- Address of a bit-field can not be accessed.
- Pointers cannot be used to access bit fields.
- Bit-fields can not be declared as static.
- No field can be longer than 32 bits (1 long word).
- It is not possible to declare arrays of bit fields.
- The ampersand operator (&) cannot be applied to fields, so there cannot be pointers to bit fields. Hence, we can not use **scanf** to read values into bit fields.

Exercises:

1. What is a structure? Explain with syntax and an example.
2. What is the difference between a structure and an array?
3. How a structure can be initialized? Explain with an example
4. How the members of a structure can be accessed? Explain when '.' is used and when '->' or * operators are used to access the fields within the structure?
5. What is a structure? How does a structure differ from an array? How are structure members assigned values and are accessed? Explain.
6. Explain array of structures with example
7. What are nested structures? Explain with an example
8. Is it possible to have structures within structures? Explain with an example
9. Is it possible to return the value of a structure using *return* statement? If the value of two or more structures is needed in the calling function, what to do? Explain with an example.
10. Write a C program to add, subtract, multiply and divide two complex numbers. Each function should return a complex number after performing the respective operations.
11. What is a union? How is it different from structure? With a suitable example show how union is declared and used in C.
12. What is the difference between structure and union?
13. Write a C program to arrange the student record based on increasing order of roll numbers, increasing order of marks, increasing order of their age. Assume the student record contains the following fields: name, age, branch, marks, roll number and address.

Chapter 5: Binary Files

What are we studying in this chapter?

- Classification of Files
- Using Binary Files
- Standard Library functions for files

- 2 hours

5.1 Files and Classification of files

We know that the function **scanf()** is used to enter the data from the keyboard and **printf()** is used to display the result on the video display unit. This works fine when the input data is very small. But, as the volume of input data increases, in most of the applications, we find it necessary to store the data permanently on the disk and read from it for the following reasons:

- It is very difficult to input large volume of data through terminals
- It is time consuming to enter large volume of data using keyboard.
- When we are entering the data through the keyboard, if the program is terminated for any of the reason or computer is turned off, the entire input data is lost.

To overcome all these problems, the concept of storing the data in disks was introduced. Here, the data can be stored on the disks; the data can be accessed as and when required and any number of times without destroying the data. The data is stored in the form of a file. Now, let us see "What is a file? How the data is stored in a file?"

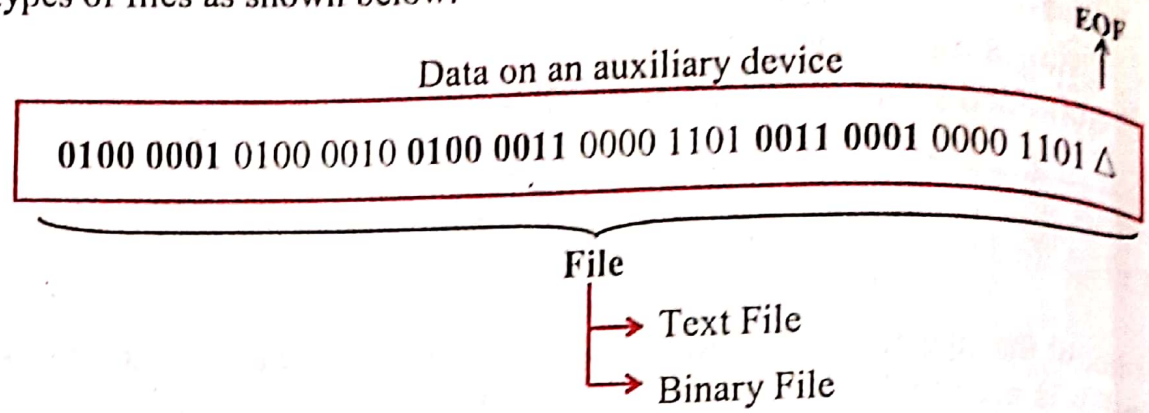
Definition: A file is defined as a collection of related data stored in auxiliary devices such as disks, CDs etc. The data is recorded on the auxiliary devices using 0s and 1s (called bits). The auxiliary devices will not assign any relationship between the various bits. The pictorial representation of data on auxiliary storage device is shown below:

0100 0001 0100 0010 **0100 0011** 0000 1101 **0011 0001** 0000 1101 Δ

EOF

Note: **EOF** stands for **E**nd **O**f **F**ile

Once we know the definition of file, let us see "What are the various types of files?" When the data stored in auxiliary device is read by the program, the data (stored in terms of 0s and 1s) can be interpreted as either a text file or a binary file. Thus, there are two types of files as shown below:



Thus, it is very clear that even though the data is stored in terms of 0s and 1s on a disk, we can interpret them either as text file or binary file.

5.1.1 Text file

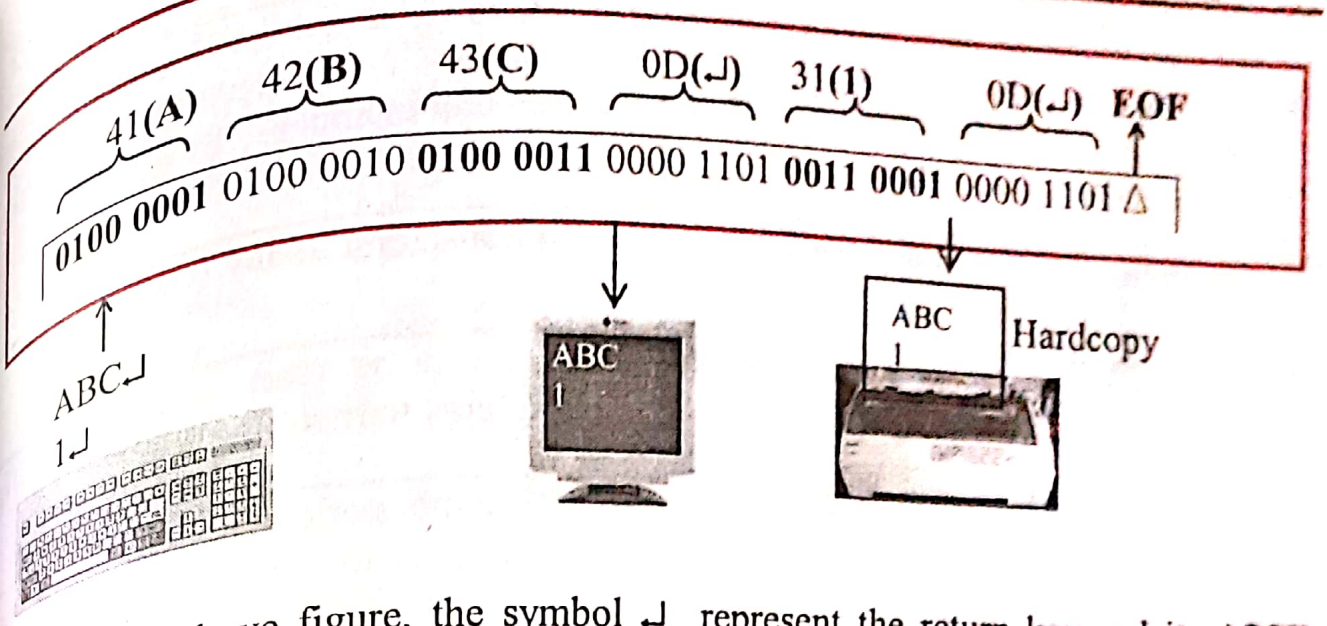
Now, let us see "What is a text file?"

Definition: A text file is the one where data is stored as a stream of characters that can be processed sequentially. In the text format, data are organized into lines terminated by newline character. The text files are in human readable form and they can be created and read using any text editor. Since text files process only characters, they can read or write only one character at a time. A **newline** may be converted to **carriage return** and **line feed** character. **Note:** newline character = carriage return + linefeed

Because of these translations, the number of characters written/read may not be the same as the number of characters written/read on the external device. Therefore, there may not be a one-to-one relationship between the characters written/read into the file and those stored on the external devices.

For example, the data 2345 requires 4 bytes with each character occupying exactly one byte. A text file can not contain integers, floating-point numbers etc. To store the data such as integers, floating point numbers etc., they must be converted to their character-equivalent formats. For example, the data 2345 in text file is stored as sequence of 4 characters '2', '3', '4' and '5'. The following figure shows

- How 6 characters are entered from the keyboard
- How the 6 characters entered from keyboard are stored in file
- How the 6 characters stored in a file are displayed on the monitor/printer



Note: In the above figure, the symbol ↵ represent the return key and its ASCII character is 0x0D (13 in decimal). The ASCII values of A, B, C are 0x41 (65 decimal), 0x42 (66 in decimal), 0x43 (67 in decimal) respectively. The ASCII values of 1, 2, 3 etc. are 0x31 (48 in decimal) , 0x32(49 in decimal) and so on. The symbol Δ represents end of file.

5.1.2 Binary file

Now, let us see "What is a binary file?"

Definition: A binary file is the one where data is stored on the disk in the same way as it is represented in the computer memory. The binary files are not in 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 not be read using any of the text editors.

For example, the data 2345 takes 2 bytes of memory and is stored as 0x0929 i.e., 0x09 is stored as one byte and 0x029 is stored as other byte. The number of characters written/read is same as the number of characters written/read on the external device. Therefore, there is one-to-one relationship between the characters written/read into the file and those stored on the external devices.

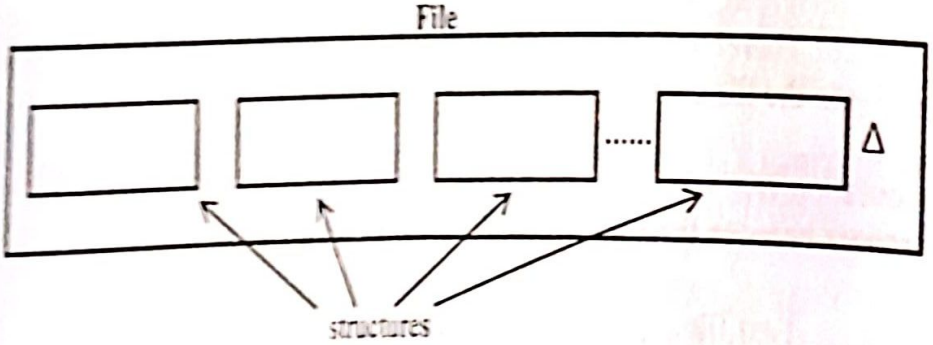
Now the question is "What is the difference between a text file and a binary file?" The differences between these two types of files are shown below:

Text file	Binary file
1. Human readable format	1. Not in human readable format

2. Data is stored as lines of characters with each line terminated by \n which may be translated into carriage return + line feed	2. Data is stored on the disk in the same way as it is represented in the computer memory
3. Translation of newline character to carriage return + line feed	3. No translation of any type
4. Data can be read using any of the text editor	4. Data can be read only by specific programs written for them
5. The number of characters written/read may not be the same as the number of characters written/read on the external device such as disk	5. The number of characters written/read is same as the number of characters written/read on the external device such as disk
6. There may not be a one-to-one relationship between the characters written/read into the file and those stored on the external devices.	6. There is one-to-one relationship between the characters written/read into the file and those stored on the external devices.
7. The data 2345 requires 4 bytes with each character requiring one byte. So, it is stored as 0x32, 0x33, 0x34, 0x35 (ASCII values) thus requiring 4 bytes	7. The data 2345 requires 2 bytes and stored as 0x0929. The data 2345 is stored as 0x09 and 0x29 thus requiring 4 bytes.

