UNIT-6

File Organization in Data Structure

FILES -CONTENTS

- File organization,
- examples of using file,
- file access methods ,
- Hashing and collision resolution techniques

What is File?

A file is a collection of data stored on a secondary storage device like hard disk.

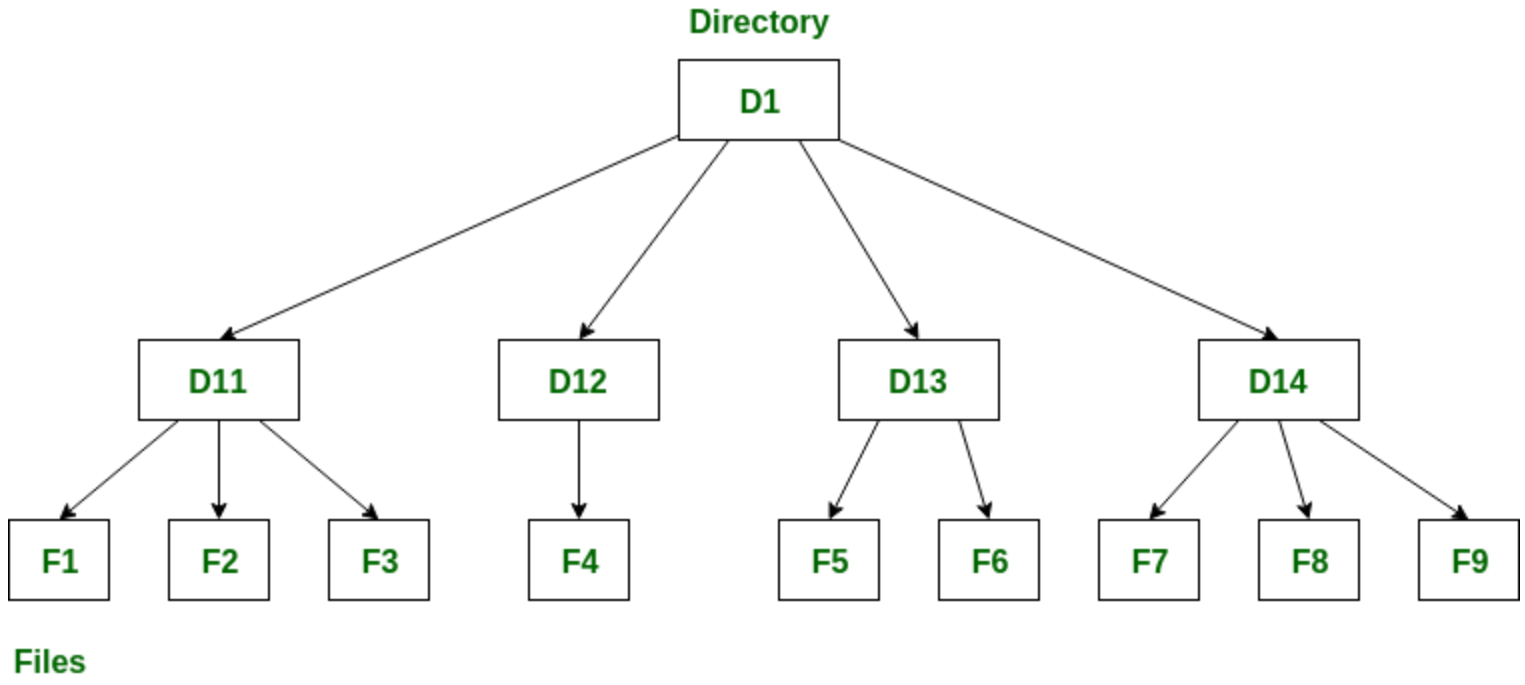
A file is basically used because real life applications involve large amounts of data and in such situations the console oriented I/O operations pose two major problems:

