not ’removed’
   if (tfile) printf("File reopened OK\n");
   else perror("Error in freopen");
  }
  return 0;
}
```

fscanf

Example

Purpose To scan and format input from a stream.

Syntax

```
#include <stdio.h>
int fscanf(FILE *stream, char *format,...);
```

Description **fscanf** is the same as **sscanf** except that the input is read from a stream (rather than a string). Encountering the end-of-file marker is equivalent to reaching the end of a string in **sscanf**.

A subsequent call of **fscanf** will continue to read data from the position in the stream at which the last call stopped (this will not be the expected position if an earlier call was not completely successful). See **sscanf** for further details.

fscanf can produce unexpected results when the end of a line is incorrectly scanned; **fgets** with **sscanf** is normally preferred.

Return value **fscanf** returns the number of input items assigned.

See also **scanf**

Example

```
// fscanf.cpp
#include <stdio.h>                // for fscanf, etc.
int main()
{ FILE *infile;
  int size,type,handle;
  char name[80],d[20];
  infile=fopen("test.tmp","r");  // file created by fprintf.cpp
  fscanf(infile,"%s%d%s%d%s%s%s%d",d,&size,d,&type,d,name,d,&handle);
  printf("size :%d\ntype :%d\nname :%s\nhandle:%d\n",
         size,type,name,handle);
  fclose(infile);
  return 0;
}
```

fseek

Purpose To set the current value of the file position indicator.

Syntax

```
#include <stdio.h>
int fseek(FILE *stream, long offset,
          int whence);
```

Description The **fseek** function sets the file position indicator for the given stream to a new value as follows. The value and meaning of *whence* is one of:

whence	**position**
SEEK_SET	Beginning of file
SEEK_CUR	Current position
SEEK_END	End-of-file

whilst *offset* is the number of characters to be added to this position. Commonly *whence* is set to **SEEK_SET** and *offset* is a value returned by an earlier call to **ftell**.

fseek discards any characters pushed back into the stream by **ungetc** and clears the end-of-file flag for the stream. An **fseek** call on an update stream (see **fopen**) can be followed by either an input or an output operation.

Return value **fseek** returns zero for success and non-zero for failure.

Notes The cause of failure is given by the value of the global variable **errno**.

See also **fsetpos**

Example See **ftell** and **rewind**.

fsetpos

Example

Purpose To set the value of the file position indicator.

Syntax

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

Description The **fsetpos** function sets the value of the file position indicator (for the given stream) as the value of the object of type **fpos_t** pointed to by *pos.* This is the value given by an earlier call of the **fgetpos** function.

fsetpos is equivalent to **fseek** (with *whence*=**SEEK_SET**).

fsetpos discards any characters pushed back into the stream by **ungetc** and clears the end-of-file flag for the stream. An **fsetpos** call on an update stream (see **fopen**) can be followed by either an input or an output operation.

Return value **fsetpos** returns zero for success and non-zero for failure.

Notes The cause of failure is given by the value of the global variable **errno**.

Example

Example

```
// fsetpos.cpp
#include <stdio.h>                  // for fsetpos, etc.
#include <stdlib.h>                 // for atoi
int main()
{ FILE *tfile;
  fpos_t line_pos[10];
  char buf[80];
  int i;
  tfile=fopen("test.$$$","r");    // file created by fwrite.cpp
  for(i=1;!feof(tfile);i++)
  {
     fgetpos(tfile,&line_pos[i]);
     printf("position of line %d:%d\n",i,line_pos[i]);
     fgets(buf,80,tfile);
  };

  while(1)
  {
    printf("Line number:"); gets(buf); i=atoi(buf);
    if (!*buf) break;
    fsetpos(tfile,&line_pos[i]);
    buf[0]='\0';
    fgets(buf,80,tfile);
    puts(buf);
  }
  fclose(tfile);
  return 0;
}
```

ftell

Example

Purpose To get the current value of the file position indicator.

Syntax

```
#include <stdio.h>
long ftell(FILE *stream);
```

Description The **ftell** function returns the current value of the file position indicator (for the given stream).

This value is then available for a later call of the **fseek** function.

Return value **ftell** returns the value of the file position indicator or -1 for failure.

See also **fgetpos**

Example See also **rewind**.

Example

```
// ftell.cpp
#include <stdio.h>                 // for ftell, etc.
#include <stdlib.h>                // for atoi
int main()
{
  FILE *tfile;
  long line_pos[10];
  char buf[80];
  int i;
  tfile=fopen("test.$$$","r");    // file created by fwrite.cpp
  for(i=1;!feof(tfile);i++)
  {
     line_pos[i]=ftell(tfile);
     printf("position of line %d:%d\n",i,line_pos[i]);
     fgets(buf,80,tfile);
  };

  while(1)
  {
    printf("Line number:"); gets(buf); i=atoi(buf);
    if (!*buf) break;
    fseek(tfile,line_pos[i],SEEK_SET); buf[0]='\0';
    fgets(buf,80,tfile);  puts(buf);
  }
  fclose(tfile);
  return 0;
}
```

fwrite

Example

Purpose To write data to a stream.

Syntax

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t s,
             size_t n,FILE *stream);
```

Description The function **fwrite** writes data from the array pointed to by *ptr* to the given stream.

n elements of size *s* are written to the stream so the number of bytes is (s*n). *ptr* is a pointer to an array of any type. For a buffered stream, the file position indicator is advanced by the number of bytes written.

Return value **fwrite** returns the number of elements of size *s* that have been successfully written or a value less than *n* if a write error occurs. If either *n* or *s* is zero then there is no change to the stream and **fwrite** returns a zero.

Notes The cause of failure is given by the value of the global variable **errno**.

See also **fread**

Example

Example

```
// fwrite.cpp
#include <stdio.h>                          // for fwrite, etc.
#include <string.h>                         // for strlen
int main()
{
  FILE *tfile;
  char buf[]="**First line**\n**Second line**\n**Third line**\n";
  int n,m;
  tfile=fopen("test.$$$","w");
  n=strlen(buf);
  printf("Number of characters in string:%d\n",n);
  m=fwrite(buf,1,n,tfile);
  printf("Number successfully written   :%d\n",m);
  if (m<n) perror("Error in fwrite                ");
  fclose(tfile);
  return 0;
}
```

getc

Purpose To get a character from an input stream.

Syntax

```
#include <stdio.h>
int getc(FILE *stream);
```

Description The function **getc** is identical to **fgetc**. See **fgetc** for further details.

See also **getchar**, **ungetc**, **fputc**

getchar

Purpose To get a character from **stdin**.

Syntax

```
#include <stdio.h>
int getchar(void);
```

Description The function **getchar** is equivalent to fgetc(stdin). See **fgetc** for further details.

gets

Example

Purpose To get a string from **stdin**.

Syntax

```
#include <stdio.h>
char *gets(char *s);
```

Description The function **gets** reads a string of characters from the standard input stream and puts it (including a terminating null) into the array pointed to by *s*.

Reading stops when a newline character is encountered but (in contrast to **fgets**) the newline is not transmitted. Reading also stops when end-of-file is encountered but this will not occur unless **stdin** is redirected using **freopen**. Otherwise, in the default state, the system will wait for input from the keyboard.

Return value **gets** returns *s* if successful.

Notes If end-of-file is encountered before any characters are transmitted then **gets** returns a **NULL** pointer and the array pointed to by *s* is unchanged.

If a read error occurs, a **NULL** pointer is returned and the array pointed to by *s* will be incomplete.
The user should ensure that the array *s* is long enough to store the input string.

See also **fputs**

Example

```
// gets.cpp
#include <stdio.h>                    // for gets, etc.
int main()
{
  char str[80];
  printf("Input:\n");
  do
  {
    gets(str);
    puts(str);
  } while (*str);
  if (ferror(stdin)) printf("ERROR:Read error in gets\n");
  return 0;
}
```

perror

Example

Purpose To print an error message to **stderr**.

Syntax

```
#include <stdio.h>
void perror(const char *s);
```

Description The **perror** function prints an error message to the standard error stream **stderr** (the console by default). The message corresponds to the current value of the global variable **errno** (see **<errno.h>**).

First, if *s* is not **NULL**, the string pointed to by *s* is printed, followed by a colon and a space (the user may supply this in order to point to the place where the error occurred). Then a message corresponding to the value of **errno** is printed (this is given by strerror(errno)) followed by a newline.

Return value None.

See also **strerror**

printf

Purpose To write formatted output to **stdout**.

Syntax

```
#include <stdio.h>
int printf(const char *format,...);
```

Description The function **printf** writes output from a list of arguments to the **stdout** stream using format specifiers given in the string pointed to by *format*.

For each argument in the list (denoted by ...) there must be a corresponding format specifier in the string pointed to by *format*. If insufficient arguments are provided then the outcome is unsafe and possibly fatal (The /CHECK option will detect this fault reliably, see also chapter 5 in the manual).

A specifier consists of a % character followed by a number of characters which describe how the corresponding argument is to be presented. Characters which are not part of a specifier are copied to the output stream.

A format specifier has the following general form:

% [flags] [width] [.precision] [size] type

where the brackets signify an optional element. The details are given below.

Type conversion specifier:

In the following table, *dddd* denotes a decimal digit (0..9) string of arbitrary length. For floating-point arguments, the number of digits after an optional decimal point will be determined by the [.precision] element or by default precision values given later.

char:	**type:**	**format of output:**
d	integer	[-]dddd
i	integer	[-]dddd
o	unsigned integer	octal form
u	unsigned integer	decimal form
x	unsigned integer	hexadecimal form, lower case letters
X	unsigned integer	hexadecimal form, upper case letters
p	pointer	hexadecimal form of a pointer
f	floating-point	[-]dddd.ddddd
e	floating-point	[-]d.dddde(+/-)ddd
E	floating-point	[-]d.ddddE(+/-)ddd the exponent contains at least two digits, if the value is

		zero the exponent is 00
g,G	floating-point	either f or e/E; trailing zeros are removed (unless the # flag is used); the e/E form is only used if the exponent is less than -4 or greater than or equal to the precision value
c	integer	conversion to unsigned char
s	pointer to string	string
%	none	the % character
n	pointer to int	A count of the number of characters written so far is stored in the location pointed to by the argument.

Size modifier:

The size of the input argument is assumed be **sizeof**(**int**) for integer values and **sizeof**(**double**) for floating-point arguments unless the size is modified by using *h* (for **short int**), *l* (for **long int**), or *L* (for **long double**).

Precision modifier:

The precision modifier begins with a full stop (.) and is followed by either a decimal digit string (denoted by *n*) representing the precision or an asterisk (*) which signifies that the precision is given by the next argument in the argument list. The details are as follows. Default values appear towards the end of the table. Initially *n* denotes a positive integer.

modifier	**type specifier**	**precision of output**
.*n*	d,i,o,u,x,X	*n* is the minimum number of digits to be printed, using zeros to pad out on the left if necessary, more than *n* characters can still be output (padding for [precision] should not be confused with padding for [width]);
.*n*	f	*n* is the number of decimal places; if necessary the last digit is rounded;
.*n*	e,E	*n* is the number of decimal places with one non-zero digit before the point (unless the value is zero); if necessary the last digit is rounded;
.*n*	g,G	*n* is the maximum number of significant digits to be printed; if necessary the last digit is rounded;
.*n*	s	*n* is the maximum number of characters to be printed;
.*n*	c	no effect
.*	any	as for .*n* but the precision *n* is supplied as the next argument in the argument list which will come before the value in question
default	d,i,o,u,x,X	*n*=1

default	f,e,E,g,G	*n*=6
default	s	terminate at null
.0 or .	d,i,o,u,x,X	*n*=1
.0 or .	f,e,E	*n*=0; no decimal point (unless # flag is used)
.0 or .	g,G	*n*=1; no decimal point (unless # flag is used)

Width modifier:

The width modifier can be used to specify the minimum width of the field for output; more characters will be printed if necessary rather than truncate the result. Where fewer than the minimum is required, the field is padded out (on the left with spaces unless a flags modifier is used, see below).

A decimal digit string representing an integer value *n* signifies that at least *n* characters are printed, padding if necessary; An asterisk (*) signifies that the width is supplied as the next argument in the argument list which will come before the value in question.

Flags modifier:

The modifiers “default”, “-”, and “0” below relate to the padding operation associated with the [width] modifier, in the absence of which these [flags] modifiers are redundant. “+” and “space” are used to force a + sign or space in a non-negative result. “#” is used to convert certain results can to what is called an “alternative form”.

modifier	**type specifier**	**format of output**
default		result is right justified in the field padding with spaces
-		result is left justified in the field padding with spaces
0	f,e,E,g,G	result is right justified in the field padding with zeros; (ignored if - flag is used)
0	d,i,o,u,x,X	result is right justified in the field padding with zeros; (ignored if - flag is used, ignored if precision is specified)
+	not c or s	forces a ’+’ sign for a signed conversion with a non-negative result
space	not c or s	forces a space for a signed conversion with a non-negative result (ignored if + flag is used)
#	o	zero is inserted before the value, and after any padding
#	x,X	0x (or 0X) is inserted before the value, and after any padding

#	e,E,f,g,G	forces a decimal point when zero precision is specified
#	g,G	forces a decimal point and prevents trailing zeros from being removed

Return value **printf** returns the number of characters output.

See also **fprintf**, **sprintf**

Examples *k*=12.

f	**printf(*f,k*)**	***f***	**printf(*f,k*)**
`"%6d"`	`    12`	`"%6.4x"`	`  000c`
`"%6.4d"`	`  0012`	`"%#6X"`	`   0XC`
`"%-6.4da"`	`0012  a`	`"%#6.4x"`	`0x0c`
`"%06d"`	`000012`	`"%0#6x"`	`0000xc`
`"%+6.4d"`	`+00012`	`"%#8.6x"`	`0x000c`

x=3.45 y=3.45e-2

f	printf(*f,x*)	*f*	printf(*f,y*)
`"%6.3f"`	`3.450`	`"%f`	`0.034500`
`"%06.2f"`	`003.45`	`"%.3e"`	`3.450e-02`
`"%-6.2fa"`	`3.45  a`	`"%-10.2ea"`	`3.45e-02  a`
`"%+6.2f"`	`+3.45`	`"%010.2E"`	`003.45E-02`
`"%#6.0f"`	`3.`	`"%#8.0e"`	`3.e-02`

putc

Purpose To write a character to a stream.

Syntax

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Description The function **putc** is identical to **fputc**. See **fputc** for further details.

See also **putchar**, **fgetc**

putchar

Purpose To write a character to **stdout**.

Syntax

```
#include <stdio.h>
int putchar(int c);
```

Description The function call putchar(*c*) is equivalent to putc(*c*,stdout). See **fputc** for further details.

See also **fgetc**

puts

Example

Purpose To write a string to **stdout**.

Syntax

```
#include <stdio.h>
int fputs(const char *s);
```

Description **puts** writes the string pointed to by *s* to the standard output stream. The terminating null is not copied but (in contrast to **fputs**) a newline character is appended.

Return value **put** returns zero or a positive value if successful, otherwise, if a write error occurs, **puts** returns **EOF**.

See also **fgets**

remove

Example

Purpose To erase a file.

Syntax

```
#include <stdio.h>
int remove(const char *filename);
```

Description The **remove** function erases the file whose name is given by the string pointed to by *filename*. The file should either be in the default directory or the name may include a DOS pathname. Relative pathnames (e.g. "..\file") are also accepted. The name should correspond to a single file; wild cards (* or ?) are not accepted. A directory cannot be accessed via the DOS PATH environment variable. **remove** is similar in some respects to the DOS ERASE command.

Return value **remove** returns zero for success and non-zero for failure.

Notes The cause of failure is given by the value of the global variable **errno** (see **perror**).

A file that is open, should be closed before being **remove**d.

Example

Example

```
// remove.cpp
#include <stdio.h>                    // for remove etc
int main()
{ char fname[80];
  while(1)
  {
    printf("Delete filename:"); gets(fname);
    if (!*fname) break;
    if (remove(fname)==0) printf("%s removed\n",fname);
    else                  perror("Error in remove");
  }
  return 0;
}
```

rename

Example

Purpose To rename a file.

Syntax

```
#include <stdio.h>
int rename(const char *oldfile,
           const char *newfile);
```

Description The **rename** function changes the name of the file whose name is given by the string pointed to by *oldfile*, to the name given by the string pointed to by *newfile*. Either the directory will be the default directory for both files, or both names will include a DOS pathname which may be different for the two files provided that the disc drive is the same.

A directory cannot be accessed via the DOS PATH environment variable. Wildcards (* or ?) are not permitted.

rename is similar to the DOS RENAME command.

Return value **rename** returns zero for success and non-zero for failure.

Notes The cause of failure is given by the value of the global variable **errno** (see **perror**). For example, if the file *newfile* already exists then **rename** will fail.

Example

```
// rename.cpp
#include <stdio.h>                    // for rename etc
int main()
{ char oldfile[80],newfile[80];
  while(1)
  {
    printf("Old name:");  gets(oldfile);
    if (!*oldfile) break;
    printf("New name:");  gets(newfile);
    if (rename(oldfile,newfile)==0)
                  printf("%s renamed to %s\n",oldfile,newfile);
    else       perror("Error in rename");
  }
  return 0;
}
```

rewind

Example

Purpose To set a file position indicator to the beginning of the file.

Syntax

```
#include <stdio.h>
void rewind(FILE *stream);
```

Description The **rewind** function sets the value of the file position indicator (for the given stream) to the beginning of the file and clears the error flag for the stream.

rewind is equivalent to fseek(*stream*,0,SEEK_SET) followed by a call to **clearerr**.

Return value None.

See also **fseek**

Example

Example

```
// rewind.cpp
#include <stdio.h>                 // for rewind, etc.
#include <stdlib.h>                // for atoi
int main()
{ FILE *tfile;
  long line_pos[10];
  char buf[80];
  int i,j;
  tfile=fopen("test.$$$","r+");   // file created by fwrite.cpp
  i=1;
  do
  {  line_pos[i++]=ftell(tfile);
     buf[0]='\0';
     fgets(buf,80,tfile);
     printf("%s",buf);
  } while (!feof(tfile));
  while(1)
  {
    printf("Change line number     :"); gets(buf); i=atoi(buf);
    if (!*buf) break;
    printf("Change character number:"); gets(buf); j=atoi(buf);
    printf("Change character to    :"); gets(buf);
    fseek(tfile,line_pos[i]+j-1,SEEK_SET);
    fputc(buf[0],tfile);
    rewind(tfile);
    do
    {
       buf[0]='\0';
       fgets(buf,80,tfile);
       printf("%s",buf);
    } while(!feof(tfile));
  }
  fclose(tfile);
  return 0;
}
```

scanf

Purpose To scan and format input from **stdin**.

