FILE HANDLING

INTRODUNCTION

All programming allocations are broadly categorized into basic two types:

- Console applications
- File applications

All previous chapter operations are *console applications* (console means computer screen). After terminations of program, the executed output will lost. We can't get the executed output in any way. To avoid such problem and for permanent operations *file applications* concept are used. In real life situation, we require to store the data permanently so that it can be used later. It is therefore necessary to have a more flexible approach where data can be stored permanently. Once we write data to a disk or a secondary storage device it becomes permanent in a sense that it can be reused. The method, which is applied for placing data on a secondary storage, is called *file*.

GENERAL CONCEPTS OF FILES

A file is a place on the disk where a group of related data is stored. In C, an extensive set of library function is available for creating and processing a file. Before going for that let us have a look on the kinds of files we can handle with C.

Types of files

There are two different types of data files present. They are:

- stream-oriented data files
- system-oriented data files

Stream-oriented data files are generally easier to work and are therefore more commonly used. Here, we will be discussing only the stream-oriented data files. Stream-oriented data files are subdivided into two categories. First is the ***text file***, consisting of consecutive characters. It consists of numeric and alpha-numeric character, symbols for different file operations. The second category of stream-oriented data files are referred to as ***unformatted*** data files. Here, data are organized into blocks containing continuous bytes of information. These blocks represent complex data structure like structures or arrays. A separate set of library functions is available for processing files of this type. These library functions provide a single instruction that can transfer the entire arrays or structures to or from data files.

System-oriented data files are used for performing operations on system informations. It uses the the system information such as system date, time, etc for various operations. Generally we will deal with stream-oriented data file applications.

Basing on the ***disk access method***, file can be divide into two types.

- Sequential access file
- Random access file

Your application determines the method you should choose. The access mode of a file determines how you read, write, change, and delete data from the file. Some of your files can be accessed in both ways, sequentially and randomly, as long as your programs are written properly and the data lends itself to both types of file access.

A ***sequential file*** must be accessed in the same order the file was written. It works same as that of accessing information form a cassette tapes. You play music in the same order it was recorded. You can quickly fast-forward or rewind through songs that you do not want to listen to, but the order of the songs must have to be followed for any operations. It is difficult, and sometimes impossible, to insert data in the middle of a sequential file. The only way to insert new information in between two file is that: "The next informations are deleted first. New informations are stored and the previous informations are restored to the next of new information”. This process of insertion of new information involves rewriting of same information multiple times.

Unlike with sequential files, you can access ***random access*** files in any order you want. Like palying songs randomly from a compact disc or a record; you can directly jump to any memory location for faster processing of information. The process does not depends on the sequence of storage on the secondary storage device. Random access files sometimes takes more programming but the main advantage is that, it provides more flexibility of file access method. Both sequential and random access file can be used for stream oriented data file operations.

ESTABLISHING BUFFER AREA

While working with a stream-oriented data file, the first step is to establish a ***buffer area***, where information is temporarily stored while being transferred between computer’s memory and the data file. This buffer area allows information to be read from or written to the data file more rapidly. So it acts as an stream (sequential flow of data in bytes) from where the inflow or outflow of data is possible.

In the program, we set a pointer to this buffer area, which communicate to the data file. Setting a pointer to the buffer area is the first job to perform while working with any stream-oriented data file. This is done by the following codes:

```
FILE *ptr;
```

Where, FILE(upper case letter is required) is special structure type that establishes the buffer area in the memory and the pointer named **ptr** has been set to that memory location.

LEARNING SEQUENTIAL FILE CONCEPTS

There are three operations you can perform on sequential disk files. You can

- Create disk files
- Add to disk files
- Read from disk files

Basing on the applications operations are performed. If you are creating a disk file for the first time, you must create the file and write the initial data to it. Suppose that you wanted to create a customer data file. So first create a new file and write your current customers to that file. The customer data might originally be in arrays or arrays of structures, pointed to with pointers, or typed into regular variables by the user.

Over time, as your customer base grows, you can add new customers to the file. When you add to the end of a file, you *append* to that file. As your customers enter your store, you would read their information from the customer data file.

However, Customer disk processing brings up one disadvantage of sequential files. Suppose that a customer wants to change his or her address in your files. Sequential access files does not allow such operation suitably. It is also difficult to remove information from sequential files. Random files will provide a much easier approach to changing and removing data. The primary approach to changing or removing data from a sequential access file is to create a new one from the old one with the updated data.

NOTE: All file functions described in this chapter use the stdio.h header file.

OPENING AND CLOSING SEQUENTIAL FILES

File operations has a close analogy with the CPU operation. The cabinet has to be opened before working within the CPU. After operation, the cabinet has to be closed. Similarly, you must have to open the file first for any operation. Then it has to be closed to avoid the corruption of file contents.

When you open a disk file, you must inform the C compiler the name of file with proper extension and what you want to do (write to, add to, or read from). C compiler and the operating system work together to make sure that the disk is ready, and they create an entry in your file directory (if the file is alredy created) for the filename. When you close a file, C compiler writes any remaining data to the file, and updates the file directory.

