FILES :

→ The input and output operations that we have performed were done through screen and keyboard only.

→ After termination of program, all the entered data is lost because primary memory is volatile

→ If the data has to be used later, then it becomes necessary to keep it in permanent storage device

→ C supports the concept of files through which data can be stored on the disk or secondary storage device

→ The stored data can be read whenever required

Definition :

" A file is a collection of related data placed on the disk "

→ The file handling is managed by library functions as well as system calls.

→ The file handling in C can be broadly classified into two types -

i) High level (standard files or stream oriented files)

ii) Low level (system oriented files)

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

→ The header file stdio.h should be included in the program to make use of the I/O functions.

→ The advantage of using stream oriented file I/O is that the I/O in files is somewhat similar to the screen, keyboard I/O. For example, we have functions like fscanf(), fprintf(), fgets(), fputs() which are equivalent the functions like scanf(), printf(), gets(), puts() etc

→ In text files newline is stored as a combination of carriage return '\r' (ASCII 13) and linefeed '\n' (ASCII 10) while in binary files newline is stored only as '\n' (ASCII 10)

→ In binary format, the data is stored in the same way as it is represented in memory so no conversions have to take place while transferring of data between memory and file. In text format, some conversions have to take place while transferring data between memory and file

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

→ The data written using binary format is not very portable since the size of data types and byte order may be different on different machines. In text format, these problems do not arise so it is considered more portable

Concept of Buffer :

→ Buffer is an area in memory where the data is temporarily stored before being written to the file

→ when we open a file, a buffer is automatically associated with its file pointer

→ whatever the data we send to the file is not immediately written to the file. First it send to the buffer and when the buffer is full then its contents are written to the file

→ when the file is closed, all the contents of the buffer are automatically written to the file even if the buffer is not full. This is called flushing the buffer. we can also flush the buffer explicitly by a function fflush()

→ The concept of buffer is used to increase efficiency. Had there been no buffer we would have to access the disk each time for writing even single byte of data. This would have taken lot of time because each time the disk is accessed, the read / write head has to be repositioned. when buffering is done, the data is collected in the buffer and data equal to the size of buffer is written to the file at a time. so the no. of times disk is accessed decreases, which improves the efficiency.

File Name :

File name is divided into two parts.

i) primary name

ii) Extension

→ primary name determines the informations contained in the file

→ Extension is used to know the type of a file

Example : $\underbrace{\text{soft}}_{\text{primary name}}.\overbrace{\text{txt}}^{\text{Extension}}$ → soft.dat ↳ Binary file

Operations involved in file :

The following operations are involved in the file

1) opening the file
2) closing the file
3) creating the file
4) displaying or reading the contents of a file or write data in the file

Opening the file :

→ A file must be opened before any I/O operations can be performed on that file. The process of establishing a connection between the program and file is called opening the file

→ fopen( ) function is used to open a file.

→ syntax <file-pointer> = fopen("file name", "mode");

< file-pointer > is a pointer of FILE type where FILE is a structure already defined inside the header file stdio.h.

→ Syntax for declaring a file pointer is ;

```
FILE * <variable-name> ;
```

→ Example :
```
FILE * fp ;
FILE * a ;
FILE * f1 ; etc
```

→ The file which is to be opened up should be provided in the "file-name" part

→ Example of opening a file   FILE *a ;

$$\begin{tikzcd}
\texttt{a} \arrow[d] & = \texttt{fopen (" APP.txt"} \arrow[d] & \texttt{, "r") ;} \arrow[d] \\
\text{file-pointer} & \text{file-name} & \text{mode}
\end{tikzcd}$$

Note :

→ The file-name that is provided inside the fopen( ) function, by-default it is assumed that it is present in "c:\TC\bin".

→ If any file which we want to open (which does not exist in the default directory i.e c:\TC\bin) then we need to provide the entire path in the file-name in fopen() function

→ Example : If we want to open a file whose name is app.doc and which exist in the drive 'D' inside the folder 'file123' then fopen( ) can be defined as follows.

```
FILE *f1 ;
f1 = fopen ( " D:\\file123\\app doc", "r" ) ;
```

If the file is present in c:\TC\bin then

```
f1 = fopen ( " app.doc", "r");
```

MODES :

→ The second argument in fopen() function represents the mode in which the file is to be opened.