Syntax

```
#include <stdio.h>
int scanf(char *format,...);
```

Description **scanf** is the same as **fscanf** except that the input is read from **stdin**. See **sscanf** for further details.

scanf can produce unexpected results when the end of a line is incorrectly scanned; **gets** with **sscanf** is normally preferred.

Return value **scanf** returns the number of input items assigned.

setbuf

Purpose To assign a new buffer to a stream.

Syntax

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Description **setbuf** is a special case of **setvbuf** to which the reader is referred for details.   If *buf* is not a **NULL** pointer then the stream is fully buffered and the size of the buffer is **BUFSIZE** (i.e. setvbuf(stream,buf,_IOFBF,BUFSIZE)).

If *buf* is a **NULL** pointer then the stream is unbuffered (i.e. setvbuf(stream,NULL,_IONBF,0)).

Notes See notes for **setvbuf**.

setvbuf

Example

Purpose To assign a new buffer to a stream.

Syntax

```
#include <stdio.h>
void setvbuf(FILE *stream, char *buf,
             int mode, size_t size);
```

Description The **setvbuf** function is used to assign a new buffer to a stream. For example, it may be used to change the buffering of one of the standard streams (**stdin**, **stdout**, **stderr**) or to create a buffer with a size or buffering strategy different from the default. It may only be used after the stream has been associated with an open file. If the old buffer is not empty then the function fails. Where possible, space for the old buffer is de-allocated. If *buf* is **NULL** then space for the new buffer is allocated by **setvbuf** otherwise *buf* points to the new buffer. In both cases the size of the buffer is *size* bytes.

The type of buffer is given by the value of *mode* as follows:

mode	**Description**
_IONBF	The file is unbuffered: The *buf* and *size* values are ignored. Input is read directly from the file. Output is written immediately to the file.
_IOLBF	The file is line buffered: Data is transmitted to the buffer whilst it is not full. The buffer is automatically flushed when a newline is encountered or when the buffer is full.
_IOFBF	The file is fully buffered: Data is transmitted to the buffer whilst it is not full. The buffer is automatically flushed when it is full.

Return value **setvbuf** returns zero for success and non-zero for failure.

Notes **setvbuf** fails if the original buffer was not empty or if a new buffer could not be created with the given *size* or if an invalid *mode* value was supplied. The cause of failure is given by the value of the global variable **errno** (see **perror**).

See also **setbuf**

Example

```
// setvbuf.cpp
#include <stdio.h>                  // for setvbuf, etc.
#include <string.h>                 // for memset
int main()
{
  char buff[21],str[80];
  FILE *outfile;
  outfile=fopen("test.tmp","w");
  setvbuf(outfile,buff,_IOFBF,20);
  memset(buff,'*',20);
  buff[20]='\0';
  while(1)
  {
    printf("input :"); gets(str);
    if (!*str) break;
    if (*str=='@') fflush(outfile);
    else           fputs(str,outfile);
    printf("buffer:%s\n",buff);
  }
  return 0;
}
```

sprintf

Purpose To write formatted output to a string.

Syntax

```
#include <stdio.h>
int sprintf(char *s, const char *format,...);
```

Description **sprintf** is the same as **printf** except that the output is directed to the string pointed to by *s* (rather than a stream). A null character is appended. See **printf** for further details.

Return value **sprintf** returns the number of characters transmitted excluding the terminating null.

See also **fprintf**

sscanf

Purpose

To scan and format input from a string.

Syntax

```
#include <stdio.h>
int sscanf(const char *s, char *format,...);
```

Description

The **sscanf** function reads input from the string pointed to by *s* under the control of the contents of the string pointed to by *format*. The string *s* contains a number of "fields" that are to be scanned and converted according to "specifiers" that appear in the format string. Normally, for each field and specifier, there will be a corresponding pointer in a list of pointers (designated by "..." above) to objects which are to receive the converted input.

If insufficient pointers are provided then the outcome is unsafe and possibly fatal (the /CHECK option will detect this fault reliably, see also chapter 5 in the manual).

The format string contains three types of objects:

- a sequence of white-space characters (as specified in the description of the **isspace** function) including spaces, tabs ('\t') and new lines ('\n'),
- a sequence of *ordinary* characters; an *ordinary* character is defined here to be a non-white-space character excluding an isolated % character,
- a sequence of characters (denoting a "specifier") which begins with a % character, followed by other characters in accordance with the rules given below.

As the format string is read, white-space characters in the format string are passed over and ignored. *Ordinary* characters are matched with corresponding characters in the input string *s*. If the characters fail to match then **sscanf** terminates at this point. If they do match, then these characters in the input string are passed over and otherwise ignored. %% in the format string is matched by % in the input string.

A specifier has the following general form:

% [skip] [width] [size] type

where the brackets signify an optional element.

If the width is not specified, a field in the input string is terminated either by a white-space character or by a character that cannot be converted according to the current specifier or by a null character (**EOF** for input from a stream using **scanf**, **fscanf** etc.). Leading white-space characters are skipped except under %c.

Type conversion specifier.

The *base* below refers to an argument in the function **strtol** or **strtoul** (one or other of these functions is implicitly called in each case where the base is given). A zero value for *base* means that the radix is 8,10,or 16 depending on leading characters in the field. Under *field expected*, *dddd* denotes a sequence of decimal digits (0..9) of arbitrary length and *n* denotes a non-zero decimal digit (1..9). The type of argument given is the default type which may be modified by using the [size] element. Where a list of alternatives appear under *char*, in this context the alternatives are interchangeable.

char:	**field expected:**	***base*:**	**default type of argument:**
d	[-]dddd	10	pointer to int
o	octal integer	8	pointer to unsigned int
x	hexadecimal integer	16	pointer to unsigned int
p	hexadecimal integer	16	pointer to void
u	dddd	10	pointer to unsigned int
i	ndddd (decimal) or 0dddd (octal) or 0Xdddd (hex)	0	pointer to int
f,e,g	see strtod		pointer to float
s	character string terminated by white-space or null or [width] value		pointer to an array of char large enough for the string with added terminating null
c	1 character or [width] characters including white-space, Use %1s for the next non-white-space character		pointer to char or array of char large enough for the sequence (no terminating null)
[	character string		pointer to an array of char large enough for the selected characters (see below) with added terminating null
n	none		pointer to an int which is to hold the number of characters read so far from the input string
%	% (i.e.%% matches %)		none

Here are some examples:

format	**input**	**output**
`"number%d"`	`"number 420"`	`int k=420`
`"value=%i"`	`"value=0x0010"`	`int k=16`
`"%e"`	`"3.45e-02"`	`double x=0.0345`

`"%s"`	`"my name is"`	`char str[]="my"`
`"%50s"`	`"my name is"`	`char str[]="my name is"`
`"%6c"`	`"my name is"`	`char str[]="my nam"`
`"%1s"`	`"  my"`	`char c='m'`

The %[specifier.

A specifier of the form %[*scan_set*] operates in a manner similar to %s but *scan_set* provides a list of the characters that may legitimately be found in the string. The string is terminated as for %s (optionally with width modifier) or when a character which is not in the list is encountered (see examples below). A specifier of the form %[^*scan_set*] is similar, but here *scan_set* provides a list of characters that may be used to terminate the string (in this case, if a space character is a valid terminator, then it must appear explicitly in *scan_set*).

The character] can be included in *scan_set* provided it is the first character of the list *scan_set* (the first after [or [^). The character ^ can be included in *scan_set* provided that it is not the first character for the form %[*scan_set*]. The minus character (-) (sometimes used to signify a range of characters) and the back slash (\) have no special significance in this context.

Here are some examples.

format	**input**	**output**
`%[abc]`	`ccbd`	`ccb`
`%2[abc]`	`ccbd`	`cc`
`%[abc]`	`cc d`	`cc`
`%[[bc]`	`cc[]`	`cc[`
`%[][c]`	`cc[]`	`cc[]`
`%[ab^]`	`^bc`	`^b`
`%[^abc]`	`defa`	`def`
`%[^abc]`	`defda`	`defd`
`%[^abc]`	`defda`	`def`

The skip modifier.

An asterisk (*) may be inserted to show that the corresponding field and conversion is to be discarded and not assigned to the next object pointed to in the list.

The width modifier.

A decimal digit string representing a positive integer, may be inserted to show the maximum number of characters that are to be read from the next field.

Fewer than this number will be read if the end of the field is reached first. If the maximum number is read then a subsequent field may follow immediately.

The size modifier.

The type of the receiving object may be modified from the default given above by using the size modifier *h* for the types d,i,o,x,X,u in order to obtain a pointer to a corresponding (signed or unsigned) **short**. *l* may similarly be used for a pointer to a **long** (the default in this implementation). *l* may also be used with the types f,e,g,E,G in order to obtain a pointer to a **double**, whilst, correspondingly, *L* indicates a pointer to a **long double**.

Return value **sscanf** returns the number of input items assigned.

Notes A return value that is less than the number of pointers given (after the format pointer) indicates an error.

If an inspection of the values assigned does not reveal the error, then %*n* may provide a useful diagnostic facility.

See also **fscanf**, **scanf**

tmpfile

Example

Purpose To create a temporary binary file.

Syntax

```
#include <stdio.h>
FILE *tmpfile(void);
```

Description The **tmpfile** function creates and opens a temporary file in the default directory with the mode “*wb+*” (see **fopen**) and a unique name. The file will be automatically **remove**d when it is closed or when the program terminates normally.

Return value **tmpfile** returns a pointer to the stream of the file created.

See also **tmpnam**

Example

```
// tmpfile.cpp
#include <stdio.h>                    // for tmpfile,printf
int main()
{  FILE *tfile=tmpfile();
  printf("Name of file opened is %s\n",tfile->name);
  return 0;
}
```

tmpnam

Example

Purpose To create a temporary file name.

Syntax

```
#include <stdio.h>
char *tmpnam(char *s);
```

Description The **tmpnam** function creates a filename which is different from any other filename in the default directory. Such a filename can be safely used to open a temporary file using the **fopen** function. A file opened in this way will not be erased when the file is closed or the program terminates. **tmpnam** (with **fopen**) is a permanent alternative to **tmpfile**.

Return value If *s* is a **NULL** pointer then **tmpnam** returns a pointer to a static string. This string will be overwritten by later calls of **tmpnam**(NULL). If *s* points to an array, then **tmpnam** returns *s*.

Notes The array pointed to by *s* should contain at least **L_tmpnam** characters. **tmpnam** should be called no more than **TMP_MAX** times.

Example

```
// tmpnam.cpp
#include <stdio.h>                    // for tmpnam,printf
int main()
{
  char tnam[L_tmpnam];
  FILE *tfile;
  printf("A name for a file is %s\n",tmpnam(tnam));
  tfile=fopen(tnam,"w+b");
  printf("A name for another file is %s\n",tmpnam(NULL));
  fclose(tfile);
  remove(tnam);
  return 0;
}
```

ungetc

Example

Purpose To push a character back into an input stream.

Syntax

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Description The **ungetc** function pushes the character *c* back into the input stream pointed to by *stream*. Pushed-back characters can be read by subsequent calls to **fgetc** (say) or **fread** but note that the order will be the reverse to that of pushing. Any number of characters may be pushed back, limited only by the memory available. An intervening call to **rewind**, **fseek** or **fsetpos** erases any pushed-back characters. The corresponding external file is unchanged by a call to **ungetc**. A successful call to **ungetc** clears the end-of-file indicator for the stream.

The file position indicator is decremented for each call of the function and the original value is reinstated after reading or discarding all pushed back characters.

Return value **ungetc** returns the character pushed back, or **EOF** if the operation fails.

Notes If *c* is equal to **EOF** then the operation fails and the stream is unchanged.

See also **fputc**

Example

Example

```
// ungetc.cpp
// Press the space bar to read the file. F1,F2 & F3 will clear
// push-back characters. Any other input character is pushed
// back into the stream.
#include <stdio.h>                // for ungetc, etc.
#include <dbos\conio.h>           // for get_key
int main()
{
  FILE *infile;
  int c;
  short k;
  long pos;
  fpos_t place;
  infile=fopen("test.tmp","r");  // file created by fprintf.cpp
  do
  {
    get_key(k);
    switch(k)
    {
     case 0x0020: c=fgetc(infile); putchar(c); break;// <space>
     case 0x013b: pos=ftell(infile);
                  fseek(infile,pos,SEEK_SET); break; // <F1>
     case 0x013c: fgetpos(infile,&place);
                  fsetpos(infile,&place);break;      // <F2>
     case 0x013d: rewind(infile);        break;      // <F3>
     default:     ungetc(k,infile);
    }
  } while (k != 13);                                 // <enter>
  fclose(infile);
  return 0;
}
```

unlink

Purpose To erase a file.

Syntax

```
#include <stdio.h>
int unlink(const char *filename);
```

Description The **unlink** function is identical to **remove**. See **remove** for further details.

vfprintf

Purpose To write formatted output to a stream.

Syntax

```
#include <stdio.h>
int vfprintf(FILE *stream,
             const char *format,va_list arg);
```

Description **vfprintf** is the same as **vsprintf** except that the output is directed to a stream. See **vsprintf** for further details.

Return value **vfprintf** returns the number of characters transmitted.

See also **vprintf**

vprintf

Purpose To write formatted output to **stdout**.

Syntax

```
#include <stdio.h>
int vprintf(const char *format,va_list arg);
```

Description **vprintf** is the same as **vsprintf** except that the output is directed to **stdout**. See **vsprintf** for further details.

Return value **vprintf** returns the number of characters transmitted.

See also **vfprintf**

vsprintf

Example

Purpose To write formatted output to a string.

Syntax

```
#include <stdio.h>
int vsprintf(char *s,
             const char *format,va_list arg);
```

Description **vsprintf** is the same as **sprintf** except that the variable argument list is replaced by a pointer *arg* to a list of arguments initialised by a call to the macro **va_start** (with possible subsequent calls to **va_arg**, see **<stdarg.h>** for further details).

From a different point of view, **sprintf** can be defined in terms of a call to **va_start** followed by a call to **vsprintf** (see example). See **sprintf** for further details.

Return value **vsprintf** returns the number of characters transmitted excluding the terminating null.

See also **vfprintf**, **vprintf**

Example

```
#include <stdarg.h>                          // for va_start
#include <stdio.h>                           // for vsprintf
int my_sprintf(char *s, const char *format,...)
{  int len;
   va_list args;
   va_start(args,format);
   len=vsprintf(s,format,args);
   return len;
}
```

Functions defined in stdlib.h

abort	exit	rand
abs	fault_malloc_failure	realloc
atexit	free	srand
atof	getenv	strtod
atoi	itoa	strtol
atol	labs	strtoul
bsearch	ldiv	system
calloc	malloc	utoa
div	qsort	

About functions defined in stdlib.h

The **<stdlib.h>** header provides prototypes for functions of general utility. It also includes the following definitions:

```
typedef unsigned int size_t;
typedef struct _div_t { int quot; int rem; } div_t;
typedef struct _ldivt { long quot; long rem; } ldiv_t;
#define NULL 0
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
#define RAND_MAX 2147483647
```

abort

Example

Purpose To force an abnormal termination of the program.

Syntax

```
#include <stdlib.h>
void abort(void);
```

Description **abort** causes the program to terminate immediately without a call to **exit** and its associated **atexit** instructions etc.

Return value **abort** does not return to its caller.

abs

Purpose To compute the absolute value of an integer.

Syntax

```
#include <stdlib.h>
int abs(int j);
```

Return value **abs** returns the absolute value of the integer *j*.

See also **labs**, **fabs**.

atexit

Example

Purpose To enter a function into a list of functions that are to be called on exiting from a program.

Syntax

```
#include <stdlib.h>
int atexit(void (*funct)(void));
```

Description The **atexit** function designates the function pointed to by *funct* as a function to be called (without arguments) upon exiting from a program (see **exit**). A function designated twice will be called twice and so on.

Return value **atexit** returns zero for successful designation, non zero for failure.

See also **abort**

atof

Purpose To convert a string to a **double**.

Syntax

```
#include <stdlib.h>
double atof(const char *nptr);
```

Description Converts the null terminated string pointed to by *nptr* into a double value.

atof(*nptr*) is equivalent to strtod(*nptr*,NULL).   See **strtod** for further details.

Return value **atof** returns the converted value.

Notes See notes for **strtod**.

See also **atoi**, **atol**, **scanf**.

atoi

Purpose To convert a string to an **int**.

Syntax

```
#include <stdlib.h>
int atoi(const char *nptr);
```

Description Converts decimal value in the null terminated string pointed to by *nptr* into an int value.

atoi(*nptr*) is equivalent to (int)strtol(*nptr*,NULL,10). See **strtol** for further details.

Return value **atoi** returns the converted value.

Notes See notes for **strtol**.

See also **atof**, **atol**, **scanf**, **strtod**.

atol

Purpose To convert a string to a **long**.

Syntax

```
#include <stdlib.h>
long atol(const char *nptr);
```

Description Converts the decimal value in the null terminated string pointed to by *nptr* into a **long** value.

atol(*nptr*) is exactly equivalent to strtol(*nptr*,NULL,10). See **strtol** for further details.

Return value **atol** returns the converted value.

Notes See notes for **strtol**.

See also **atof**, **atoi**, **scanf**, **strtod**, **strtoul**.

bsearch

Example

Purpose To search an array for a given match.

Syntax

```
#include <stdlib.h>
void *bsearch(const void *key,
  const void *base, size_t nmemb,
  size_t size,
  int (*compare)(const void *,const void *));
```

Description The **bsearch** function searches an ordered array of *nmemb* objects of size *size*. *base* points to the first element in the array and *key* provides the object for which a match is required. *compare* points to a comparison function which is provided by the user and which takes two arguments that point to the key object and an array element respectively. This function should be coded to return a negative integer when the key object is less than the array element, zero when a match occurs, and a positive integer when the key object is greater than the array element.

Return value **bsearch** returns a pointer to a matching element, or a **NULL** pointer if no match was found. If there is more than one match then **bsearch** may not necessarily point to the first occurrence.

See also **lfind**, **lsearch**, **qsort**.

