Files

A files is a collection of related data placed on the disk.

The file handling in C can be broadly category in two types:

1. High Level (standard files or stream oriented files).
2. Low level(system oriented files).

- High level file handling is managed by library functions while low level file handling is managed by system calls.
- The high level handling is commonly used since it is easier to manage and hides most of the details from the programmer.

We will discuss here high level file handling only.

There are two ways of storing data in files:

Binary format

Text format

In text format ,data is stored as lines of characters with each line terminated by a newline('\n').

In binary format ,data is stored on the disk in the same way as it is represented in the computer memory.

Text files are in human readable form and they can be created and read using any text editor.

Binary files are not in a human readable form and they can be created and read only by specific programs written for them .

The binary data stored in the file can't be read using a text editor.

- The integer 1679 will take only 2 bytes in a binary file while it will occupy 4 bytes in a text file because in binary file it is stored as an integer while in text file it is stored in a sequence of 4 characters i.e '1' ,'6', '7', '9' .
- The hexadecimal value of 1679 is 0x068F, so in binary format it is represented by the two bytes 0x06 and 0x8F.
- In a text file, this number 1679 is represented by bytes 0x31,0x36, 0x37,0x39(ASCII values.)

Binary – 0000 0110 1000 1111

Text – 0011 0001 0011 0110 0011 0111 0011 1001

Both text files and binary files keep record of the length of the file, and identify the end of file when this length is reached.

In text files, there is one more way to detect the end of file.

The character with ASCII value 26 is considered to be end of file character in text files .

All input functions stop reading from a text file when this character is encountered and return an end of file signal to the program.

In binary files no such character represents the end of file.

The input and output operations in binary files take less time as compared to that of text files because in binary files no conversions have to take place.

However the data written using binary format is not very portable since the size of data types and byte order may be different on different machines.

In text format ,these problems do not arise so it is considered more portable.

Concept of Buffer

- Buffer is an area in memory where the data is temporarily stored before being written to the file.
- When we open a file, a buffer is automatically associated with its file pointer.
- Whatever data we send to the file is not immediately written to the file.
- First it is sent to the buffer and when the buffer is full then contents are written to the file.
- When the file is closed ,all the contents of buffer are automatically written to the file.
- This is called flushing the buffer,we can also explicitly flush the buffer by a function **fflush()** .
- The concept of buffer is used to increase efficiency.
- Had there been no buffer we would have to access the disk each time for writing even single byte of data.
- This would have taken lot of time because each time the disk is accessed ,the read/write head has to be repositioned.
- When buffereing is done,the data is collected in the buffer and data equal to the size of buffer is written to the file at a time,so the number of times disk is accessed decreases,which improves the efficiency.

The steps for file operations in C programming are as follows:

Open a file

Read the file or write data in the file

Close the file

The function fopen() and fclose() are used for opening and closing the files respectively.

Opening a Files

A files must be opened before any I/O operations can be performed on that file.

The process of establishing a connection between the program and file is called opening the file.

A structure named FILE is defined in the file stdio.h that contains all information about the file like name, status , size ,current position ,end of file status etc..

All these details are hidden from the programmer and the operating system takes care of all these things.

```
type struct{
        ……
        …….
            } FILE;
```

A file pointer is a pointer to structure of type FILE .

Whenever a file is opened , a structure of type FILE is associated with it ,and a file pointer that points to this structure identifies this file. The function fopen() is used to open a file.

Declaration:

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

fopen() function takes two strings as arguments, the first one is the name of the file to be opened and the second one is the mode that decides which operations(read, write, append etc.) are to be performed on file.

On success , fopen() returns returns a pointer of type FILE and on error it returns NULL

The return value of fopen() is assigned to a FILE pointer declared previously.

FILE *fp1,*fp2;

fp1=fopen(“myfile.txt”,”w”);

fp2=fopen(“yourfile.dat”,”r”);

The name of file is limited to FILENAME_MAX characters.

After opening the file with fopen(), the name of file is not used in the program for any operation on it.

Whenever we have to perform any operation on the file , we will use the file pointer returned by fopen() function.

So the name of file is sometimes known as external name,while the file pointer associated with it is known as its internal name.

The second argument represents the mode in which the file is to be opened .

FILE OPEN WITH fopen()

To open a file, the function fopen() is used.

This function returns a pointer to a file.