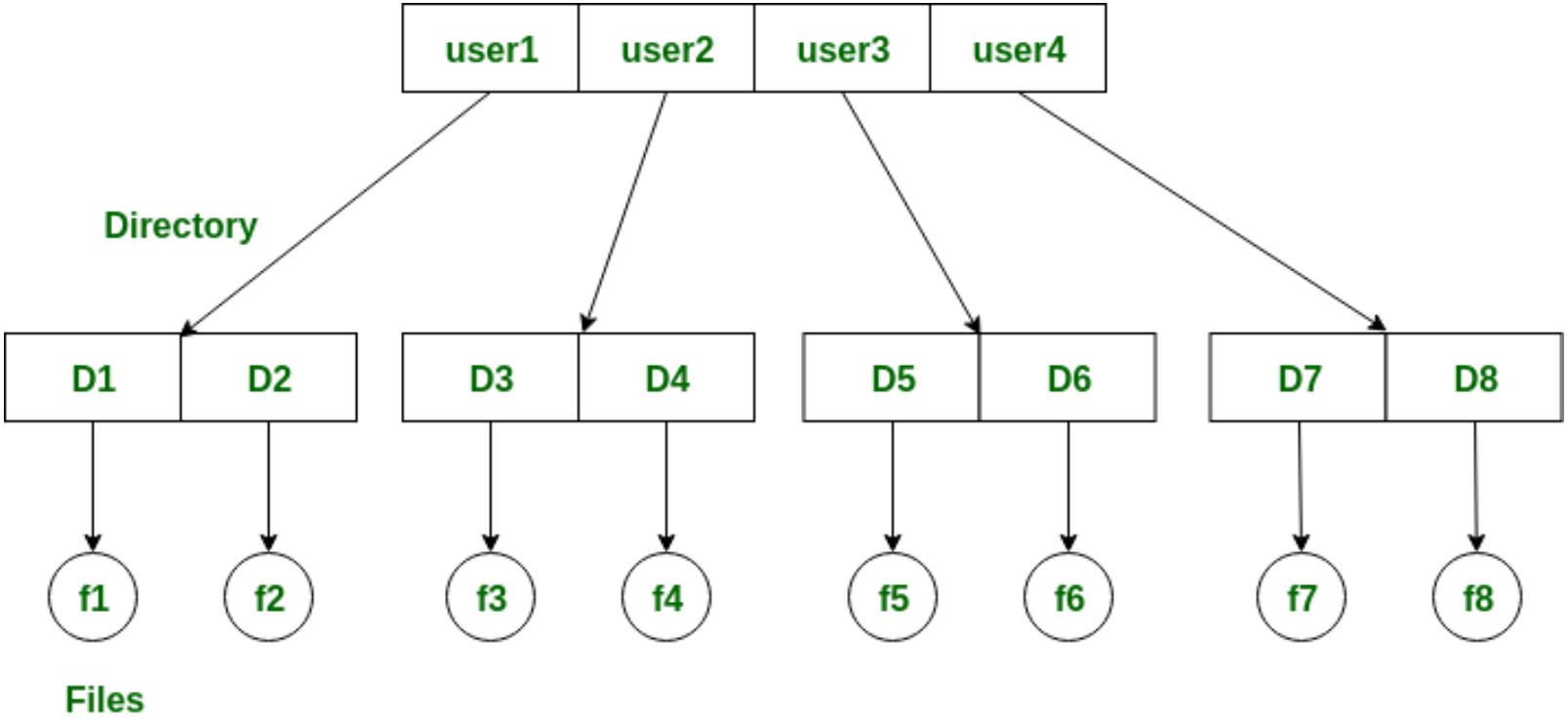
- First, it becomes cumbersome and time consuming to handle huge amount of data through terminals.
- Second, when doing I/O using terminal, the entire data is lost when either the program is terminated or computer is turned off. Therefore, it becomes necessary to store data on a permanent storage (the disks) and read whenever necessary, without destroying the data.

File is a collection of records related to each other. The file size is limited by the size of memory and storage medium.

Structures of Directory in Operating System

- A **directory** is a container that is used to contain folders and files. It organizes files and folders in a hierarchical manner.




user1
user2
user3
user4
Directory
D1
D2
D3
D4
D5
D6
D7
D8
f1
f2
f3
f4
f5
f6
f7
f8
Files

Physical versus Logical Files

- ***Physical File***: A collection of bytes stored on a disk or tape.
- ***Logical File***: A "Channel" (like a telephone line) that hides the details of the file's location and physical format to the program.

When a program wants to use a particular file, "data", the operating system must find the physical file called "data" and make the hookup by assigning a logical file to it. This logical file has a logical name which is what is used inside the program.