Example

```
#include <stdio.h>                  // for printf,gets
#include <string.h>                 // for strcmpi
#include <stdlib.h>                 // for bsearch,atoi
#define PTR int(*)(const void*,const void*)
int num_order(const int  *k, const int  *e)
{ return (*k-*e);       }
int lex_order(const char *k, const char **e)
{ return strcmpi(k,*e); }
int main()
{
  char str[10];
  int i,a[]={1,3,5,16,17,26,32,54,67,83,91,91};// ordered array
  int nmemb=sizeof(a)/sizeof(int);
  for(i=0; i<nmemb; i++) printf("%d ",a[i]);
  while(1)
  {
    printf("\nkey:"); gets(str); int key=atoi(str);
    if(key==0) break;
    int *ptr=(int*)bsearch(&key,a,nmemb,sizeof(int),
                         (PTR)num_order);
    if (ptr) printf("%d\n",*ptr);
    else     printf("Not found\n");
  };
  char *b[]={"Andrew","Elaine","Gillian","Helen","Jonathan",
             "Paul"};
  for(i=0; i<6; i++) printf("%s ",b[i]);
  while(1)
  {
    printf("\nkey:"); gets(str);
    if(str[0]=='\0') break;
    char **ptr=(char**)bsearch(str,b,6,sizeof(char *),
                             (PTR)lex_order);
    if (ptr) printf("%s\n",*ptr);
    else     printf("Not found\n");
  };
  return 0;
}
```

calloc

Example

Purpose To allocate and clear a block of memory dynamically.

Syntax

```
#include <stdlib.h>
void *calloc(size_t n, size_t s);
```

Description **calloc** allocates a block of memory of (*n* * *s)* bytes for *n* objects of size *s*. Thus if the objects are of type *t* then *s*=**sizeof**(*t*). Each byte is cleared to 0. If *n*=0 or *s*=0 then a unique pointer is returned.

Return value **calloc** returns a pointer to the new block.

Notes If there is insufficient memory available then **calloc** returns a **NULL** pointer.

See also **malloc**, **realloc**, **free**, **fault_malloc_failure**.

Example

Example

```
#include <stdio.h>                    // for printf,gets
#include <stdlib.h>                   // for calloc,free
#include <string.h>                   // for strlen,strcpy
int main()
{ char *s1,s2[80];
  do
  {
    printf("string:");  gets(s2);
    s1=(char *)calloc(strlen(s2)+1,sizeof(char));
    strcpy(s1,s2);
    printf("memory:%s\n",s1);
    free(s1);
  } while (s2[0]);
  return 0;
}
```

div

Example

Purpose To compute the quotient and remainder after dividing two integers.

Syntax

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Description The **div** function computes the quotient and remainder of the fraction *numer/denom*. If the result is returned to *r* (of type **div_t**) then *r.quot* is the integer value obtained by division and truncation towards zero. *r.rem* is the integer remainder defined as (*numer* - *r.quot*denom*). The result is identical to the coding:

```
     r.quot  =  numer/denom;
     r.rem   =  numer%denom;
```

which is simpler.

Return value **div** returns a structure of type **div_t** consisting of the quotient and remainder.

Notes If *denom* is zero then *r.quot* is set to **HUGE_VAL**, *r.rem* is set to zero and **errno** is set to **ERANGE** (see **error.h**).

See also **ldiv**.

Example

Example

```
#include <stdio.h>                    // for printf,gets
#include <stdlib.h>                   // for div,atoi
int main()
{  char str[80];
  int numer,denom;
  div_t r;
  do
  {
    printf("numerator  :");  gets(str); numer=atoi(str);
    printf("denominator:");  gets(str); denom=atoi(str);
    r=div(numer,denom);
    printf("quotient   :%d\n",r.quot);
    printf("remainder  :%d\n",r.rem);
  } while (str[0]) ;
  return 0;
}
```

exit

Example

Purpose To cause a program to terminate normally.

Syntax

```
#include <stdlib.h>
void exit(int status);
```

Description The **exit** function calls upon any functions that have been listed by the **atexit** function in reverse order of their designation. The **exit** function then terminates the program by completing any tasks which are normally unfinished such as the flushing of any unwritten buffers and the closure of any open streams. *status*=**EXIT_SUCCESS** is used to represent successful termination, whilst *status* = **EXIT_FAILURE** denotes unsuccessful termination. A call of return *n* from the **main** segment of a program has the effect of **exit**(*n*).

Return value **exit** does not return to its caller.

See also **abort**.

Example

Example

```
#include <stdio.h>                  // for printf,gets
#include <stdlib.h>                 // for abort,exit,atexit
void atexit_do(void)
{printf("\natexit_do called on exit....\n");}
int main()
{ char str[10];
  do
  { printf("\nchar:");  gets(str);
    if (str[0]=='@')  atexit(atexit_do);
    else
    { printf("This was in the output buffer...");
      switch (str[0])
      { case 'a': abort();
        case 's': exit(EXIT_SUCCESS);
        case 'f': exit(EXIT_FAILURE);
        case '0': return 0;
        case '1': return 1;
      } //switch
    }   //if
  } while (*str);
  return 0;
}
```

fault_malloc_failure

Example

Purpose To provide a warning message when **malloc** fails.

Syntax

```
#include <stdlib.h>
int fault_malloc_failure(int flag);
```

Description **malloc**, **calloc** and **realloc** each return a **NULL** pointer when there is insufficent memory available for the call.

If this failure is not trapped explicitly then **fault_malloc_failure** may be used to force a run time error. *flag* is set to a non-zero value to enable this behaviour, or to zero to return to the default state.

Return value **fault_malloc_failure** returns zero when the trap is enabled, non-zero when disabled.

See also **free**.

Example

```
#include <stdlib.h>              // for fault_malloc_failure,free
int main()
{
  fault_malloc_failure(1);
  char *s=malloc(1000000000);
  fault_malloc_failure(0);
  free(s);
  return 0;
}
```

free

Example

Purpose To free a block of memory that has been allocated by **calloc**, **malloc**, or **realloc**.

Syntax

```
#include <stdlib.h>
void free(void *ptr);
```

Description The **free** function makes previously allocated memory available for reallocation.

ptr must be a value that has been returned by **calloc**, **malloc** or **realloc**. The system keeps a record of such pointers together with the size of each associated block.

free(NULL) has no effect.

Return value None.

Notes If a record of a pointer is not found (because it has not been set by **malloc**, **calloc** or **realloc**, or it has already been removed by **free** or **realloc**) then the program terminates with a run time failure message.

getenv

Example

Purpose To read an entry in the DOS “environment” list.

Syntax

```
#include <stdlib.h>
char *getenv(const char *name);
```

Description **getenv** searches a list in the so-called DOS environment region of memory for items such as “COMSPEC”, “PATH”, “PROMPT”, etc., which have default values or values assigned using the DOS SET command. *name* is a pointer to such a string which should not contain any lower case letters or the equals sign (=).

Return value **getenv** returns a pointer to the value associated with *name* or a **NULL** pointer if no such string is found.

See also **system**.

Example

```
#include <stdio.h>                  // for printf,gets
#include <stdlib.h>                 // for getenv
int main()
{  char *ptr,str[80];
  do
  {
    printf("name       :");  gets(str);
    // enter "PATH","COMSPEC","PROMPT",etc from DOS SET command
    ptr=getenv(str);
    if (ptr) printf("environment:%s\n",ptr);
    else     printf("ERROR      :environment not found\n");
  } while (*str);
  return 0;
}
```

itoa

Example

Purpose To convert an integer to a string.

Syntax

```
#include <stdlib.h>
char *itoa(int val, char *str, int rad);
```

Description **itoa** converts *val* into a string using the radix *rad* (in the range 2..36) and puts the result into the string pointed to by *str*. If *val* is negative then **itoa** sets the radix to 10 and the string begins with a minus sign.

Return value **itoa** returns a pointer to *str*.

Notes The string pointed to by *str* should be long enough to accommodate the result which may be up to 40 bytes long. A value of *rad* outside of the stipulated range will cause a fatal error.

See also **utoa**, **sprintf**.

Example

```
#include <stdio.h>                    // for printf,gets
#include <stdlib.h>                   // for itoa,atoi
int main()
{ char str[40];
  int val,rad;
  while(1)
  { printf("value  :");  gets(str); val=atoi(str);
    if(val==0) break;
    printf("radix  :");  gets(str); rad=atoi(str);
    printf("return :%s\n",itoa(val,str,rad));
  }
  return 0;
}
```

labs

Purpose To compute the absolute value of an integer.

Syntax

```
#include <stdlib.h>
long labs(long j);
```

Return value **labs** returns the absolute value of the integer *j*.

See also **abs**, **fabs**.

ldiv

Purpose To compute the quotient and remainder after dividing two integers.

Syntax

```
#include <stdlib.h>
ldiv_t ldiv(long numer, long denom);
```

Description The **ldiv** function is equivalent to **div**. See **div** for further details.

malloc

Example

Purpose To allocate a block of memory dynamically.

Syntax

```
#include <stdlib.h>
void *malloc(size_t s);
```

Description **malloc** allocates a block of memory of *s* bytes.
If *s*=0 then a unique pointer is returned.

Return value **malloc** returns a pointer to the new block.

Notes If there is insufficient memory available then **malloc** returns a **NULL** pointer.

See also **calloc**, **realloc**, **free**, **fault_malloc_failure**.

Example

```
#include <stdio.h>                    // for printf,gets
#include <stdlib.h>                   // for malloc,free
#include <string.h>                   // for strlen,strcpy
int main()
{ char *s1,s2[80];
  do
  { printf("string:");  gets(s2);
    s1=(char *)malloc(sizeof(char)*(strlen(s2)+1));
    strcpy(s1,s2);
    printf("memory:%s\n",s1);
    free(s1);
  } while (s2[0]);
  return 0;
}
```

qsort

Example

Purpose To sort an array into a given order.

Syntax

```
#include <stdlib.h>
void qsort(const void *base,
 size_t nmemb, size_t size,
 int (*compare)(const void *, const void *));
```

Description **qsort** is similar to **bsearch** but **qsort** reorders the given array according to the function pointed to by *compare*. If two elements compare as equal (i.e. **compare* returns zero) then their order in the sorted array is unspecified. See **bsearch** for details of the properties required of the function pointed to by *compare*.

Example

```
// qsort.cpp
#include <stdio.h>                  // for printf
#include <string.h>                 // for strcmpi
#include <stdlib.h>                 // for qsort
#define PTR int(*)(const void*,const void*)
int num_order(const int *e1,   const int  *e2)
{ return (*e1-*e2);        }
int lex_order(const char **e1, const char **e2)
{ return strcmpi(*e1,*e2); }
int main()
{
  int i,a[]={56,63,13,1,92,76,0,34,23};
  char *b[]={"Jonathan","Gillian","Helen","Andrew","Elaine",
             "Paul"};
  int nmemb=sizeof(a)/sizeof(int);
  qsort(a,nmemb,sizeof(int),(PTR)num_order);
  for(i=0; i<nmemb; i++) printf("%d ",a[i]);
  printf("\n");
  nmemb=sizeof(b)/sizeof(char *);
  qsort(b,nmemb,sizeof(char *),(PTR)lex_order);
  for(i=0; i<nmemb; i++) printf("%s ",b[i]);
  return 0;
}
```

rand

Example

Purpose To return a pseudo-random integer value.

Syntax

```
#include <stdlib.h>
int rand(void);
```

Description Note that on re-running the program, the same sequence is generated using successive calls of **rand** unless a varying "seed" is provided via **srand** or **date_time_seed** .

rand() % n   gives a result in the range 0..(n-1).

Return value **rand** returns a pseudo-random integer value in the range 0..**RAND_MAX**.

realloc

Example

Purpose To change the size of a block created by **malloc**, **calloc** or **realloc**.

Syntax

```
#include <stdlib.h>
void *realloc(void *ptr,size_t s);
```

Description **realloc** changes the size of the block pointed to by *ptr*. If *ptr* is not **NULL** then it must point to a block which has been created by **malloc**, **calloc** or **realloc**. A list of such pointers and the size of the associated blocks is kept by the system. The contents of the new block will be the same as the old up to the smaller of the old and the new sizes. If the size increases, the extra memory will not be cleared. If the size is zero, then **realloc** behaves exactly like the function **free**. If *ptr* is **NULL** then **realloc** behaves exactly like **malloc**.

Return value **realloc** returns a pointer to the new block or a **NULL** pointer if **realloc** is set to behave like the function **free**.

Notes If there is insufficient memory available then **realloc** returns a **NULL** pointer.

If a record of a pointer is not found (because it has not been set by **malloc**,**calloc** or **realloc**, or it has already been removed by **free** or **realloc**) then the program terminates with a run time failure message.

Example

Example

```
#include <stdio.h>                  // for printf,gets
#include <stdlib.h>                 // for realloc
#include <string.h>                 // for strlen,strcpy
int main()
{
  char s2[80];
  char *s1=NULL;
  size_t len=1;
  do
  {
    printf("string:");  gets(s2);
    len+=strlen(s2);
    s1=realloc(s1,len);
    strcat(s1,s2);
    printf("memory:%s\n",s1);
  } while (s2[0]);
  realloc(s1,0);
  return 0;
}
```

srand

Example

Purpose To enter a new “seed” for the pseudo-random integer generator **rand**.

Syntax

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Description The argument of **srand** provides a seed for the function **rand**. That is, it sets an initial value which is used to start a process for generating a pseudo-random sequence. The default value of this *seed* is 1. Some means should be used to vary the seed value otherwise re-running the program will provide the same sequence of pseudo-random values.

See also **date_time_seed**

Example

Example

```
#include <stdio.h>                    // for printf,gets
#include <stdlib.h>                   // for rand,srand
#include <time.h>                     // time
int main()
{
  char str[10];
  union  { unsigned int i[2]; time_t d;} t;
  t.d=time(NULL);
  srand(t.i[0]);
  do
  { printf("rand :%d",rand() % 1000); gets(str); }
  while (!*str);
  return 0;
}
```

strtod

Example

Purpose	To convert a string to a **double**.

Syntax

```
#include <stdlib.h>
double strtod(const char *nptr,
              char **endptr);
```

Description

Converts the null terminated string pointed to by *nptr* into a **double** value.

The string is assumed to contain in order:

- an optional *initial* part consisting of:

white-space characters *c*	(isspace(*c*)>0)

- a *mandatory* part containing at least one digit and containing in order:

an optional sign	(+ or -)
optional digits *d*	(isdigit(*d*) > 0)
an optional decimal point	(.)
optional digits *d*	(0..9 i.e. isdigit(*d*) > 0)

- an optional *exponent* part containing in order:

the letter e	(e or E)
an optional sign	(+ or -)
one or more digits *d*	(isdigit(*d*) > 0)

- an optional *final* part consisting of *delimiters* c

  (isdigit(*c*) = = 0).

Return value

strtod returns the **double** value if conversion has proved successful or a zero if not.

If *endptr* is not **NULL** then **strtod** sets **endptr* to point to the *final* part of the string. This is useful for error detection.

Notes

If the conversion leads to an overflow condition then **strtod** returns either plus or minus **HUGE_VAL** (depending on the sign of the value) and **errno** is set to **ERANGE**.

If the conversion leads to an underflow condition then **strtod** returns a zero and **errno** is set to **ERANGE** (see **<errno.h>**).

See also

atof, **scanf**.

Examples

+2.456e-6	(OK)
-.12e4	(OK)
.4	(OK)
1.0 e4	(gives 1.0)

2e	(gives 2.0)
e2	(gives 0)

Example

```
#include <stdio.h>                    // for printf,gets,perror
#include <stdlib.h>                   // for strtod
#include <errno.h>                    // for errno,EZERO
int main()
{
  char str[80],*endptr;
  do
  {
    printf("string :");   gets(str);
    printf("double :%g\n",strtod(str,&endptr));
    printf("endpart:%s\n",endptr);
    if(errno) { perror(NULL); errno=EZERO; }
  } while (*str);
  return 0;
}
```

strtol

Example

Purpose To convert a string to a **long**.

Syntax

```
#include <stdlib.h>
long strtol(const char *nptr, char **endptr,
            int radix);
```

Description Converts the null terminated string pointed to by *nptr* into a **long** value using the given *radix*.

The string is assumed to contain in order:

- an optional *initial* part consisting of white-space characters *c* (isspace(*c*) > 0)
- a *mandatory* part containing:

  an optional sign (+ or -)
  and one or more *numeric* characters.
- an optional *final* part consisting of non-*numeric* characters.

For this purpose a *numeric* character is any character that is permitted for the given *radix*.

e.g.	*radix*=10;	0..9
	radix=8;	0..7
	radix=16;	0..9, A..F, and a..f
	radix=36;	0..9, A..Z, and a..z.

If *radix*=0 then **strtol** will assign the radix to one of 8, 10 and 16 depending on the initial characters of the *mandatory* part. i.e. a leading 0x or 0X gives a radix of 16, otherwise a leading 0 gives a radix of 8, otherwise the radix is 10.

Return value **strtol** returns the **long** value if conversion has proved successful or a zero if not.

If *endptr* is not **NULL** then **strtol** sets **endptr* to point to the *final* part of the string. This is useful for error detection.

Notes If the conversion leads to an overflow condition then **strtol** returns either **LONG_MAX** or **LONG_MIN** (depending on the sign of the value) and **errno** is set to **ERANGE** (see **<errno.h>** and **<limits.h>**).

If radix>36 or radix<2 (radix0) then **strtol** returns zero.

See also **atoi**, **atol**, **strtoul**, **scanf**.

Example If *radix* = 0

238ag	gives decimal	238

0238ag	gives octal	23
0x238ag	gives hexadecimal	238A

Example

Example

```
#include <stdio.h>                  // for printf,gets,perror
#include <stdlib.h>                 // for strtol
#include <errno.h>                  // for errno,EZERO
int main()
{
  char str[80],*endptr;
  int radix;
  do
  {
    printf("radix  :");  gets(str);
    radix=strtol(str,NULL,10);
    printf("string :");  gets(str);
    printf("long   :%ld\n",strtol(str,&endptr,radix));
  //printf("ulong  :%lu\n",strtoul(str,&endptr,radix));
    printf("endpart:%s\n",endptr);
    if(errno){ perror(NULL); errno=EZERO; }
  } while (*str);
  return 0;
}
```

strtoul

Example

Purpose To convert a string to an **unsigned long**.

Syntax

```
#include <stdlib.h>
unsigned long strtoul(const char *nptr,
                      char **endptr, int radix);
```

Description Converts the null terminated string pointed to by *nptr* into an **unsigned long** value using the given *radix*.

strtoul operates in the same manner as **strtol** except for the following error condition and the fact that a leading minus sign is not permitted.

Return value **strtoul** returns the **unsigned long** value if conversion has proved successful or a zero if not.

If *endptr* is not **NULL** then **stroul** sets **endptr* to point to the “final” part of the string.

Notes If the conversion leads to an overflow condition then **strtoul** returns **ULONG_MAX** and **errno** is set to **ERANGE** (see **<error.h>** and **<limits.h>**).

See also **atoi**, **atol**, **scanf**.

system

Example

Purpose To issue a DOS command.

Syntax

```
#include <stdlib.h>
int system(const char *com);
```

Description The **system** function provides a means of executing a DOS command from within a C program. *com* points to a string containing the DOS command. The current DOS PATH environment entry is used to access files which are not in the default directory. If *com* is a **NULL** pointer then the DOS EXIT command is executed.