To open a file, you must call the ***fopen( )*** function. To close a file, call the ***fclose( )*** function. The syntax of both functions are:

filePtr = fopen("filename", "file access mode");
and
fclose(filePtr);
Where,
→The *filePtr* is a special type of pointer that points only to files, not to data variables. You must define a file pointer with *FILE **, as a definition in the stdio.h header file. Your operating system handles the exact location of your data in the disk file. You do not want to worry about memory location where the file is stored. So, the *filePtr* is used for opening the file such that it can point the memory location where the file resides in the disk for data read and write operation.
→ The *fileName* is a string (or a character pointer that points to a string) containing a valid filename with proper extension for your computer. Filename can be specified in uppercase or lowercase letters.
→ The *file access mode* specifies the file operation type. The different *file access modes* (those with a plus sign are used in random file processing) are:

Mode	**Description**
"r"	Opens a file for reading
"w"	Opens a file for writing (creates it)
"a"	Opens a file for appending (adding to it)
"r+"	Opens a file for update (reading and writing)
"w+"	Opens a file for update (creates it, then allows reading and writing)
"a+"	Opens a file for update (reads the entire file, or writes to the end of it)

Sometimes, the file access modes are specified with a ***'t'*** or a ***'b'***, such as "rt" or "wb+". The *t* represents the file is a *text file* and is the default mode. Each of the file access modes listed in the above table are equivalent to using *t* after the access mode letter. i.e. "rt" is identical to "r", and so on. A text file is an ASCII file, compatible with most other programming languages and applications. Text files do not always contain text, in the word processing sense of the word. Any data you need to store can go in a text file. Programs that read ASCII files, can read data from the text files specified in the *fopen( )*. The *b* in the file access mode represents the file is a *binary file*.

BINARY MODES

If you specify *b* inside the access mode rather than *t*, C compiler creates or reads the file in a binary format. Binary data files are "squeezed"—that is, they take less space than text files. The disadvantage of using binary files is that other programs cannot always read the data files. Only C programs written to access binary files (using the *b* access mode) can read and write to them. The advantage of binary files is that you save disk space because your data files are more compact. Other than the access mode in the fopen( ) function, you use no additional commands to access binary files with your C programs. The binary format is a system-specific file format. In other words, not all computers can read a binary file created on another computer.

Here is a complete list of binary file access modes:

“rb”, “wb”, “ab”, “rb+”, “r+b”, “wb+”, “w+b”, “ab+”, “a+b”

If you open a file for writing (using access modes of “w”, “wt”, “wb”, or “w+”), C creates the file. If a file by that name already exists, C overwrites the old file with no warning. When opening files, you must be careful that you do not overwrite existing data you want to save.

If an error occurs during the opening of a file, C does not return a valid file pointer. Instead, C returns a file pointer equal to the value NULL. NULL is defined in *stdio.h*. For example, if you open a file for output, but use a disk name that is invalid, C cannot open the file and will make the file pointer point to NULL. Always check the file pointer when writing disk file programs to ensure that the file opened properly.

NOTE: Most of the programmers prefer to open all files at the beginning of their programs and close them at the end. This is not always best. Open files immediately before you access them and close them when you are done with them. This protects the files contents. Keep them open only as long as needed. It is good programming practice to close a file explicitly using fclose function. However most C compiler automatically close a data file at the end of program execution.

The general form of file program is represented as:

```
#include <stdio.h>
void main( ){
        FILE *ptr ;
        ptr= fopen(“sample.dat”,”w”);
        . . . . . . .
        . . . . . . .
        fclose(ptr);
        . . . . . .
        . . . . . .
}
```

E.g. Suppose that you want to create a file for storing your house payment records for the last year. Here are the first few lines in the program that would create a file called house.dat on your disk:

```
#include <stdio.h>
void main( ){
    FILE *filePtr;                      // Declares a file pointer
    filePtr = fopen(“house.dat”, “w”);  // Creates the file
    ……………
    ……………
}
```

The rest of the program writes data to the file. The program never has to refer to the filename again. The program uses the filePtr variable to refer to the file. Examples in the next few sections illustrate how. There is nothing special about filePtr, other than its name (although the name is meaningful in this case). You can name file pointer variables XYZ or a908973 if you like, but these names are not meaningful.

You must include the stdio.h header file because it contains the definition for the FILE * declaration. You do not have to worry about the physical FILE's specifics. The filePtr "points" to data in the file as you write it. Put the FILE *declarations in your programs where you declare other variables and arrays.

NOTE: Because files are not part of your program, you might find it useful to declare file pointers globally. Unlike data in variables, there is rarely a reason to keep file pointers local. Before finishing with the program, you should close the file. The following fclose( ) function closes the house file:

```
fclose(filePtr);       /* Closes the house payment file */
```

2. If you like, you can put the complete pathname in the filename. The following opens the household payment file in a subdirectory on the D disk drive:

```
        filePtr = fopen("d:\mydata\house.dat", "w");  // Creates the file
```

3. If you like, you can store a filename in a character array or point to it with a character pointer. Each of the following sections of code is equivalent:

```
        char fn[ ] = "house.dat";          // Filename in character array
        filePtr = fopen(fn, "w");          // Creates the file
        char *myfile="house.dat";           // Filename pointed to
        filePtr = fopen(myfile, "w");        // Creates the file
/* Let the user enter the filename */
        printf("What is the name of the household file? ");
        gets(filename);            // Filename must be an array or character pointer
        filePtr = fopen(filename, "w");      // Creates the file
```

This fclose( ) function closes the open file, no matter which method you used to open the file:

```
        fclose(filePtr);       /* Closes the house payment file */
```

4. Check the return value from fopen( ) to ensure that the file opened properly. Here is code after fopen( ) that checks for an error:

```
#include <stdio.h>
void main( ){
    FILE *filePtr;            // Declares a file pointer
    filePtr = fopen("house.dat", "w"); // Creates the file
    if (filePtr == NULL){
        printf("Error opening file.\n");
    }
    else{
        /* Rest of output commands go here */
    }
}
```

5. You can open and write to several files in the same program. Suppose that you wanted to read data from a payroll file and create a backup payroll data file. You would have to open the current payroll file using the "r" reading mode, and the backup file in the output "w" mode.

For each open file in your program, you must declare a different file pointer. The file pointers that your input and output statements use determine which file they operate on. If you have to open many files, you can declare an array of file pointers.