File Organization

- **File organization ensures that records are available for processing. It is used to determine an efficient file organization for each base relation.**

- *For example, if we want to retrieve employee records in alphabetical order of name. Sorting the file by employee name is a good file organization.*
- *However, if we want to retrieve all employees whose marks are in a certain range, a file is ordered by employee name would not be a good file organization.*

Types of File Organization

- **There are three types of organizing the file:**

  1. Sequential access file organization
  2. Direct access file organization
  3. Indexed sequential access file organization

1. Sequential access file organization

Storing and sorting in contiguous block within files on tape or disk is called as **sequential access file organization**.

- In sequential access file organization, all records are stored in a sequential order. The records are arranged in the ascending or descending order of a key field.
- Sequential file search starts from the beginning of the file and the records can be added at the end of the file.
- In sequential file, it is not possible to add a record in the middle of the file without rewriting the file.

Advantages of sequential file

- It is simple to program and easy to design.
- Sequential file is best used if sufficient storage space.

Disadvantages of sequential file

- Sequential file is time consuming process.
- It has high data redundancy.
- Random searching is not possible.

2. Direct access file organization

- Direct access file is also known as random access or relative file organization.
- In direct access file, all records are stored in direct access storage device (DASD), such as hard disk. The records are randomly placed throughout the file.
- The records does not need to be in sequence because they are updated directly and rewritten back in the same location.
- This file organization is useful for immediate access to large amount of information. It is used in accessing large databases.
- It is also called as hashing.

Advantages of direct access file organization
Direct access file helps in online transaction processing system (OLTP) like online railway reservation system.

- In direct access file, sorting of the records are not required.
- It accesses the desired records immediately.
- It updates several files quickly.
- It has better control over record allocation.

Disadvantages of direct access file organization

- Direct access file does not provide back up facility.
- It is expensive.
- It has less storage space as compared to sequential file.

3. Indexed sequential access file organization

- Indexed sequential access file combines both sequential file and direct access file organization.
- In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.

Continued..

- This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.

- The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.

Advantages of Indexed sequential access file organization

- In indexed sequential access file, sequential file and random file access is possible.
- It accesses the records very fast if the index table is properly organized.
- The records can be inserted in the middle of the file.
- It provides quick access for sequential and direct processing.
- It reduces the degree of the sequential search.

Disadvantages of Indexed sequential access file organization

- Indexed sequential access file requires unique keys and periodic reorganization.
- Indexed sequential access file takes longer time to search the index for the data access or retrieval.
- It requires more storage space.
- It is expensive because it requires special software.
- It is less efficient in the use of storage space as compared to other file organizations.

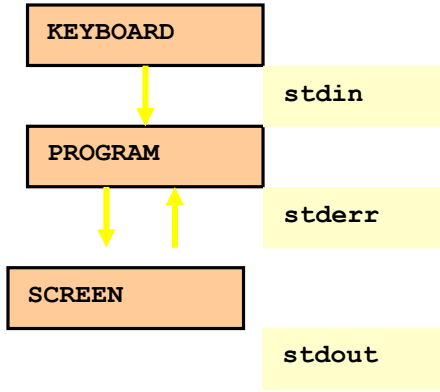
INTRODUCTION TO FILES

STREAMS IN C

- In C, the standard streams are termed as pre-connected input and output channels between a text terminal and the program (when it begins execution). Therefore, stream is a logical interface to the devices that are connected to the computer.
- Stream is widely used as a logical interface to a file where a file can refer to a disk file, the computer screen, keyboard, etc. Although files may differ in the form and capabilities, all streams are the same.
- The three standard streams in C languages are- standard input (stdin), standard output (stdout) and standard error (stderr).

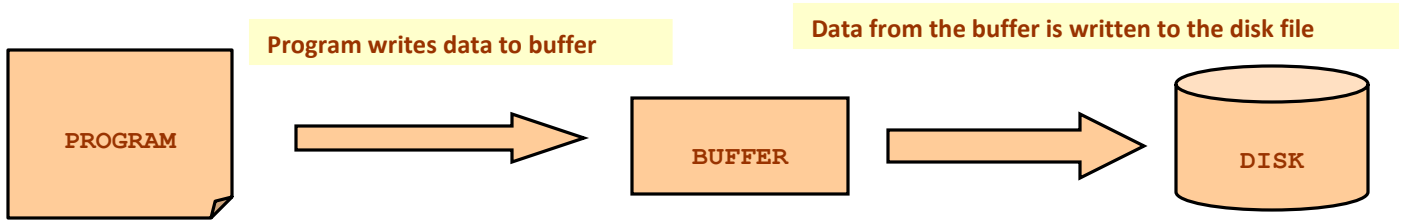
STREAMS IN C contd.

