# **NAME :- ARUN A. BHOYAR**

**SUB:- ELECTRONICS**

**SEM:-VI**

**UNIT:- IV ( page 1 to 3 already taught to the students)**

**TOPIC:- FILE I/O FUNCTIONS IN C**

## Standard File I/O

The C standard library I/O functions allow you to read and write data to both files and devices. There are no predefined file structures in C, all data being treated as a sequence of bytes. These I/O functions may be broken into two different categories : stream I/O and low-level I/O.

The stream I/O functions treat a data file as a stream of individual characters. The appropriate stream function can provide buffered, formatted or unformatted input and output of data, ranging from single characters to complicate special structures. Buffering streamlines the I/O process by providing temporary storage for data which takes away the burden from the system of writing each item of data directly and instead allows the buffer to fill before causing the data to be written.

The low-level I/O system on the other hand does not perform any buffering or formatting of data --instead it makes direct use of the system's I/O capabilities to transfer usually large blocks of information.

###  Stream I/O

The C I/O system provides a consistent interface to the programmer independent of the actual device being accessed. This interface is termed a **stream** in C and the actual device is termed a **file**. A device may be a disk or tape drive, the screen, printer port, etc. but this does not bother the programmer because the stream interface is designed to be largely device independent. All I/O through the keyboard and screen that we have seen so far is in fact done through special standard streams called **stdin** and **stdout** for input and output respectively. So in essence the console functions that we have used so far such as printf(), etc. are special case versions of the file functions we will now discuss.

There are two types of streams : text and binary. These streams are basically the same in that all types of data can be transferred through them however there is one important difference between them as we will see.

#### Text Streams

A text stream is simply a sequence of characters. However the characters in the stream are open to translation or interpretation by the host environment. For example the newline character, '\n', will normally be converted into a carriage return/linefeed pair and ^Z will be interpreted as EOF. Thus the number of characters sent may not equal the number of characters received.

#### Binary Streams

A binary stream is a sequence of data comprised of bytes that will not be interfered with so that a one-to-one relationship is maintained between data sent and data received.

#### Common File Functions

|  |  |
| --- | --- |
| fopen()  | open a stream  |
| fclose() | close a stream  |
| putc()& fputc() | write a character to a stream  |
| getc()& fgetc() | read a character from a stream |
| fprintf()& fscanf  | formatted I/O |
| fgets() & fputs() | string handling |
| fseek()  | position the file pointer at a particular byte |
| feof() | tests if EOF |

#### Opening and Closing Files

A stream is associated with a specific file by performing an open operation. Once a file is opened information may be exchanged between it and your program. Each file that is opened has a unique file control structure of type FILE ( which is defined in <stdio.h> along with the prototypes for all I/O functions and constants such as EOF (-1) ). A **file pointer** is a pointer to this FILE structure which identifies a specific file and defines various things about the file including its name, read/write status, and current position. A file pointer variable is defined as follows

 FILE \*fptr ;

The fopen() function opens a stream for use and links a file with that stream returning a valid file pointer which is positioned correctly within the file if all is correct. fopen() has the following prototype

 FILE \*fopen( const char \*filename, const char \*mode );

where filename is a pointer to a string of characters which make up the name and path of the required file, and mode is a pointer to a string which specifies how the file is to be opened. The following table lists some values for mode.

|  |  |
| --- | --- |
| r | opens a text file for reading (must exist) |
| w | opens a text file for writing (overwritten or created) |
| a | append to a text file |
| rb | opens a binary file for reading |
| wb | opens a binary file for writing |
| ab | appends to a binary file |
| r+ | opens a text file for read/write (must exist) |
| w+ | opens a text file for read/write |
| a+ | append a text file for read/write |
| rb+ | opens a binary file for read/write |
| wb+ | opens a binary file for read/write |
| ab+ | append a binary file for read/write |

If fopen( ) cannot open "test.dat " it will a return a NULL pointer which should always be tested for as follows.

 FILE \*fp ;

 if ( ( fp = fopen( "test.dat", "r" ) ) == NULL )

 {

 puts( "Cannot open file") ;

 exit( 1) ;

 }

This will cause the program to be exited immediately if the file cannot be opened.

The fclose() function is used to disassociate a file from a stream and free the stream for use again.

 fclose( fp ) ;

fclose() will automatically flush any data remaining in the data buffers to the file.

#### Reading & Writing Characters

Once a file pointer has been linked to a file we can write characters to it using the fputc() function.

 fputc( ch, fp ) ;

If successful the function returns the character written otherwise EOF. Characters may be read from a file using the fgetc() standard library function.

 ch = fgetc( fp ) ;

When EOF is reached in the file fgetc( ) returns the EOF character which informs us to stop reading as there is nothing more left in the file.