Here is a way you can open the two payroll files:

```
#include <stdio.h>
void main( ){
      FILE *fileIn;                          // Input file
      FILE *fileOut;                         // Output file
      fileIn = fopen("payroll.dat", "r");  // Existing file
      fileOut = fopen("payroll.BAK", "w");      // New file
      …………………
      …………………
}
```

When you finish with these files, be sure to close them with these two fclose() function calls:

```
fclose(fileIn);
         fclose(fileOut);
```

WORKING WITH TEXT FILES

As mentioned earlier, text files are handled character-wise or string-wise. When creating a new text file with a specially written program , the usual approach is to enter the information from the keyboard and then write it to the data file. If the data file consists of individual characters, the library functions getchar and putc can be used; **getchar** is used to read individual character from the keyboard, **putc** is used to write a character to the file. It takes the character and the concerned pointer to the file as its parameter.

Now consider the following program, which writes a line of text to a file.

```
#include<stdio.h>
void main( ){
     FILE *ptr;
     char c;
     ptr=fopen("sample.txt","w");
     do {
          c=getchar( );
          putc(c,ptr);
     }while(c!='\n');
     fclose(ptr);
}
```

The line of text, which will be entered to the console, will be written to the file. Now check it out. Go to the directory in which you are working (in which you have saved your program e.g. c:\TC\BIN). You will find a text file named sample. Open the file by double clicking it. You will find the line of text, which you had entered during the program execution.

You can read a text file on the console as the output of the program. For this you have to write a program that reads from a file and write on the console. Here, we will be using **getc** function to read from the file and **putchar** function to write on the console.

Consider the following program.

```
#include<stdio.h>
#include<stdlib.h>
void main(){
    FILE *ptr;
    char c;
    ptr=fopen("sample.txt","r");
    if(ptr==NULL)/*When file is not found*/
        printf("ERROR: file is NOT FOUND");
    else
        do{
            c=getc(ptr); /* Reading from the File */
            putchar(c);  /* Writing on the console */
        }while(!feof(ptr));
        fclose(ptr);
}
```

Here, we have used **feof** function, which indicates an end of file condition of a file. This function returns a non-zero value (TRUE) if an end of file condition has just been detected, and a value of zero (FALSE) if an end of file is not detected.

With this above program you can read any file. Just change the file name with the file name you want to read. Try one! Give the name of the file in which you have saved the file in place of "sample.txt" and run the program. You will find the source program as output.

WRITING TO A FILE

Any input or output function that requires a device performs input and output with files. The most common file I/O functions are
→ getc( ) and putc( )
→ fprintf( ) and fscanf( )
→ fgets( ) and fputs( )

The only difference between fscanf( ) and scanf( ) is its first parameter. The first parameter to fscanf( ) must be a file pointer locating the physical memory location where the file resides.
The following function reads three integers from a file pointed by the filePtr:
fscanf(filePtr, "%d %d %d", &num1, &num2, &num3);

As with scanf( ), you do not have to specify the & before string variable names. The following fscanf( ) reads a string from the disk file:
fscanf(filePtr, "%s", name);

There are many ways present to write data to a disk file. Most of the time, more than one function will work. For instance, if you write many names to a file, both fputs( ) and fprintf( ) may be defined together for operation. You also can write the names using putc( ). You should use whichever function you are most comfortable with for the data being written. If you want a newline character (\n) at the end of each line in your file, the fprintf( ) and fputs( ) probably are easier than putc( ), but all three will do the job. fprintf( ) and fputs( ) can be used for string operations.

NOTE: Each line in a file is called a *record.* By putting a newline character at the end of file records, to provides better readability and understandability.

1. Program using fputs( )

```
#include <stdio.h>
void main( ){
     FILE *fp;
     fp = fopen(“BIODATA.DAT”, “w”);    /* Creates a new file */
     fputs(“Purna Chandra Sethi\n”, fp);
     fputs(“Information Technology\n”, fp);
     fputs(“CET, BBSR\n”, fp);
     fclose(fp);                  /* File close operation */
}
```

2. Program to writes the numbers from 1 to 100 to a file

```
#include <stdio.h>
void main( ){
    int i;
    FILE *fp;
    fp = fopen(“NUMS.txt”, “wt”);   /* Creates a new file */
    if(fp == NULL)
        printf(“Error: File not found.\n”);
    else{
       for(i=1; i<101; i++)
           fprintf(fp,“%d\t“,i);    /* Writes the data */
    }
    fclose(fp);
}
```

The numbers are not written one per line, but with a tab (four characters space) between each of them. The format of the ***fprintf( )*** control string determines the format of the output data. When writing data to disk files, keep in mind that you will have to read the data later. You will have to use “mirror-image” input functions to read data from the files. Notice that this program opens the file using the “wt” access mode. This is equivalent to the “w” access mode because C compiler by default considers the files as text files.

READING FROM A FILE

As soon as the data is in a file, you must be able to read that data. You must open the file in a read access mode. There are several ways to read data. You can read character by character or a string at a time. The choice depends on the format of the data. If you stored numbers using fprintf( ), you might want to use a mirror-image fscanf( ) to read the data.

Files you open for read access (using “r”, “rt”, and “rb”) must exist already. Otherwise, the C compiler will return error. You cannot read a file that does not exist. fopen( ) returns NULL if the file does not exist when you open it for read access. A word can be read at a time from a file using fscanf( ) by %s format specifier.

Another event happens when reading files. Eventually, you read all the data. Subsequent reading produces errors because there is no more data to read. C provides a solution to the end-of-file occurrence. If you attempt to read from a file that you have completely read the data from, C returns the value EOF, defined in stdio.h. To find the end-of-file condition, be sure to check for EOF when performing input from files.