Commands are issued using the MS-DOS command processor COMMAND.COM with the amount of memory specified in the E/ parameter of this command. The default value of 160 can be increased by using the COMSPACE command which is to be found in the DBOS.DIR directory. The general form of this command is

COMSPACE D'<number>'

where <number> specifies the number of bytes (in decimal) to be reserved (e.g. `COMSPACE D'1024'` reserves 1K). As a guideline, use the size of the real mode .EXE or .COM file plus 10%. It is not possible to issue a command which uses DBOS (e.g. another Salford C++ program).

Return value **system** returns zero for success and a non-zero value for failure.

See also **getenv**.

Example

Example

```
#include <stdio.h>                    // for printf,gets
#include <stdlib.h>                   // for system
int main()
{
  char str[80];
  system("ver");
  do
  {
    printf("\n\ncommand>"); gets(str);
    system(str);
  } while(*str);
  return 0;
}
```

utoa

Purpose To convert an unsigned integer to a string.

Syntax

```
#include <stdlib.h>
char *utoa(unsigned int val, char *str,
                                int rad);
```

Description **utoa** converts the **unsigned int** *val* into a string using the radix *rad* and puts the result into the string pointed to by *str*. It is identical to **itoa** except that **utoa** does not include the option of a leading minus sign (see **itoa** for further details).

Functions defined in string.h

- index
- memchr
- memcmp
- memcpy
- memmove
- memset
- rindex
- stpcpy
- strcat
- strchr
- strcmp
- strcmpi
- strcmpl
- strcoll
- strcpy
- strcspn
- strdup
- strend
- strerror
- stricmp
- strlen
- strlwr
- strncat
- strncmp
- strncpy
- strpbrk
- strrchr
- strspn
- strstr
- strtok
- strupr
- strxfrm

About functions defined in string.h

The **<string.h>** header gives prototypes for a number of functions which are useful for processing arrays of characters or objects which are treated as characters. It includes the following definitions:

```
typedef unsigned int size_t;
#define NULL 0
```

index

Purpose To locate the position of the first occurrence of a given character in a null terminated string.

Syntax

```
#include <string.h>
char *index(const char *s, int c);
```

Description **index** is an alternative name for **strchr**. See **strchr** for further details.

memchr

Example

Purpose To locate the position of the first occurrence of a given character in an array.

Syntax

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Return value **memchr** returns a pointer to the first occurrence of *c* in the array of size *n* pointed to by *s*.

A **NULL** return means that the character has not been found.

Notes A **NULL** is returned when *n*=0.

It is important to ensure that *n* is not negative.

See also **strchr**, **strrchr**.

Example

```
#include <stdio.h>                         //for printf,gets
#include <string.h>                        //for memchr
#include <stdlib.h>                        //for atoi
int main()
{ size_t n;
  char *ptr,c,str[10];
  char s[]="abcdefghijklmnopqrstuvwxyz";
  do
  { printf("Source   :%s\n",s);
    printf("n        :"); gets(str); n=atoi(str);
    printf("c        :"); gets(str); c=str[0];
    ptr=(char*)memchr(s,c,n);
    if (ptr) printf("points to:%c\n",*ptr);
    else     printf("not found\n");
  } while (n);
  return 0;
}
```

memcmp

Example

Purpose To compare two arrays of *n* characters.

Syntax

```
#include <string.h>
int memcmp(const void *s1, const void *s2,
size_t n);
```

Description Compares up to *n* characters in corresponding positions in the arrays pointed to by *s1* and *s2* until elements are found to differ.

Return value Either **memcmp** returns a positive or negative value depending on whether *s1*[i]-*s2*[i] is positive or negative, for the index i where elements first differ;

or **memcmp** returns zero if the first *n* elements of the two arrays are identical.

memcmp returns a positive value when *n*=0.

Notes *n* should not be greater than the length of the shorter array.

See also **strcmp**, **strncmp**, **strcmpl**.

Example

```
#include <stdio.h>                         //for printf,gets
#include <string.h>                        //for memcmp
#include <stdlib.h>                        //for atoi
int main()
{ size_t n;
  char str[10];
  char s1[]="abcdef";
  char s2[]="abcdfe";
  do
  { printf("Source 1       :%s\n",s1);
    printf("Source 2       :%s\n",s2);
    printf("n              :"); gets(str); n=atoi(str);
    printf("memcmp(s1,s2,n):%d\n",memcmp(s1,s2,n));
    printf("memcmp(s2,s1,n):%d\n",memcmp(s2,s1,n));
  } while (n);
  return 0;
}
```

memcpy

Example

Purpose To copy an array of *n* characters.

Syntax

```
#include <string.h>
void *memcpy (void *dest, const void *srce,
size_t n);
```

Description Copies *n* characters from the object pointed to by *srce* to the object pointed to by *dest*.

Return value **memcpy** returns the value of *dest*.

Notes If the two arrays of characters overlap then the effect of **memcpy** should be regarded as unpredictable and the alternative **memmove** should be used.

If *n* is greater than the length of *dest* or *srce* then the effect of **memcpy** should be regarded as unpredictable. It is important to ensure that *n* is not less than zero.

See also **strcpy**, **strncpy**, **strdup**.

Example

```
#include <stdio.h>                         //for printf,gets
#include <string.h>                        //for memcpy,memmove
#include <stdlib.h>                        //for atoi
int main()
{
  unsigned int n,s,d;
  do
  {
    char mem1[]="abcdefghijklmnopqrstuvwxyz";
    char mem2[]="abcdefghijklmnopqrstuvwxyz";
    char str[10];
    printf("Copy from letter    :");  gets(str); s=str[0]-’a’;
    printf("Copy to letter      :");  gets(str); d=str[0]-’a’;
    printf("Number of characters:");  gets(str); n=atoi(str);
                                            printf("%s\n",mem1);
    memcpy (&mem1[d],&mem1[s],n);    printf("%s\n",mem1);
    memmove(&mem2[d],&mem2[s],n);    printf("%s\n",mem2);
  } while (n);
  return 0;
}
```

memmove

Example

Purpose To copy an array of *n* characters.

Syntax

```
#include <string.h>
void *memmove(void *dest, const void *srce,
size_t n);
```

Description Copies *n* characters from the object pointed to by *srce* to the object pointed to by *dest*. **memmove** first copies the source into a temporary array of *n* characters that does not overlap either of the objects involved.

Return value **memmove** returns the value of *dest*.

Notes If *n* is greater than the length of *dest* or *srce* then the effect of **memmove** should be regarded as unpredictable. It is important to ensure that *n* is not less than zero.

See also **memcpy**, **strcpy**, **strdup**.

memset

Example

Purpose To set an array of *n* characters to a particular value.

Syntax

```
#include <string.h>
void *memset (void *s, int c, size_t n);
```

Description Copies the (**unsigned char**) value of *c* into the *n* characters of the array pointed to by *s*.

Return value **memset** returns the value of *s*.

Notes It is important to ensure that *n* is not negative.

See also **memcpy**.

Example

```
#include <stdio.h>                         // for printf,gets
#include <string.h>                        // for memset
#include <stdlib.h>                        // for atoi
int main()
{
  char s[40],str[10],c;
  size_t n;
  memset(s,'*',40);
  printf("c     :"); gets(str); c=str[0];
  printf("n     :"); gets(str); n=atoi(str);
  printf("memset:%.40s\n",memset(s,c,n));
  return 0;
}
```

rindex

Purpose To locate the position of the last occurrence of a given character in a null terminated string.

Syntax

```
#include <string.h>
void *rindex(const char *s, int c);
```

Description **rindex** is an alternative name for **strrchr**. See **strrchr** for further details.

stpcpy

Example

Purpose To copy one null terminated string into another.

Syntax

```
#include <string.h>
char* stpcpy(char *dest, const char *src);
```

Description **stpcpy** is identical to **strcpy** except for the return value.   See **strcpy** for further details.

Return value **stpcpy** returns a pointer to the null terminator of *dest*, i.e. *dest* + strlen(*src*).

Example

```
#include <stdio.h>                            //for printf,gets
#include <string.h>                           //for stpcpy
int main()
{
  char srce[80],dest[80],*ptr;
  printf("source       :");   gets(srce);
  ptr=stpcpy(dest,srce);
  printf("destination  :%s\n",dest);
  printf("Pointer after:%c\n",*(ptr-1));
  return 0;
}
```

strcat

Example

Purpose To append one null terminated string to the end of another.

Syntax

```
#include <string.h>
char *strcat(char *dest, const char *srce);
```

Description Appends a copy of the string pointed to by *srce* (including its null terminator) on to the end of the string pointed to by *dest* (the initial character of *srce* replaces the original terminating null of *dest*).

A source string of zero length is quite acceptable.

Return value **strcat** returns the value of *dest*.

Notes If the two strings overlap then the effect of **strcat** should be regarded as unpredictable.

If strlen(*srce*) + strlen(*dest*) + 1 is greater than the length of the array *dest* then **strcat** may contaminate other areas of memory.

See also **strncat**

Example

```
#include <stdio.h>                                  //for printf,gets
#include <string.h>                                 //for strcat
int main()
{
   char srce[80],dest[80];
   dest[0]='\0';
   do
   {
     printf("source     :");   gets(srce);
     printf("destination:%s\n",strcat(dest,srce));
   } while (srce[0]);
   return 0;
}
```

strchr

Example

Purpose To locate the position of the first occurrence of a given character in a null terminated string.

Syntax

```
#include <string.h>
char *strchr(const char *s, int c);
```

Return value *strchr* returns a pointer to the first occurrence of *c* in the string pointed to by *s* or a **NULL** pointer if the character has not been found.   If *c*='\0' then **strchr** returns a pointer to the null terminator at the end of the string.

See also **memchr**, **strrchr**, **strstr**.

Example

```
#include <stdio.h>                          //for printf,gets
#include <string.h>                         //for strchr
int main()
{
  char *ptr,c,str[10];
  char s[]="aaabbcdddeefggghhijjjkkl";
  do
  { printf("Source   :%s\n",s);
    printf("c        :"); gets(str); c=str[0];
    ptr=strchr(s,c);
    if (ptr) printf("points to position:%d\n",ptr-s+1);
    else     printf("not found\n");
  } while (ptr);
  return 0;
}
```

strcmp

Example

Purpose To compare two null terminated strings.

Syntax

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Description Compares characters in corresponding positions in the strings pointed to by *s1* and *s2* until elements are found to differ or the end of one of the strings is reached.

Return value Either **strcmp** returns a positive or negative value depending on whether *s1*[i]-*s2*[i] is positive or negative for the index i where elements first differ; or zero if the two strings are identical.

(e.g. negative if *s1* is lexicographically less than *s2*.)

See also **strcmpl**, **strncmp**, **memcmp**.

Example strcmp("dog","cat") is positive.
strcmp("dog","dogs") is negative.
strcmp("","a") is negative.
strcmp("","") is zero.

Example

```
#include  <stdio.h>                       //for printf,gets
#include  <string.h>                      //for strcmp
int main()
{ char s1[80],s2[80];
   do
   { printf("Source 1               :",s1); gets(s1);
     printf("Source 2               :",s2); gets(s2);
     printf("strcmp(s1,s2):%d\n",strcmp(s1,s2));
     printf("strcmp(s2,s1):%d\n",strcmp(s2,s1));
   } while (s1[0]);
   return 0;
}
```

strcmpi

Purpose To compare two null terminated strings without regard to the case of alphabetic characters.

Syntax

```
#include <string.h>
int strcmpi(const char *s1, const char *s2);
```

Description **strcmpi** is an alternative name for **strcmpl**. See **strcmpl** for further details.

strcmpl

Example

Purpose To compare two null terminated strings without regard to the case of alphabetic characters.

Syntax

```
#include <string.h>
int strcmpl(const char *s1, const char *s2);
```

Description Compares characters in corresponding positions in the strings pointed to by *s1* and *s2* until elements are found to differ or the end of one of the strings is reached.

The comparison is insensitive to case, so that 'a' matches 'A' etc.

Return value Either **strcmpl** returns a positive or negative value depending on whether *s1*[i]-*s2*[i] is positive or negative for the index i where elements first differ; or zero if the two strings are equivalent. (e.g. negative if *s1* is lexicographically before *s2*.)

See also **strcmp**, **memcmp**.

strcoll

Purpose To compare two null terminated strings in relation to the current locale.

Syntax

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

Description The current implementation supports only the “C” locale (see **<locale.h>**) in which **strcoll** is identical to **strcmp**.

Return value See **strcmp**.

strcpy

Example

Purpose To copy one null terminated string into another.

Syntax

```
#include <string.h>
char *strcpy (char *dest, const char *srce);
```

Description Copies the string pointed to by *srce* into that pointed to by *dest* up to and including the terminating null.

Return value **strcpy** returns the value of *dest*.

Notes If the two strings overlap then the effect of **strcpy** should be regarded as unpredictable.

Care should be taken to ensure that the destination array is long enough to accommodate the source string.

See also **memcpy**, **stpcpy**, **strdup**, **strncpy**.

Example

```
#include <stdio.h>                                   //for printf,gets
#include <string.h>                                  //for strcpy
int main()
{  char srce[80],dest[80];
   printf("source     :");   gets(srce);
   printf("destination:%s\n",strcpy(dest,srce));
   return 0;
}
```

strcspn

Example

Purpose To find the length of the initial part of a string which is made up entirely from characters that do not appear in another string.

Syntax

```
#include <string.h>
size_t strcspn(const char *s1,
               const char *s2);
```

Description **strcspn** returns the length of the initial segment of string *s1* that consists entirely of characters not from string *s2*. That is, **strcspn** tests if *s1*[n] does not appear in the string *s2* for n=0,1,2.., until *s1*[n] is found in *s2* or the end of *s1* is reached.

strcspn is equivalent to the code:

```
  size_t len=0;
  while (*sl && !strchr(s2,*s1)) {s1++;len++}
  return len;
```

Return value **strcspn** returns the length of the initial part of *s1*.

See also **strpbrk**, **strspn**.

Example

Example

```
#include <stdio.h>                         //for printf,gets
#include <string.h>                        //for strcspn
int main()
{
  char s1[]="abcdefghijklmnopqrstuvwxyz";
  char s2[80];
  size_t n;
  do
  {
    printf("s1       :%s\n",s1);
    printf("s2       :");  gets(s2);
    n=strcspn(s1,s2);
    printf("strcspn   :%d\n",n);
  } while (n);
  return 0;
}
```

strdup

Purpose To allocate memory and copy a string to it.

Syntax

```
#include <string.h>
char* strdup(const char *str);
```

Description **strdup** duplicates a string by calling **malloc** and by copying the string pointed to by *str* to the newly allocated memory. The user is responsible for freeing the memory when it is no longer needed.

Return value **strdup** returns a pointer to the duplicated string or a **NULL** pointer if space could not be allocated.

See also **strcpy**, **memcpy**, **stpcpy**.

strend

Example

Purpose To obtain a pointer to the null terminator of a string.

Syntax

```
#include <string.h>
char *strend(char *s);
```

Return value **strend** returns a pointer to the null terminator.

See also **strchr**

Example

Example

```
#include <stdio.h>                              //for printf,gets
#include <string.h>                             //for strend
int main()
{
   char *ptr,str[80];
   printf("source             :");  gets(str);
   ptr=strend(str);
   printf("length             :%d\n",ptr-str);
   printf("strend points after:%c\n",*(ptr-1));
   return 0;
}
```

strerror

Example

Purpose To obtain a message corresponding to a given error number.

Syntax

```
#include <string.h>
char *strerror (int errnum);
```

Return value **strerror** returns a pointer to a string which describes the implementation dependent error condition that corresponds to *errnum*. The message is inserted into a static buffer which is overwritten on each call of **strerror**.

If *errnum* is out of range then the message “unknown error” appears.

See also **perror**

Example

```
#include <stdio.h>                                  // for printf
#include <string.h>                                 // for strerror
int main()
{
  int i;
  for (i=-1;i<100;printf("%d %s\n",i++,strerror(i)));
  return 0;
}
```

stricmp

Purpose To compare two null terminated strings without regard to the case of alphabetic characters.

Syntax

```
#include <string.h>
int stricmp(const char *s1, const char *s2);
```

Description **stricmp** is an alternative name for **strcmpl**. See **strcmpl** for further details.

strlen

Example

Purpose To obtain the length of a null terminated string.

Syntax

```
#include <string.h>
size_t strlen(const char *s);
```

Return value **strlen** returns the number of characters that come before the terminating null of the string pointed to by *s*.

Example

```
#include <stdio.h>                           //for printf,gets
#include <string.h>                          //for strlen
int main()
{
   char str[80];
   do
   {
     printf("string:");   gets(str);
     printf("length:%d\n",strlen(str));
   } while (str[0]);
   return 0;
}
```

strlwr

Example

Purpose To change any upper case letters in a string to lower case.

Syntax

```
#include <string.h>
char *strlwr(char *s);
```

Description **strlwr** changes any letters in the range A..Z to corresponding lower case letters a..z leaving other characters unchanged.

Return value **strlwr** returns a pointer to *s*.

See also **tolower**

Example

Example

```
#include <stdio.h>                         //for printf,gets
#include <string.h>                        //for strlwr
int main()
{
   char str[80];
   do
   {
     printf("string:");   gets(str);
     printf("strlwr:%s\n",strlwr(str));
   } while (str[0]);
   return 0;
}
```

strncat

Example

Purpose To append up to *max_len* characters from one null terminated string to the end of another.

Syntax

```
#include <string.h>
char *strncat(char *dest, const char *srce,
size_t max_len);
```

Description Appends either strlen(*srce*) or *max_len* characters (which- ever is the smaller) from the string pointed to by *srce* on to the end of the string pointed to by *dest* (the initial character of *srce* replaces the original terminating null of *dest*). A terminating null is appended to the result. A source string of zero length is quite acceptable.

Return value **strncat** returns the value of *dest*.

Notes If the two strings overlap then the effect of **strncat** should be regarded as unpredictable.

If strlen(*srce*) + strlen(*dest*) + 1 is greater than the length of the array *dest* then **strncat** may contaminate other areas of memory.

See also **strcat**.

Example

Example

```
#include <stdio.h>                         //for printf,gets
#include <string.h>                        //for strncat
#include <stdlib.h>                        //for atoi
int main()
{
   char srce[80],dest[80],str[10];
   size_t max_len;
   dest[0]='\0';
   do
   {
     printf("source     :"); gets(srce);
     printf("max length :"); gets(str);  max_len=atoi(str);
     printf("destination:%s\n",strncat(dest,srce,max_len));
   } while (srce[0]);
   return 0;
}
```

strncmp

Example

Purpose To compare up to a maximum number of characters from two null terminated strings.