→ The basic modes of a file are ;

1) "r" → read mode
2) "w" → write mode
3) "a" → append mode

Some other modes of a file are ;

4) "r+" → both read and write
5) "w+" → both read and write
6) "a+" → both append and read

1) "r" (read) :

If we want to display or read the content of a file then file should be opened in the "r" mode i.e in the read mode.

Example :
```
FILE *f ;
f = fopen("app.txt", "r");
```

Steps followed by the system when a file will be opened in the "r" mode :

Step1 : system will search for the file specified in the filename part whether it is present or not

Step2 : If the file is present then fopen() function returns the address of first character of the file to the file pointer. Otherwise fopen() function returns NULL value to the file pointer

2) "W" (write) :

If we want to create a file or to overwrite the earlier contained in the file then file should be opened in the "w" or write mode

Example FILE *f ;
f = fopen ("app.txt", "W") .

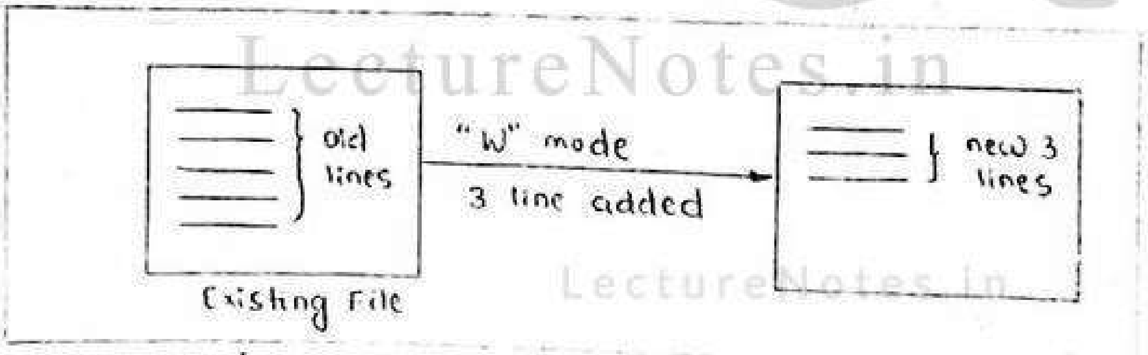
Steps followed by a system while opening a file in "w" mode :

Step 1 : System search for a specified file or for a given file which is there in the file name part

Step 2 If the file is found then fopen() will return the address of first character to the address of the file pointer

Step 3 : If the file is not present then a file will be created in the given name and file pointer will point to the first bit position of the file

Step 4 Suppose a given file is not present and there exist no enough memory for creation of the file then fopen( ) will return NULL to the file pointer



("W" (Write) mode)

3) "a" (append) :

If we want to add some new content at the end of the file then file should be opened in the append mode ("a" mode).

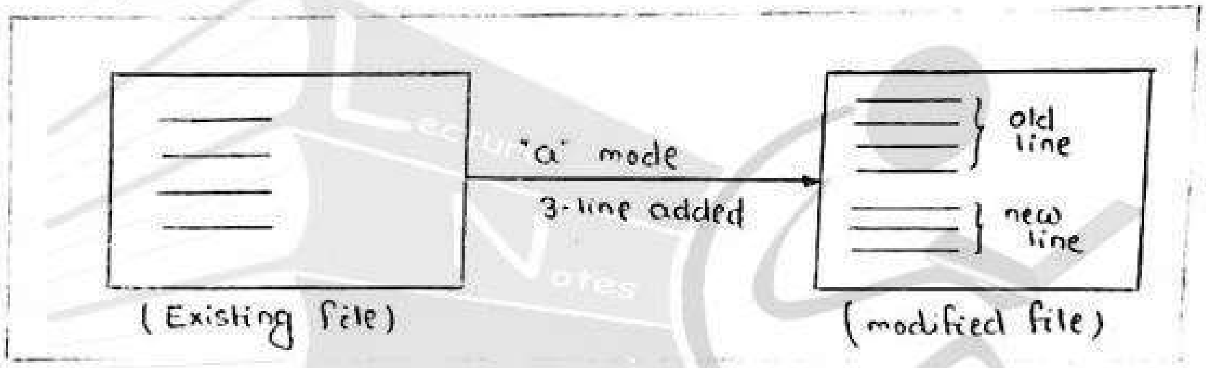
Example FILE *f,
f= fopen (" app.txt". "a") ;