5.2 Using Binary files

Normally the binary files are associated with structures as shown in figure below:



Each rectangle in the figure represents a structure. At the end of a file there is a file marker (Δ) which denotes **end of file (EOF) marker**. The four steps that are used during the manipulation of binary files are shown below.

Steps in using the files
- Declare a file pointer variable
- Open a file
- Read the data from the file or write the data into file
- Close the file

5.2.1 Declare a file pointer variable

We know that all the variables are declared before they are used. Likewise, a file pointer variable also should be declared. Now, let us see "How to declare a file pointer variable?" A file pointer variable **fp** should be declared as a pointer to a structure of type FILE as shown below:

```
#include <stdio.h>

FILE *fp;              /* Here, fp is a pointer to a structure FILE */

void main()
{
      .....            /* File operations */
}
```

Note that FILE is the derived data type which is defined already in header file **stdio.h** as a structure. The structure details are hidden from the programmer and this structure is used to store information about a file.

Note: The file pointer **fp** can be used either as a local or global variable. In the above syntax, the file pointer **fp** is used as a global variable. The file pointer **fp** can be used to open a file, update a file and to close a file in sequence.

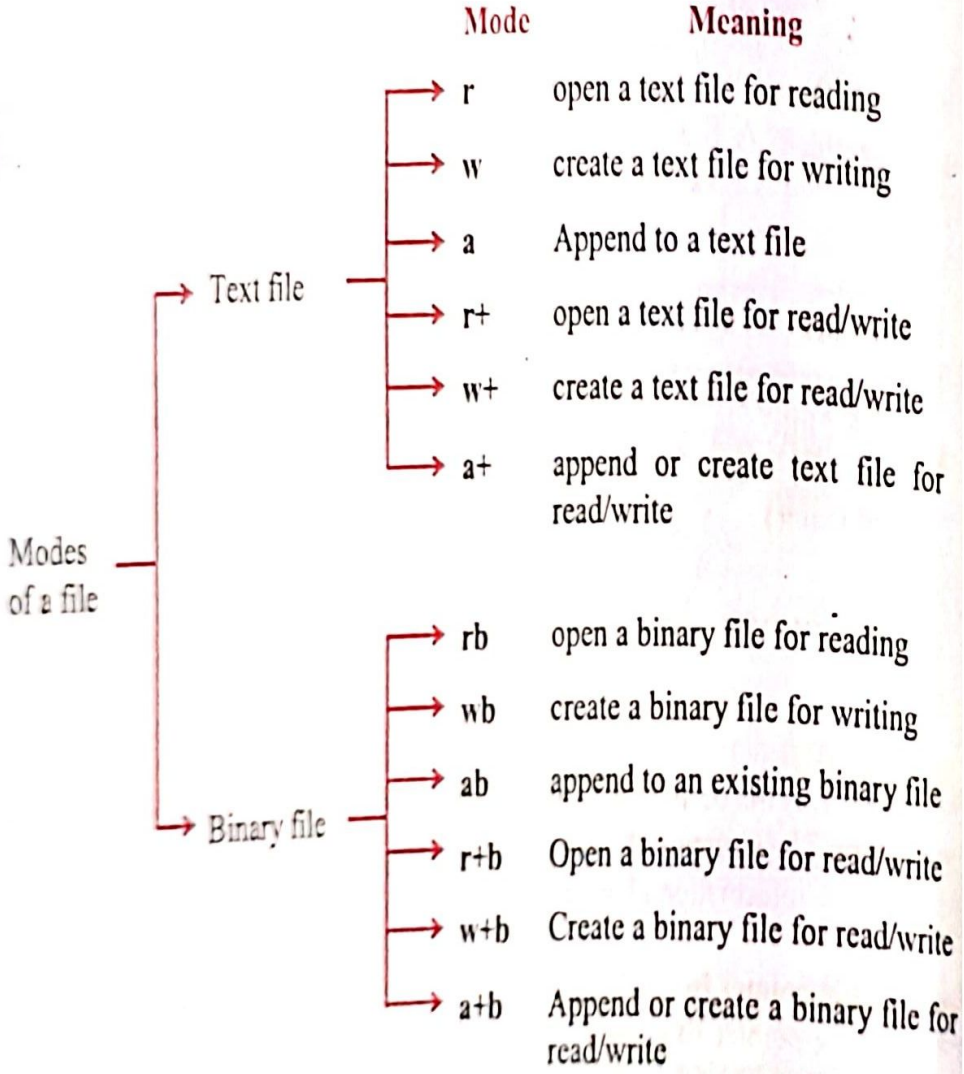
Analogy: The information of all the students in a college is maintained by the college office. The information of a particular student such as name, address, branch and semester details along with CET ranking, percentage of marks in PUC etc. is stored in a file. If we want to read, write or update the student details we have to open a file,

update a file and finally close it. The same sequence of operations can be carried with respect to files in C also. Here too, before updating a file, it has to be opened after updating the file, it has to be closed.

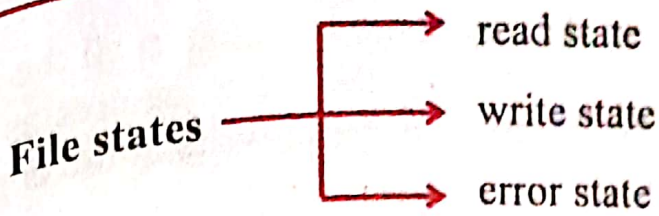
Note: The first operation to be done before updating the file is opening a file. The operation to be done after updating is closing the file.

5.2.2 States and modes of a file

Before opening a file, let us see "What are the various modes in which a file can opened/created successfully?" The various modes in which a file can opened/created successfully along with meanings are shown below:



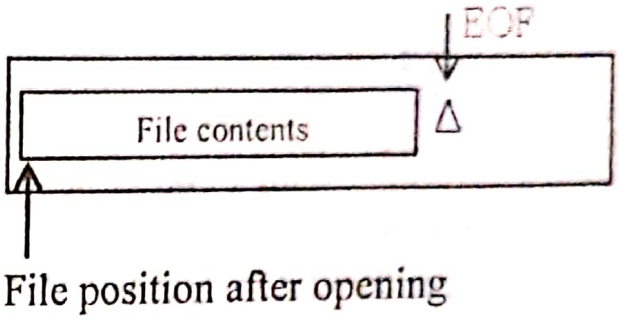
Note: It is clear from the above figure that the file can be either in read or write sta If the file can not be opened successfully or can not be created then it is an error a the file state will be error state. This results in three possible file states as show below:



Now, let us discuss each of the modes in detail:

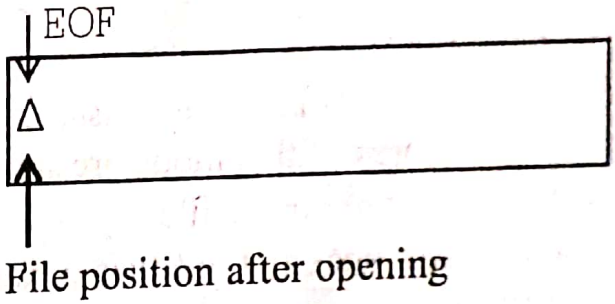
r **(read mode)** This mode is used for opening an existing file to perform read operation. The various features of this mode are shown below:

- Used only for the text file
- If the file does not exist, an error is returned
- The contents of the file are not lost
- The file pointer points to the beginning



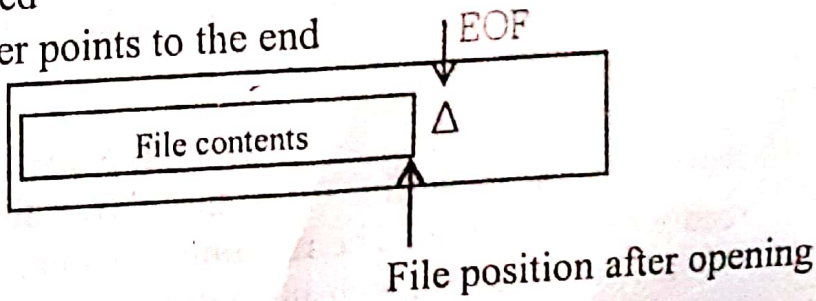
w **(write mode)** This mode is used to create a file. The various features of this mode are shown below:

- Used only for the text file
- If the file does not exist, a file is created.
- If the file already exists, the original contents of the file are lost
- The file pointer points to the beginning



a **(append mode)** This mode is used to insert the data at the end of the existing file. The various features of this mode are shown below:

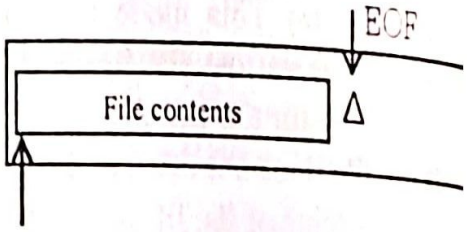
- Used only for the text file
- If the file does not exist, a file is created
- If the file already exists, the file pointer points to the end
- The new data is inserted at the end
- Existing data can not be modified



Note: If we have + as the suffix for the above modes, then in all the above three cas the file is opened for read/write operation with the same features as specified earlie For example, r+ (read/write), w+(read/write), a+(read/write at the end)

rb (read mode) This mode is used for opening an existing file to perform rea operation. The various features of this mode are shown below:

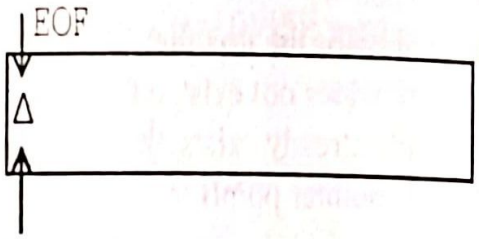
- Used only for the binary file
- If the file does not exist, an error is returned
- The contents of the file are not lost
- The file pointer points to the beginning