For Example :- Program to copy a file byte by byte

 #include <stdio.h>

 void main()

 {

 FILE \*fin, \*fout ;

 char dest[30], source[30], ch ;

 puts( "Enter source file name" );

 gets( source );

 puts( "Enter destination file name" );

 gets( dest ) ;

 if ( ( fin = fopen( source, "rb" ) ) == NULL ) // open as binary as we don’t

 {// know what is in file

 puts( "Cannot open input file ") ;

 puts( source ) ;

 exit( 1 ) ;

 }

 if ( ( fout = fopen( dest, "wb" ) ) == NULL )

 {

 puts( "Cannot open output file ") ;

 puts( dest ) ;

 exit( 1 ) ;

 }

 while ( ( ch = fgetc( fin ) ) != EOF )

 fputc( ch , fout ) ;

 fclose( fin ) ;

 fclose( fout ) ;

 }

**NB :** When any stream I/O function such as fgetc() is called the current position of the file pointer is automatically moved on by the appropriate amount, 1 character/ byte in the case of fgetc() ;

#### Working with strings of text

This is quite similar to working with characters except that we use the functions fgets() and fputs() whose prototypes are as follows :-

 int fputs( const char \*str, FILE \*fp ) ;

 char \*fgets( char \*str, int maxlen, FILE \*fp ) ;

For Example : Program to read lines of text from the keyboard, write them to a file and then read them back again.

 #include <stdio.h>

 void main()

 {

 char file[80], string[80] ;

 FILE \*fp ;

 printf( "Enter file Name : " );

 gets( file );

 if (( fp = fopen( file, "w" )) == NULL )//open for writing

 {

 printf( "Cannot open file %s", file ) ;

 exit( 1 ) ;

 }

 while ( strlen ( gets( str ) ) > 0 )

 {

 fputs( str, fp ) ;

 fputc( '\n', fp ) ; /\* must append \n for readability -- not stored by gets() \*/

 }

 fclose( fp ) ;

 if (( fp = fopen( file, "r" )) == NULL )//open for reading

 {

 printf( "Cannot open file %s", file ) ;

 exit( 1 ) ;

 }

 while (fgets( str, 79, fptr ) != EOF )// read at most 79 characters

 puts( str ) ;

 fclose( fp ) ;

 }

#### Formatted I/O

For Example :- Program to read in a string and an integer from the keyboard, write them to a disk file and then read and display the file contents on screen.

 #include <stdio.h>

 #include <stdlib.h>

 void main()

 {

 FILE \*fp ;

 char s[80] ;

 int t ;

 if ( ( fp = fopen( "test.dat", "w" ) ) == NULL )

 {

 puts( "Cannot open file test.dat") ;

 exit(1) ;

 }

 puts( "Enter a string and a number") ;

 scanf( "%s %d", s, &t );

 fprintf( fp, "%s %d", s, t );

 fclose( fp ) ;

 if ( ( fp = fopen( "test.dat", "r" ) ) == NULL )

 {

 puts) "Cannot open file") ;

 exit(1) ;

 }

 fscanf( fp, "%s %d" , s , &t ) ;

 printf( "%s, %d\n", s, t ) ;

 fclose( fp ) ;

 }

**Note :** There are several I/O streams opened automatically at the start of every C program.

 stdin --- standard input ie. keyboard

 stdout --- standard output ie. screen

 stderr --- again the screen for use if stdout malfunctions

It is through these streams that the console functions we normally use operate. For example in reality a normal printf call such as

 printf( "%s %d", s, t ) ;

is in fact interpreted as

 fprintf( stdout, "%s %d", s, t ) ;

#### fread() and fwrite()

These two functions are used to read and write blocks of data of any type. Their prototypes are as follows where size\_t is equivalent to unsigned.

size\_t fread( void \*buffer, size\_t num\_bytes, size\_t count, FILE \*fp ) ;

size\_t fwrite( const void \*buffer, size\_t num\_bytes, size\_t count, FILE \*fp ) ;

where **buffer** is a pointer to the region in memory from which the data is to be read or written respectively, **num\_bytes** is the number of bytes in each item to be read or written, and **count** is the total number of items ( each num\_bytes long ) to be read/written. The functions return the number of items successfully read or written.

For Example :-

 #include <stdio.h>

 #include <stdlib.h>

 struct tag {

 float balance ;

 char name[ 80 ] ;

 } customer = { 123.232, "John" } ;

 void main()

 {

 FILE \*fp ;

 double d = 12.34 ;

 int i[4] = {10 , 20, 30, 40 } ;

 if ( (fp = fopen ( "test.dat", "wb+" ) ) == NULL )

 {

 puts( "Cannot open File" ) ;

 exit(1) ;

 }

 fwrite( &d, sizeof( double ), 1, fp ) ;

 fwrite( i, sizeof( int ), 4, fp ) ;

 fwrite( &customer, sizeof( struct tag ), 1, fp ) ;

 rewind( fp ) ; /\* repositions file pointer to start \*/

 fread( &d, sizeof( double ), 1, fp ) ;

 fread( i, sizeof( int ), 4, fp ) ;

 fread( &customer, sizeof( struct tag ), 1, fp ) ;

 fclose( fp ) ;

 }

**NB :** Unlike all the other functions we have encountered so far fread and fwrite read and write **binary** data in the same format as it is stored in memory so if we try to edit one these files it will appear completely garbled. Functions like fprintf, fgets, etc. read and write displayable data. fprintf will write a double as a series of digits while fwrite will transfer the contents of the 8 bytes of memory where the double is stored directly.