Steps that is followed by the system while opening the file in "a" mode :-

Step1 : It will search for the existence of files provided in the file name part if the given file is present or not . If the given file is present then fopen() returns the address of last character of the given file

step2 : If file is not present then a new file will be created in the given name and file content will point to the final bit position

step3 : If there arises any system problem while opening the file or if there exists no enough memory to create a new file then fopen() return NULL to the file pointer

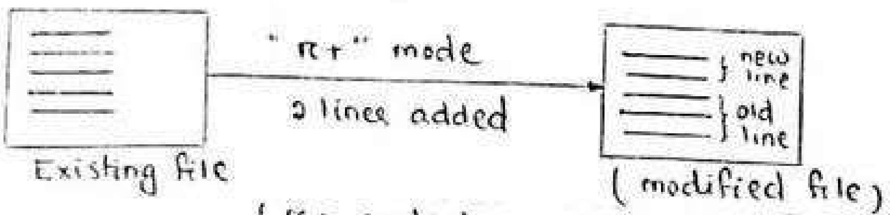


("a" (append) mode)

NOTE : All these modes r, w, a, should be taken as string kind thats why it is always represented as "r", "w" and "a".

Except these 3 modes there are also exists another 3 modified modes 1) "r+" 2) "w+" & 3) "a+"

1) "r+" : operations involved are read and write . Since it is also write the operations, the new contents will modify the earlier content at the begining.



(r+ mode)

2. If we try to create a file but there is no space on the disk or we don't have write permission.

3. If we try to create a file that already exists and we don't have permission to delete that file

4. Operating sytem limits the no of files that can be opened at a time and we are trying to open more files than that number.

→ we can give full name (path name) to open a file. Suppose we want to open a file in DOS whose path is "E:\book\name.txt", the we will have to write as;

```
fp = fopen("E:\\book\\name.txt", "r");
```

Here we have to use double backslash because single back-slash inside string is considered as an escape character, '\b' and '\n' will be regarded as escape sequences if we use single backslash. In UNIX, a single forward slash can be used.

→ Never give the mode in single quotes, since it is a string not a character constant.

```
fr = fopen(" file.dat", 'w'); /* Error */
```

Closing a File:

→ The file that was opened using fopen() function must be closed when no more operations are to be performed on it.

→ After closing the file, connection between file and program is broken

→ Declaration: $\boxed{\text{fclose (file-pointer);}}$

→ On closing the file, all the buffers associated with it are flushed i.e all the data that is in the buffer is written to the file.

→ The buffers allocated by the system for the file are free after the file is closed so that these buffers can be available for other files

→ Some systems have a limitations on the number of files that can be opened at a time, so we should close the files that are not currently in use so that we can open other files

→ Although all the files are closed automatically when the program terminates normally, but sometimes it may be necessary to close the file by fclose() e.g. when we have to reopen the file in some other mode or when we exceed the number of opened files permitted by the system

→ Moreover, it is a good practice to close files explicitly by fclose() when no more operations are to be performed on the file because it becomes clear to the reader of the program that the file has no use now

→ fclose() returns EOF on error and 0 on success (EOF is a constant defined in stdio.h and its value is -1). An error in fclose() may occur when there is not sufficient space on the disk or when the disk has been taken out of the drive.