File position after opening

wb (write mode) This mode is used to create a file. The various features of this mod are shown below:

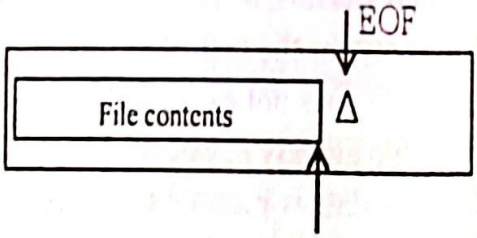
- Used only for the binary file
- If the file does not exist, a file is created.
- If the file already exists, the original contents of the file are lost
- The file pointer points to the beginning



File position after opening

ab (append mode) This mode is used to insert the data at the end of the existing file The various features of this mode are shown below:

- Used only for the binary file
- If the file does not exist, a file is created
- If the file already exists, the file pointer points to the end
- The new data is inserted at the end
- Existing data can not be modified



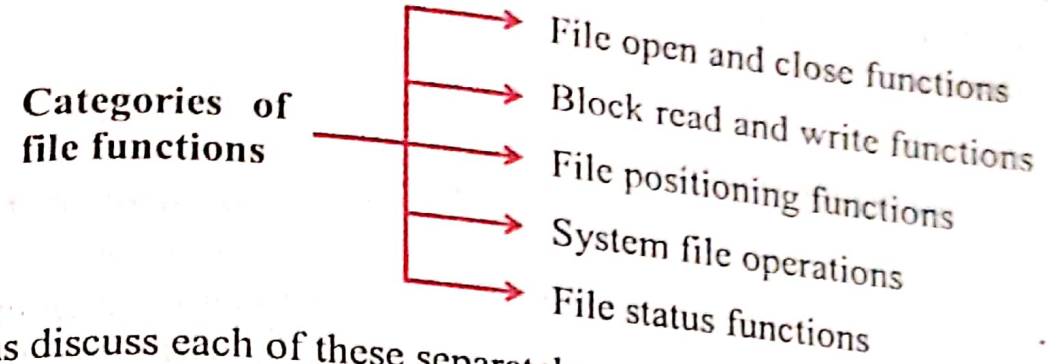
File position after opening

Note: Having a suffix + for the above three modes, then the file is opened fo read/write operation. The remaining features remain same. The modes rb+, wb+ and ab+ are same as r+b, w+b and a+b respectively.

Note: A mode with substring + is called update mode. This is because; the substring + which is part of the mode as explained earlier represent read/write operation.

5.3 Standard library functions for files

Now, let us see "What are the standard library functions that are used for binary files?" The various standard file library functions that we discuss with respect to binary files are shown below:



Now, let us discuss each of these separately

5.3.1 File open and close functions

Once we know various modes in which a file can be opened or created, let us see "How to open a file?" The file should be opened before reading a file or before writing into a file. The syntax to open a file for either read/write operation is shown below:

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

where

- **fp** is a file pointer of type FILE
- **filename** holds the name of the file to be opened. The filename should be a valid identifier.
- **mode** details are provided in section 5.2.2. This informs the library function about the purpose of opening a file.

Return values The function may return the following:

- File pointer of type FILE if successful
- NULL if unsuccessful

If the file pointer **fp** is not NULL, the necessary data can be accessed from the specified file. If a file cannot be opened successfully for some reason, the function returns NULL. We can use this NULL character to test whether a file has been successfully opened or not using the statement:

```
if (fp == NULL)
{
        printf("Error in opening the file\n");
        exit(0);
}

/* Using fp access file contents */
.....
.....
```

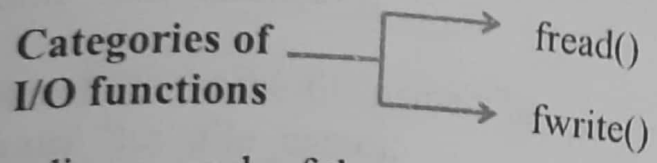
Now, let us see "When to close and how to close a file?" When we no longer need a file, we should close the file. This is the last operation to be performed on a file. A file can be closed using close function. This ensures that all buffers are flushed and all the links to the file are broken. Once the file is closed, to access the file, it has to be re-opened. If a file is closed successfully, 0 is returned otherwise EOF is returned. The syntax is shown below:

```
#include <stdio.h>

void main()
{
        FILE *fp;

        ......
        .......
        fp = fopen("t.c", "rb");
        .....
        .....
        if ( fclose(fp) == EOF)
        {
                printf("Error in closing the file\n");
                return;
        }
        ........
        ........
}
```

Note: If a program is terminated, all the opened files will be closed automatically. But, as a programming practice, it is better to close all the opened files once we know that the file pointers are not required.

5.3.2 Block read and write (Block I/O) functions

We know that data are stored in the memory in the form of 0's and 1's. When we read and write the binary files, the data are transferred just as they are found in memory and hence there are no format conversions. Now, let us see "What are the functions that are required to transfer the block of data to/from binary files?" The various input and output functions that transfer block of data are shown below:



Now, let us discuss each of these one by one.

5.3.2.1 File read – fread()

Now, let us see "What is the syntax of fread? How does the function work?" This function is used to read a block of data from a given file. The prototype of fread() which is defined in header file "stdio.h" is shown below:

```
int fread (void *ptr, int size, int n, FILE *fp);
```

Before executing the function, we have to pass the following parameters:

- **fp** is a file pointer of an opened file from where the data has to be read from.
- **ptr:** The data read from the file should be stored in memory. For this purpose, it is required to allocate the sufficient memory and address of the first byte is stored in **ptr**. We say that **ptr** now points to buffer.
- **n** is the number of items to be read from the file.
- **size** is the length of each item in bytes.

Working: When the fread() function is executed, the function reads **n** number of items, each of length **size** bytes (**n * size** bytes) from the file associated with **fp**. The data read from the file is copied into the memory location pointed to by **ptr**.

Returns

- Number of items successfully read
- If no items have been read or when error has occurred or end-of-file is encountered, the function returns 0 (zero).

Note: Since this function will not return EOF, we have to check for possible error using **feof()** or **ferror()** functions.

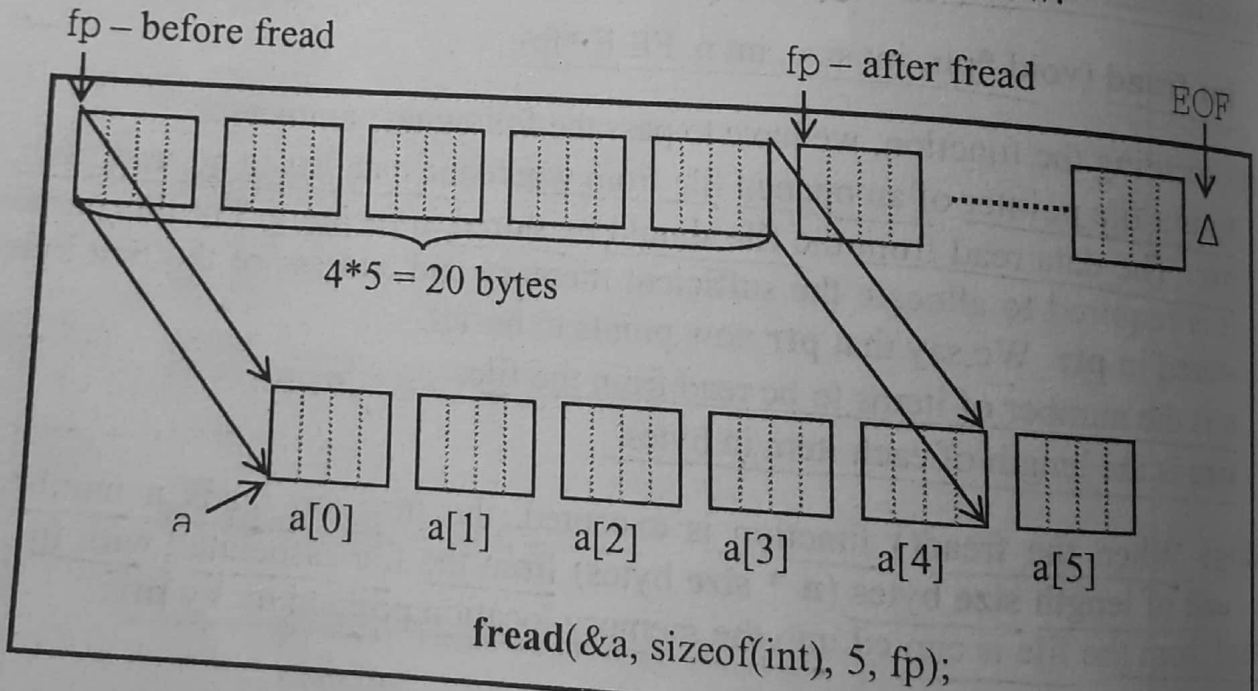
Example 5.3.2.1.1: Read a floating point value from the file and store it in variable f.

```
fread(&f, sizeof(float), 1, fp);
```

Example 5.3.2.1.2: C function to read 5 integers from the file and store it in array a.

```
#include <stdio.h>

void main()
{
        int a[10];
        ..........
        ..........
        fread(&a, sizeof(int), 5, fp);
        ..........
        ..........
}
```

The pictorial representation after executing fread() is shown below:



fread(&a, sizeof(int), 5, fp);

Note: 20 bytes (5 integers) are read from the file from the file pointer **fp** to the buffer area identified by **a**. Here, location **a** is the place where the data read from the file is stored. So, **a** is called buffer area. Note the position of file pointer **fp** before and after **fread** operation.

Now, let us write a program to read **n** integers from the binary file. Assume the file "**test.dat**" contains the integers stored in binary form.