Program to reads the contents of one file and copy it to another

The program must open two files — the first for reading and the second for writing. The file pointer determines which of the two files is being accessed:

```
/* Program */
#include <stdio.h>
#include <stdlib.h>
void main( ){
    char inFilename[12];      /* Holds original filename */
    char outFilename[12];    /* Holds backup filename */
    int  i;                  /* Input character */
    FILE *fpi;
    FILE *fpo;

    printf("\nEnter the name of the file you want to read : ");
    gets(inFilename);

    printf("\nEnter the name of the file to which copy operation to be performed");
    gets(outFilename);
    fpi=fopen(inFilename, "r");
    fpo=fopen(outFilename, "w");
    if (fpi==NULL){
        printf("\n%s does not exist\n", inFilename);
        exit( );
    }
    printf("\nCopying ...\n");
    while (((i = getc(fpi)) != EOF)
        putc( i , fpo);
    printf("\nThe file contents are copied.\n");
    fclose(fpi);
    fclose(fpo);
}
```

WORKING WITH UNFORMATTED DATA FILES

As mentioned earlier, unformatted data file uses blocks of data, where each block consists of fixed number of contiguous bytes. Each block will generally represent a complex data structure, such as a structure or an array. For example, a data file may consist of multiple structures having the same composition or it may contain multiple arrays of same type and size. For such a file it desirable to read entire block from the data file or write the entire block to the data file, rather than reading or writing the individual components as in the previous case.

The library functions **fread** and **fwrite** read from the file and write to the file respectively. Each of these functions requires four arguments: a pointer to the data block, the size of the block, the number data blocks to be transferred and stream pointer. Thus, a fwrite function may be written as below:

fwrite(&data_block, sizeof(data_block), n, ptr)

where,

n→ Number of data blocks
ptr→ Pointer to the concerned file
data_block→ Name of the data structure e.g. structure or array.

Given below is a program, which writes name, branch and roll number of a student. All above data are members of a structure named student.

```
#include <stdio.h>

struct student {
    char name[30];
    char branch[25];
    int roll;
};
struct student input(struct student); //function prototype

void main( ){
    FILE *ptr;
    struct student s;
    ptr=fopen("student.dat","w");
    s=input(s);
    fwrite(&s,sizeof(student),1,ptr);
    printf("\nAbove Data has been written to the File");
    fclose(ptr);
}

struct student input(struct student s){
    int i;
    printf("\nPlease Enter the Following Data:\n");
    printf("\nName : ");
    scanf("%s",s.name);
    printf("\nBranch : ");
    scanf("%s",s.branch);
    printf("\nRoll Number : ");
    scanf("%d",&s.roll);
    return(s);
}
```

The **output** of the above program is:

```
Please Enter the Following Data:
Name : Anand
Branch : CSE
Roll Number : 7
```

Now let us process the above file. For this first we have to read the file and then perform operation on the data read from the file.

Consider the following program.

/* Processing The student File*/

```
#include<stdio.h>
#include<stdlib.h>
struct student {
     char name[30];
     char branch[25];
     int roll;
};
void output(struct student);
void main( ){
    FILE *ptr;
    struct student s;
    ptr=fopen("student.dat","r");
    if(ptr==NULL)
        printf("ERROR: File NOT FOUND");
    else{
        fread(&s,sizeof(student),1,ptr);
        output(s);
    }
    fclose(ptr);
}
void output(struct student s){
    printf("Reading From The File\n");
    printf("---------------------\n");
    printf("Name : %s",s.name);
    printf("\nBranch : %s", s.branch);
    printf("\nRoll Number : %d",s.roll);
}
```

The output of the above program is as below:

```
Reading From The File
---------------------
Name : Anand
Branch : CSE
Roll Number : 7
```

/*Program to Process the student file contents*/

```
#include <stdio.h>
#include <stdlib.h>
struct student {
    char name[30];
    char branch[25];
    int roll;
    int mark[6];
};
void output(struct student);
void main( ){
    FILE *ptr;
    struct student s;
    ptr=fopen("student.dat","r");
    if(ptr==NULL)
        printf("ERROR: File NOT FOUND");
    else{
        fread(&s,sizeof(student),1,ptr);
        output(s);
    }
    fclose(ptr);
}
```

```
void output(struct student s){
    int i, sum=0;
    float avg;
    printf("Reading From The File\n");
    printf("---------------------\n");
    printf("Name : %s",s.name);
    printf("\nBranch : %s", s.branch);
    printf("\nRoll Number : %d",s.roll);
    for(i=0; i<6; i++){
        printf("\nMarks of subject%d : %d",i+1, s.mark[i]);
        sum=sum+s.mark[i];
    }
    avg=sum/6.0;
    printf("\n\nAverage of marks :%2.2f", avg);
}
```

The **output** of the above program is as below:

```
Reading From The File
---------------------
Name : Anand
Branch : CSE
Roll Number : 7
Marks of Subject1 : 60
Marks of Subject2 : 50
Marks of Subject3 : 70
Marks of Subject4 : 80
Marks of Subject5 : 60
Marks of Subject6 : 40
Average of marks : 60.00
```

WRITING TO A PRINTER

The fopen( ) and other output functions were not designed to just write to files. They were designed to write to any device, including files, the screen, and the printer. If you need to write data to a printer, you can treat the printer as if it were a file. The following program opens a FILE pointer using the MS-DOS name for a printer located at LPT1 (the MS-DOS name for the first parallel printer port):

```
/* Prints to the printer device */
#include <stdio.h>
void main( ){
    FILE *prnt;
    prnt = fopen(“LPT1”, “w”);
    fprintf(prnt, “Printer line 1\n”);  // 1st line printed
    fprintf(prnt, “Printer line 2\n”);  // 2nd line printed
    fprintf(prnt, “Printer line 3\n”);  // 3rd line printed
    fclose(prnt);
}
```

Make sure that your printer is turned on and that it has paper before you run this program.
When you run the program, you see this printed on the printer:

```
Printer line 1
Printer line 2
Printer line 3
```

RANDOM FILE RECORDS

Sequential file processing is slow unless you read the entire file into arrays and process them in memory. Random files provide you a way to read individual pieces of data from a file in any order needed and process them one at a time. Generally, you write a stream of characters to a disk file and access that data either sequentially or randomly by reading it into variables and structures. The process of randomly accessing data in a file is simple. Consider the data files of a large credit card organization. When you make a purchase, the store calls the credit card company to get an authorization. Millions of names are present in the credit card company's files. There is no quick way the credit card company could read every record sequentially from the disk. Sequential files does not provide a quick access. In many situations, looking up individual records in a data file with sequential access is not feasible. The credit card companies must use a random file access so that their computers can go directly to your record, just as you go directly to a song on a compact disc.