→ If more than one files are opened, then we can close all the files by calling fclose() for each file

```
        fclose (fptr1);
        fclose (fptr2);
        ---------
        ---------
```

→ We can also close multiple files by calling a single function fcloseall(). It closes all the opened files.

→ Declaration $\boxed{\text{fcloseall ( );}}$

→ On error, fcloseall() returns EOF and on success it returns the no of files closed. We can write it as –

```
    n = fcloseall ();
    if ( n = = EOF)
        printf (" Could not close all open files\n");
    else
        printf (" %d files successfully closed\n", n);
```

End of File :

→ The file reading functions need to know the end of file so that they can stop reading.

→ When the end of file is reached, the operating system sends an end-of-file signal to the program.

→ When the program receives this signal, the file reading functions return EOF, which is a constant defined in the file stdio.h and its value is -1.

→ EOF is an integer value, so the return value of the function is assigned to an integer variable.

→ The value EOF is not present at the end of the file, it is returned by the file reading functions when end of file is reached.

Structure of a General File program :

```
void main()
    {
        FILE *fp;
        fp = fopen("filename", "mode");
        - - - - - - - -
        - - - - - - - -
        - - - - - - - -
        fclose(fp);
    } /* End of main */
```

Predefined File pointers :

Three predefined constant file pointers are opened automati--cally when the program is executed.

	File pointer	Device
1.	stdin	Standard input device (Keyboard)
2	stdout	Standard output device (Screen)
3.	stderr	Standard error output device (Screen)

Functions used for File I/O :

Character I/O - fgetc(), fputc(), getc(), putc()

String I/O - fgets(), fputs()

Integer I/O - getw(), putw()

Formatted I/O - fscanf(), fprintf()

Record I/O - fread(), fwrite()

<1> Character I/O :

a) fputc() :

→ The function fputc() takes one character, it may be a constant and a variable and print this value to the current file pointer position and the file pointer is incremented

→ Syntax: fputc (char variable or Char Constant , file pointer) ;

→ Example: fputc (ch , f1);

→ on success it returns an integer representing the character written, and on error it returns EOF

```
/* program to understand the use of fputc() function */
#include<stdio.h>
void main()
{   FILE *f1 ;
    int ch ;
 if ( f1 = fopen ( "myfile.txt" , "w" )) = = NULL)
      {
        printf ( " File does not exist\n" ) ;
        exit() ;
      }
else  {   printf ( "Enter text  \n" ) ;
       /* press ctrl+z in DOS and ctrl+d in UNIX to stop
          reading characters */
```

```
  while ( ch = getchar()) != EOF )
      fputc (ch, f1) ;
  }
  fclose (f1) ;
}
```

<u>output</u> : Enter text :

```
        Mamata Garanayak
        lecturer in NMIET .
  ^Z
```

After the execution of this program, this text along with the ^Z character will be written to the file myfile.txt.

b) <u>fgetc() :</u>

→ This function reads a single character from a given file and increments the file pointer position.

→ On success it returns the character after converting it to an int without sign extension. On end of file on error it returns EOF.

→ <u>Syntax</u> $\boxed{\text{char variable = fgetc (file pointer) ;}}$

→ <u>Example</u> : ch = fgetc (f1) ;

```
/* program to understand the use of fgetc() */
 # include <stdio.h>
 void main( )
 {   FILE *f1 ;
     char ch ;
  if (( f1 = fopen( "myfile.txt", "r")) = = NULL)
          printf (" This file does not exist \n") ;

  else {    while ((ch = fgetc(f1)) != EOF)
                 printf ("%c", ch) ;
       }
    fclose (f1) ;
 }
```

Output : Mamata Gurunayak
lecturer in NMIET

▨ NOTE : The while loop that we have written in the program is equivalent to the code

```
    ch = fgetc(f1) ;
  while ( ch != EOF )
       { printf ( "%c", ch) ;
         fgetc(f1) ;
       }
```

The value returned by fputc() and fgetc() is not of type char but is of type int. This is because these functions returns an integer value EOF (-1) on end of file or error. The variable ch that is used to store the character read from the file, is also declared to be of int type for this reason only.