Example 5.3.2.1.3: Program to read a file of integers from the file "test.dat"

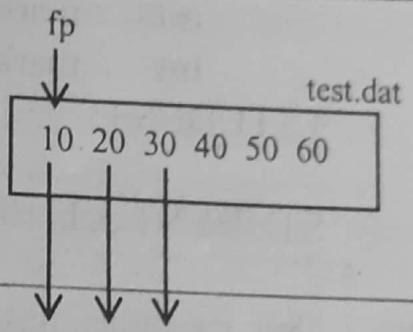
```
#include <stdio.h>

void main()
{
    FILE *fp;
    int     a[10], n, i;
    char    file_name[20];

    printf("Enter the file name\n");
    scanf("%s",file_name);

    printf("Enter the number of integers\n");
    scanf("%d",&n);

    if ((fp = fopen(file_name,"rb")) == NULL)
    {
        printf("Error in opening the file\n");
        return;
    }

    fread(a, sizeof(int), n, fp);

    for (i = 0; i < n; i++)
    {
        printf("%d\n",a[i]);
    }
}
```

TRACING

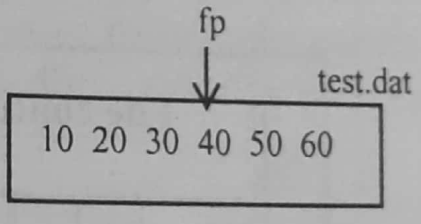
Input

Enter the file name
test.dat

Enter number of integers
3



10 20 30 (After fread)
a[0] a[1] a[2]



Output

10 20 30

Note: The sequence of operations carried out during the execution of above prog are:

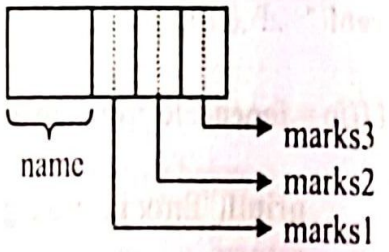
- The file "test.dat" is opened and fp points to the first byte of opened file as sh in figure.
- Since n = 3, three numbers are read from the file
- The three integers are copied into a[0], a[1] and a[2] respectively.
- After reading the file pointer fp points to 40 as shown in figure.
- Three items 10, 20 and 30 are displayed on the screen.

Now, let us see "How to read the data from a file into a structure?" For this to happ let us consider the following structure definition:

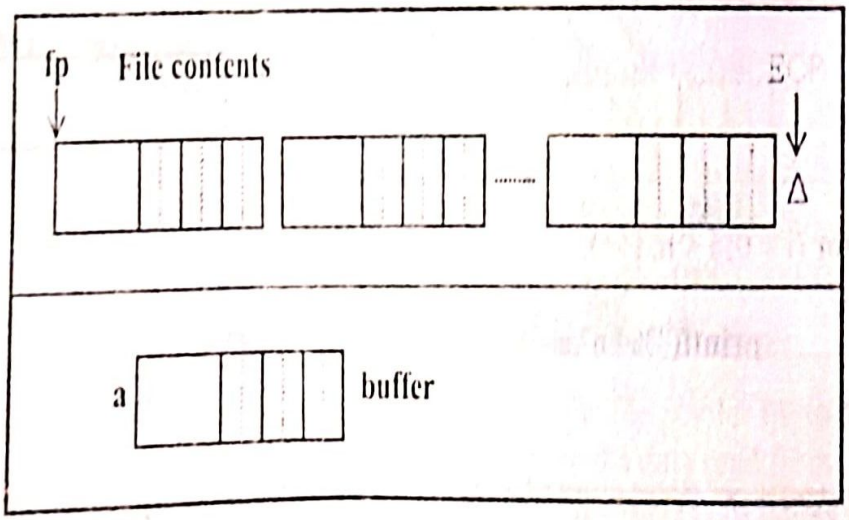
Example 5.3.2.1.4: Read the information of a structure from the file using fread().

```
typdef struct
{
        char   name[25];
        int    marks1;
        int    marks2;
        int    marks3;
} STUDENT;

STUDENT a, b[10];
```

Pictorial representation of structure



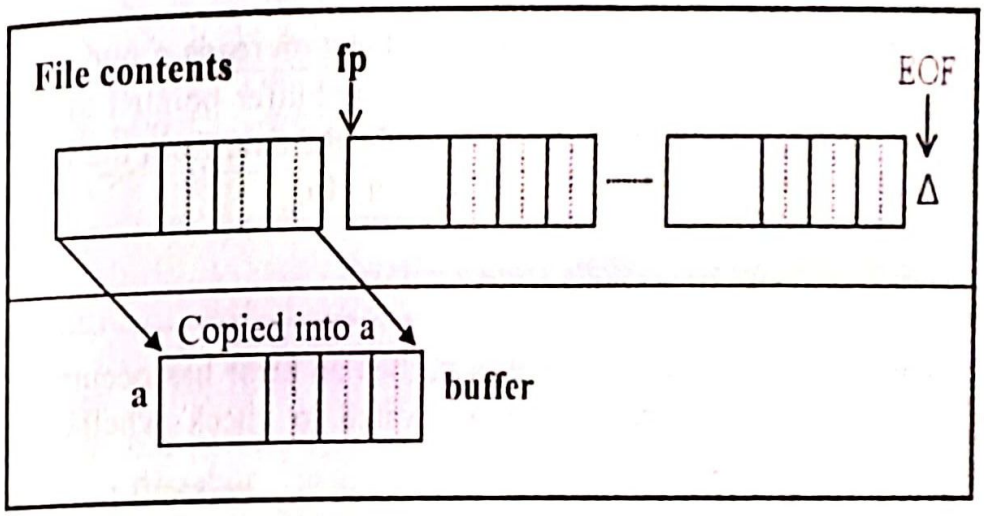
Note: The structure has four fields. The memory is allocated for the struct variables a and b. The pictorial representation of the memory allocated for variabl and file contents before fread() is executed is shown below:



w, let us see what will happen if the following statement is executed.

```
fread(a, sizeof(STUDENT), 1, fp);
```

e function reads a single structure from the file and store it in the variable a. The torial representation of the file pointer and variable a after fread operation is wn below:



te:After reading the first structure from the file, the file pointer fp is automatically nts to next structure.

ample 5.3.2.1.5:For the structure definition shown in previous example, "What l happen when the following statement is executed?"

```
fread (b, sizeof(b), 10, fp);
```

lution:The function reads 10 structures from the file and store them in the variable ne after the other starting from b[0], b[1],.....b[9]. The file pointer fp points to eleventh structure in the file.

3.2.2 File write – fwrite()

w, let us see "What is the syntax of fwrite? How does the function work? This ction is used to write a block of data into a given file. The prototype of fwrite() ch is defined in header file "stdio.h" is shown below:

```
int fwrite (void *ptr, int size, int n, FILE *fp);
```

Before executing the function, we have to pass the following parameters:

- fp is a file pointer of an opened file into which the data has to be written.
- ptr: The data to be written into the file should be stored in the memory. Th address of the first byte is stored in ptr. We say that ptr now points to buffer.
- n is the number of items to be written into the file.
- size is the length of each item in bytes.

Working: When fwrite() function is executed, the function reads n number of items each of length size bytes (i.e., n*size bytes) from the buffer pointed to by ptr an writes into the file associated with fp. In other words, the data from the buffer whic is pointed by ptr is written into the file associated with fp.

Returns Number of items successfully written

Note: If number of items written is less than n, then an error has occurred. It is the responsibility of the programmer to use this value to check whether the write operation is successful or not.

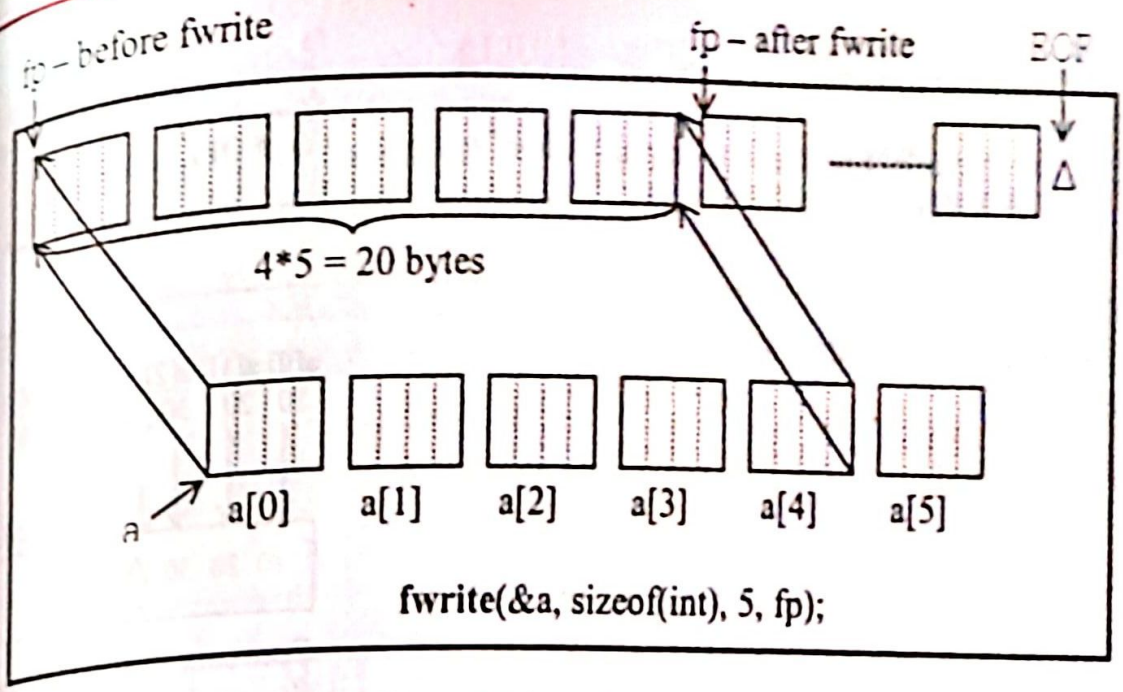
Example 5.3.2.2.1: Write a floating point value from variable f into file

```
fwrite(&f, sizeof(float), 1, fp);
```

Example 5.3.2.2.2: C function to write 5 integers stored in array a into the file

```
#include <stdio.h>

void main()
{
    int a[5] = {10, 20, 30, 40, 50, 60, 70, 80};
    .....
    .....
    /* Here, file is opened successfully in write mode */
    .....
    fwrite(&a, sizeof(int), 5, fp);
    .....
    .....
}
```

The pictorial representation after executing fwrite () is shown below:



Note: 20 bytes (5 integers) are read from the buffer **a** and are written into the file associated with file pointer **fp.** Note the position of file pointer **fp** before and after executing fwrite function.