Syntax

```
#include <string.h>
int strncmp(const char *s1, const char *s2,
size_t max_len);
```

Description Compares characters in corresponding positions in the strings pointed to by *s1* and *s2* until elements are found to differ or the end of one of the strings is reached or *max_len* characters have been compared.

Return value Either **strncmp** returns a positive or negative value depending on whether *s1*[i]-*s2*[i] is positive or negative for the index i where elements first differ; or zero if the two strings are identical up to the index *max_len*. (e.g. negative if *s1* is lexicographically less than *s2*.)

See also **strcmp**, **memcmp**.

Example strncmp("dog","dogs",3) is positive.
strncmp("dog","dogs",5) is negative.
strncmp("dog","dog",5) is zero.
strncmp("dog","dog",0) is zero.

Example

Example

```
#include <stdio.h>                    //for printf,gets
#include <string.h>                   //for strncmp
#include <stdlib.h>                   //for atoi
int main()
{
  size_t n;
  char str[10],s1[80],s2[80];
  do
  { printf("Source 1      :");  gets(s1);
    printf("Source 2      :");  gets(s2);
    printf("max_len       :");  gets(str); n=atoi(str);
    printf("strncmp(s1,s2,n):%d\n",strncmp(s1,s2,n));
    printf("strncmp(s2,s1,n):%d\n",strncmp(s2,s1,n));
  } while (n);
  return 0;
}
```

strncpy

Example

Purpose To copy up to a maximum number of characters from one string into another.

Syntax

```
#include <string.h>
char *strncpy (char *dest, const char *srce,
                            size_t max_len);
```

Description Copies not more than *max_len* characters from the string pointed to by *srce* into that pointed to by *dest*.

If *max_len* is greater than the length of *srce* then the remainder is padded with null characters.

If *max_len* is less than or equal to the length of *srce* then the truncated part is copied without a terminating null.

Return value **strncpy** returns the value of *dest*.

Notes If the two strings overlap then the effect of **strncpy** should be regarded as unpredictable.

Care should be taken to ensure that the destination array is long enough to accommodate *max_len* characters (i.e. *max_len* ≤ strlen(*dest*) + 1).

It is important to ensure that *max_len* is not negative.

See also **memcpy**, **strcpy**, **strdup**, **stpcpy**.

Example

```
#include <stdio.h>                         //for printf
#include <string.h>                        //for strncpy
#include <stdlib.h>                        //for atoi
int main()
{
   char str[10];
   size_t max_len;
   char srce[]="1234567890";                     // 10 chars
   do
   {
     char dest[]="********************";         // 20 chars
     printf("\n\nsrce       :%s\n",srce);
     printf("dest       :%s\n",dest);
     printf("max.number?:");  gets(str);  max_len=atoi(str);
     printf("dest       :%s\n",strncpy(dest,srce,max_len));
     if (max_len>21)
       printf("Error:potential corruption     \n");
     if(srce[0]!='1')
     printf("Error:actual corruption to srce\n");
   } while (max_len);
   return 0;
}
```

strpbrk

Example

Purpose To find the first occurrence in one string of any of the characters from another string.

Syntax

```
#include <string.h>
char *strpbrk(const char *s1,
              const char *s2);
```

Description **strpbrk** scans the string *s1*, for the first occurrence of any character appearing in *s2*. That is, **strpbrk** tests if *s1*[n] does not appear in the string *s2*, for n=0,1,2... until *s1*[n] is found in *s2* or the end of *s1* is reached.

strpbrk is equivalent to the code:

```
while (*s1 && !strchr(s2,*s1)) s1++;
return *s1? (char *)s1:NULL;
```

Return value **strpbrk** returns a pointer to the first occurrence in the string pointed to by *s1* of any of the characters from the string pointed to by *s2*.

If no character from *s2* occurs in *s1* then a **NULL** pointer is returned.

See also **strcspn**.

Example

```
#include <stdio.h>                                //for printf,gets
#include <string.h>                               //for strpbrk
int main()
{
  char s1[]="abcdefghijklmnopqrstuvwxyz";
  char s2[80];
  char *ptr;
  do
  {
    printf("s1       :%s\n",s1);
    printf("s2       :");  gets(s2);
    ptr=strpbrk(s1,s2);
    if (ptr) printf("Points to:%c\n",*ptr);
    else     printf("None in s1\n");
  } while (ptr);
  return 0;
}
```

strrchr

Example

Purpose To locate the position of the last occurrence of a given character in null terminated string.

Syntax

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Return value **strrchr** returns a pointer to the last occurrence of *c* in the string pointed to by *s* or a **NULL** pointer if the character has not been found.

If *c*='\0' then **strrchr** returns a pointer to the null terminator at the end of the string.

See also **memchr**, **strstr**.

Example

```
#include <stdio.h>                                  //for printf,gets
#include <string.h>                                 //for strrchr
int main()
{
  char *ptr,c,str[10];
  char s[]="aaabbcdddeefggghhijjjkkl";
  do
  {
    printf("Source   :%s\n",s);
    printf("c        :"); gets(str); c=str[0];
    ptr=strrchr(s,c);
    if (ptr) printf("points to position:%d\n",ptr-s+1);
    else     printf("not found\n");
  } while (ptr);
  return 0;
}
```

strspn

Example

Purpose To find the length of the initial part of one string which is made up entirely from characters that appear in another string.

Syntax

```
#include <string.h>
size_t strspn(const char *s1,
              const char *s2);
```

Description **strspn** tests if *s1*[n] appears in the string *s2*, for n=0,1,2... until *s1*[n] is not found in *s2* or the end of *s1* is reached.

strspn is equivalent to the code:

```
size_t len=0;
while (*s1 && strchr(s2,*s1)) {s1++;len++;}
return len;
```

Return value **strspn** returns the length of the initial part of *s1*.

See also **strcspn**,**strpbrk**.

Example

```
#include <stdio.h>                         //for printf,gets
#include <string.h>                        //for strspn
int main()
{
  char s1[]="abcdefghijklmnopqrstuvwxyz";
  char s2[80];
  size_t n;
  do
  {
    printf("s1       :%s\n",s1);
    printf("s2       :"); gets(s2);
    n=strspn(s1,s2);
    printf("strspn   :%d\n",n);
  } while (n);
  return 0;
}
```

strstr

Example

Purpose To locate the position of the first occurrence of a given string in another string.

Syntax

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Return value **strstr** returns a pointer to the first occurrence of the string pointed to by *s2* (excluding the terminating null) in the string pointed to by *s1*.

A **NULL** pointer is returned if *s2* has not been found. *s1* is returned if *s2*[0]='\0'.

Example

Example

```
#include <stdio.h>                               //for printf,gets
#include <string.h>                              //for strstr
int main()
{
  char *ptr,s2[80];
  char s1[]="aaabbcdddeefggghhijjjkkl";
  do
  {
    printf("s1      :%s\n",s1);
    printf("s2      :"); gets(s2);
    ptr=strstr(s1,s2);
    if (ptr) printf("points to position:%d\n",ptr-s1+1);
    else     printf("not found\n");
  } while (ptr);
  return 0;
}
```

strtok

Example

Purpose To split a string into *tokens* which are separated by given *delimiters*.

Syntax

```
#include <string.h>
char *strtok(char *str, const char *del);
```

Description A *delimiter* is defined here to be one of the ASCII characters listed in the string pointed to by *del*. A *token character* is defined to be any ASCII character which is not a *delimiter*. A *token* is defined to be a string of *token characters*.

On the first call of **strtok**, *str* is set to point to the string which is to be scanned. Subsequently if *str* is set to **NULL** the initial string is scanned for further *tokens* terminated by one of the *delimiters* listed in *del* (which may be the same or different from before).

Initially **strtok** searches *str* for the first *token character* thus passing over any leading *delimiters*. If no *token character* is found then **strtok** returns a **NULL** pointer. If a *token character* is found then **strtok** searches for the next *delimiter* which marks the end of that *token*. This *delimiter* is replaced by a null and a pointer to the next position is saved for subsequent calls of **strtok**. **strtok** then returns a pointer to the given *token*.

Each subsequent call of **strtok** with *str*=**NULL**, continues the search in like manner.

It can be assumed that the saved pointer value will not be changed by any other library function.

Return value **strtok** returns a pointer to the next null terminated *token* in the initial *str* or a **NULL** pointer if no further *tokens* can be found.

If there are no *token characters* in the string then **strtok** returns a **NULL** pointer.

If there are no *delimiters* in the string then **strtok** returns *str*.

Example

```
#include <stdio.h>                          //for printf,gets
#include <string.h>                         //for strtok
int main()
{
  char s1[80],s2[80];
  printf("string    :"); gets(s1);
  printf("delimiters:"); gets(s2);
  char *ptr=s1;
  int i=1;
  while (ptr=strtok(ptr,s2))                //assignment
  {
    printf("token %d   :%s\n",i++,ptr);
    ptr=NULL;
  }
  return 0;
}
```

strupr

Example

Purpose To change any lower case letters in a string to upper case.

Syntax

```
#include <string.h>
char *strupr(char *s);
```

Description **strupr** changes any letters in the range a..z in the string pointed to by *s*, to corresponding upper case letters A..Z leaving other characters unchanged.

Return value **strupr** returns a pointer to *s*.

See also **toupper**

Example

Example

```
#include <stdio.h>                           //for printf,gets
#include <string.h>                          //for strupr
int main()
{
   char str[80];
   do
   {
     printf("string:");   gets(str);
     printf("strupr:%s\n",strupr(str));
   } while (str[0]);
   return 0;
}
```

strxfrm

Purpose To transform a string in relation to the current locale.

Syntax

```
#include <string.h>
size_t strxfrm(char *dest, const char *src,
size_t n);
```

Description The current implementation supports only the “C” locale (see **<locale.h>**) in which **strxfrm** is identical to **strncpy** except for the return value.

Return value **strxfrm** returns the length of the transformed string (excluding the terminating null).

Functions defined in time.h

asctime	difftime	mktime
clock	gmtime	strftime
ctime	localtime	time

About functions defined in time.h

The **<time.h>** header contains a number of functions for obtaining the current date and time of day using the following definitions:

```
#define NULL 0
#define CLOCKS_PER_SEC 1
typedef double time_t;
typedef double clock_t;
typedef unsigned int size_t;

struct tm
{
  int  tm_sec;   /* Seconds after the minute [0..61] */
  int  tm_min;   /* Minutes after the hour [0..59] */
  int  tm_hour   /* Hours since midnight [0..23] */
  int  tm_mday;  /* Day of the month [1..31] */
  int  tm_mon;   /* Month of the year [0..11] */
  int  tm_year;  /* Year since 1900 */
  int  tm_wday;  /* Day of week [0..6] 0=Sunday */
  int  tm_yday;  /* Days since January 1st [0..365] */
  int  tm_isdst; /* Daylight Saving Time flag */
};
```

The first six items of this structure are packed (in an unspecified manner) into a **double** in order to provide an object of type **time_t**. The first form will be described as the *unpacked time* (also called the *broken down time*); the second form as the *packed time*. Either form can be used to store the so-called *calendar time* which is a term used for the date (according to the Gregorian calendar) and Greenwich Mean Time (with the seconds stored as an integer).

The input and output for a number of the relevant functions is summarised in the following table.

function	**input**	**output**
time	none	*packed time*
mktime	*unpacked time*	*packed time*
gmtime	*packed time*	*unpacked time*
localtime	*packed time*	*unpacked time*
asctime	*unpacked time*	string
ctime	*packed time*	string

strftime	*unpacked time*	string

asctime

Example

Purpose To convert an unpacked *calendar time* into a string.

Syntax

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

Description *timeptr* points to a **tm** structure containing the unpacked *calendar time*.

Return value **asctime** returns a pointer to a static string of the form:

```
Thu Nov 14 14:10:40 1991\n\0
```

A subsequent call of **asctime** will overwrite the static string (**asctime** and **ctime** use the same static string).

clock

Example

Purpose To provide the processor time in seconds.

Syntax

```
#include <time.h>
clock_t clock(void);
```

Description The **clock** function provides a measure of the elapsed time since the program started. Two calls of the function can be used to measure the time which has elapsed between passing the associated points in the program.

Return value **clock** returns a value which (when divided by **CLOCKS_PER_SEC**) gives the elapsed time in seconds.

Example

Example

```
#include <stdio.h>                       // for printf
#include <time.h>                        // for clock etc.
int main()
{ clock_t start=clock();
  printf("Wait....... and press return\n"); getchar();
  clock_t end=clock();
  printf("Elapsed time:%.2lf seconds\n",
           (end-start)/CLOCKS_PER_SEC);
  return 0;}
```

ctime

Example

Purpose To unpack a *calendar time* and convert it to a string.

Syntax

```
#include <time.h>
char *ctime(const time_t *timer);
```

Description **ctime** combines the operations of the functions **localtime** and **asctime** and is equivalent to asctime(localtime(*timer*)).

Return value **ctime** returns a pointer to a static string of the form:

```
Thu Nov 14 14:10:40 1991\n\0
```

A subsequent call of **ctime** will overwrite its own static string and the static **tm** structure associated with **localtime**. **asctime** and **ctime** use the same static string.

localtime and **ctime** use the same static **tm** structure.

Example

```
#include <stdio.h>                  // for printf
#include <time.h>                   // for ctime,time,asctime etc.
int main()
{ time_t t=time(NULL);
  printf("%s",ctime(&t));
  printf("%s",asctime(localtime(&t)));
  return 0;
}
```

difftime

Example

Purpose To compute the difference of two *calendar times*.

Syntax

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

Return value **difftime** returns the difference as a whole number of seconds between two values returned by the **time** or **mktime** functions.

Example

Example

```
#include <stdio.h>                      // for printf
#include <time.h>                       // for difftime,time etc.
int main()
{
  char c;
  time_t start=time(NULL);
  printf("Wait....... and press return\n"); c=getchar();
  time_t end=time(NULL);
  printf("Elapsed time:%.0lf seconds\n",difftime(end,start));
  return 0;
}
```

gmtime

Example

Purpose To unpack a *calendar time*.

Syntax

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

Description **gmtime** unpacks the contents of the object pointed to by *timer* into a static **tm** structure.

Return value **gmtime** returns a pointer to its own static **tm** structure which is overwritten on each call of **gmtime**.

Example

```
#include <stdio.h>                // for printf
#include <time.h>                 // for gmtime,time,asctime etc.
int main()
{ struct tm *unpacked_time;
  time_t packed_time=time(NULL);
  unpacked_time=gmtime(&packed_time);
  printf("%s",asctime(unpacked_time));
  return 0;
}
```

localtime

Example

Purpose To unpack a *calendar time*.

Syntax

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

Description **localtime** unpacks the contents of the object pointed to by *timer*. It is identical to **gmtime** but has its own **tm** structure (there is no implicit link provided between the two, nor is there any implicit adjustment for daylight saving time).

Return value **localtime** returns a pointer to its own static **tm** structure which is overwritten on each call of **localtime**.

mktime

Example

Purpose To pack a *calendar time*.

Syntax

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

Description The function **mktime** packs the *calendar time* stored in the structure pointed to by *timeptr* into an object which can be supplied to other functions of this group.

mktime also evaluates the day of the week and the number of days since January 1st and stores these values in the **tm** structure.

Return value **mktime** returns the packed *calendar time*.

Example

```
#include <stdio.h>                    // for printf
#include <stdlib.h>                   // for atoi
#include <time.h>                     // for mktime etc.
int main()
{
  char *weekday[]={" Sun"," Mon"," Tues"," Wednes"," Thurs",
               "  Fri"," Satur","n unknown "};
  char str[10];
  struct tm birth;
  printf("What is your date of birth?\n");
  printf("Year :"); gets(str); birth.tm_year=atoi(str)-1900;
  printf("Month:"); gets(str); birth.tm_mon =atoi(str)-1;
  printf("Date :"); gets(str); birth.tm_mday=atoi(str);
  if(mktime(&birth)==-1) birth.tm_wday=7;
  printf("You were born on a%sday\n",weekday[birth.tm_wday]);
  return 0;
}
```

strftime

Example

Purpose To format an unpacked *calendar time* for output.

Syntax

```
#include <time.h>
size_t strftime(char *s, size_t maxsize,
const char *format,
                   const struct tm *timeptr);
```

Description **strftime** formats the *unpacked time* in the structure pointed to by *timeptr* using the formatting sequence given in the string pointed to by *format*. No more than *maxsize* characters (including the terminating null) are placed into the string pointed to by *s*.

In a manner similar to **printf**, the formatting sequence consists of a mixture of conversion specifiers made up of pairs of characters (the first of which is a % character), and other characters which are copied unchanged. The details appear in the following table.

Return value **strftime** returns the number of characters placed into *s* excluding the terminating null.

Notes If the total number of characters required (including the terminating null) is greater than *maxsize* then **strftime** returns zero and the string pointed to by *s* may not be terminated at the expected point.
s and *format* should point to distinct non-overlapping strings.

Specifier	**Meaning**	**Range or Example**
%a	abbreviated weekday name	Sun to Sat
%A	full weekday name	Sunday to Saturday
%b	abbreviated month name	Jan to Dec
%B	full month name	January to December
%c	date and time	e.g.Thur Nov 14 14:10:40 1991
%d	day of the month	01 to 31
%H	hour, 24 hour clock	00 to 23
%I	hour, 12 hour clock	01 to 12
%j	day of year	001 to 366
%m	number of the month	01 to 12
%M	minute	00 to 59
%p	before or after noon	“AM” or “PM”

%S	second	00 to 61
%U	week number in year where the first Sunday is first day of week one	00 to 53
%w	number of weekday where 0 is Sunday	0 to 6
%W	week number in year where the first Monday is first day of week one	00 to 53
%x	date	e.g. Thur Nov 14 1991
%X	time	e.g. 14:10:40
%y	year number without the century	00 to 99
%Y	year with century	e.g. 1992
%Z	Greenwich Mean Time	“GMT”
%%		“%”

Example

```
#include <stdio.h>                // for printf
#include <time.h>                 // for strftime,time,gmtime etc.
#include <stdlib.h>               // for atoi
int main()
{
  char format[80],str[80];
  size_t maxsize,nchars;
  struct tm *timeptr;
  time_t t;
  do
  {
    t=time(NULL);
    timeptr=gmtime(&t);
    printf("maxsize      :"); gets(str); maxsize=atoi(str);
    printf("format string:"); gets(format);
    nchars=strftime(str,maxsize,format,timeptr);
    printf("strftime     :%s\n",str);
    printf("nchars       :%d\n\n",nchars);
  } while (maxsize);
  return 0;
}
```

time

Example

Purpose To provide the current *calendar time*.

Syntax

```
#include <time.h>
time_t time(time_t *timer);
```

Return value **time** returns the current *calendar time* in packed form. This form can be supplied to other time functions. If *timer* is not **NULL** then it points to an object which also contains the return value.