Syntax: file_pointer = fopen(“datafile”,”mode”);

Before this assignment, file_pointer and fopen() should be declared as FILE pointer type variables.

file_pointer: This is a logical name given to the data file and throughout the program, the datafile (physical file) is referred by this file pointer.

datafile: A filename in which data is stored or retrieved. This is the physical file name for the data in the secondary storage device.

mode: A file in C can be opened in various way. This mode decides the read/write operations with the data file

```
FILE *fp;
fp = fopen(“filename”,”mode”);
```

The first statement declares the variable fp as a “pointer to data type FILE”.

FILE is a structure that is defined in the I/O library.

The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp.

This pointer ,which contains all the information about the file is subsequently used as a communication link between the system and the program.

FILE OPEN MODES IN fopen()

w (write) mode: Write only

a (append) mode: appending/adding data

r (read) mode: read only

w+ (write + read) mode: same as w except both for reading and writing.

a+ (append+ read): same as a except both for reading and writing.

r+ (read+ write) mode: The existing file is opened to the beginning for both reading and writing.

When trying to open a file , one of the following things may happen:

1. When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.

2. When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exists.

3. When the mode is 'reading', and if it exists, then the file is opened with the current contents safe otherwise an error occurs.

```
FILE *p1,*p2;
p1 = fopen("data", "r");
p2= fopen("results", "w");
```

The file data is opened for reading and results is opened for writing .

In case, the results file already exists, its content are deleted and the file is opened as a new file.

Closing a FILE

A file must be closed as soon as all operations on it have been completed.

This ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken.

Another instance where we have to close a file is when we want to reopen the same file in a different mode.

The I/O library supports a function to do this for us.

It takes the following form:

```
fclose(file_pointer);
```

This would close the file associated with the FILE pointer file_pointer.

```
……..
……..
FILE *p1,*p2;
p1=fopen(“INPUT”,“w”);
p2=fopen(“OUTPUT”,“r”);
……..
……..
fclose(p1);
fclose(p2);
……
```

This program opens two files and closes them after all operations on them are completed. Once a file is closed, its file pointer can be reused for another file.

As a matter of fact all files are closed automatically whenever a program terminates.

INPUT/OUTPUT OPERATIONS ON FILES

- getc and putc
- getw and putw
- fprint and fscanf

The getc and putc Functions

The simplest file I/O functions are getc and putc.

These are analogous to getchar and putchar functions and handle one character at a time.

Assume that a file is opened with mode **w** and file pointer **fp1**.

Then, the statement becomes

putc(c,fp1);

writes the character contained in the character variable c to the file associated with **FILE** pointer **fp1 .**

Similarly **getc** is used to read a character from a file that has been opened in read mode.

c=getc(fp2); would read a character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of getc or putc.

The getc will return an end-of-file ,when end of file has been reached .

Therefore the reading must be terminated when EOF is encountered .

WAP to read data from the keyboard ,write it to a file called INPUT, again read the same data from the INPUT file, and display it on the screen.

A program and the related input and output data.

We enter the input data via the keyboard and the program writes it character by character to the file INPUT.

The end of the data is indicated by entering an EOF character, which is control-Z in the reference system.

```
#include<stdio.h>
      main()
      {
      File   *f1;
      char c;
      printf("Data Input\n\n");
      f1=fopen("INPUT","w");                /*Open the file INPUT*/
      while(c=getchar()!=EOF)               /* Get a character from keyboard*/
      putc(c,f1);                           /* Write a character  to INPUT */
      fclose(f1);                           /*Close the file INPUT */
       printf("Data  Output\n\n");
       f1=fopen("INPUT","r");               /*Reopen the file INPUT*/
       while(c=getchar(f1)!=EOF)            /* Read a character from INPUT*/
      printf("%c",c);                       /* Display a character on screen*/
      fclose(f1);                            /*Close the file INPUT */
      }
```

Output
Data Input
This is a program to test the file handling
Features on this system^Z

Data Output
This is a program to test the file handling
Features on this system

The getw and putw Functions

The getw and putw are integer-oriented functions.

They are similar to the getc and putc functions and are used to read and write integer values.

These functions would be useful when we deal with only integer data.