- **Standard input (stdin):** Standard input is the stream from which the program receives its data. The program requests transfer of data using the *read* operation. However, not all programs require input. Generally, unless redirected, input for a program is expected from the keyboard.
- **Standard output (stdout):** Standard output is the stream where a program writes its output data. The program requests data transfer using the *write* operation. However, not all programs generate output.
- **Standard error (stderr):** Standard error is basically an output stream used by programs to report error messages or diagnostics. It is a stream independent of standard output and can be redirected separately. No doubt, the *standard output* and *standard error* can also be directed to the same destination.
- A stream is linked to a file using an open operation and disassociated from a file using a close operation.



BUFFER ASSOCIATED WITH FILE STREAM

- When a stream linked to a disk file is created, a buffer is automatically created and associated with the stream. A buffer is nothing but a block of memory that is used for temporary storage of data that has to be read from or written to a file.
- Buffers are needed because disk drives are block oriented devices as they can operate efficiently when data has to be read/ written in blocks of certain size. The size of ideal buffer size is hardware dependant.
- The buffer acts as an interface between the stream (which is character-oriented) and the disk hardware (which is block oriented). When the program has to write data to the stream, it is saved in the buffer till it is full. Then the entire contents of the buffer are written to the disk as a block.



Similarly, when reading data from a disk file, the data is read as a block from the file and written into the buffer. The program reads data from the buffer. The creation and operation of the buffer is automatically handled by the operating system. However, C provides some functions for buffer manipulation. The data resides in the buffer until the buffer is flushed or written to a file.

USING FILES IN C

To use files in C, we must follow the steps given below:

Declare a file pointer variable
Open the file
Process the file
Close the file

Declaring a file pointer variable

There can be a number of files on the disk. In order to access a particular file, you must specify the name of the file that has to be used. This is accomplished by using a file pointer variable that points to a structure FILE (defined in stdio.h). The file pointer will then be used in all subsequent operations in the file. The syntax for declaring a file pointer is

FILE *file_pointer_name;

For example, if we write

FILE *fp;

Then, fp is declared as a file pointer.

An error will be generated if you use the filename to access a file rather than the file pointer

Opening a File

- A file must be first opened before data can be read from it or written to it. In order to open a file and associate it with a stream, the fopen() function is used. The prototype of fopen() can be given as:
- FILE *fopen(const char *file_name, const char *mode);
- Using the above prototype, the file whose pathname is the string pointed to by file_name is opened in the mode specified using the mode. If successful, fopen() returns a pointer-to-structure and if it fails, it returns NULL.

MODE	DESCRIPTION
r	Open a text file for reading. If the stream (file) does not exist then an error will be reported.
w	Open a text file for writing. If the stream does not exist then it is created otherwise if the file already exists, then its contents would be deleted
a	Append to a text file. if the file does not exist, it is created.
rb	Open a binary file for reading. B indicates binary. By default this will be a sequential file in Media 4 format
wb	Open a binary file for writing
ab	Append to a binary file
r+	Open a text file for both reading and writing. The stream will be positioned at the beginning of the file. When you specify "r+", you indicate that you want to read the file before you write to it. Thus the file must already exist.
w+	Open a text file for both reading and writing. The stream will be created if it does not exist, and will be truncated if it exist.
a+	Open a text file for both reading and writing. The stream will be positioned at the end of the file content.
r+b/ rb+	Open a binary file for read/write
w+b/wb+	Create a binary file for read/write
a+b/ab+	Append a binary file for read/write

OPENINING A FILE contd.