Reading and writing files randomly is similar to thinking of the file as a big array. With arrays, you know that you can add, print, or remove values in any order. You do not have to start the first array element, sequentially looking at the next one, until you get the element you need. Similarly, the random access file can be accessed in any order.

Most random file records are fixed-length records. That is, each record (usually a row in the file) takes the same amount of disk space. Most sequential files consists of variable-length records. When you are reading or writing data sequentially, there is no need for fixed-length records because you input each value one character, word, string, or number at a time, looking for the data you want. With fixed-length records, your computer can better calculate exactly where the search record is located on the disk.
Although you waste some disk space with fixed-length records (because of the spaces that pad some of the fields), the advantages of random file access compensate for the "wasted" disk space.

OPENING RANDOM ACCESS FILES

Just as with sequential files, you must open random access files before reading or writing to them. You can use any of the read access file modes for random access file operation. The file access modes are:

Mode	**Description**
"r+"	Opens a file for update (reading and writing)
"w+"	Opens a file for update (creates it, and then allows reading and writing)
"a+"	Opens a file for update (reads the entire file, or writes to the end of it)
"r+t"	Opens a text file for update (reading and writing; same as "r+")
"w+t"	Opens a text file for update (creates it, and then allows reading and writing; same as "w+")
"a+t"	Opens a text file for update (reads the entire file, or writes to the end of it; same as "a+")
"r+b"	Opens a binary file for update (reading and writing)
"w+b"	Opens a binary file for update (creates it, and then allows reading and writing)
"a+b"	Opens a binary file for update (reads the entire file, or writes to the end of it)

THE FSEEK( ) FUNCTION

C programming uses a function called ***fseek( )*** that enables us to read to a specific point in a random access data file. Syntax of *fseek( )* is:

fseek(filePtr, longNum, origin);

where,

→ **filePtr** is the pointer to the file that you want to access, initialized with an fopen( ) statement.
→ **longNum** is the number of bytes in the file you want to skip. Skipping the bytes on the disk is much faster than reading them. If longNum is negative, C compiler skips backward in the file (this allows for rereading of data several times). Because data files can be large, you must declare longNum as a long integer to hold a large number of bytes.
→ **origin** is a value that tells C compiler where to begin the skipping of bytes specified by longNum.
The origin can be any of the three values:

Description	Named Origin	Constant
Beginning of file	SEEK_SET	0
Current file position	SEEKCUR	1
End of file	SEEK_END	2

The words SEEK_SET, SEEKCUR, and SEEK_END are defined in stdio.h as the constants 0, 1, and 2, respectively. Actually, the file pointer plays a much more important role than just "pointing to the file" on the disk. The file pointer continually points to the exact location of the next byte to read or write. In other words, as you read data, from either a sequential or a random access file, the file pointer increments with each byte read. By using fseek( ), you can move the file pointer forward or backward in the file.
E.g. To transfer the file pointer back at the beginning the following fseek( ) must have to be used.

```
fseek(fp, 0L, SEEK_SET); // Positions the file pointer at the beginning
```

The constant 0L passes a long integer 0 to the fseek( ) function. Without the L, C compiler would pass a regular integer that would not match the fseek( ) prototype, which is located in stdio.h. This fseek( ) function literally reads "move the file pointer 0 bytes from the beginning of the file."

```
#include <stdio.h>
void main( ){
    FILE *p;
    p=fopen("abc.txt" , "w");
    fseek(p, 0l, SEEK_SET);
    fprintf(p, "HELLO");
    fclose(p);
}
```

You also can close then reopen a file to position the file pointer back at the beginning, but using fseek( ) is a more efficient method. Of course, you could have used regular I/O functions such as printf( ) to write to the screen, instead of having to open the screen as a separate device.

Similarly, the statement "fseek(filePtr, 30L, SEEK_SET);" positions file pointer at the 30th bytes from the beginning of the file. To point to the end of a data file, you can use the fseek( ) function to position the file pointer at the last byte. Subsequent fseek( )s should then use a negative longNum value to skip backward in the file. The following fseek( ) function makes the file pointer point to the end of the file:

fseek(filePtr, 0L, SEEK_END); //positions file pointer at the end

This fseek( ) function literally reads "move the file pointer 0 bytes from the end of the file."

SEEKCUR can also be used in the sam way as that of SEEK_SET and SEEK_END for operation at appropriate position as the refernce of current cursor position.

OTHER HELPFUL I/O FUNCTIONS

Several more disk I/O functions are available that you might find useful. They are mentioned here for completeness. As you write more powerful programs in C, you will find a use for many of these functions when performing disk I/O. Each of these functions is prototyped in the stdio.h header file:

- feof(fp) can be used to test for an end-of-file condition when reading binary files. Unlike text files, C might mistake the binary data for the end-of-file marker. The feof( ) function ensures that the end-of-file condition is properly tested.
- fread(array, size, count, fp) reads the amount of data specified by the integer count, each data size being size in bytes (use the sizeof( ) function), into the array or pointer specified by array. fread( ) is called a buffered I/O function. fread( ) enables you to read much data with a single function call.
- fwrite(array, size, count, fp) writes count array elements, each being size size, to the file specified by fp. fwrite( ) uses a buffered I/O function. fwrite( ) enables you to write much data in a single function call.
- remove(fp) erases the file pointed to by fp. remove( ) returns 0 if the file was erased successfully and "–1" if an error occurred.
- rewind(fp) positions the file pointer at the beginning of the file.