Functions defined in signal.h

signal
raise

About functions defined in signal.h

The **<signal.h>** header provides some definitions together with the prototypes for two functions which can be used to handle the reporting of external conditions during the execution of a program.   The definitions are listed here for reference.

```
typedef int sig_atomic_t;

#define SIG_DFL (void (*)(int))0
#define SIG_IGN (void (*)(int))-1
#define SIG_ERR (void (*)(int))-2
                         // * signifies ANSI extension
#define SIGINT     100   // Control-break
#define SIGFPE     101   // Floating point fault
#define SIGKEY     102   // Key press or release*
#define SIGALRM    103   // Alarm clock interrupt*
#define SIGMOUSE   104   // Mouse event*
#define SIGRESERVE 105   // Down to pages reserve*
#define SIGSEGV    106   // Invalid access to storage
                         // (G.P. exception)
#define SIGILL     107   // Illegal instruction
#define SIGABRT    200   // Abnormal termination
                         // (e.g. call to abort)
#define SIGTERM    201   // Termination request sent
                         // to program

#define SIGUSR1    202   // User-defined signal 1*
#define SIGUSR2    203   // User-defined signal 2*
#define SIGUSR3    204   // User-defined signal 3*
#define SIGUSR4    205   // User-defined signal 4*
```

signal

Example

Purpose To select from possible responses to a signal number.

Syntax

```
#include <signal.h>
void (*signal(int sig,
      void (*func)(int)))(int);
```

Description The value of *func* can be assigned in one of three ways corresponding to the different ways in which the receipt of the signal number *sig* is to be subsequently handled. If the value of *func* is **SIG_DFL**, then default handling for that signal will occur. If its value is **SIG_IGN**, the signal will be ignored. Thirdly, if *func* points to a function then that function will be called when the signal *sig* occurs.

Return value If the request can be honoured, **signal** returns the value of *func* for the most recent call to **signal** and the specified value of *sig*. Otherwise, a value of **SIG_ERR** is returned and a positive value is stored in **errno**.

Example

```
#include <stdio.h>          // for sprintf
#include <signal.h>         // for signal
#include <dbos\mouse.h>     // for hide_mouse_cursor etc.
#include <dbos\graphics.h>  // for draw_text etc.

volatile terminate=0;

void handler(int dummy)
{ short ih,iv,istat;
  static count=0;
  char text[80];
  hide_mouse_cursor();
  sprintf(text,"In handler, count = %d   ", ++count);
  clear_screen_area(10,80,300,150,0);
  draw_text(text,10,80,7);
  get_mouse_position(ih,iv,istat);
  sprintf(text,"ih = %d,iv = %d,istat = %d      ",ih,iv,istat);
  draw_text("                  ",10,120,7);
  draw_text(text,10,120,7);
  if (istat==1) terminate=1;
  signal(SIGMOUSE, handler);
  display_mouse_cursor();
  set_mouse_interrupt_mask((short)1);
}
main()
{ int i;
  char text[80];
  clear_screen();
  display_mouse_cursor();

  signal(SIGMOUSE, handler);
  set_mouse_interrupt_mask((short)1);
  vga();
  i=0;
  while(1)
    {
    sprintf(text,"i = %3d  ",i);
    clear_screen_area(10,10,200,60,0);
    draw_text(text,10,10,2);
    i=(i+1)%10000;
    if (terminate)
      {
      hide_mouse_cursor();
      text_mode();
      set_mouse_interrupt_mask((short)0);
      exit(1);
      }
    }
}
```

raise

Purpose To send a signal to the executing program.

Syntax

```
#include <signal.h>
int raise(int sig);
```

Description The **raise** function sends the signal *sig* to the executing program.

Return value The **raise** function returns zero if successful, non-zero if unsuccessful.

Functions defined in stdarg.h

va_start

va_arg

va_end

About functions defined in stdarg.h

The **<stdarg.h>** header provides prototypes for three macros which can be used within a function call to advance through an argument list of variable length. This mechanism allows functions with varying numbers of arguments of differing types (like **printf**) to be coded.

The function definition must include a list of fixed arguments (i.e. at least one). In what follows, the rightmost fixed argument is designated by *parmN*. This list is followed by ,... (a comma and three full-stops) representing the variable part of the argument list.

As the list of variable length is processed by the function, the current argument *ap* of type *type* is initialised by the **va_start** macro, advanced by **va_arg** macro, and terminated by the **va_end** macro.

va_start

Example

Purpose To initialise a list of arguments of variable length.

Syntax

```
#include <stdarg.h>
void va_start(va_list ap,parmN);
```

Description The **va_start** macro initialises *ap* as the first argument in an argument list of variable length. *parmN* is the rightmost argument in a list of fixed arguments which must come before the list of variable length.

Return value None.

va_arg

Example

Purpose To advance through a list of arguments of variable length.

Syntax

```
#include <stdarg.h>
type va_arg(va_list ap,type);
```

Description The **va_arg** macro expands to an expression which has the type and value of the next argument in the call. *ap* is initialised by **va_start** and advanced by each call of **va_arg** in such a way that **va_arg** returns the values of successive arguments in turn.

Return value Initially **va_arg** returns the first argument to the right of *parmN* (in **va_start**). Successive calls of **va_arg** return the values of the following arguments in order.

va_end

Example

Purpose To terminate a list of arguments of variable length.

Syntax

```
#include <stdarg.h>
void va_end(va_list ap);
```

Description The **va_end** macro assigns a terminating value to *ap* after it has been used as the current argument in a list of varying length. A call to **va_arg** prevents *ap* from being used again without an intervening call to **va_start**.

Return value None.

Example

```
//Sums a list of numbers terminated by zero.
#include <stdio.h>                // for printf
#include <stdarg.h>               // for va_start,va_arg,va_end
int sum(int first,...)
{  int total=first,next=first;
   va_list ap;  va_start(ap,first);
   while(next) { next=va_arg(ap,int);  total+=next; }
   va_end(ap); return total;
}
int main()
{  printf("The sum is:%d\n",sum(1,2,3,4,5,0));
   return 0;
}
```

Functions defined in assert.h

assert

About functions defined in assert.h

The **<assert.h>** header defines the **assert** macro which may be used in a program in order to force a program to abort under conditions supplied by the user.

assert

Example

Purpose To put diagnostic information into a program.

Syntax

```
#include <assert.h>
void assert(int expression);
```

Description When *expression* evaluates to zero, the **assert** macro sends diagnostic information to the **stderr** stream and then calls the **abort** function. This information includes the expression, and the file name and line number where the expression was found. If *expression* evaluates to non-zero, then **assert** has no tangible effect.

The **<assert.h>** header refers to the macro **NDEBUG** which may be defined by the user in order to switch off the diagnostic facility. For this purpose simply insert:

```
#define NDEBUG
```

before the call to

```
#include <assert.h>
```

or use the /DEFINE option in the compiler command line.

Return value None.

Example

```
#include <stdio.h>                  // for printf,gets
#include <math.h>                   // for acos
#include <stdlib.h>                 // for strtod
#include <errno.h>                  // for errno
#include <assert.h>                 // for assert
int main()
{
   double x,y;
   char str[80];
   do
   {
      printf("Input x:"); gets(str); x=strtod(str,NULL);
      y=acos(x);
      assert(errno==0);
      printf("acos(x):%lf\n",y);
   } while(x);
   return 0;
}
```

Functions defined in locale.h

setlocale
localeconv

About functions defined in locale.h

The **<locale.h>** header provides prototypes for two functions which relate to the current “locale”.   In the present context, a locale is a particular country or nation with (for example) its own currency punctuation conventions.   One function (**setlocale**) enables the user to select from the various locales provided by the system.   The other (**localeconv**) provides access to the predefined conventions for that locale.

At the present stage of development, only the "C" locale is supported in which all of the standard fields take their default state.

setlocale

Example

Purpose To select one of the locales supported by the system.

Syntax

```
#include <locale.h>
char *setlocale(int category,
                const char *locale);
```

Description Currently only the "C" locale is supported so for the time being this function is redundant. *locale* points to a string defining the locale whilst *category* is one of the macros LC_ALL, LC_COLLATE, LC_CTYPE, LC_MONETARY, LC_NUMERIC, and LC_TIME representing aspects of the locale.

Return value If successful, **setlocale** returns a pointer to a string describing the former locale, otherwise it returns a NULL pointer.

Example

```
#include <stdio.h>                // for printf
#include <locale.h>               // for setlocale
int main()
{
   char *old;
   old=setlocale(LC_ALL,"C");
   printf("The former locale was %s\n",old);
   return 0;
}
```

localeconv

Purpose To obtain details of the current locale.

Syntax

```
#include <locale.h>
struct lconv *localeconv(void);
```

Return value The **localeconv** function returns a pointer to a **lconv** structure which contains details the current locale conventions. Currently only the "C" locale is supported in which all of the standard fields take their default state according to the C standard.

Functions defined in setjmp.h

About functions defined in setjmp.h

The **<setjmp.h>** header declares the **jmp_buf** type and gives prototypes for two functions which are used to perform non-local goto commands.   These functions were designed to deal with errors and exceptions encountered in user defined functions written in assembler code.

setjmp

Purpose To prepare for a non-local jump.

Syntax

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Description The **setjmp** function saves its calling environment in *env* in preparation for a later call to the **longjmp** function. The call to **longjmp** returns control to the position of the corresponding **setjmp** function.

Return value Initially **setjmp** returns 0. Following a transfer of control by a call to **longjmp**, **setjmp** returns *val* (a parameter in **longjmp**).

longjmp

Example

Purpose To execute a non-local jump.

Syntax

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Description The **longjmp** function is used to return control to the position of the **setjmp** function which was used to assign values in the *env* **jmp_buf** structure. In effect, **longjmp** calls **setjmp** and returns control to the calling point of **setjmp.** *val* is given a value which is to be used as the return for **setjmp.** A zero value for *val* is not permitted and is automatically changed to 1.

Return value The **longjmp** function does not return to it own calling point.

Example

```
#include <stdio.h>                 // for printf
#include <setjmp.h>                // for setjmp,longjmp
int main()
{
   jmp_buf env;
   int val=setjmp(env);
   printf("val:%d\n",val);
   if(val) return 0;
   longjmp(env,1);
}
```

The error.h header

The **<errno.h>** header defines certain macros which relate to the reporting of error conditions and declares an associated external variable **errno** of type **int** which is supplied by the system.   This variable is initialised to **EZERO** (=0) at the beginning of a program and is set to non-zero by certain functions in the system library when certain non-fatal error conditions arise.   The value of **errno** is unique to a particular type of error which can be identified by a call to **strerror** or **perror**.   After a call to one of the associated library functions, the user should test the current value of **errno** and (if appropriate) reset the value to **EZERO** before initiating a response to the given error condition.

Example

```
#include <stdio.h>                 // for printf,gets,perror
#include <math.h>                  // for acos
#include <stdlib.h>                // for strtod
#include <errno.h>                 // for errno
int main()
{
   double x,y;
   char str[80];
   do
   {
      printf("Input x:"); gets(str); x=strtod(str,NULL);
      y=acos(x);
      if(errno==EZERO) printf("acos(x):%lf\n",y);
      else {perror("ERROR in acos"); errno=EZERO;}
   } while(x);
   return 0;
}
```

The limits.h header

The standard numerical limits given in the **<limits.h>** header are:

```
#define CHAR_BIT        8
#define SCHAR_MIN       -128
#define SCHAR_MAX       127
#define UCHAR_MAX       255U
#define CHAR_MIN        SCHAR_MIN
#define CHAR_MAX        SCHAR_MAX
#define MB_LEN_MAX      1
#define SHRT_MIN        -32768
#define SHRT_MAX        32767
#define USHRT_MAX       65535U
#define INT_MIN         -2147483648
#define INT_MAX         2147483647
#define UINT_MAX        4294967295U
#define LONG_MAX        2147483647
#define LONG_MIN        -2147483648
#define ULONG_MAX       4294967295U
```

The float.h header

The limits given in <float.h> are

```
#define FLT_RADIX             2
#define FLT_ROUNDS            1

#define FLT_DIG               6
#define FLT_MANT_DIG          24
#define FLT_MAX_10_EXP        +38
#define FLT_MAX_EXP                 +128
#define FLT_MIN_10_EXP        -37
#define FLT_MIN_EXP                 -125
#define FLT_MAX               3.40e38
#define FLT_MIN               1.175e-38
#define FLT_EPSILON                 1.19e-7

#define DBL_DIG               15
#define DBL_MANT_DIG          53
#define DBL_MAX_10_EXP        +308
#define DBL_MAX_EXP                 +1024
#define DBL_MIN_10_EXP        -307
#define DBL_MIN_EXP                 -1021
#define DBL_MAX               1.80e308
#define DBL_MIN               2.225e-308
#define DBL_EPSILON                 2.22e-16

#define LDBL_DIG              19
#define LDBL_MANT_DIG         64
#define LDBL_MAX_10_EXP       +4932
#define LDBL_MAX_EXP          +16384
#define LDBL_MIN_10_EXP       -4931
#define LDBL_MIN_EXP          -16381
#define LDBL_MAX              1.21e4932
#define LDBL_MIN              3.32e-4931
#define LDBL_EPSILON          1e-17
```

The sizeof standard types

type	sizeof
char	1
short int	2
int	4
long int	4
float	4
double	8
long double	10

The ANSI C standard defines sizeof(char)=1 byte.

Functions defined in clib.h

bdos, _bdos
bioscom, _bios_serialcom
biosequip, bios_equiplist
bioskey, _bios_keybrd
biosmemory, _bios_memsize
biosprint, _bios_printer
biostime, _bios_timeofday
cgets, _cgets
chmod, _chmod
cprintf, _cprintf
cscanf, _cscanf
_dos_getdate
_dos_gettime
_dos_setdate
_dos_settime
ecvt, _ecvt
_exit
fcvt, _fcvt
fileno
_fpreset
gcvt, _gcvt
_get_file_structure
getch, _getch
getche, _getche
getdate
gettime
inport,_inpw
inportb, inp
inport32
isascii, __isascii
kbhit, _kbhit
lfind, _lfind
_lrotl
_lrotr
lsearch, _lsearch
ltoa
memicmp, _memicmp
_OFFSET
outport, _outpw
outportb, _outp
outport32
putch, _putch
_rotl
_rotr
_SEGMENT
setdate
setdisk
setmode
settime
_strerror
strnicmp, _strnicmp
strrev, _strrev
swab, _swab
toascii, __toascii
_tolower
_toupper
ultoa, _ultoa
ungetch, _ungetch

About functions defined in clib.h

The **<clib.h>** header gives prototypes for a number of general functions. It includes the following definitions:

```
struct date
{
    int da_year;            // current year
    char da_day;            // day of month
    char da_mon;            // month (1=Jan)
};

struct ftime
{
    unsigned ft_tsec;        // Two seconds.
    unsigned ft_min;         // Minutes.
    unsigned ft_hour;        // Hours.
    unsigned ft_day;         // Day.
    unsigned ft_month;       // Month.
    unsigned ft_year;        // Year - 1980.
};

struct __time
{
    unsigned char ti_min;
    unsigned char ti_hour;
    unsigned char ti_hund;
    unsigned char ti_sec;
};

struct _dostime_t
{
   unsigned char hour;
   unsigned char minute;
   unsigned char second;
   unsigned char hsecond;
};
struct _dosdate_t
{
   unsigned char day;
   unsigned char month;
   unsigned char dayofweek;
   unsigned int year;
};
```

bdos, _bdos

Purpose To make a DOS/BIOS call using only the DX and AL registers.

Syntax

```
#include <clib.h>
int bdos(int dosfun, unsigned dosdx,
                      unsigned dosal);
```

Description **bdos** performs the software interrupt given by *dosfun* with the values of DX and AL given by *dosdx* and *dosal* respectively.

Return value **bdos** returns the value of AX after the interrupt.

Notes **_bdos** is a alternative name for **bdos**.

See also **intdos**, **intdosx**, **int86**, **int86x**.

bioscom, _bios_serialcom

Example

Purpose To perform serial I/O via an RS-232 port.

Syntax

```
#include <clib.h>
int bioscom(int cmd, char abyte, int port);
```

Description **bioscom** uses BIOS interrput 0x14 to perform serial I/O services. The value of *cmd* is one of:

0 Initialise Port Parameters
1 Send one character
2 Receive one character.
3 Get port status.

In all commands the value of port is the serial port number to which the command should be issued.

cmd=0

abyte contains the intended serial port parameters, with bit settings BBBPPSCC.

BBB	**Baud Rate**
000	110 baud
001	150 baud
010	300 baud
011	600 baud
100	1200 baud
101	2400 baud
110	4800 baud
111	9600 baud

PP	**Parity Code**
00	None
01	Odd
10	None
11	Even

S	**Number of stop bits**
0	One stop bit
1	Two stop bits

CC	Character size
10	7-bit character
11	8-bit character

cmd=1 *abyte* contains the character to be sent.

cmd=2 In this case, the value of *abyte* is unused.

cmd=3 In this case also, the value of *abyte* is unused.

Return value *cmd*=0, the return value is not meaningful.

cmd=1, bits 8 to 15 of the return value are as defined below in entries 0x8000 to 0x0100.

cmd=2, bits 8 to 15 of the return value are as defined below in entries 0x8000 to 0x0100, bits 0 to 7 contain the character received from the port.

cmd=3, the lower 16 bits of the return value are given by:

0x8000	Time out
0x4000	Transfer shift register empty
0x2000	Transfer holding register empty
0x1000	Break detect
0x0800	Framing error
0x0400	Parity error
0x0200	Overrun error
0x0100	Data ready
0x0080	Received line signal detect
0x0040	Ring indicator
0x0020	Data set ready
0x0010	Clear to send
0x0008	Data receive line signal detect
0x0004	Trailing edge ring detector
0x0002	Delta data set ready
0x0001	Delta clear to send

Notes **_bios_serialcom** is alternative name for **bioscom**.

See also **in** , **out** , **inport**, **inportb**, **inport32**, **outport**, **outportb**, **outport32**.

biosequip, _bios_equiplist

Example

Purpose To get the peripheral equipment list.

Syntax

```
#include <clib.h>
int biosequip(void);
```

Description **biosequip** gets the value of the equipment list code word using BIOS interrupt 0x11.

Return value **biosequip** returns the value of the equipment list code word which is in the form PPMURRRUFFVVUUCI.

PP	Number of printers installed
M	1 if internal modem installed
RRR	Number of RS-232 ports installed
U	Unused
FF	Number of floppy disk drives minus 1 (0= one drive)
VV	Internal video mode.
00	Reserved
01	40 -by- 25 colour
10	80 -by- 25 colour
11	80 -by- 25 monochrome
U	Unused
C	1 if math coprocessor installed
I	1 if initial program load diskette installed

Notes **_bios_equiplist** is an alternative name for **bioequip**.

bioskey, _bios_keybrd

Example

Purpose To interface the BIOS keyboard services directly.