<C> getc() and putc() :

→ The operation of getc() and putc() are exactly similar to that of fgetc() and fputc(), the only difference is that the former two are defined as macros while the latter two are functions.

<2> Integer I/o :

a. putw() :

→ This function writes an integer value to the file pointed by a file pointer

→ on success, it returns the integer written to the file and on error it returns EOF.

→ Syntax : $\boxed{\text{putw (integer variable or constant , file pointer) ;}}$

→ Example
```
putw ( 25, f1) ;
putw ( num, fp) ; etc
```

```
/* program to understand the use of putw function */
#include <stdio.h>
void main()
{
    FILE *fptr;
    int value;
  fptr = fopen("num.dat", "wb");          wb → Binary file opened
                                               in write mode.
  for ( value = 1 ; value <= 30 ; value++)
        putw (value, fptr);
    fclose (fptr);
}
```

output : This program will write integers from 1 to 30 into the file "num.dat".

b) getw() :

→ This function returns the integer value from the file associated with file pointer.

→ It returns the next integer from the input file on success and EOF on error or end of file

→ Syntax: $\boxed{\text{integer variable = getw(file-pointer);}}$

→ Example: value = getw(fptr);

```
/* program to understand the use of getw() Function */
 #include <stdio.h>
 void main()
    {
      FILE *fptr;
      int value;
    fptr = fopen("num.dat","rb");        rb → Binary file opened
                                                in read mode
  while ((value = getw(fptr)) != EOF)
             printf ("%d\t", value);
       fclose (fptr);
   }
```

Output : This program will read and print integers from the file "num dat" which was created

NOTE : ~~This program will read as~~ If getw() is used with text files then it will stop reading if integer 26 is present in the file because in text files end of file is denoted by ASCII 26 which is also a valid integer value so getw() should not be used with text files

The value of EOF is -1 which is a valid integer value so this program will work efficiently if -1 is not present in file, if -1 exists in the file then getw() will stop reading and all the values beyond -1 will be left unread so we should use feof() to check ~~and~~ end of file and ferror() to check error .

<3> String I/O :

a. fputs() :

→ This function writes the null terminated string pointed to by str to file

→ The null character that marks the end of string is not written to the file

→ On success it returns the last character written and on error it returns EOF

→ Syntax $\boxed{\text{fputs (string variable , file-pointer);}}$

→ Example fputs (str , fp) ;

```
/* program to understand the use of fputs() */
 # include <stdio h>
 void main( )
 {  FILE *fptr ;
    char  str[80] ;
    fptr = fopen("test.txt", "w") ,
    printf (" To stop entering , press ctrl+d in UNIX & ctrl+z in DOS\n");
```

```
  while  ( gets(str) ! = NULL )
        fputs( str, fptr);
  fclose ( fptr);
}
```

Suppose we enter the text after running the program -.

yester-day is history
Tommorrow is mystery.
Today is a gift
^Z

When the first line of text is entered and enter key is pressed, the function gets() converts the newline character to the null character and the array str contains "yesterday is history" (20 characters + 1 null character) Now str is written to the file text.txt using fputs() The null character is not written to the file, so only 20 characters are written

The function puts() prints the string on the screen. The difference between fputs() and puts() is that puts() translates null character to a newline, but fputs() doesn't. fputs() will write a newline character to the file only if it is contained in the string.

b) <u>fgets():</u>

→ The function is used to read characters from a file and these characters are stored in the string pointed to by str

→ It reads atmost n-1 characters from the file where n is the second argument.

→ fptr is a file pointer which points to the file from which characters are read.

→ This function returns the string pointed to by str on success and on error on end of file it returns NULL.