The general forms of getw and putw are:

putw(***integer, fp***);

getw(***fp***);

A file named DATA contains a series of integer numbers. Code a program to read these numbers and then write all ‘odd’ numbers to a file to be calle ODD and all ‘even’ numbers to a file to be called EVEN.

```
#include<stdio.h>
    main()
    {
    File   *f1,*f2,*f3;
    int number, i;
    printf(“Contents of Data file\n\n”);
    f1=fopen(“DATA”,“w”); /* Create DATA file*/
    for(i=1;i<=30;i++)
    {
            scanf(“%d”,&number);
            if(number==-1)
            break;
            putw(number,f1);
    }
    fclose(f1);

    f1=open(“DATA”,“r”);
    f2=open(“ODD”,“w”);
    f3=open(“EVEN”,“w”);
```

```
while((number=getw(f1)!=EOF)
    {
        if(number%2==0)
        putw(number,f3);
        else
        putw(number,f2);
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
    f2=fopen("ODD","r");
    f3=fopen("EVEN","r");
    printf("\n\n Contents of ODD file\n\n);
    while((number=getw(f2)!=EOF)
        printf("%4d",number);
    printf("\n\n Contents of EVEN file\n\n);
    while((number=getw(f3)!=EOF)
        printf("%4d",number);
    fclose(f2);
    fclose(f3);
    }
```

Output

Content of DATA file

111 222 333 444 555 666 777 888 999 000 121 232 343 454 565 -1

Content of ODD file

111 333 555 777 999 121 343 565

Content of DATA file

222 444 666 888 0 232 454

The fprintf and fscanf Functions

So far we have seen functions, that can handle only one character or integer at a time.

The functions fprintf and fscanf perform I/O operations that are identical to printf and scanf functions, except of course that they work on files.

The general form of fprintf is

fprintf(fp,"control string",list);

where fp is a file pointer associated with a file that has been opened for writing.

The control string contains output specifications for the items in the list. The list may include variables, constants and strings.

fprintf (f1,"%s %d %f",name,age,7.5);

The general format of fscanf is

fscanf(fp,"control string",list);

fscanf(f2,"%s %d", item ,&quantity);

Write a program to open a file named INVENTORY and store in it the following data:

Item Name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Extend the program to read this data from the file INVENTORY and display the inventory table with the value of each item.

```
#include<stdio.h>
    main()
    {
    FILE *fp;
    int number, quantity, i;
    float price,value;
    char item[10], filename[10];
    printf(“Input filename\n”);
    scanf(“%s”,filename);
    fp=fopen(“filename”,”w”);
    printf(“Input inventory data\n\n”);
    printf(“Item name  Number  Price   Quantity\n”);
    for(i=1;i<=3;i++)
    {
          fscanf(stdin,”%s%d%f%d”, item , &number, &price,&quantity);
          fprintf(fp,”%s %d %.2f %d”,item,number,price,quantity);
    }
    fclose(fp);
    fprintf(stdout,”\n\n”);
```

```
    fp=fopen(filename,”r”);
     printf(“Item name  Number  Price   Quantity   Value\n”);
    for(i=1;i<=3;i++)
    {
          fscanf(fp,”%s%d%f%d”, item , &number, &price,&quantity);
          value=price*quantity;
          fprintf(stdout,”%s %d %.2f %d %11.2f”,item , number,
    price,quantity,value);
    }
    fclose(fp);
}
```

Output

Input file name

INVENTORY

Input inventory data

Item Name	Number	Price	Quantity
AAA-1	111	17.50	115
BBB-2	125	36.00	75
C-3	247	31.75	104

Item Name	Number	Price	Quantity	value
AAA-1	111	17.50	115	2012.50
BBB-2	125	36.00	75	2700.00
C-3	247	31.75	104	3302.00

Error Handling During I/O Operations

It is possible that an error may occur during I/O operations on a file.Typical error situations include:

1. Trying to read beyond the end-of-file mark
2. Device overflow
3. Trying to use a file tat has not been opened.
4. Trying to perform an operation on afile,when the file is opened for another type of operation.
5. Opening a file with an invalid filename.
6. Attempting to write to a write-protected file.

feof and ferror that can help us detect I/O errors in the files .

The **feof** function can be used to test for an end of file condition.

It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read,and returns zero otherwise.

If fp is a pointer to file that has just been opened for reading, then the statement

```
if(feof(fp))
printf("End of Data.\n");
```

Would display the message "End of Data." on reaching the end of file condition.

The **ferror** function reports the status of the file .

It also takes a FILE pointer as its argument and returns a nonzero integer value if an error has been read etected up to that point , during processing . It returns zero otherwise.

If fp is a pointer to file that has just been opened for reading, then the statement

```
if(ferror(fp)!=0)
printf("An error has occured.\n");
```

Would display the error message , if the reading is not successful.

We know that whenever the file is opened using fopen function,a file pointer is returned.

If the file cannot be opened then the function returns he NULL pointer .