Syntax

```
#include <clib.h>
int bioskey(int cmd);
```

Description Performs BIOS keyboard services directly using interrupt 0x16. The parameter *cmd* can take the values:

0	Read next character. This function will not return until a character is read
1	Report if character ready. This function will not remove the character from the keyboard buffer.
2	Get shift status

Return value *cmd*=0:

If the character read is an ASCII character, bits 0 to 7 contain the ASCII character code, and bits 8 to 15 contain the standard PC keyboard scan code.

If the character read is an extended ASCII code, bits 0 to 7 contain the extended ASCII code, and bits 8 to 15 are NULL.

cmd=1:

0 if there is no character waiting in the keyboard buffer. Otherwise, bits 0 to 15 of the returned value are as for *cmd=0*.

cmd=2:

Bits 0 to 7 contain the shift status given by:

0x01	Right shift active
0x02	Left shift active
0x04	Ctrl active
0x08	Alt active
0x10	Scroll lock active
0x20	Num lock active
0x40	Caps lock active
0x80	Insert state active

Notes Multiple states can be active simultaneously.

_bios_keybrd is an alternative name for **bioskey**

See also dos_key_waiting , key_waiting , get_key , get_dos_key

biosmemory, _bios_memsize

Purpose To return the size of the basic memory.

Syntax

```
#include <clib.h>
int biosmemory(void);
```

Description **biosmemory** uses interrupt 0x12 to get the size of the basic memory.

Return value **biosmemory** returns the size of the basic memory in 1K blocks. This does not include display adapter memory, extended memory or expanded memory.

Notes **_bios_memsize** is an alternative name for **biosmemory**.

biosprint, _bios_printer

Example

Purpose Performs printer I/O using BIOS services directly.

Syntax

```
#include<clib.h>
int biosprint(int cmd, int abyte, int port);
```

Description **biosprint** is used to perform various printer operations using BIOS interrupt 0x17.

cmd defines which printer operation is to be executed. It can be one of the following:

0	Send byte to printer; sends the character in *abyte* to printer number *port*.
1	Initialise printer; Initialises printer number *port*.
2	Get printer status.

Return value The return value for all values of *cmd* is the printer status defined by:

0x01	Time out
0x02	Unused
0x04	Unused
0x08	I/O error
0x10	Printer selected
0x20	Out of paper
0x40	Printer acknowledgement
0x80	Printer not busy

Notes Multiple states can be active simultaneously.

_bios_printer is an alternative name for **biosprint.**

biostime, _bios_timeofday

Example

Purpose To get or set the current BIOS timer.

Syntax

```
#include <clib.h>
long biostime(int cmd, long newtime)
```

Description **biostime** either gets or sets the BIOS timer using BIOS interrupt 0x1A. This timer counts in ticks since midnight at a rate of around 18.2 ticks per second. A *cmd* value of 0 will get the current clock value. A *cmd* value of 1 will cause the timer to be set to the value of *newtime*.

Return value If *cmd* = 0, the return value is the value of the tick count.

Notes **_bios_timeofday** is an alternative name for **biostime**.

cgets, _cgets

Purpose To get a string from the console.

Syntax

```
#include <clib.h>
char *cgets(char *str)
```

Description The **cgets** function reads characters from **stdin** until either a '\n' is encountered or until the maximum number of characters to be read has been reached. This maximum number is passed to the function in *str[0]* and the actual number of characters read is returned in *str[1]*. The characters read start at *str[2]* and end with a NULL terminator.

Notes The array pointed to by *str* must be at least *str[0]* + 2 bytes long. **_cgets** is an alternative name for **cgets.**

These functions are not fully compatible with similar Borland/Microsoft functions which input directly from the the console.

chmod, _chmod

Purpose To change the access permissions pertaining to a file.

Syntax

```
#include <clib.h>
#include <sys\stat.h>
int chmod(const char *path, int amode)
```

Description The function *chmod* is used to change the access mode of the file specified by *path*. The specified filename should not contain any wild cards.

The permissible values for amode are:

S_IREAD	Permission to read
S_IWRITE	Permission to write
S_IREAD | S_IWRITE	Permission to read and write

These constants are defined in **sys\stat.h**.

Since under DOS any existing files automatically have read permission, **chmod** only affects the write permission on a file.

Return value On success **chmod** returns 0. Otherwise **chmod** returns **EOF** and sets **errno** to either **ENOENT** or **EACCES**.

Notes **_chmod** is an alternative name for **chmod**.

cprintf, _cprintf

Purpose To put text to **stdout**.

Syntax

```
#include <clib.h>
int cprintf(const char *format,...);
```

Description The **cprintf** function performs the same operation as the **printf** function. See **printf** for further details.

Notes **_cprintf** is an alternative name for **cprintf**. These functions are not fully compatible with similar Borland/Microsoft functions which output directly to the screen.

cscanf, _cscanf

Purpose To get text from **stdin**.

Syntax

```
#include <clib.h>
int cscanf(const char *format,...);
```

Description The **cscanf** function performs the same operation as the **scanf** function. See **scanf** for further details.

Notes **_cscanf** is an alternative name for **cscanf**. These functions are not fully compatible with similar Borland/Microsoft functions which input directly from the console.

_dos_getdate

Purpose Gets the system date and stores it in a structure of type **_dosdate_t**.

Syntax

```
#include <clib.h>
void getdate(struct _dosdate_t *date)
```

Description **_dos_getdate** gets the system date and stores it in the structure pointed to by *date*

Return value None.

See also **getdate**

_dos_gettime

Purpose Gets the system time.

Syntax

```
#include <clib.h>
void _dos_gettime(struct _dostime_t *time)
```

Description The **_dos_gettime** function fills in the **_dostime_t** structure pointed to by *time* with the current system time.

Return value None.

See also **gettime**

_dos_setdate

Purpose Sets the system time and date.

Syntax

```
#include <clib.h>
unsigned _dos_setdate
         (struct _dosdate_t *date);
```

Description **_dos_setdate** sets the system date to that contained in the **_dosdate_t** structure pointed to by *date*.

Return value Returns 0 for success, non-zero for failure. On failure **errno** is set to **EINVAL** showing an invalid date was specified.

See also **setdate**

_dos_settime

Purpose To set the current system time.

Syntax

```
#include <clib.h>
unsigned _dos_settime
           (struct _dostime_t *time);
```

Description **_dos_settime** sets the current system time according to the contents of the structure pointed to by *time*.

Return value Returns 0 for success, non-zero for failure.   On failure **errno** is set to **EINVAL** showing an invalid time was specified..

See also **settime**

ecvt, _ecvt

Purpose Converts a floating point number to a string.

Syntax

```
#include <clib.h>
char *ecvt(double value, int ndig, int *dec,
int *sign)
```

Description **ecvt** converts the floating point value stored in *value* to a string of *ndig* digits. The last digit is rounded. If *value* is negative, the integer pointed to by *sign* is non-zero, otherwise it is zero.

dec stores the position of the decimal point relative to the beginning of the string. (If *dec* < 0 then the decimal point lies to the left of the string).

Return value **ecvt** returns a pointer to a static buffer containing the string. This buffer is overwritten on each call to **ecvt**.

Notes The returned string does not contain a decimal point.

_ecvt is an alternative name for **ecvt**.

See also **fcvt**, **gcvt**

_exit

Purpose To terminate a program's execution.

Syntax

```
#include <clib.h>
void _exit(int status);
```

Description **_exit** terminates the program currently being executed and passes the integer contained in *status* to the calling environment. This function does not flush any buffers, or call any exit functions.

fcvt, _fcvt

Purpose Converts a floating point number to a string.

Syntax

```
#include <clib.h>
char *fcvt(double value, int ndig, int *dec,
int *sign);
```

Description **fcvt** converts the value held in *value* to a string of length *ndig*+1 digits, which is accurate to *ndig* digits. It is similar to **ecvt**, except that it causes *dec* to be greater than 0.

Return value **fcvt** returns a pointer to a static buffer containing the string generated. This static buffer is overwritten on each call to **fcvt**.

Notes The returned string does not contain a decimal point.

_fcvt is an alternative name for **fcvt**.

fileno

Purpose To get the handle associated with a stream.

Syntax

```
#include <clib.h>
int fileno(FILE *stream)
```

Description **fileno** is a macro which returns the handle corresponding to *stream*.

Return value **fileno** returns the integer file handle associated with *stream*.

_fpreset

Purpose To reset the state of the floating point system.

Syntax

```
#include <clib.h>
void _fpreset(void);
```

Description **_fpreset** reinitialises the state of the floating point maths co-processor.

Return value None.

gcvt, _gcvt

Purpose To convert a floating point number into a string.

Syntax

```
#include <clib.h>
 char *gcvt(double value,int prec, char *d);
```

Description The **gcvt** function converts the floating point number contained in *value* to a string in standard form. This string contains *prec* significant figures and a decimal point.

If necessary, this string will also contain a '-' sign, and an exponent of the form 'e[+-][0-9][0-9]'. This string is stored in *d*. For this reason, the array *d* should be of length *prec*+4.

Notes **_gcvt** is an alternative name for **gcvt**.

_get_file_structure

Purpose To get a pointer to a file given the handle.

Syntax

```
#include <clib.h>
FILE* _get_file_structure(int handle);
```

Description **_get_file_structure** gets a pointer to the file structure corresponding to *handle*.

Return value The **_get_file_structure** function returns a pointer to the FILE structure corresponding to *handle* or NULL if there is no such file.

getch, _getch

Purpose Gets a character from the keyboard without echoing to the screen.

Syntax

```
#include <clib.h>
int getch(void)
```

Description **getch** gets a character directly from the keyboard without echoing to the screen. **getch** does not use **stdin**.

Return value **getch** returns the character read.

Notes **_getch** is an alternative name for **getch**.

getche, _getche

Purpose Gets a character from the keyboard and echos to the screen.

Syntax

```
#include <clib.h>
int getche(void)
```

Description **getche** gets a character directly from the keyboard without using **stdin**.

Return value **getche** returns the character read.

Notes **_getche** is an alternative name for **getche**.

getdate

Purpose Gets the system date and stores it in a structure of type **date**.

Syntax

```
#include <clib.h>
void getdate(struct date *datep)
```

Description **getdate** gets the system date and stores it in the structure pointed to by *datep*

Return value None.

See also **_dos_getdate**

gettime

Purpose Gets the system time.

Syntax

```
#include <clib.h>
void gettime(struct __time *timep)
```

Description The **gettime** function fills in the **_ _time** structure pointed to by *timep* with the current system time.

Return value None.

See also **_dos_gettime**

inport, _inpw

Purpose Read two bytes from an I/O port.

Syntax

```
#include <clib.h>
int inport(int portid);
```

Description The **inport** function reads two bytes of data from the I/O port specified by *portid*.

Return value **inport** returns the two bytes read from the I/O port.

Notes This function assumes that the low-byte is transmitted first.

_inpw is an alternative name for **inport**.

inportb, inp

Purpose Reads one byte from an I/O port.

Syntax

```
#include <clib.h>
unsigned char inportb(int portid);
```

Description The **inportb** function reads one byte of data from the I/O port specified by *portid*.

Return value The **inportb** function returns the value read from the port.

Notes **inp** is an alternative name for **inportb**.

inport32

Purpose To read a 32-bit word from an I/O port.

Syntax

```
#include <clib.h>
long inport32(int portid);
```

Description The **inport32** function reads a 32-bit word from the I/O port specified in *portid*.

Return value **inport32** returns the word read from the I/O port specified in *portid*.

Notes This function assumes that the low-byte is transmitted first.

isascii, __isascii

Purpose To ascertain whether an integer is an ASCII character.

Syntax

```
#include <clib.h>
int isascii(int c);
```

Description **isascii** is a macro which classifies integers as to whether they are ASCII characters.

Return value **isascii** returns non-zero for true, and zero for false.

Notes **__isascii** is an alternative name for **isascii**.

kbhit, _kbhit

Purpose To check if there is a key waiting in the keyboard buffer.

Syntax

```
#include <clib.h>
int kbhit(void);
```

Description **kbhit** performs the same function as **key_waiting**. See **key_waiting** for further details.

Notes **_kbhit** is a alternative name for **kbhit**.

lfind, _lfind

Purpose To perform a linear search.

Syntax

```
#include <clib.h>
void *lfind(const void *key,const void *base,
  size_t *num, size_t width,
  int (*compare)(const void *,const void *));

#include <clib.h>
void *_lfind(const void *key,
  const void *base,
  size_t *num, size_t width,
  int (__cdecl*compare)
      (const void *, const void *));
```

Description The **lfind** function performs a linear search through the list pointed to by *base* for the item pointed to by *key*. Each item in the list should be of size *width* and the number of items in the list should be stored in the integer pointed to by *num*. This function uses a user defined comparison routine *compare* which should return 0 if **param1* = = **param2*.

Return value **lfind** returns a pointer to the first matching element in the list if one is found. Otherwise **lfind** returns NULL.

Notes Unlike **bsearch**, **lfind** does not require the list to be sorted.

_lfind is an alternative name for **lfind**. **__cdecl** is provided for compatibility on porting but otherwise has no effect.

See also **bsearch**, **lsearch**.

_lrotl

Purpose To rotate a **long int** left.

Syntax

```
#include <clib.h>
unsigned long _lrotl(unsigned long val,
                     int n);
```

Description **_lrotl** rotates the value held in *value* left by *n* bits.

Return value **_lrotl** returns the rotated long integer.

_lrotr

Purpose To rotate a **long int** right.

Syntax

```
#include <clib.h>
unsigned long _lrotr(unsigned long val,
                     int n);
```

Description **_lrotr** rotates the value held in *value* right by *n* bits.

Return value **_lrotr** returns the rotated long integer.

lsearch, _lsearch

Purpose To perform a linear search.

Syntax

```
#include <clib.h>
void *lsearch(const void *key,
   const void *base,
   size_t *num, size_t width,
   int (*compare)(const void *,const void *))

#include <clib.h>
void *_lsearch(const void *key,
   const void *base,
   size_t *num, size_t width,
   int (__cdecl *compare)
       (const void *,const void*))
```

Description The **lsearch** function performs a linear search through the list pointed to by *base* for the item pointed to by *key*.

Each item in the list should be of size *width* and the number of items in the list should be stored in the integer pointed to by *num*. This function uses a user defined comparison routine *compare* which should return 0 if **param1* = = **param2*. If the item pointed to by *key* is not in the list, it is added to the end of the list. For this reason, it should be ensured that the list has enough allocated memory to hold an extra element before calling **lsearch**.

Return value **lsearch** returns a pointer to the first item in the list which matches *key*. If *key* has been added to the list, then *num* is incremented.

Notes **_lsearch** is an alternative name for **lsearch**. **__cdecl** is provided for compatibility on porting but otherwise has no effect.

See also **lfind**, **bsearch**

ltoa

Purpose To convert a **long** to a string.

Syntax

```
#include <clib.h>
char *ltoa(long value, char *string,
           int radix);
```

Description The **ltoa** function is identical to the **itoa** function. For more details see **itoa**.

memicmp, _memicmp

Purpose To compare two arrays of characters, without regard to case.

Syntax

```
#include <clib.h>
int memicmp(const void *s1, const void *s2,
size_t n);
```

Description The **memicmp** function compares the first *n* characters of two arrays without regard to character case.

Return value **memicmp** returns an integer which is

> 0 if s1 greater than s2

< 0 if s2 greater than s1

= 0 if s1 is the same as s2 except possibly for case differences.

Notes **_memicmp** is an alternative name for **memicmp**.

See also **memcmp**

_OFFSET

Purpose To obtain the byte offset into a segment.

Syntax

```
#include <clib.h>
unsigned _OFFSET(void *address);
```

Return value Returns the byte offset into a segment for a given address. This function should only be used for the real mode address.

See also _SEGMENT, **doscom**

outport, _outpw

Purpose To send two bytes of data to an I/O port.

Syntax

```
#include <clib.h>
void outport(int portid, int value);
unsigned _outpw(unsigned portid,
                unsigned value);
```

Description These functions send the two bytes of data stored in bits 0 to 15 of *value* to the port specified by *portid*.

Return value **_outpw** returns the data output.

Notes **outport** sends the low-byte first.

outportb, _outp

Purpose To send a byte to an I/O port.

Syntax

```
#include <clib.h>
void outportb(int portid,
              unsigned char value);
int _outp(unsigned portid, int value)
```

Description These functions send the byte contained in *value* to the port specified in *portid*.

Return value **_outp** returns the data output.

outport32

Purpose To output 32 bits to an I/O port.

Syntax

```
#include <clib.h>
void outport32(int portid, long value);
```

Description **outport32** sends the word contained in *value* to the port specified in *portid*.

Return value None.

Notes **outport32** sends the low-byte first.

putch, _putch

Purpose To put a character to **stdout**.

Syntax

```
#include <clib.h>
int putch(int c);
```

Description **putch** puts the character specified by *c* to **stdout**.

Return value **putch** returns non-zero on success and zero on failure.

Notes **_putch** is an alternative name for **putch**. These functions are not fully compatible with similar Borland/Microsoft functions which output directly to the screen.

_rotl

Purpose To rotate an integer left.

Syntax

```
#include <clib.h>
unsigned int _rotl(unsigned int val, int n);
```

Description **_rotl** rotates the value held in *val* left by *n* bits.

Return value The **_rotl** function returns the rotated integer.

_rotr

Purpose To rotate an integer right.

Syntax

```
#include <clib.h>
unsigned int _rotr(unsigned int val, int n);
```

Description **_rotr** rotates the value held in *val* right by *n* bits.

Return value The **_rotr** function returns the rotated integer.

_SEGMENT

Purpose To obtain the segment number.

Syntax

```
#include <clib.h>
unsigned _SEGMENT(void *address);
```

Return value Returns the segment number.   This function should only be used for the real mode address.

Notes The functions _SEGMENT and _OFFSET can be used in conjunction with **doscom**   to obtain the segment/offsets required by DOS for INT 21 functions.   They should not be used to pass segments and offsets for extended memory addresses!

See also _OFFSET

setdate

Purpose Sets the system time and date.

Syntax

```
#include <clib.h>
void setdate(struct date *datep);
```

Description **setdate** sets the system date to that contained in the **date** structure pointed to by *datep*.

Return value None.

See also **_dos_setdate**

setdisk

Purpose To set the current disk drive.

Syntax

```
#include <clib.h>
int setdisk(int drive);
```

Description Sets the current drive to that specified by *drive*. 0 corresponds to drive A, 1 to B, etc.

Return value None.

setmode

Purpose To set the mode of an open file to ‘binary’ or ‘text’.