Many of these are helpful functions that you can duplicate using what you already know. For instance, the rewind( ) function simply positions the file pointer at the beginning of a file. rewind( ) is the third method you have seen that does this. These three methods all position the file pointer at the beginning of the file:

→ fclose(fp);   //closes the file, so the cursor position will transfer top the beginning of file
→ fopen(fp, "r");   //opens the file in read mode, so the cursor position will point to the beginning of file
→ fseek(fp, 0L, SEEK_SET); //SEEK_SET represents the beginning of file
→ rewind(fp);   // locates the file cursor position at the beginning of file

COMMAND LINE ARGUMENTS

The function main( ) can accept two parameters. They are: ***argc*** and ***argv***. These are commonly known as command line arguments. The syntax of command line argument is:

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

The first parameter is of type *int* and second is an *array of pointers to string* type. The parameters are commonly defined as: argc (argument counter), argv (argument vector). These are used to access the arguments at the command prompt. The programs are created first, compiled and executable (.exe extension) at the command prompt. Then arguments are supplied to it. The first argument is always the name of the file. The arguments argc and argv can be used to access the command lines. The parameter *argc* represents the number of arguments on the command line. All the arguments supplied at the command line are stored internally as strings and their addresses are stored in the array of pointers named *argv*.

Program to unserstand command line arguments

```
#include <stdio.h>
void main(int argc, char *argv[ ]){
    int i;
    printf("Argc = %d\n", argc);
    for(i=0 ; i<argc ; i++)
        printf("argv[%d] = %s\n", i, argv[i]);
}
```

Suppose, the name of the program is *command.c* and it is executed on the command prompt as:

```
        C commands are interesting and powerful
```

The variable *argc* will have 6 values. Since, total six arguments are supplied at the command prompt, each word is considered as an argument. The first string "C" is located by the base address *argv[0]* and rest strings are considered sequentially. Hence, the output will be:

```
        Argc = 6
        argv[0] = C
        argv[1] = commands
        argv[2] = are
        argv[3] = interesting
        argv[4] = and
        argv[5] = powerful
```

Program that copies a file to another file. The names of two files are sent as command lines.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main(int argc, char *argv[ ]){
    FILE *source, *dest;
    int c;
    clrscr( ) ;
    if(argc !=3){
        printf("Wrong number of arguments");
        exit(1);
    }
    if((source=fopen(argv[1] , "r"))==NULL){
        printf("Can not open the source file");
        exit(1);
    }
    if((dest=fopen(argv[2] , "w"))==NULL){
        printf("Can not open the destination file");
        exit(1);
    }
   while((c=fgetc(source))!=EOF)
        fputc(dest , c) ;
   fclose(source) ;
   fclose(dest) ;
   getch( ) ;
}
```

The topic of file in C is very vast. Here we have made an effort to scratch the surface of the important topic only.

PREVIOUS YEARS QUESTIONS AND ANSWERS

1.a. What are the different ways of accessing a file know?
  b. What kind of file access(es) is/are possible with C-programming?
  c. What are the different modes that a file in C can be handled. [2004-BPUT]

Ans: **(a) Different ways of accessing file:**

Basing on the ***disk access method***, file accessing can be divide into two types.

- Sequential accessing file
- Random accessing file

The application determines the method which will be appropriate to implement. The access mode of a file determines how to read, write, change, and delete data from the file. Some of the files can be accessed in both ways, sequentially and randomly, as long as the programs are written properly and the data lends itself to both types of file access.

A ***sequential file*** must be accessed in the same order the file was written. It works same as that of accessing information form a cassette tapes. Playing of music is done in the same order it was recorded. We can quickly fast-forward or rewind through songs that we do not want to listen to, but the order of the songs must have to be followed for any operations. It is difficult, and sometimes impossible, to insert data in the middle of a sequential file. The only way to insert new information in between two file is that: "The next informations are deleted first. New informations are stored and the previous informations are restored to the next of new information". This process of insertion of new information involves rewriting of same information multiple times.

Unlike with sequential files, we can access ***random access*** files in any order we want. Like palying songs randomly from a compact disc or a record; we can directly jump to any memory location for faster processing of information. The process does not depends on the sequence of storage on the secondary storage device. Random access files sometimes takes more programming but the main advantage is that, it provides more flexibility of file access method. Both sequential and random access file can be used for stream oriented data file operations.

While working with a stream-oriented data file, the first step is to establish a ***buffer area***, where information is temporarily stored while being transferred between computer's memory and the data file. This buffer area allows information to be read from or written to the data file more rapidly. So it acts as an stream (sequential flow of data in bytes) from where the inflow or outflow of data is possible.
In the program, we set a pointer to this buffer area, which communicate to the data file. Setting a pointer to the buffer area is the first job to perform while working with any stream-oriented data file. This is done by the following codes:

```
FILE *ptr;
```

Where, FILE(upper case letter is required) is special structure type that establishes the buffer area in the memory and the pointer named `ptr` has been set to that memory location.

(b) Kind of file:

There are two different types of data files present. They are:

- stream-oriented data files
- system-oriented data files

Stream-oriented data files are generally easier to work and are therefore more commonly used. Here, we will be discussing only the stream-oriented data files. Stream-oriented data files are subdivided into two categories. First is the ***text file***, consisting of consecutive characters. It consists of numeric and alpha-numeric character, symbols for different file operations. The second category of stream-oriented data files are referred to as

unformatted data files. Here, data are organized into blocks containing continuous bytes of information. These blocks represent complex data structure like structures or arrays. A separate set of library functions is available for processing files of this type. These library functions provide a single instruction that can transfer the entire arrays or structures to or from data files.

System-oriented data files are used for performing operations on system informations. It uses the the system information such as system date, time, etc for various operations. Generally we will deal with stream-oriented data file applications.

(c) **Modes of file**

Text file modes:

Mode	**Description**
"r"	Opens a file for reading
"w"	Opens a file for writing (creates it)
"a"	Opens a file for appending (adding to it)
"r+"	Opens a file for update (reading and writing)
"w+"	Opens a file for update (creates it, then allows reading and writing)
"a+"	Opens a file for update (reads the entire file, or writes to the end of it)

Sometimes, the file access modes are specified with a ***'t'*** or a ***'b'***, such as "rt" or "wb+". The *t* represents the file is a *text file* and is the default mode. Each of the file access modes listed in the above table are equivalent to using *t* after the access mode letter. i.e. "rt" is identical to "r", and so on. A text file is an ASCII file, compatible with most other programming languages and applications. Text files do not always contain text, in the word processing sense of the word. Any data you need to store can go in a text file. Programs that read ASCII files, can read data from the text files specified in the *fopen( )*. The *b* in the file access mode represents the file is a *binary file*.

Binary file modes:

If you specify *b* inside the access mode rather than *t*, C compiler creates or reads the file in a binary format. Binary data files are "squeezed"—that is, they take less space than text files. The disadvantage of using binary files is that other programs cannot always read the data files. Only C programs written to access binary files (using the *b* access mode) can read and write to them. The advantage of binary files is that you save disk space because your data files are more compact. Other than the access mode in the fopen( ) function, you use no additional commands to access binary files with your C programs. The binary format is a system-specific file format. In other words, not all computers can read a binary file created on another computer.

Here is a complete list of binary file access modes:

"rb", "wb", "ab", "ab+", "a+b", "wb+", "w+b", "ab+", "a+b"

If you open a file for writing (using access modes of "w", "wt", "wb", or "w+"), C creates the file. If a file by that name already exists, C overwrites the old file with no warning. When opening files, you must be careful that you do not overwrite existing data you want to save.

If an error occurs during the opening of a file, C does not return a valid file pointer. Instead, C returns a file pointer equal to the value NULL. NULL is defined in *stdio.h*. For example, if you open a file for output, but use a disk name that is invalid, C cannot open the file and will make the file pointer point to NULL. Always check the file pointer when writing disk file programs to ensure that the file opened properly.

2. Write C program to read the contents of file “f1” and paste the contents at the beginning of another file “f2” without taking the help of any extra file. [2004S-BPUT]

Program:

```
#include <stdio.h>
#include <conio.h>
void main( ){
    FILE *fp1, *fp2;
    clrscr( );
    fp1=fopen(“f1” , “r”);
    if(fp1==NULL)
       printf(“File not found”);
    else
       printf(“File1 successfully opened”);
    fp2=fopen(“f2” , “w”);
    fp2=fseek(fp2,0,0);
    while(fp1!=EOF){
        fp2=fputc(fgetc(fp1), fp2);
    }
    fclose(fp1);
    fclose(fp2);
    getch( );
}
```

3. Write a C program to count the number of characters, words, lines, spaces and tabs in the input supplied from a file. [2004S-BPUT]

Program:

```
#include <stdio.h>
#include <conio.h>
void main( ){
     char ch;
     int noc, nol, nos, not;
     FILE *fp;
     clrscr( );
     noc= nol=nos=not=0;
     fp=fopen(“abc.txt”, “r”);
     if(fp==NULL)
        printf(“File does not exist”);
     else{
        while(1){
            ch=getc(fp);
            if(ch==EOF)
                break;
            else if((ch>64&&ch<91) || (ch>96&&ch<123))
                noc++;
            else if(ch==’ ’)
                nos++;
            else if(ch==’\n’)
                nol++;
            else if(ch==’\t’)
```

```
                not++;
            }
        }
        printf("\nNumber of characters = %d", noc);
        printf("\nNumber of words = %d", nos+1);
        printf("\nNumber of lines = %d", nol);
        printf("\nNumber of spaces = %d", nos);
        printf("\nNumber of tabs = %d", not);
        fclose(fp);
        getch( );
}
```

4. What do you mean by file? Write the program segment to open a file. [2005-BPUT]
Ans: A file is a collection of related information defined by the creator. File operation is generally done for permanent operation such that the file information can be reused in later applications.
The program segment to open a file is:

```
        FILE *ptr;
        ptr = fopen("file name with proper extension", "mode");
E.g.    FILE *ptr;
        ptr = fopen("abc.txt" , "r"); // It represents the file is opened in read mode
```

5. What is a file? How many files are there in C? Write the syntax for opening a file, reading a file and
closing a file.
[2005-BPUT)
Ans: A file is a collection of related information defined by the creator. File operation is generally done for permanent operation such that the file information can be reused in later applications.

There are ***two types of file*** present in C programming.

- Binary file
- Text file

If a file is specified as the binary type, the file input/output functions does not interprete the contents of the file when they are read from or write to the file. But, if the file is specified as the text type, the file input output functions interprete the contents of the file.

The basic difference between between these two types of files is that: ASCII characters are considered for binary (machine code) operation. Until the end of file is reached the file process continues. Where as, in case of text file the enter key press represents end of line. This concept is generally applied in DOS for file operation

When a file is created, corresponding four files with same name but different extension are created.
E.g. Let the file name is First.c

- First.c → contains the original source code.
- First.exe → executable file.
- First.obj → object file which contains the machine codes.
- First.bak → back up file used for security purpose.

Syntax for opening a file:

```
        FILE *ptr;
        ptr = fopen("file name with proper extension" , "File mode");
```

The different text file modes are:

r → Opening the file for reading only. If the file is not present then the compilert will retrun NULL representing file is
not available.
w → Create a file for read and write operation. If the file is already present then the exsting information will be over
written on the current contents.
a → Open an existing file in append mode that is the contents will be written at the end of file.
r+ → Open an existing file for both read and write operation.
w+ → Create a new file for update operation. If it is already present, the new contents will be over written on the
existing contents
a+ → Open an existing file in append mode i.e. open the file for update at the end of file.
Similiarly, the binary file modes can be defined: rb, wb, ab, rb+, wb+, ab+
Syntax for reading a file:

```
FILE *ptr;
ptr=fopen("file name with proper extension" , "r");
```

Syntax for closing a file:

```
FILE *ptr;
fclose(ptr);      //ptr is the FILE pinter variable which is used for opening the file
```

6. Write a C program to create a file for RD account in a Bank assuming the file structure as RD account number, installment amount and number of installments. [2005-BPUT]
Ans:

```
#include <stdio.h>
struct RD{
    int acno;
    int insamt;
    int insno;
}r;
void main( ){
    FILE *p;
    p = fopen("rdacc.txt" , "a");
    printf("\nEnter the account number, installment amount and number of installments : ");
    fscanf(p , "%d%d%d", &r.acno, &r.insamt, &r.insno);
    fprintf("\n\nAccount Number : %d\nInstallment amount : %d
            \nNumber of installments : %d", r.acno, r.insamt, r.insno);
    fclose(p);
}
```

7. Write a C program to open a file named "BPUT" and write a line of text in it by reading the text from the keyboard. [2005S-BPUT]
Program:

```
#include <stdio.h>
#include <conio.h>
void main( ){
    FILE *fp;
    char str[30];
    clrscr( );
    fp=fopen("BPUT" , "a");
    printf("\nEnter a string : ");
    gets(str);
    fprintf(fp, “%s”, str);
    fclose(fp);
    printf("\nFile successfully opened updated and closed");
    getch( );
}
```

8. Write a C program to remove the comment lines i.e. text with // and /*…*/ from an input C program file. Use command line arguments. [2006-BPUT]

Ans:

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

void main( ){
    FILE *p,*p1;
    char name[15];
    char c1, c2, found='n';
    printf("\nEnter the file name : ");
    scanf("%s",name);
    if((p=fopen(name,"r")) == NULL){
       printf("\nError in opening the file.");
       exit(0);
    }
    p1=fopen(name,"w");
    c2=getc(p);
    c1=getc(p);
    do{
       if(c2=='/'&&c1=='*')
         found='y';
       else if(found=='n')
         fputc(c2,p1);
       if(c2=='*'&&c1=='/'){
         found='n';
         c2=fgetc(p);
         continue;
       }
      c2=c1;
    }while((c1=fgetc(p))!=EOF);
    fclose(p);
    fclose(p1);
}
```

9. Write a program that will accept the country name and its capital, to generate a data file containing the list of countries and their corresponding capital in a structure. [2007S-BPUT]

Ans:

```
#include <stdio.h>
#include <conio.h>
struct country{
   char name[30];
```

```
    char capital[25];
};

void main( ){
    struct country c[100];
    int i, n;
    FILE *p;
    clrscr( );
    p = fopen("country.txt" , "a");
    printf("\nEnter the number of countries  : ");
    scanf("%d", &n);
    printf("\nEnter the name of the countries and the corresponding capitals : ");
    for( i=0 ; i<n ; i++ )
        scanf("%s%s", c[i].name, c[i].capital);
    printf("\nName of country\tCapital\n");
    for( i=0 ; i<n ; i++ )
        printf("\n%s\t %s", c[i].name, c[i].capital);
    for(i=0 ; i<n ; i++)
        fprintf(p,"\nName : %s\tCapital : %s", c[i].name, c[i].capital);
    fclose(p);
    getch( );
}
```

10. Write an iterative C program that will encode or decode a line text. To encode a line of text, proceed as follows:

- Convert each character, including blank space to its ASCII equivalent.
- Generate a positive random integer. Add this to the ASCII equivalent of each character. The same random integer will be used for the entire line of text. [2007S-BPUT]

Ans:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main( ){
    char str[30];
    int i, n;
    clrscr( );
    printf("\nEnter a string : ");
    gets(str);
    printf("\nString is : %s",str);
    /*Conversion to ASCII equivalent*/
    i=0;
    printf("\nASCII equivalent String : ");
    while(str[i]!='\0'){
        printf("%d",str[i]);
        i++;
    }
    /*Random Number operation Section*/
    n=rand( );
    printf("\nString with sum of ASCII and random number : ");
    i=0;
    while(str[i]!='\0'){
       printf("%d",(int)str[i]+n);
       i++;
    }
    getch( );
}
```

11. Write a complete C program to create a file that contains only integers. Create another file and copy contents of the previous file into this newly created file. [2009-BPUT]

Program:

```
#include <stdio.h>
void main( ){
    char inFilename[12];      /* Holds original filename */
    char outFilename[12];    /* Holds backup filename */
    int  i;                   /* Input character */
    FILE *fpi;
    FILE *fpo;

    printf("\nEnter the name of the file that contains only integer to read : ");
    gets(inFilename);

    printf("\nEnter the name of the file to which copy operation to be performed");
    gets(outFilename);

    if ((fpi=fopen(inFilename, "r"))=  =NULL){
        printf("\n%s does not exist\n", inFilename);
        exit( );
    }
    printf("\nCopying ...\n");
    while ((i = getc(fpi)) != EOF)
        putc( i , fpo);
    printf("\nThe file contents are copied.\n");
    fclose(fpi);
    fclose(fpo);
}
```

HOME ASSIGNMENT

1. Different between console application and file application using suitable example.
2. Briefly explain the access modes.
3. Write a program to create two text files. Provide a user defined function that accepts the names of the above two files as arguments and returns 1, if the number of characters is same in both the files or else it will return 0.
4. What is command line argument? Write a C program to copy the content of one file to the other using command line argument.
5. Write a C program to create a student information system for a class of students using file.