The fopen() can fail to open the specified file under certain conditions that are listed below:
Opening a file that is not ready for usage
Opening a file that is specified to be on a non-existent directory/drive
Opening a non-existent file for reading
Opening a file to which access is not permitted

```
        FILE *fp;
fp = fopen("Student.DAT", "r");
if(fp==NULL)
{
        printf("\n The file could not be opened");
        exit(1);
}
OR
char filename[30];
FILE *fp;
gets(filename);
fp = fopen(filename, "r+");
if(fp==NULL)
{
        printf("\n The file could not be opened");
        exit(1);
}
```

CLOSING A FILE USING FCLOSE()

- To close an open file, the fclose() function is used which disconnects a file pointer from a file. After the fclose() has disconnected the file pointer from the file, the pointer can be used to access a different file or the same file but in a different mode.
- The fclose() function not only closes the file but also flushed all the buffers that are maintained for that file
- If you do not close a file after using it, the system closes it automatically when the program exits. However, since there is a limit on the number of files which can be open simultaneously; the programmer must close a file when it has been used. The prototype of the fclose() function can be given as,
- fclose(fp);
- Here, fp is a file pointer which points to the file that has to be closed. The function returns an integer value which indicates whether the fclose() was successful or not. A zero is returned if the function was successful; and a non-zero value is returned if an error occurred.

READ DATA FROM FILES

C provides the following set of functions to read data from a file.
fscanf() fgets() fgetc() fread()

fscanf()- The fscanf() is used to read formatted data from the stream. The syntax of the fscanf() can be given as,

int fscanf(FILE *stream, const char *format,…);

The fscanf() is used to read data from the *stream* and store them according to the parameter *format* into the locations pointed by the additional arguments.

```
#include<stdio.h>
main()
{               FILE *fp;
                char name[80];
        int roll_no;
        fp = fopen("Student.DAT", "r");
                if(fp==NULL)
                {                               printf("\n The file could not be opened");
                                exit(1);
                 }
                 printf("\n Enter the name and roll number of the student : ");
                  fscanf(stdin, "%s %d", name, &roll_no);  /* read from keyboard */
                 printf("\n NAME : %s \t ROLL NUMBER = %d", name, roll_no);

// READ FROM FILE- Student.DAT
                fscanf(fp, "%s %d", name, &roll_no);
                 printf("\n NAME : %s \t ROLL NUMBER = %d", name, roll_no);
            fclose(fp);

}
```

fgets()

fgets() stands for *file get string*. The fgets() function is used to get a string from a stream. The syntax of fgets() can be given as:

char *fgets(char *str, int *size*, FILE **stream*);

The **fgets()** function reads at most one less than the number of characters specified by size (gets size - 1 characters) from the given stream and stores them in the string str. The fgets() terminates as soon as it encounters either a newline character or end-of-file or any other error. However, if a newline character is encountered it is retained. When all the characters are read without any error, a '\0' character is appended to end the string.

```
FILE *fp;
char str[80];
fp = fopen("Student.DAT", "r");
if(fp==NULL)
{
        printf("\n The file could not be opened");
        exit(1);
}
while (fgets(str, 80, fp) != NULL)
        printf("\n %s", str);
printf("\n\n File Read. Now closing the file");
fclose(fp);
```

fgetc()

- The fgetc() function returns the next character from stream, or EOF if the end of file is reached or if there is an error. The syntax of fgetc() can be given as

  int fgetc(FILE *stream);

- **fgetc** returns the character read as an **int** or return **EOF** to indicate an error or end of file.
- fgetc() reads a single character from the current position of a file (file associated with *stream)*. After reading the character, the function increments the associated file pointer (if defined) to point to the next character. However, if the stream has already reached the end of file, the end-of-file indicator for the stream is set.

Program on fgetc()

```
int main ()
{
    // open the file
    FILE *fp = fopen("test.txt","r");

    // Return if could not open file
    if (fp == NULL)
      return 0;

    do
    {
        // Taking input single character at a time
        char c = fgetc(fp);

        // Checking for end of file
        if (feof(fp))
            break ;

        printf("%c", c);
    }  while(1);

    fclose(fp);
    return(0);
}
```

fread()