→ <u>Syntax</u>: fgets (string variable, string length, file pointer);

→ <u>Example</u>: fgets (str, 60, fptr);

```
/* program to understand the use of fgets() */
#include <stdio.h>
void main()
{
    FILE *fptr;
    char str[20];
    fptr = fopen("test.txt","r");
    while ( fgets( str, 20, fptr) != NULL)
          puts( str);
     fclose (fptr);
}
```

output:

```
Yesterday is histor
yTommorrow is Myster
yToday is a gift Tha
t's why it is calle
d present.
```

When fgets() was called with second argument as 20, then it read 19 characters from the file.

gets() reads characters from the standard input while fgets() reads from a file. The difference between fgets() and gets() is that fgets() does not replace the newline character read by the null character, while gets() does. If fgets() read a newline, then both newline and null character will be present in the final string.

In gets() it may be possible that input is more than the size of array, since c does not check for array bounds. So an overflow may occur but in fgets() we can limit the size of input with the help of second argument.

<4> Formatted I/O:

→ The formatted I/O functions can input and output a combination of all of these in a formatted way. Formatting in files is generally used when there is a need to display data on terminal or print data in some format.

a. fprintf() :

→ This function is same as the printf() function but it writes formatted data into the file instead of the standard output (screen).

→ This function has same parameters as in printf() but it has one additional parameter which is a pointer of file type, that points to the file to which the output is to be written.

→ It returns the number of characters output to the file on success and EOF on error.

→ Syntax : $\boxed{\text{fprintf(file-pointer, "formatted String", var1, var2 ...);}}$

→ Example : fprintf (fptr, "%s", sum);

```
/* program to understand the use of fprintf() */

# include <stdio.h>
void main()
{
   FILE *fp ;
   char name[10];
   int age ;
   fp = fopen ( "rec.dat", "w") ;
  printf( "Enter your name and age :");
  scanf ( "%s%d", name, &age);
 fprintf ( fp, "My name is %s and age is %d", name, age) ;
  fclose( fp) ;
}
```

```
/* program to understand the use of fprintf() */
#include <stdio h>
struct student
  { char name [20];
     float marks ;
  } stu ;
```

```
void main()
  {
    FILE *fp;
   int i,n;
   fp = fopen ("students.dat", "w");
  printf ("Enter number of records :");
  scanf (" %d", &n);
 for ( i=1 ; i<=n ; i++)
      {
          printf (" Enter the name and marks :");
          scanf (" %s%f ", stu.name , &stu.marks);
       fprintf ( fp, "%s %f" , stu.name , stu.marks);
    }
}
```

b) fscanf() :

→ This function is similar to the scanf() function but it reads data from file instead of standard input, so it has one more parameter which is a pointer of file type and it points to the file from which data will be read.

→ Syntax : $\boxed{\text{fscanf (file-pointer, "formatted string", \&var1 , \&var2....);}}$

→ Example : fscanf (fp, "%s %d", &str, &num);

```
/* program to understand the use of fscanf() */
 # include <stdio.h>
 struct student
  { char name [20];
     float marks;
  } stu;
void main()
 {  FILE ~~*fopen()~~ , *fp;
   fp = fopen (" students.dat", "r");
   printf (" Name \tMARKS\n");
 while ( fscanf (fp, " %s%f", stu name , &stu.marks) != EOF)
```

```
    printf ("%s\t%f\n", stu.name , stu.marks) ;
  fclose (fp) ;
 }
```

▨ The file pointers stdout and stdin are automatically opened. If we use these file pointers in the functions fprintf() and fscanf(), then there function cells become equivalent to printf() and scanf().

→ If we replace the file pointer fp by stdout then -

```
    fprintf ( stdout , "My age is %s", age) ; is equivalent to
      printf ( "My age is %d", age) ;
```

→ If we replace the file pointer fr by stdin then -

```
   fscanf ( stdin , "%s%d", name , &age) ; is equivalent to
    scanf ( "%s%d", name , &age) ;
```

⟨5⟩ Block Read / Write :

→ It is useful to store blocks of data into file rather than individual elements

→ Each block has some fixed size. it may be a structure or an array.

→ It is easy to read the entire block from file or write the entire block to the file.

→ There are two useful functions for this purpose - fread() and fwrite().

→ Although we can read or write any type of data varying from a single character to arrays and structures through there functions, there are mainly used to read and write structures and arrays.