This facility is can be used to test whether a file has been opened or not.

```
if(fp==NULL)
printf("File coud not be opened.\n");
```

WAP to illustrate error handling in file operations.

```
#include<stdio.h>
main()
{
char  *filename;
FILE *fp1,*fp2;
int i,number;
fp1=fopen("TEST","w");
for(i=10;i<=100;i+=10)
putw(i,fp1);
fclose(fp1);
printf("\n Input Filename\n");
open_file:
      scanf("%s",filename);
if((fp2=fopen(filename,"r"))==NULL)
{
 printf("\n Cannot open the  file\n");
 printf("\n Type filename again\n");
goto open_file;
}
```

```
else
        for(i=1;i<=20;i++)
        {
                number=getw(fp2);
                if(feof(fp2))
                {
                        printf("\n Ran out of Data");
                        break;
                }
        fclose(fp2);
        }
```

Output

```
Input Filename
TETS
Cannot open the file
Type filename again
TEST
10
20
30
40
50
60
70
80
90
100
```

Random Access To Files

Whenever we are interested in accessing only a particular part of a file and not in reading the other parts of the file then we use the functions **fseek, ftell** and **rewind** available in the I/O library.

ftell takes a file pointer and return a number of type long that corresponds to the current position.

This function is useful in saving the current position of a file, which can be used later in the program.

It takes the following form:

n=ftell(fp);

n would give the relative offset(in bytes) of the current position.

This means that n bytes have already been read (or written).

rewind takes a file pointer and resets the position to start of the file.

For, ex: rewind(fp);

n=ftell(fp);

Remember, the first byte in the file is numbered as 0, second as 1, and so on .

This function helps us in reading a file more than once, without having to close and open the file.

Remember that whenever a file is opened for reading or writing , arewind is done implicitly.

fseek function is used to move the file position to a desired location within the file.

It takes the following form:

fseek (file_ptr, offset,position);

file_ptr *is a pointer to the file concerned,* ***offset*** *is a number or variable of type long,and* ***position*** *is an integer number.*

The offset specifies the number of positions(bytes) to be moved from the location specified by position.

The position can take one of the following three values:

Value	Meaning
0	Beginning of file
1	Current position
2	End of file

The offset may be positive, meaning move forward, or negative, meaning move backward.

Operations of **fseek** Function

Statement	Meaning
fseek(fp,0L,0)	Go to the beginning(similar to the beginning).
fssek(fp,0L,1)	Stay at the current position(rarely used).
fseek(fp,0L,2)	Go to the end of the file, past the last character of the file.
fseek(fp,m,0)	Move to (m+1)th byte in the file.
fseek(fp,m,1)	Go forward by m bytes.
fseek(fp,-m,1)	Go backward by m bytes from the current position.
fseek(fp,-m,2)	Go backward by m bytes from the end.

When the operation is successful, fseek returns a zero. If we attempt to move the file pointer beyond the file boundaries, an eror occurs and fseek returns -1.

It is good practice to check whether an error has occurred or not, before proceeding further.

Write a program that uses that uses the function ftell and fseek.

We have created a file RANDOM with following contents:

Position----------→ 0 1 2 ……25

Character Stored----------→ A B C……..Z

We are reading the file twice .

First we are reading the content of every fifth position and printing its value along with its position on the screen.

The second time, we are reading the contents of the file from the end and printing the same on the screen.

During the first reading the file pointer crosses the end-of-file mark when the parameter **n** of **fseek(fp,n,0)** becomes 30.

Therefore, after printing the content of position 30,the the loop is terminated.

For reading the file from the end,we use the statement

fseek(fp,-1L,2); to position the file pointer to the last character.

Since every read causes the position to move forward by one position, we have to move back by two positions to read the next character.

fseek(fp,-2L,1); in the while statement also tests whether the file pointer has crossed the file boundary or not.

```
#include<stdio.h>
    main()
    {
    FILE *fp;
    long n;
    char c;
    fp=fopen("RANDOM","w");
    while((c=getchar())!=EOF)
    putc(c,fp);
    printf("No. of characters entered=%ld\n",ftell(fp));
     fclose(fp);
     fp=fopen("RANDOM","r");
    n=0L;
     while(feof(fp)==0)
    {
            fseek(fp,n,0);
            printf("position of %c is %ld\n",getc(fp),ftell(fp));
            n=n+5L;
    }
    putchar('\n');
```

```
    fseek(fp,-1L,2);
    do
    {
        putchar(getc(fp));
    }
    while(!fseek(fp,-2L,1));
    fclose(fp);
}
```

Output

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ^Z
No. of characters entered =26
Position of A is 0
Position of F is 5
Position of K is 10
Position of P is 15
Position of U is 20
Position of Z is 25
Position of     is 30
ZYXWVUTSRQPONMLKJIHGFEDCBA
```