Now, let us write a program to write **n** integers into the binary file. Assume the file is "test.dat".

mple 5.3.2.2.3: Program to write n integers into a file "test.dat"

```
#include <stdio.h>

void main()
{
    FILE *fp;
    int     a[6] = {10, 20, 30, 40, 50, 60};
    int     n, i;
    char    file_name[20];

    printf("Enter the file name\n");
    scanf("%s",file_name);
```

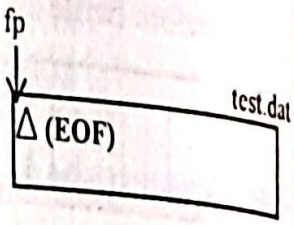
TRACING

Input
Enter the file name
test.dat

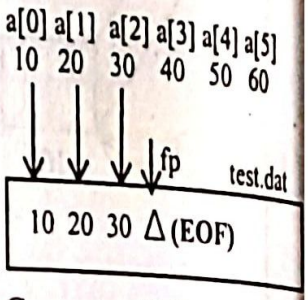
```
    printf("Enter the number of integers\n");
    scanf("%d",&n);
```

Enter number of integers
3

```
if ((fp = fopen(file_name,"wb")) == NULL)
{
        printf("Error in creating the file\n");
        return;
}
```




```
fwrite (a, sizeof(int), n, fp);
```



Contents of file after fwrit

```
}
```

Note: The sequence of operations carried out during the execution of above progra are:

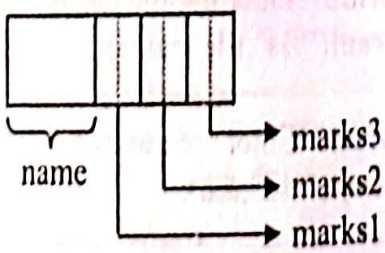
- The file "test.dat" is opened in write mode and **fp** points to the beginning of file a shown in figure. Observe that the file is empty.
- Since **n = 3**, three numbers are read from the buffer and are written into fi associated with file pointer **fp**.
- After writing the file pointer **fp** points to end-of-file
- Three items 10, 20 and 30 are displayed on the screen.

Now, let us see "How to write the data from a structure into a file?" For this happen, let us consider the following structure definition:

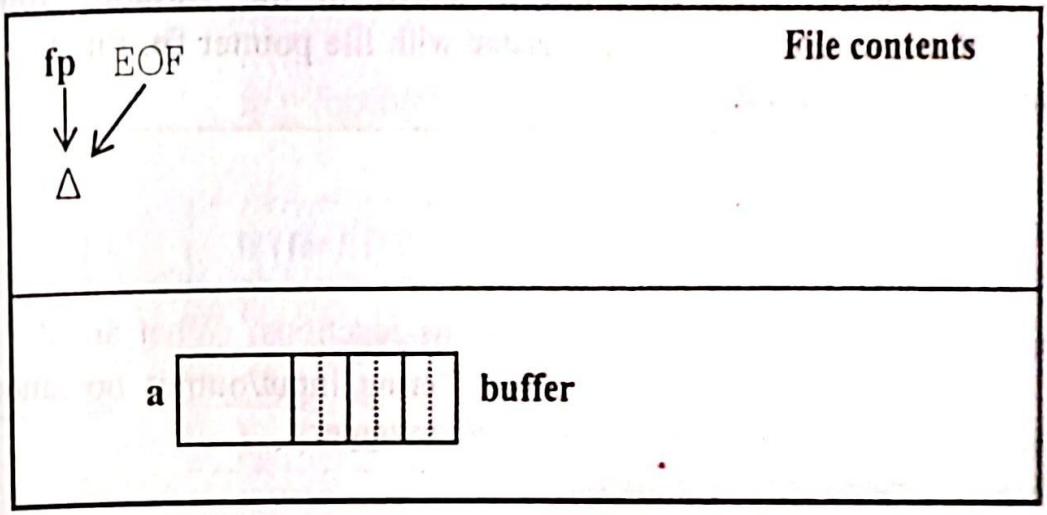
Example 5.3.2.2.4: Write the information of a structure into the file using fwrite().

```
typdef struct
{
        char   name[25];
        int    marks1;
        int    marks2;
        int    marks3;
} STUDENT;

STUDENT a, b[10];
```

Pictorial representation of structure



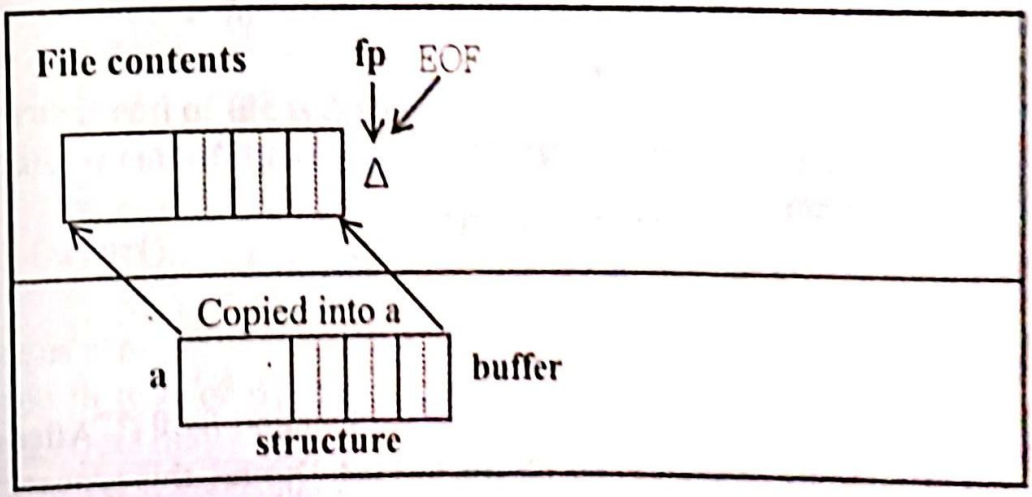
The structure has four fields. The memory is allocated for the structure
les a and **b.** Assume the structure **a** is initialized to appropriate values. The
al representation of the memory allocated for variable **a** and file contents before
() is executed is shown below:



let us see what will happen if the following statement is executed.

```
fwrite(a, sizeof(STUDENT), 1, fp);
```

function writes the values of structure identified by variable **a** into the file
iated with file pointer **fp.** The contents of file after executing **fwrite** is shown
v:



After writing the structure into the file, the file pointer points to end-of-file.

Example 5.3.2.2.5: For the structure definition shown in previous example, "Wh will happen when the following statement is executed?"

```
fwrite (b, sizeof(b), 10, fp);
```

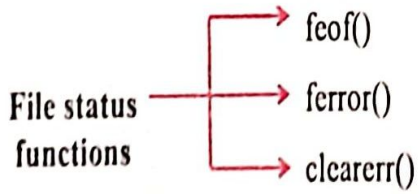
Solution: The function reads 10 structures stored in the variable from b[0] b[1],.....b[9] and writes into the file associated with file pointer fp. Finally, the fi pointer fp points to end-of-file.

5.3.3 File status functions

Now, let us see "What is the need for file status functions? What are the vario functions that inform the status of the file?" During input/output operations, t following operations may be performed by the programmer:

- Reading beyond end of file marker
- Opening a file which is not existing
- Input invalid file name
- Attempting to write to a write-protected file etc.,

Such read and write errors results in abnormal behavior of the program and so shou not be ignored. If proper error checking is not provided, there may be premat termination of the program or we may get incorrect output. For this reason C has be provided with various file status functions which can detect such errors. The vario file status functions provided in C language are shown below:



5.3.3.1 feof()

Now, let us see "Why feof() is used? How to use the function feof()?" After openi the file in read mode, the contents of the file can be read. During, this process we m encounter end of file. This function is used to detect whether end of file is reach The syntax to check whether end of file is reached or not is shown below:

```
#include <stdio.h>

void main()
{
        FILE *fp;
        ……..
        ……..
        fp = fopen(char *filename, char *mode)
        ……..
        ……..
        if (feof(fp))
        {
                printf("End of file is reached\n");
                exit(0);
        }
        ……..
        ……..
}
```

Syntax

fp is a file pointer of type FILE
filename holds the name of the file to be opened
mode details are provided in section 5.2.2. This informs the library function about the purpose of opening a file.

rn values The function accepts file pointer fp as a parameter and following are returned.
true if end of file is detected (true – nonzero value)
false if end of file is not detected. (false – zero value)

.2 ferror()

let us see "Why ferror() is used? How to use the function ferror()" A file can ened in read or write mode. During reading from a file or writing into a file ing errors may occur:
Modifying read only file
Trying to read a file which is opened in write mode and so on.

Example 5.3.2.2.5: For the structure definition shown in previous example, "Wha will happen when the following statement is executed?"

```
fwrite (b, sizeof(b), 10, fp);
```

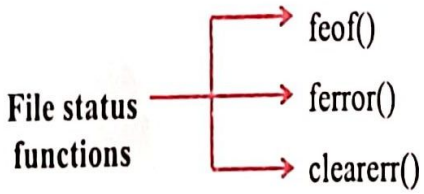
Solution: The function reads 10 structures stored in the variable from b[0 b[1],.....b[9] and writes into the file associated with file pointer fp. Finally, the fil pointer fp points to end-of-file.

5.3.3 File status functions

Now, let us see "What is the need for file status functions? What are the vario functions that inform the status of the file?" During input/output operations, th following operations may be performed by the programmer:

- Reading beyond end of file marker
- Opening a file which is not existing
- Input invalid file name
- Attempting to write to a write-protected file etc.,

Such read and write errors results in abnormal behavior of the program and so shou not be ignored. If proper error checking is not provided, there may be prematu termination of the program or we may get incorrect output. For this reason C has bee provided with various file status functions which can detect such errors. The variou file status functions provided in C language are shown below:



5.3.3.1 feof()

Now, let us see "Why feof() is used? How to use the function feof()?" After openin the file in read mode, the contents of the file can be read. During, this process we ma encounter end of file. This function is used to detect whether end of file is reache The syntax to check whether end of file is reached or not is shown below:

```
#include <stdio.h>

void main()
{
        FILE *fp;
        ........
        ........
        fp = fopen(char *filename, char *mode)
        ........
        ........
        if (feof(fp))
        {
                printf("End of file is reached\n");
                exit(0);
        }
        ........
        ........
}
```

Syntax

fp is a file pointer of type FILE
filename holds the name of the file to be opened
mode details are provided in section 5.2.2. This informs the library function about the purpose of opening a file.

n values The function accepts file pointer **fp** as a parameter and following are returned.
true if **end of file** is detected (true – nonzero value)
false if **end of file** is not detected. (false – zero value)

2 ferror()

et us see "Why ferror() is used? How to use the function ferror()?" A file can ned in read or write mode. During reading from a file or writing into a file ng errors may occur:
Modifying read only file
Trying to read a file which is opened in write mode and so on.

The macro ferror() is used to detect an error that occurred during read or wri operations on a file. The syntax to check whether any error has occurred or not shown below:

```
#include <stdio.h>

void main()
{
        FILE *fp;
        int   c;

        fp = fopen("test", "w");

        c = getc(fp);

        if (ferror(fp))                    Syntax
        {
                printf("Error in reading the file\n");
                return;
        }

}
```

Note: Here, file is opened in write mode and using the function getc(), we are try to read from a file. This is an error and this error is detected by the macro ferror() displays the message "Error in reading the file". The prototype declaration

```
int ferror(FILE *fp);
```

is available in the header file "stdio.h"

Return values The function accepts file pointer **fp** as a parameter and follow values are returned.