Syntax

```
#include <clib.h>
int setmode(int handle,int amode)
```

Description **setmode** sets the mode of the open file corresponding to *handle* to the mode specified in *amode*. *amode* is one of **O_BINARY** or **O_TEXT** according to whether the mode is to be binary or text.

Return value **setmode** returns 0 on success. If an error occurs, **setmode** returns -1 and sets **errno** to **EINVAL** - invalid argument.

settime

Purpose To set the current system time.

Syntax

```
#include <clib.h>
void settime(struct time *timep);
```

Description **settime** sets the current system time according to the contents of the structure pointed to by *timep*.

Return value None.

See also **_dos_settime**

_strerror

Purpose Constructs a user defined error message string.

Syntax

```
#include <clib.h>
 char* _strerror(const char *s);
```

Description If *s* is NULL, the return value points to the most recently generated error message. If *s* is not NULL, the return value contains the string *s*, a colon, a space, the most recently generated system error message, and a new line. *s* should be less than 95 characters long.

Return value The **_strerror** function returns a pointer to a static buffer which contains the error message string. This buffer is overwritten on each call to **_strerror**.

strnicmp, _strnicmp

Purpose To compare up to a maximum number of characters from two NULL terminated strings without regard for case.

Syntax

```
#include <clib.h>
int strnicmp(const char *s1,const char *s2,
size_t max_len);
```

Description The **strnicmp** function compares characters in corresponding positions in the strings pointed to by *s1* and *s2* without regard for character case until elements are found to differ, or the end of one of the strings is reached, or *max_len* characters have been compared.

Return value Either **strnicmp** returns a positive or negative value, depending on whether *s1*[i]-*s2*[i] is positive or negative for the index i where elements first differ in any respect but case;

or zero if the two strings are identical up to the index *max_len*. (e.g. negative if *s1* is lexicographically less than *s2*.)

Notes **_strnicmp** is an alternative name for **strnicmp.**

See also **strncmp**

strrev, _strrev

Purpose Reverse the contents of a string.

Syntax

```
#include <clib.h>
char *strrev(char *s);
```

Description **strrev** reverses the contents of the string pointed to by *s*.

Return value **strrev** returns a pointer to the start of the reversed string.

Notes **_strrev** is an alternative name for **strrev**.

swab, _swab

Purpose To swap bytes.

Syntax

```
#include <clib.h>
void swab(char *from, char *to, int nbytes);
```

Description **swab** copies *nbytes* bytes from the array pointed to by *from* to the string pointed to by *to*. The low-byte and high-byte of each 16-bit word are swapped.

Return value None.

Notes **_swab** is an alternative name for **swab**.

toascii, __toascii

Purpose Converts characters to ASCII.

Syntax

```
#include <clib.h>
int toascii(int c);
```

Description **toascii** is a macro which converts integers into ASCII characters by masking off all but the seven least significant bits, giving a value in the range [0,127].

Return value **toascii** returns a value in the range [0,127].

Notes **__toascii** is an alternative name for **toascii**.

_tolower

Purpose To convert an upper case character to lower case.

Syntax

```
#include <clib.h>
int _tolower(int ch);
```

Description **_tolower** is a macro which takes *ch*, an upper case character, and returns the corresponding lower case character.

Return value **_tolower** returns the ASCII code for a lower case letter.

Notes If *ch* is not an upper case character, its action is undefined

_toupper

Purpose Converts an upper case character to lower case.

Syntax

```
#include <clib.h>
int _toupper(int ch);
```

Description **_toupper** is a macro which takes *ch*, an upper case ASCII character, and converts it to the corresponding lower case character.

Return value **_toupper** returns the ASCII code for an upper case letter.

Notes If *ch* is not an lower case character, its action is undefined.

ultoa, _ultoa

Purpose Convert a unsigned long number to a string.

Syntax

```
#include <clib.h>
char *ultoa(unsigned long value,
            char *string, int radix);
```

Description **ultoa** performs the same function as **utoa**. For more details see the entry for **utoa**.

Notes **_ultoa** is an alternative name for **ultoa**.

ungetch, _ungetch

Purpose To push a character back into the keyboard buffer.

Syntax

```
#include <stdio.h>
int ungetch(int ch);
```

Description **ungetch** pushes the 16 bit scan code/ASCII pair *ch* back into the keyboard buffer.   Up to 16 characters can be pushed back (512 characters if HOTKEY77 is installed).

Return value **ungetch** returns *ch* if successful, otherwise it returns **EOF**.

Notes **_ungetch** is an alternative name for **ungetch**.

See the manual for details of this DBOS function.

Example

```
//  This program uses the functions bioskey and bioscom to
//  implement a very simple algorithm which will allow
//  communication with a modem.
//
#include <stdio.h>
#include <clib.h>

#define COM1            0
#define COM2            1
#define COM_INIT        0
#define COM_SEND        1
#define COM_RECIEVE     2
#define COM_STATUS      3
#define KEY_READ        0
#define KEY_READY       1

#define COM_PORT        COM2

void main(void)
{
    int             ch;
    unsigned int    service, data, status;

    //
    //  Initialise the modem.
    //
    data = 0x3 | 0 | 0x40;
    bioscom(COM_INIT, data, COM_PORT);
    printf("Connecting to COM%d\nQ will exit\n", COM_PORT + 1);

    while (1)
        {
        //
        // Is there a character waiting to be read from the
        // modem?
        status = 0x0100 & bioscom(COM_STATUS, 0, COM_PORT);
        if (status == 0x0100)
            {
            ch = 0xff & bioscom(COM_RECIEVE, 0, COM_PORT);
            putchar(ch);
            }
        //
        //  Now check if the user has pressed a key.
        //
        if (bioskey(KEY_READY))
            {
            ch = bioskey(KEY_READ) & 0xff;
            if ((ch == 'Q') || (ch == 'q'))
                {
                printf("Goodbye\n");
                exit(0);
                }
            //
            //  Now send the data to the modem.
            //
            status = 0;
            while (status != 0x2000)
              status = 0x2000 & bioscom(COM_STATUS,0,COM_PORT);
            status = bioscom(COM_SEND, ch, COM_PORT);
            if ((status & 0x8000) == 0x8000)
                printf("Error sending data %c to modem\n", ch);
            }
        }
}
```

Example

```
//  Obtain the system information using biosequip function.
//
#include <stdio.h>
#include <clib.h>

void main(void)
{
    int result = biosequip();

    printf("System configuration = %4x:\n", result);
    if (result & 0x0020)
        printf("Coprocessor is installed\n");
      else
        printf("Coprocessor is not installed\n");
    printf("There are %d floppy disk drives\n",
                   ((result & 0x00c0) >> 6) + 1);
    printf("There are %d RS-232 ports installed\n",
                   ((result & 0x0e00) >> 9) + 1);
}
```

Example

```
//
//  Use bioskey to examime the key board status.
//
#include <stdio.h>
#include <clib.h>

void main(void)
{
    int old_result, result;

    result = bioskey(2);
    old_result = result;

    while (1)
        {
        //
        //  Decode the key - Remember, more than one state may
        //  be active at any one time.
        //
        if (result & 0x01)
            printf("Right SHIFT key pressed\n");
        if (result & 0x02)
            printf("Left SHIFT key pressed\n");
        if (result & 0x04)
            printf("CTRL key pressed\n");
        if (result & 0x08)
            printf("ALT key pressed\n");
        if (result & 0x10)
            printf("Scroll Lock pressed\n");
        if (result & 0x20)
            printf("Num Lock pressed\n");
        if (result & 0x40)
            printf("CAPS LOCK pressed\n");
        if (result & 0x80)
            printf("Insert is active\n");

        while (result == old_result)
            result = bioskey(2);
        old_result = result;
        putchar('\n');
        }
}
```

Example

```
//
//  Use biosprint to send data to a printer.
//
#include <stdio.h>
#include <clib.h>

void main(void)
{
    //
    //  Get the printer status.
    //
    printf("Waiting for the printer to come online\n");
    while (!(biosprint(2, 0, 0) & 0x7f));
    printf("Printing\n");
    //
    //  Now print the string one character at a time.
    //
    for(char *p = "This is a test printer string\r\n"; *p; p++)
        {
        biosprint(0, *p, 0);
        while (!(biosprint(2, 0, 0) & 0x7f));
        }
}
```

Example

```
//
//  This program illustrates the use of the biostime function
//  to implement a very simple profiling operation.
//
//  Two methods are used, one calculates the time from a known
//  value, the other resets the time.
//
#include <stdio.h>
#include <clib.h>

void main(void)
{
    long int    ticks, new_ticks, i;

    ticks = biostime(0, 0);
    printf("Current time is %ld\n", ticks);
    //
    //  Now for the code to time.
    //
    for (i = 0; i < 10; i++)
        printf("Hello, world\n");
    //
    //  Calculate the elapsed time.
    //
    new_ticks = biostime(0, 0);
    printf("That for loop took %g seconds\n",
           (new_ticks - ticks) / 18.2);
    //
    //  Reset the time.
    //
    biostime(1, 0);
    //
    //  Now for the same loop.
    //
    for (i = 0; i < 10; i++)
        printf("Hello, world\n");
    //
    //  Calculate the elapsed time.
    //
    ticks = biostime(0, 0);
    printf("This loop took %ld ticks\n", ticks);
```

Functions defined in dir.h

chdir	findnext	getdisk
exists	getcwd	searchpath
findfirst		

About functions defined in dir.h

The **<dir.h>** header includes prototypes for functions which relate to disc drive and directory information and includes the following structure definition.

```
struct  ffblk   {
    char        ff_reserved[21];  // reserved by DOS
    char        ff_attrib;        // DOS file attribute
    unsigned    ff_ftime;         // DOS access time
    unsigned    ff_fdate;         // DOS access date
    long        ff_fsize;         // file size
    char        ff_name[13];      // file name
    char        ff_path_name[129] // fully qualified
};                             // path name
```

chdir

Purpose To change the current directory.

Syntax

```
#include <dir.h>
int chdir(const char *path);
```

Description **chdir** changes the current working directory to that pointed to by *path*. This directory must already exist. *path* may include the corresponding drive but the active drive cannot be changed using this function.

Return value **chdir** returns zero for success and -1 for failure. The global variable **errno** is set when an error occurs.

See also **attach**

exists

Example

Purpose To test if a file with a given name already exists.

Syntax

```
#include <dir.h>
int exists(const char *pathname);
```

Description **exists** searches the current directory for a file with the given name or, if a full path name is given, the given directory is searched.

Return value **exists** returns a non-zero value for success or zero if a file with the given name cannot be found.

Example

Example

```
// exists.cpp
#include <stdio.h>                // for printf
#include <dir.h>                  // for exists
main()
{
    printf("COMMAND.COM");
    if (!exists("command.com"))
      printf("not ");
    printf("found in this directory\n");
    printf("EXISTS.CPP ");
    if (!exists("EXISTS.CPP"))
      printf("not ");
    printf("found in this directory\n");
}
```

findfirst

Example

Purpose To search a directory for a file or type of file.

Syntax

```
#include <dir.h>
int findfirst(const char *path,
              struct ffblk *f, int attrib);
```

Description **findfirst** uses the DOS function 0x4E to search for the first matching file or files given by the specification pointed to by *path*. This contains an optional drive specifier, the path, and the name of the file to be found. The file name may include wild card characters (? or *).

If a matching file is found, then the **ffblk** structure *f* is filled with the DOS file-directory information (please refer to your DOS reference manual for details).

attrib contains the DOS file attribute data which defines the type of file that is acceptable in the search. A zero value means that all files are acceptable. Otherwise the details are as follows.

bit value	**meaning**
1	read-only attribute
2	hidden file
4	system file
8	volume label
16	directory
32	archive

Return value **findfirst** returns zero for success -1 for failure. On failure, the global variable **errno** is set with a value which indicates the cause of failure.

See also **files**

findnext

Example

Purpose To continue a **findfirst** search.

Syntax

```
#include <dir.h>
int findnext(struct ffblk *f);
```

Description **findnext** can be used to find more files that match the *path* of a preceding call to **findfirst**. *f* is filled with directory information as before. One file name is returned for each call of the function until **findnext** returns -1.

Return value **findnext** returns zero on success and -1 on failure to find a matching file. On failure, the global variable **errno** is set with a value which indicates the cause of failure.

Example

```
// find.cpp
#include <stdlib.h>          // for exit
#include <stdio.h>           // for printf
#include <dir.h>             // for findfirst, findnext
main()
{
    struct ffblk file_block;
    if (findfirst("*.cpp", &file_block, 0)  0)
      {
        printf("cannot find file\n");
        exit(1);
      }
    do
      printf("File name : %s\n", file_block.ff_name);
    while (!findnext(&file_block));
}
```

getcwd

Example

Purpose To get the current working directory.

Syntax

```
#include <dir.h>
char *getcwd(char *path, int path_len);
```

Description **getcwd** gets the full path name, including the drive, of the current working directory and returns it in the string of length *path_len* pointed to by
path. *path_len* should include the terminating null.

If *path* is **NULL** then the function uses **malloc** to allocate *pathlen* bytes for the string. A pointer to this string is returned via the function name and thus the memory can be freed at a later point using **free**.

Return value If *path* is not **NULL** on input, **getcwd** returns *path* on success, **NULL** on failure. If *path* is **NULL** on input, **getcwd** returns a pointer to the allocated memory. When an error occurs, the global variable **errno** is set to indicate the cause of failure.

Example

```
// getcwd.cpp
#include <stdio.h>        // for printf
#include <dir.h>          // for getcwd
main()
{
    char buffer[80];
    if (getcwd(buffer,80))
      printf("Current working directory is %s\n", buffer);
    else
      printf("Cannot obtain current working directory.\n");
}
```

getdisk

Example

Purpose To get the current drive number.

Syntax

```
#include <dir.h>
int getdisk();
```

Return value **getdisk** returns the current drive number with 0 corresponding to drive A, 1 for B and so on.

Example

```
// getdisk.cpp
#include <stdio.h>      // for printf
#include <dir.h>        // for getdisk
main()
{
    printf("You are currently logged onto drive %c\n",
          'A' + getdisk());
}
```

searchpath

Example

Purpose To search the DOS path for a file.

Syntax

```
#include <dir.h>
char *searchpath(const char *file);
```

Description **searchpath** searches in the current directory and then along the DOS path (PATH=... in the DOS environment) for the file pointed to by *file*.

Return value When successful, **searchpath** returns a pointer to a static buffer containing the full path name which is overwritten on each call of the function.   Otherwise **searchpath** returns a **NULL** pointer.

Example

Example

```
// search.cpp
#include <stdio.h>                  // for printf etc.
#include <dir.h>                    // for searchpath
main()
{
    char  file_name[80], *path;
    printf("\n\nEnter file name : ");
    gets(file_name);
    while (*file_name)
    {
      if (path = searchpath(file_name))
          printf("File exists : %s\n", path);
      else
          printf("Cannot find file on path.\n");
      printf("\n\nEnter file name : ");
      gets(file_name);
    }
}
```

Functions defined in dos.h

int86	intdosx
int86x	getdfree
intdos	

About functions defined in dos.h

The **<dos.h>** header defines some structures and one union together with associated functions which can be used to interface with BIOS and DOS services. The details are as follows:

```
struct WORDREGS {
  unsigned short int ax;
  unsigned short int bx;
  unsigned short int cx;
  unsigned short int dx;
  unsigned short int si;
  unsigned short int di;
  unsigned short int cflag;
  unsigned short int flags;
};

struct BYTEREGS {
  unsigned char al;
  unsigned char ah;
  unsigned char bl;
  unsigned char bh;
  unsigned char cl;
  unsigned char ch;
  unsigned char dl;
  unsigned char dh;
};
union REGS {
  struct WORDREGS x;
  struct BYTEREGS h;
};
struct SREGS {
  unsigned short es;
  unsigned short cs;
  unsigned short ss;
  unsigned short ds;
};
struct  dfree {          // free disk space information
    unsigned df_avail;  // available clusters
    unsigned df_total;  // total clusters
    unsigned df_bsec;   // bytes per sector
    unsigned df_sclus;  // sectors per cluster
};
```

int86

Purpose To make a DOS/BIOS call without segment registers.

Syntax

```
#include <dos.h>
int int86(int intno,union REGS *inregs ,
                     union REGS *outregs);
```

Description **int86** performs the software interrupt given by *intno*. Register values are copied from *inregs* (excluding the flags) before the interrupt and copied to *outregs* after the interrupt. The returned flags are copied to the *x.flags* field in *outregs* and the carry flag is also copied to *x.cflag*.

Return value **int86** returns the value of AX after the interrupt.

int86x

Purpose To make a DOS/BIOS call with segment registers.

Syntax

```
#include <dos.h>
int int86x(int intno,union REGS *inregs ,
                     union REGS *outregs,
                     struct SREGS *segregs);
```

Description **int86x** performs the software interrupt given by *intno*. Register values are copied from *inregs* (excluding the flags) and from *segregs->ds* and *segregs->es* before the interrupt (*cs* and *ss* are ignored). Then the registers are copied to *outregs* and to *segregs->ds* and *segregs->es* after the interrupt. The returned flags are copied to the *x.flags* field in *outregs* and the carry flag is also copied to *x.cflag*.

Return value **int86x** returns the value of AX after the interrupt.

intdos

Purpose To make a DOS call without segment registers.

Syntax

```
#include <dos.h>
int intdos(union REGS *inregs ,
           union REGS *outregs);
```

Description **intdos** performs the software interrupt 0x21. Register values are copied from *inregs* (excluding the flags) before the interrupt and copied to *outregs* after the interrupt. The returned flags are copied to the *x.flags* field in *outregs* and the carry flag is also copied to *x.cflag*.

Return value **intdos** returns the value of AX after the interrupt.

intdosx

Purpose To make a DOS call with segment registers.

Syntax

```
#include <dos.h>
int intdosx(union REGS *inregs ,
            union REGS *outregs,
            struct SREGS *segregs);
```

Description **intdosx** performs the software interrupt 0x21. Register values are copied from *inregs* (excluding the flags) and from *segregs->ds* and *segregs->es* before the interrupt (*cs* and *ss* are ignored). Then the registers are copied to *outregs* and to *segregs->ds* and *segregs->es* after the interrupt. The returned flags are copied to the *x.flags* field in *outregs* and the carry flag is also copied to *x.cflag*.

Return value **intdosx** returns the value of AX after the interrupt.

Example See **mouse_light_pen_emulation**.

getdfree

Purpose To get disk free space information.

Syntax

```
#include <dos.h>
void getdfree(unsigned char drive,
              struct dfree *dtable);
```

Description **getdfree** gets the disk information for the drive with number *drive* (0 denotes the default drive, 1 for A and so on). The **dfree** structure is filled with the relevant data.

Return value None. In the event of an error, *df_sclus* in the **dfree** structure is set to 0xffff.