- The fread() function is used to read data from a file. Its syntax can be given as
- int fread(void *str, size_t size, size_t num, FILE *stream);
- The function fread() reads *num* number of objects (where each object is *size* bytes) and places them into the array pointed to by str. The data is read from the given input stream.
- Upon successful completion, *fread*() returns the number of bytes successfully read. The number of objects will be less than num if a read error or end-of-file is encountered. If *size* or *num* is 0, *fread*() will return 0 and the contents of str and the state of the stream remain unchanged. In case of error, the error indicator for the stream will be set.
- The **fread()** function advances the file position indicator for the stream by the number of bytes read.

```
FILE *fp;
    char str[11];
            fp = fopen("Letter.TXT", "r+");
            if(fp==NULL)
            {
                    printf("\n The file could not be opened");
                    exit(1);
            }
            fread(str, 1, 10, fp);
            str[10]= '\0';
            printf("\n First 9 characters of the file are : %s", str);
    fclose(fp);
```

WRITING DATA TO FILES

- C provides the following set of functions to write data to a file.

  fprintf() fputs() fputc() fwrite()

fprintf()

- The fprintf() is used to write formatted output to stream. Its syntax can be given as,

  int fprintf (FILE * stream, const char * format, ...);
- The function writes to the specified *stream*, data that is formatted as specified by the *format* argument After the *format* parameter, the function can have as many additional arguments as specified in *format*
- The parameter format in the fprintf() is nothing but a C string that contains the text that has to be written on to the stream.

```
        FILE *fp;
        int i;
  char name[20];
  float salary;
  fp = fopen("Details.TXT", "w");
        if(fp==NULL)
        {       printf("\n The file could not be opened");
                exit(1);
        }
        for(i=0;i<10;i++)
  {             puts("\n Enter your name : ");
                gets(name);
                fflush(stdin);
                puts("\n Enter your salary : ");
                scanf("%f", &salary);
                fprintf(fp, " (%d) NAME : [%-10.10s] \t SALARY " %5.2f", i, name, salary);
}
fclose(fp);
```

fputs()

The fputs() is used to write a line into a file. The syntax of fputs() can be given as

int fputs(const char *str, FILE *stream);

The **fputs()** writes the string pointed to by str to the stream pointed to by stream. On successful completion, fputs() returns 0. In case of any error, fputs() returns EOF.

```
#include<stdio.h>
main()
{
        FILE *fp;
        char feedback[100];
    fp = fopen("Comments.TXT", "w");
        if(fp==NULL)
        {
                printf("\n The file could not be opened");
                exit(1);
        }
        printf("\n Kindly give the feedback on this book : ");
        gets(feedback);
        fflush(stdin);
        fputs(feedback, fp);
    fclose(fp);
}
```

fputc()

The fputc() is used to write a character to the stream.

int fputc(int *c*, FILE **stream*);

The *fputc*() function will write the byte specified by *c* (converted to an **unsigned char**) to the output stream pointed to by *stream.* Upon successful completion, *fputc*() will return the value it has written. Otherwise, in case of error, the function will return EOF and the error indicator for the stream will be set.

```
#include<stdio.h>
main()
{
        FILE *fp;
        char feedback[100];
        int i;
   fp = fopen("Comments.TXT", "w");
        if(fp==NULL)
        {
                printf("\n The file could not be opened");
                exit(1);
        }
        printf("\n Kindly give the feedback on this book : ");
        gets(feedback);
        for(i=0; i<feedback[i];i++)
                fputc(feedback[i], fp);
        fclose(fp);
}
```

fwrite()

The *fwrite()* is used to write data to a file. The syntax of fwrite can be given as,

int fwrite(const void *str, size_t size, size_t count, FILE *stream);

The *fwrite*() function will write, from the array pointed to by s*tr*, up to count objects of size specified by *size*, to the stream pointed to by *stream*.

The file-position indicator for the stream (if defined) will be advanced by the number of bytes successfully written. In case of error, the error indicator for the stream will be set.

```
main(void)
{    FILE *fp;
   size_t count;
   char str[] = "GOOD MORNING ";
   fp = fopen("Welcome.txt", "wb");
   if(fp==NULL)
   {                printf("\n The file could not be opened");
                    exit(1);
   }
   count = fwrite(str, 1, strlen(str), fp);
   printf("\n %d bytes were written to the files", count);
  fclose(fp);
}
```

fwrite() can be used to write characters, integers, structures, etc to a file. However, fwrite() can be used only with files that are opened in binary mode.