- **true** if an error is detected. (true – nonzero value)
- **false** if an error is not detected. (false – zero value)

3.3.3 clearerr()

w, let us see "What is the significance of the function clearer()?" During reading l writing of a file an error may occur. Once an error occurs, the subsequent calls to ror() returns **true** until the error status is cleared. So, to clear the error status the ıction clearerr() is used. The prototype of this function is:

```
void clearer(FILE *fp);
```

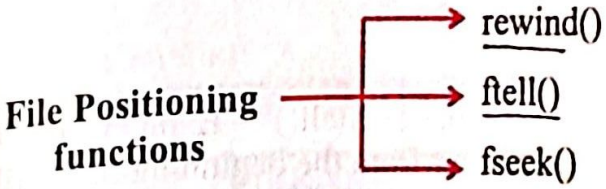
l is defined in the **"stdio.h"**

3.4 File Positioning Functions

w, let us see "What is the need for file positioning functions? What are the various e positioning functions?" The file positioning functions are required in the lowing scenarios:

- For accessing the random data from the file
- For changing the state of the file (from read to write or from write to read)

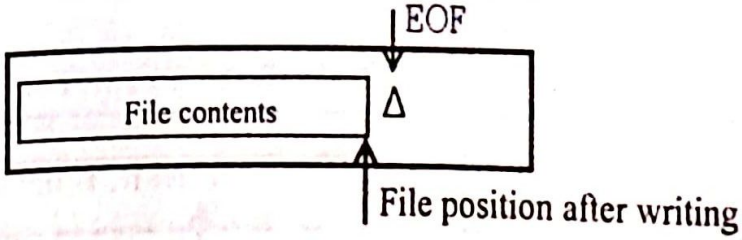
e various file status functions provided in C language are shown below:



w, let us discuss each of these one by one.

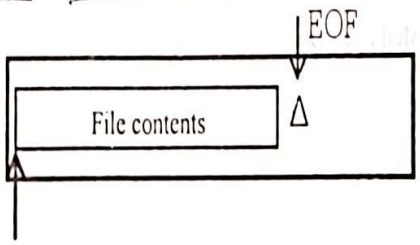
3.4.1 Rewind file – rewind()

w, let us see "What is the need for the function rewind()?" Explain. The function ind() is used to set the file pointer to the beginning of the file. When the data is tten into the file, the file pointer points to the end of file as shown in figure below:



this to happen, it is required that file should have been opened in **write mode.** w, to process the entire file, we may have to start reading from the beginning of the

file. In such situations, we have to close the file and open it in **read mode.** Instead, w can use the function rewind() which will move the file pointer to the beginning of th file without closing and re-opening the file. After executing the function rewind(), th file pointer points to the beginning of the file as shown below:



File position after rewind

Earlier we wrote the data and now we are reading the data i.e., read and writ operations are being performed on file. So, always the file should be opened i read/write mode using the mode "r+" or "rb+". The syntax is shown below:

rewind(fp);

whose prototype is defined in header file "stdio.h" as:

void rewind(FILE *fp);

5.3.4.2 current location – ftell()

Now, let us see "What is the need for the function ftell()?" Explain. The functio ftell() gives the current position of the file pointer from the beginning of the file. So, i always gives the number of bytes from the beginning of the file relative to zero. Th syntax is shown below:

ftell(fp)

where fp is the file pointer. The prototype is defined in header file "stdio.h" as

long ftell(**FILE** *fp);

For example, consider the following figures:

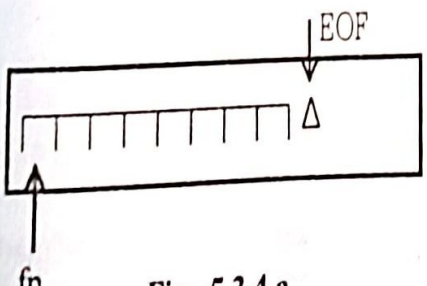


Fig. 5.3.4.a

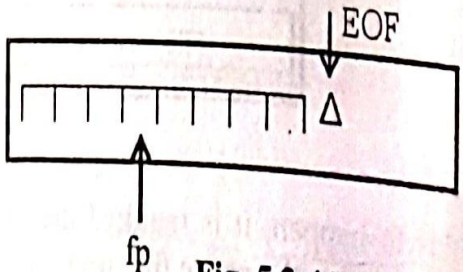


Fig. 5.3.4.b

Note that with respect to figure 5.3.4.a, ftell(fp) returns 0 and with respect to figure 5.3.4.b, the function ftell(fp) returns 3.

5.3.4.3 Set position – fseek()

Now, let us see "What is the need for the function fseek()?" Explain. The function fseek() is used to set the file pointer at the specified position. In general, it is used to move the file pointer to the desired position within the file. The file pointer can be moved backwards or forward any number of bytes. The syntax is shown below:

Syntax

```
fseek(fp, offset, start_point);
```

where

- **fp** is a file pointer.
- **offset** can take positive, negative or zero value and specifies the number of bytes to be moved from the location specified by **start_point.**
- **start_point** can take one of the values as shown below:

Constant	Value	Position of the file
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

Note: The above constants and the following prototype

```
int fseek(FILE *fp, long offset, int start_point);
```

is defined in header file "stdio.h".

Return values The function returns the following values

- value 0 on success
- non-zero value on failure

If offset is positive, the file pointer fp moves forwards. If it is negative, the file pointer fp moves backwards.

Example 5.3.4.3.1: Various methods of moving the file pointer **fp** within the file

- fseek(fp, 0, 0) – file pointer **fp** moves to the beginning of the file
- fseek(fp, 0, 1) - file pointer **fp** stays in the current position and is rarely used
- fseek(fp, 0, 2) - file pointer **fp** moves to the end of file
- fseek(fp, n, 0) - file pointer **fp** moves n bytes from the beginning of the file
- fseek(fp, n, 1) – file pointer **fp** moves n bytes from the current position of file pointer
- fseek(fp, -n, 1) – file pointer **fp** moves n bytes backwards from the current position of file pointer
- fseek(fp, -n, 2) – file pointer **fp** moves n bytes backwards from the end of file.

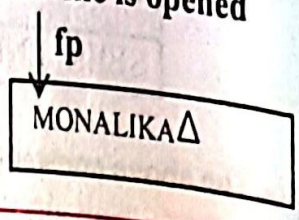
The following program computes the size of a given file.

Example 5.3.4.3.2: C Program to print the size of a given file

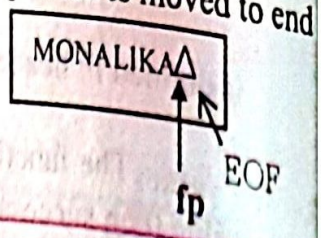
```
#include <stdio.h>
#include <process.h>

long file_size(FILE *fp)
{
        long length;

        /* Move file pointer to end of the file */
        fseek(fp, 0L, 2);

        /* Obtain the current position of file pointer */
        length = ftell(fp);
        return length;
}
```

TRACING

When file is opened



file pointer is moved to end



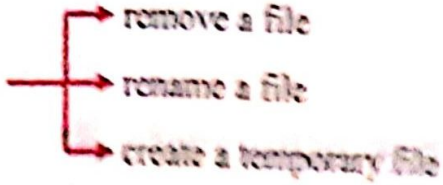
length = 8

```
id main(void)

    FILE  *fp;
    char   file_name[20];

    printf ("Enter the file name\n");
    scanf("%s",file_name);

    fp = fopen(file_name,"r");

    if (fp == NULL)
    {
          printf("Error in opening the file\n");
          exit(0);
    }

    printf("Filesize of %s is %ld bytes\n",file_name, file_size(fp));

    fclose(fp);
```

ote: What is the difference between rewind(fp) and fseek(fp,0L,0)? Both the ctions moves the file pointer to the beginning of the file. So, Functionality is same, syntax and number of parameters are different.

3.5 System File Operations

far we have seen various file functions that are used to access the contents of the . There are some functions that operate on whole files instead of contents. These ctions use operating system calls and are called system file operations. Now, let us "What are system file operations?" The various system file operations that are ported are shown below:



w, let us discuss each of these one by one.

5.3.5.1 Remove a file – remove()

Now, let us see "How to remove or delete a file?" The function **remove()** is used to delete or remove a file. The prototype declaration is shown below:

```
int remove (char *filename);
```

This declaration is available in the header file "stdio.h". The parameter **filename** is a pointer to the name of the file. "What does this function return?"

Return values The function returns the following values

- value 0 on success
- non-zero value on failure

Example 5.3.5.1.1: Program to delete a file

```
#include <stdio.h>

void main(void)
{
        char   file_name[20];
        int            status;

        printf ("Enter the file name\n");
        scanf("%s",file_name);


        status = remove(file_name);

        if (status != 0)
        {
                printf("Error: File can not be deleted\n");
                return;
        }

        printf("The file %s is successfully deleted\n",file_name);
}
```

5.2 Rename a file – rename()

, let us see "How to rename a file?" The function rename() is used to rename a . The prototype declaration is shown below:

```
int rename (char *old_filename, char *new_filename);
```

declaration is available in the header file "stdio.h". After executing the function, filename is renamed as **new_filename** and old file name does not exist "What this function return?"

n values The function returns the following values

- value 0 on success
- non-zero value on failure

5.3.5.2.1: Program to rename a file

```
ude <stdio.h>

main(void)

  char   old_file_name[20];
  char   new_file_name[20];
  int    status;

  printf ("Enter the old file name\n");
  scanf("%s",old_file_name);

  printf ("Enter the new file name\n");
  scanf("%s",new_file_name);

  status = rename(old_file_name, new_file_name);

  if (status != 0)
  {
        printf("Error: File can not be renamed\n");
        return;
  }

  printf("The file is successfully renamed\n");
```

Syntax

```
int getc(FILE *fp);
```

On success, the function returns the character pointed to by file pointer fp. The file pointer fp will automatically point to next character in the input stream. If there is any error or end-of-file is encountered, the function returns EOF.