#### Random Access I/O

The fseek() function is used in C to perform random access I/O and has the following prototype.

 int fseek ( FILE \*fp, long num\_bytes, int origin ) ;

where **origin** specifies one of the following positions as the origin in the operation

 SEEK\_SET --- beginning of file

 SEEK\_CUR --- current position

 SEEK\_END --- EOF

and where **num\_bytes** is the offset in bytes to the required position in the file. fseek() returns zero when successful, otherwise a non-zero value.

For Example if we had opened a file which stored an array of integers and we wish to read the 50th value we might do the following

 fseek ( fp, ( 49 \* sizeof( int ) ), SEEK\_SET ) ;

 fscanf( fp, "%d", &i ) ;

from anywhere in the program.

###  Low -- Level I/O

Low level file input and output in C does not perform any formatting or buffering of data whatsoever, transferring blocks of anonymous data instead by making use of the underlying operating system's capabilities.

Low level I/O makes use of a file handle or descriptor, which is just a non-negative integer, to uniquely identify a file instead of using a pointer to the FILE structure as in the case of stream I/O.

As in the case of stream I/O a number of standard files are opened automatically :-

**standard input --- 0**

**standard output --- 1**

**standard error --- 2**

The following table lists some of the more common low level I/O functions, whose prototypes are given in <io.h> and some associated constants are contained in <fcntl.h> and <sys\stat.h>.

|  |  |
| --- | --- |
| open() | opens a disk file |
| close() | closes a disk file |
| read() | reads a buffer of data from disk |
| write() | writes a buffer of data to disk |

The open function has the following prototype and returns -1 if the open operation fails.

 int open ( char \*filename, int oflag [, int pmode] ) ;

where filename is the name of the file to be opened, oflag specifies the type of operations that are to be allowed on the file, and pmode specifies how a file is to be created if it does not exist.

**oflag** may be any logical combination of the following constants which are just *bit flags* combined using the bitwise OR operator.

|  |  |
| --- | --- |
| O\_APPEND | appends to end of file |
| O\_BINARY | binary mode |
| O\_CREAT | creates a new file if it doesn't exist |
| O\_RDONLY | read only access |
| O\_RDWR | read write access |
| O\_TEXT | text mode |
| O\_TRUNC | truncates file to zero length |
| O\_WRONLY | write only access  |

**pmode** is only used when O\_CREAT is specified as part of oflag and may be one of the following values

**S\_IWRITE**

**S\_IREAD**

**S\_IREAD | S\_IWRITE**

This will actually set the read / write access permission of the file at the operating system level permanently unlike oflag which specifies read / write access just while your program uses the file.

The close() function has the following prototype

 int close ( int handle ) ;

and closes the file associated with the specific handle.

The read() and write() functions have the following prototypes

 int read( int handle, void \*buffer, unsigned int count ) ;

 int write( int handle, void \*buffer, unsigned int count ) ;

where handle refers to a specific file opened with open(), buffer is the storage location for the data ( of any type ) and count is the maximum number of bytes to be read in the case of read() or the maximum number of bytes written in the case of write(). The function returns the number of bytes actually read or written or -1 if an error occurred during the operation.

Example : Program to read the first 1000 characters from a file and copy them to another.

#include <io.h>

#include <fcntl.h>

#include <sys\stat.h>

void main()

{

char buff[1000] ;

int handle ;

handle=open(" test.dat", O\_BINARY|O\_RDONLY, S\_IREAD | S\_IWRITE );

if ( handle == -1 ) return ;

if ( read( handle, buff, 1000 ) == 1000 )

 puts( "Read successful");

else

 {

 puts( Read failed" ) ;

 exit( 1 );

}

close( handle ) ;

handle = open("test.bak",

 O\_BINARY|O\_CREAT|O\_WRONLY| O\_TRUNC,

 S\_IREAD | S\_IWRITE ) ;

if ( write( handle, buff, 1000 ) == 1000 )

 puts( "Write successful") ;

else

 {

 puts( "Write Failed") ;

 exit( 1 ) ;

 }

close( handle ) ;

}

Low level file I/O also provides a seek function **lseek** with the following prototype.

long \_lseek( int *handle*, long *offset*, int *origin* );

**\_lseek** uses the same origin etc. as *fseek()* but unlike fseek() returns the offset, in bytes, of the new file position from the beginning of the file or -1 if an error occurs.

For Example : Program to determine the size in bytes of a file.

 #include <stdio.h>

 #include <io.h>

 #include <fcntl.h>

 #include <sys\stat.h>

 void main()

 {

int handle ;

 long length ;

 char name[80] ;

 printf( “Enter file name : ” ) ;

 gets( name ) ;

 handle=open( name,O\_BINARY| O\_RDONLY, S\_IREAD | S\_IWRITE );

 lseek( handle, 0L, SEEK\_SET ) ;

 length = lseek( handle, 0L, SEEK\_END ) ;

 close( handle ) ;

 printf( “The length of %s is %ld bytes \n”, name, length ) ;

 }