DETECTING THE END-OF-FILE

In C, there are two ways to detect the end-of-file

While reading the file in text mode, character by character, the programmer can compare the character that has been read with the EOF, which is a symbolic constant defined in stdio.h with a value -1.

```
while(1)
{           c = fgetc(fp);   // here c is an int variable
            if (c==EOF)
                        break;
            printf("%c", c);
 }
```

The other way is to use the standard library function feof() which is defined in stdio.h. The feof() is used to distinguish between two cases

When a stream operation has reached the end of a file

When the EOF ("end of file") error code has been returned as a generic error indicator even when the end of the file has not been reached

The prototype of feof() can be given as:

int feof(FILE *fp);

Feof() returns zero (false) when the end of file has not been reached and a one (true) if the end-of-file has been reached.

```
while( !feof(fp)
{           fgets(str, 80, fp);
            printf("\n %s", str);
}
```

ERROR HANDLING DURING FILE OPERATIONS

It is not uncommon that an error may occur while reading data from or writing data to a file. For example, an error may arise

When you try to read a file beyond EOF indicator

When trying to read a file that does not exist

When trying to use a file that has not been opened

When trying to use a file in un-appropriate mode. That is, writing data to a file that has been opened for readi

When writing to a file that is write-protected

The function ferror() is used to check for errors in the stream. Its prototype can be given as

int ferror (FILE *stream);

It returns a zero if no errors have occurred and a non-zero value if there is an error. In case of an error, the programmer can determine which error has occurred by using the perror().

```
FILE *fp;
        char feedback[100];
        int i;
  fp = fopen("Comments.TXT", "w");
        printf("\n Kindly give the feedback on this book : ");
        gets(feedback);
        for(i=0i<feedback[i];i++)
                fputc(feedback[i], fp);
        if(ferror(fp))
        {       printf("\n Error writing in file");
                exit(1);
        }
        fclose(fp);
```

```c
#include <stdio.h>

int main () {
  FILE *fp;

  /* first rename if there is any file */
  rename("file.txt", "newfile.txt");

  /* now let's try to open same file */
  fp = fopen("file.txt", "r");
  if( fp == NULL ) {
    perror("Error: ");
    return(-1);
  }
  fclose(fp);

  return(0);
}
```

File- program1

C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information to the file.

```
#include <stdio.h>
int main()
{
  char name[50];
  int marks, i, num;
  printf("Enter number of students: ");
  scanf("%d", &num);
  FILE *fp;
  fptr = (fopen("C:\\student.txt", "a"));
  if(fp == NULL)
  {
     printf("Error!");
     exit(1);
  }

  for(i = 0; i < num; ++i)
  {
    printf("For student%d\nEnter name: ", i+1);
    scanf("%s", name);

    printf("Enter marks: ");
    scanf("%d", &marks);

    fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
  }
  fclose(fp);
  return 0;
}
```

FUNCTIONS FOR SELECTING A RECORD RANDOMLY

fseek()

fseek() is used to move file pointer associated with a given file to a specific position.

Syntax:

int fseek(FILE *pointer, long int offset, int position)

pointer: pointer to a FILE object that identifies the stream.

offset: number of bytes to offset from position

position: position from where offset is added.

returns:

zero if successful, or else it returns a non-zero value

position defines the point with respect to which the file pointer needs to be moved. It has three values:

SEEK_END : It denotes end of the file.

SEEK_SET : It denotes starting of the file.

SEEK_CUR : It denotes file pointer’s current position.

fseek()

```
#include <stdio.h>

int main()
{
   FILE *fp;
   fp = fopen("test.txt", "r");

   // Moving pointer to end
   fseek(fp, 0, SEEK_END);

   // Printing position of pointer
   printf("%ld", ftell(fp));

   return 0;
}
```

Output:

rewind()

rewind() is used to adjust the position of file pointer so that the next I/O operation will take place at the beginning of the file. It's prototype can be given as

void rewind(FILE *f);

rewind() is equivalent to calling fseek() with following parameters: fseek(f,0L,SEEK_SET);

```
int main()
{
  FILE *fp = fopen("test.txt", "r");

  if ( fp == NULL ) {
    /* Handle open error */
  }

  /* Do some processing with file*/

  rewind(fp);  /* no way to check if rewind is successful */

  /* Do some more precessing with file */

  return 0;
}
```