Syntax

```
int putc(int ch, FILE *fp);
```

On success, writes the character stored in ch to the stream pointed to by the file pointer fp. If there is any error or end-of-file is encountered, the function returns EOF.

mple 5.4.1.1: Program to copy one file to other file using getc() and putc()

```
clude <stdio.h>
clude <process.h>

d main()

    FILE  *fp1;          /* Points to the input file */
    FILE  *fp2;          /* Points to the output file */
    char   file_1[10];   /* Name of the input file */
    char   file_2[10];   /* Name of the output file */
    int    ch;           /* used to store character read */

    printf("Enter the input file\n");
    scanf("%s",file_1);

     /* Open the input file only in read mode */
    if ( (fp1 = fopen(file_1,"r")) == NULL)
    {
          printf("Opening input file failed\n");
          exit(0);
    }

    printf("Enter the output file\n");
    scanf("%s",file_2);

     /* Open the output file only in write mode */
    fp2 = fopen(file_2,"w");
```

5.3.5.3 Create a temporary file – tmpfile()

Now, let us see "What is the purpose of using tmpfile() function?" This function used to create a temporary binary file in "wb+" mode. So, the user can read or wr from this temporary file. This file is created temporarily because; it is automatica deleted when the program is terminated. The prototype declaration is shown below:

```
FILE *tmpfile();
```

This declaration is defined in the header file "stdio.h "What does this funct return?"

Return values The function returns the following values

- Pointer to a file of type FILE success
- NULL on failure

To create a temporary file, we must define a file pointer and then open it as shown below:

```
FILE *fp;

fp = tempfile();
```

5.4 Using text files

The various file handling functions that are associated with text files are shown below:

Text file handling functions
- → Character related functions Ex: getc(), putc()
- → String related functions Ex:fscanf(), fprintf()

Now, let us discuss each of these functions in coming sections.

5.4.1 getc(), putc()

Now, let us see "What is the use of getc() and putc() functions? Explain with synt The function getc() is a macro defined in stdio.h that gets one character from the f pointer specified. The function putc() is also a macro that outputs a character to a f stream specified using file pointer. The syntax of getc() and putc() functions a shown below:

5.3.5.3 Create a temporary file – tmpfile()

Now, let us see "What is the purpose of using tmpfile() function?" This functio used to create a temporary binary file in "wb+" mode. So, the user can read or w from this temporary file. This file is created temporarily because; it is automatic deleted when the program is terminated. The prototype declaration is shown below

```
FILE *tmpfile();
```

This declaration is defined in the header file "stdio.h "What does this funct return?"

Return values The function returns the following values

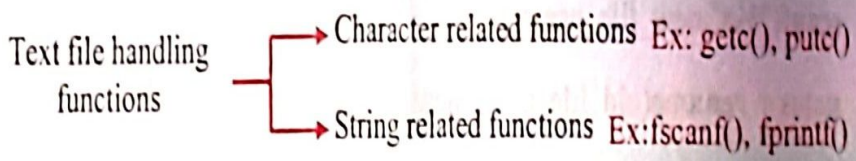
- Pointer to a file of type FILE success
- NULL on failure

To create a temporary file, we must define a file pointer and then open it as shown below:

```
FILE *fp;

fp = tempfile();
```

5.4 Using text files

The various file handling functions that are associated with text files are shown below:



Now, let us discuss each of these functions in coming sections.

5.4.1 getc(), putc()

Now, let us see "What is the use of getc() and putc() functions? Explain with syn The function getc() is a macro defined in stdio.h that gets one character from the pointer specified. The function putc() is also a macro that outputs a character to a stream specified using file pointer. The syntax of getc() and putc() functions shown below:

Syntax

```
int getc(FILE *fp);
```

On success, the function returns the character pointed to by file pointer fp. The file pointer fp will automatically point to next character in the input stream. If there is any error or end-of-file is encountered, the function returns EOF.

Syntax

```
int putc(int ch, FILE *fp);
```

On success, writes the character stored in ch to the stream pointed to by the file pointer fp. If there is any error or end-of-file is encountered, the function returns EOF.

Example 5.4.1.1: Program to copy one file to other file using getc() and putc()

```
include <stdio.h>
include <process.h>

oid main()

    FILE  *fp1;          /* Points to the input file */
    FILE  *fp2;          /* Points to the output file */
    char  file_1[10];    /* Name of the input file */
    char  file_2[10];    /* Name of the output file */
    int   ch;            /* used to store character read */

    printf("Enter the input file\n");
    scanf("%s",file_1);

     * Open the input file only in read mode */
    if ( (fp1 = fopen(file_1,"r")) == NULL)
    {
        printf("Opening input file failed\n");
        exit(0);
    }

    printf("Enter the output file\n");
    scanf("%s",file_2);

     * Open the output file only in write mode *
    fp2 = fopen(file_2,"w");
```

```
    if ( fp2 == NULL )
    {
        printf("Opening outfile failed\n");
        exit(0);
    }

    /* read till end of file is encountered */
    while ( (ch = getc(fp1)) != EOF)
    {

        putc(ch, fp2);        /* write each character read into output file */
    }

    /* close both input and output files */
    fclose(fp1);
    fclose(fp2);
}
```

Note: The above program is self-explanatory with proper comments and the rea
should type the program, execute and understand how the program works.

Example 5.4.1.2: C Program to concatenate two input files

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

void main()
{
    FILE *fp1;          /* Pointer to first input file */
    FILE *fp2;          /* Pointer to second input file */
    FILE *fp3;          /* Pointer to output put file */

    char file1[20];     /* First input file name */
    char file2[20];     /* Second input file name */
    char file3[20];     /* Output file name */
    char ch;            /* Holds the character from both input files */

    printf("Enter the first file\n");
    scanf("%s",file1);
```

```
/* Open the first input file only in read mode */
if ( (fp1 = fopen(file1,"r")) == NULL )
{
        printf("Error in opening the first file\n");
        exit(0);
}
printf("Enter the second file\n");
scanf("%s",file2);
/* Open the second input file only in read mode */
if ( (fp2 = fopen(file2,"r")) == NULL )
{
        printf("Error in opening the second file\n");
        fclose(fp1);
        exit(0);
}
printf("Enter the third file\n");
scanf("%s",file3);
/* Open the output file only in write mode */
if ( (fp3 = fopen(file3,"w")) == NULL)
{
        printf("Error in opening the third file\n");
        fclose(fp1);
        fclose(fp2);
        exit(0);
}
/* Input file is copied to output file */
while ( (ch = getc(fp1)) != EOF) putc(ch,fp3);
/* Input file is appended to output file */
 while ( (ch = getc(fp2)) != EOF) putc(ch,fp3);
fclose(fp1);
fclose(fp2);
fclose(fp3);
```

Example 5.4.1.3: Program for counting the characters, blanks, tabs and lines in file.

```
#include <stdio.h>

void main()
{
        FILE *fp;                       /* Pointer to the input file name */

        char  file_name[20];            /* Input file name */
        char  ch;                       /* Holds the character read from file */

        int   char_count = 0;           /* Holds count of all characters typed */
        int   blank_count = 0;          /* Holds count of blank characters */
        int   tab_count = 0;            /* Holds count of only tabs */
        int   line_count = 0;           /* Holds count of lines */

        printf("Enter the file name\n");
        scanf("%s",file_name);

        /* Open file only in read mode */
        if ( (fp = fopen(file_name,"r")) == NULL )
        {
                printf("Error in opening the file\n");
                exit(0);
        }

        /* read each character till end of file encountered */
        while ( ( ch = getc(fp) ) != EOF )
        {
                char_count++;                   /* Update character count */

                if ( ch == ' ' ) blank_count++;         /* Update blank count */

                if ( ch == '\n' ) line_count++;         /* Update line count */

                if ( ch == '\t' ) tab_count++;          /* Update tab count */
        }

        fclose(fp);

        printf("Number of characters = %d\n", char_count);
        printf("Number of tabs = %d\n", tab_count);
        printf("Number of lines = %d\n", line_count);
        printf("Number of blanks = %d\n", blank_count);
}
```

2 fscanf(), fprintf()

v, let us see "What is the use of fscanf() function? Explain with syntax"

function of **fscanf** and **scanf** are exactly same. But, instead of getting the input n the keyboard, we get the input from the file specified. Because input is read n the file, extra parameter file pointer has to be passed as the parameter. Rest of functionality of fscanf remains same as scanf. The syntax of fscanf() is shown ow:

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

re
fp is a file pointer
format **string** and **list** have the same meaning as in scanf() i.e., the variables specified in the **list** will take the values from the file specified by **fp** using the specifications provided in format string.

ample 5.4.2.1: Explain what will happen if the following statement is executed.

```
fscanf(fp, "%d %s %f", &id, name, &salary);
```

ution: After executing fscanf, the values for the variables **id**, **name** and **salary** are ained from the file associated with file pointer **fp**. This function returns the number tems that are successfully read from the file.

w, let us see "What is the use of fprintf() function? Explain with syntax"

e function of **fprintf** and **printf** are exactly same. But, instead of displaying the ult on the screen, the result is sent to the file. Since file is used, extra parameter file nter has to be passed as parameter. Rest of the functionality of fprintf remains ne as printf. The syntax of fprintf() is shown below:

```
fprintf(fp, "format string", list);
```

ere
fp is a file pointer associated with a file that has been opened for writing.
format string and **list** have the same meaning as in printf() function i.e., the values of the variables specified in the **list** will be written into the file associated with file pointer **fp** using the specifications provided in format string.

Example 5.4.2.2: Explain what will happen if the following statement is executed.

```
fprintf(fp, "%d %s %f", id, name, salary);
```

Solution: After executing fprintf, the values of the variables **id, name and salary** are written into the file associated with file pointer **fp**. This function returns the number of items that are successfully written into the file.

Example 5.4.2.3: Structure definition for the following table:

USN	Name	Marks1	Marks2	Marks3
+ve integer	25 characters	+ve integer	+ve integer	+ve integer

Solution: The above table can be represented using structure definition as shown below:

```
typedef struct
{
        int     usn;
        char    name[20];
        int     marks1;
        int     marks2;
        int     marks3;
} STUDENT;
```

Note: In the above definition, **STUDENT** is the user-defined data type

Example 5.4.2.4: C function to search for **key_usn** in the table shown in example 5.6.2.3.

```
/* function to search for USN */
int search_record(int key_usn, FILE *fp)
{
        STUDENT st;
```

```
/* While not end of file */
for (;;)
{
        /* Obtain the student details from the file */
        fscanf(fp,"%d %s %d %d %d",
                &st.usn, st.name, &st.marks1, &st.marks2, &st.marks3);

        if (feof(fp)) break;                    /* If end of file quit */

        if (key_usn == st.usn) return 1;        /* search successful */
}
return 0;                                       /* Search failed */
```

mple 5.4.2.5: C function to append the student record at the end of file
he structure shown in example 5.6.2.3.