→ For using these functions, the file is generally opened in binary mode (e.g. "wb", "rb"):

a. fwrite ( ):

→ This function is used for writing an entire block to a given file.

→ size-t is defined in stdio h as -

typedef unsigned int size-t ;

→ ptr is a pointer which points to the block of memory that contains the information to be written to the file , size denotes the length of each item in bytes , n is the no. of items to be written to the file and fptr is a FILE pointer which points to the file to which the data is written.

→ If successful, fwrite() will write n items or total (n * size) bytes to the file and will return n . on error or end of file it will return a number less than n.

→ To write a <u>single float value</u> contained in variable fval to the file

- fwrite (&fval, sizeof (float), 1, fp) ;

→ To write an <u>array of integers</u> arr[10] to the file

- fwrite (arr, sizeof (arr), 10, fp);

→ To write only <u>first 5 elements</u> from the above array to the file

- fwrite (arr, sizeof (int), 5, fp);

Here in 3rd argument we will send size of integer, because here the items that we are writing are integers not array.

→ To write a <u>structure variable</u> which is defined as ;

```
struct record
      {  char name[20] ;
         int roll ;
          float marks ;
      }; student ;
```

- fwrite (& student , sizeof (student), 1, fp);

→ Here sizeof operator is used ~~intead~~ instead of sending the size directly, so that our program becomes portable because the size of data types may vary on different computers.
Moreover if new elements are added to our structure, we need not recalculate and change the size in our program.

```
/* program to understand the use of fwrite() */
#include <stdio.h>
struct record
 {   char name[20];
     int roll;
     float marks;
  } student,
void main()
 {   int n,i;
    FILE * fp;
   fp = fopen("stu.dat","wb");
  if (fp == NULL)
     { printf("Error in opening file\n");
       exit(1);
     }
  printf("Enter the no. of records:");
   scanf("%d",&n);
  for (i=0; i<n; i++)
          { printf("Enter name :");
         scanf("%s", student.name),
         printf(" Enter roll no :");
          scanf("%d", &student.roll);
          printf("Enter mark:");
          scanf("%f", &student.marks);
        fwrite(&student, sizeof(student),1, fp);
  }
  fclose(fp);
}
```

Here ptr is a pointer which points to the block of memory which receives the data read from the file. size is the length of each item in bytes, n is the no. of items to be read from the file and fptr is the file pointer which points to the file from which data is read.

→ on success it reads n items from the file and return n, if error or end of file occurs then it returns a value less than n. we can use feof() and ferror() to check these conditions.

→ To read a <u>single float value</u> from the file and store it in variable fval

→ fread (&fval, sizeof (float), 1, fp) ;

→ To read <u>array of integers</u> from file and store them in an array arr[10]

→ fread (arr, sizeof (arr), 1, fp);

→ To read <u>5 integers</u> from file and store them in first 5 elements of an array arr[10]

→ fread (arr, sizeof (int), 5, fp) ;

→ To read a <u>structure variable</u> that is defined as -

```
struct record
    {
      char name [20] ;
      int roll ;
      float marks ;
    } student ;
```

→ fread (&student, sizeof (student), 1, fp);

```
/* program to understand the use of fread() */
#include <stdio.h>
struct record
  { char name[20];
    int roll;
    float marks;
  } Student;
void main()
 { FILE *fp;
   fp = fopen("stu.dat", "rb");
  if (fp == NULL)
    {
      printf("Error in opening file \n");
      exit(1);
    }
  printf(" \nNAME\t ROLL \t MARKS\n");
  while (fread(&student, sizeof(student), 1, fp) == 1)
      {
        printf("%s\t", student.name);
        printf("%d\t", student.roll);
        printf("%f\n", student.marks);
      }
  fclose(fp);
 }
```

output :

```
        NAME      ROLL     MARKS
        Mamata     20      549.21
        Sarita     21      642.03
        Bijan      22      781.92
```

The fread() returns the no. of records successfully read, so it will return 1 till there are records in the file and will return a no. less than 1 when there will be no records in the file