```
o append the student record at the end of the file */
 append_record(FILE *fp)

  STUDENT   st;      /* Holds the details of a student */

  printf("USN   : ");          scanf("%d",&st.usn);
  printf("Name  : ");          scanf("%s",st.name);
  printf("Marks 1: ");         scanf("%d",&st.marks1);
  printf("Marks 2: ");         scanf("%d",&st.marks2);
  printf("Marks 3: ");         scanf("%d",&st.marks3);

  /* Write the student details into the file */
  /* Note: Insert \n at the end of each record as a delimiter */
  fprintf(fp,"%d %s %d %d %d\n",
          st.usn, st.name, st.marks1, st.marks2, st.marks3);
```

mple 5.4.2.6: C function to display the contents of the file for the structure shown
ample 5.6.2.3.

```
display_records(FILE *fp)

  STUDENT st;
```

```
        printf("USN  Name    Marks1  Marks2  Marks3\n");
        for (;;)
        {
                fscanf(fp, "%d %s %d %d %d",
                        &st.usn, st.name, &st.marks1, &st.marks2, &st.marks3);

                if (feof(fp)) break;

                printf("%-5d %-10s %-5d\t %-5d\t  %-5d\n",
                        st.usn, st.name, st.marks1, st.marks2, st.marks3);
        }
}
```

Example 5.4.2.7: The complete C program to create a sequential file with at lea records, each record having the following structure:

USN	Name	Marks1	Marks2	Marks3
+ve integer	25 characters	+ve integer	+ve integer	+ve integer

where USN stands for University Seat Number, Marks1, Marks2 and Marks3 are marks scored by a student in subject1, subject2 and subject3 respectively. Let us v necessary functions to

- Display all the records in the file
- To search for a specific record based on USN. Display suitable message

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

typedef struct
{
        int     usn;
        char    name[20];
        int     marks1;
        int     marks2;
        int     marks3;
} STUDENT;
```

lude: **Example 5.4.2.4:** C function to search for the key USN */

lude: **Example 5.4.2.5:** C function to append the student record */

lude: **Example 5.4.2.6:** C function to display the student records */

```
nain()

 STUDENT   st;          /* Student details are stored */
 char      fname[10];   /* File name where student info is stored */
 FILE      *fp;         /* File pointer to access the contents*/
 int       key_usn;     /* University seat number */
 int       flag;        /* Holds either true or false */
 int       choice;      /* Select the various options add,search,display *

printf("Enter the file name\n");
scanf("%s",fname);

for (;;)
{
        printf("1:Insert Record  2:Search record\n");
        printf("3:Display record 4:Quit\n");
        printf("Enter the choice\n");
        scanf("%d",&choice);

        switch(choice)
        {
                case 1:
                        fp = fopen(fname,"a+");

                        if (fp == NULL)
                        {
                                printf("Fopen failed\n");
                                break;
                        }

                        append_record(fp);

                        fclose(fp);

                        break;
```

```
            case 2:
                fp = fopen(fname,"r");
                if (fp == NULL)
                {
                    printf("File open failed\n");
                    break;
                }

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

                flag = search_record(key_usn,fp);

                if (flag == 0)
                    printf("Unsuccessful search\n");
                else
                    printf("Successful search\n");

                fclose(fp);

                break;
            case 3:
                fp = fopen(fname,"r");

                if (fp == NULL)
                {
                    printf("File opened failed\n");
                    break;
                }

                display_records(fp);

                fclose(fp);
                break;

            default:
                exit(0);
        }
    }
}
```

```
                case 2:
                        fp = fopen(fname,"r");
                        if (fp == NULL)
                        {
                                printf("File open failed\n");
                                break;
                        }

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

                        flag = search_record(key_usn,fp);

                        if (flag == 0)
                                printf("Unsuccessful search\n");
                        else
                                printf("Successful search\n");

                        fclose(fp);

                        break;
                case 3:
                        fp = fopen(fname,"r");

                        if (fp == NULL)
                        {
                                printf("File opened failed\n");
                                break;
                        }

                        display_records(fp);

                        fclose(fp);
                        break;

                default:
                        exit(0);
        }
    }
}
```

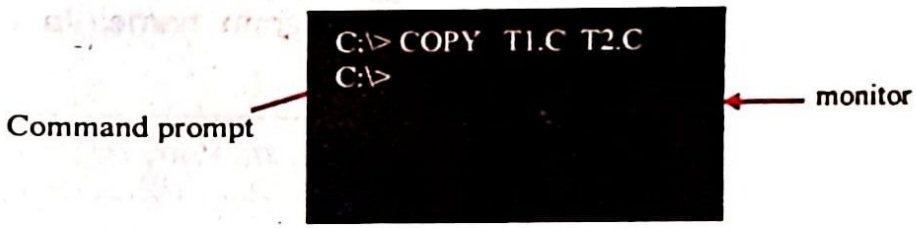
Note: Even though the program is lengthy, the program is straightforward and easier to understand. By looking at the comments the reader is required to understand the logic.

5.5 Command line arguments

The operating systems such as MS-DOS provide a command line interface. Now, shall see "What is command line interface?"

Definition: The interface which allows the user to interact with the computer providing instructions in the form of typed commands (which are in text form called command line interface. In the command prompt (see the figure below), user types the commands. These commands are made up of equal-sized alphanume and other symbols. Keyboard is the input device to enter these commands.

The screen shot of MS-DOS (MicroSoft Disk Operating System) is sh... below:



The above function COPY accepts the file T1.C and copies the contents of T1.C ... the T2.C. In the command prompt, if the user types "COPY T1.C T2.C" the ... function of COPY program accepts three parameters namely COPY, T1.C and T2.C These parameters are called command line arguments. Now, we shall see command line arguments?"

Definition: As parameters are passed to the functions, on similar lines, parameters can also be passed to the function main whenever the program is invoked from the command prompt. The words that are typed at the command prompt are passed to the function main of the program which is being invoked at the command prompt. These arguments that are passed by the operating system to the function main when the program is invoked are called command line arguments. To access the command line parameters the function main should have the following format:

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

where

- **argc** is an integer value. Here, **argc** means argument count
- **argv** is an array of strings. Here, **argv** means argument vector. Those strings are part of the command prompt are copied into argv[0], argv[1] and so on.

Example 5.5.1: Argument vector and argument count with respect to the command

C:\> COPY T1.C T2.C

argv[0]⟶ COPY
argv[1]⟶ T1.C
argv[2]⟶ T2.C

Since there are three strings at the command line prompt, integer variable argc has value 3. The value for the variable argc is not explicitly passed as the parameter argv[0] will always have the first parameter i.e, program name. In our exam argv[0] will have the string "COPY".

5.5.1 Display contents of file

Now, let us write a program to display the contents of file. The file if specified in command line has to be opened and the contents have to be displayed. If file is specified in the command line, the user should be prompted for the file and that should be opened and displayed. The complete program is shown below

Example 5.5.1: C Program to accept a file either through command line or specified by user during run time and displays the contents

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

void main(int argc, char *argv[])
{
        FILE *fp;          /* File pointer used to display a file */
        char fname[10];    /* File to be opened and displayed */
        char ch;           /* Holds the character to be displayed */
```

```
if (argc == 1)
{
        printf("Enter the file name:");        /* No command line file name */
        scanf("%s",fname);
}
else
{
        strcpy(fname, argv[1]);                /* Command line file exists */
}

fp = fopen(fname, "r");

if (fp == NULL)
{
        printf("File opening error\n");
        exit(0);
}

printf("The contents of the file are:\n");
printf("-------------------------\n");

/* File opened successfully and display the contents */
while ( (ch = getc(fp)) != EOF)
{
        printf("%c",ch);
}
```

disp.c

e: Let us assume, the file name of the above program is "disp.c". Before uting this program make sure that the file by name "Hello" exists in the current tory from where we are running the executable file "disp". Also make sure that e text is stored in the file "Hello". Use any editor, create a file called "Hello" and r the following text:

```
Hello how are you
This is the first program using command line arguments
Good luck
Have Fun
```

lly save the program in the file name "Hello". Now, compile and execute the gram as shown below:

Output

```
C:\ disp hello
The contents of the file are:
------------------------------
Hello how are you
This is the first program using command line arguments
Good luck
Have Fun
```

Output

```
C:\>disp
Enter the file name: Hello
The contents of the file are:
Hello how are you
This is the first program using command line arguments
Good luck
Have Fun
```

Note: In the first output argv[0] corresponds to disp and argv[1] corresponds to the string "Hello". In the second output argv[1] is NULL. In this program, in the beginning of the function main we are checking whether a file has been specified in the command line. If file is not specified, the value of argc will be 1 and so the user is prompted to enter the file name during run time. But, if the value of argc is greater than 1, the control comes to else statement and the file specified in the command line is copied to fname which will be used for opening. Rest of the statements in the program are self-explanatory.

5.5.2 Obtain text and create file (use only command line parameters)

Let us consider another program which accepts a file name followed by a text only through command line parameters. The text appearing in the command line should be stored in a file which is also passed as a command line parameter. The corresponding C program is shown below:

5.5.2: C Program to accept a file name and text through command line arguments and create a file with the text

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

main(int argc, char *argv[])
{
  FILE  *fp;              /* File pointer used to display a file */
  int    i;               /* To access command line parameters */
  char   str[30];         /* Temporary storage */

  if (argc == 1)
  {
        printf("File name missing in the command line\n");
        exit(0);
  }

  if (argc == 2)
  {
        printf("File name should be followed by some text\n");
        exit(0);
  }

  /*Command line has file name and text */
  /* So, create and open the file */
  fp = fopen(argv[1], "w");

  if (fp == NULL)
  {
        printf("The file can not be opened for writing\n");
        exit(0);
  }

  /* Write text in command line into file */
  for (i = 2; i < argc; i++)
  {
        fprintf(fp,"%s  ",argv[i]);     /* Note: A space after %s is required */
  }
  fclose(fp);
```

5.5.3 Copy from one file to other file (Implementing copy command)

Let us write a C program to copy contents of a file to another file accepting the fil names as the command line arguments.

Example 5.5.3: C Program to implement copy command

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

void main(int argc, char *argv[])
{
        FILE  *fp1;          /* Source file pointer */
        FILE  *fp2;          /* Destination file pointer */
        int    ch;           /* used to store character read */

        if (argc != 3)
        {
                printf("Src/dest files missing in the command line\n");
                exit(0);
        }

        fp1 = fopen(argv[1], "r");
        if (fp1 == NULL)
        {
                printf("The source file can not be opened for reading\n");
                exit(0);
        }

        fp2 = fopen(argv[2], "w");
        if (fp2 == NULL)
        {
                printf("The destination file can not be opened for writing\n");
                exit(0);
        }

        /* read till end of file is encountered */
        while ( (ch = getc(fp1)) != EOF)
                putc(ch, fp2);        /* write each character read into output file */

        fclose(fp1);
        fclose(fp2);
}
```