UNIT - 6

FILE HANDLING IN C

File Management

So far we have been using scanf and printf functions for reading and writing data to the console. This is fine as long as the data is less. However, many real world problems involve large amounts of data. In such situations, the console oriented I/O pose two major problems. They are:

1. It becomes difficult and time consuming to handle large volumes of data through console.
2. The entire data is lost when either the program is terminated or the computer is turned off.

It is therefore necessary to have a more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This is achieved by using the concept of files.

A file is a collection of related data stored on a disk. C supports a wide range of functions that have the ability to perform basic file operations, which include:

- Naming a file
- Opening a file
- Reading data from a file
- Writing data to a file
- Closing a file

C library provides various pre-defined functions for handling files. Some of them are listed below:

Function	Operation
fopen()	Creates a new file / opens an existing file
fclose()	Closes a file which has been opened for use
getc()	Reads a character from the file
putc()	Writes a character to the file
fprintf()	Write data values to a file
fscanf()	Reads a set of data values from a file
getw()	Reads an integer from the file
putw()	Writes an integer to a file
fseek()	Sets the position to the desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the beginning of the file

Defining and Opening a File

If the user wants to store data or read data from a file in the secondary memory, the user must specify certain things about the file to the operating system. They are:

1. Filename
2. Data Structure
3. Purpose

Filename is the name of the file. It is a collection of characters that make up a valid filename for the operating system. It may contain two parts: a primary name and optional period with the etxtension. Some valid file names are:

abc.txt
prog.c
sample.java
store

Data Structure of a file is defined a **FILE** in the C library. Therefore, all files should be declared as type **FILE** before they are used. **FILE** is a predefined data type. When we open a file, we must specify the

purpose. For example, we may want to write data or read data from a file. The syntax for declaring and opening a file is:

```
FILE *fp;

fp = fopen("filename","mode");
```

The first statement declares the variable **fp** as a pointer to the data type **FILE**. The second statement opens the file whose name is **filename** and assigns an identifier to the **FILE** type pointer **fp**. This pointer, which contains all the information about the file is subsequently used as a communication link between the system and the program. The **mode** specifies the purpose of opening the file. Mode can be one of the following:

- **r** opening the file for reading data from it
- **w** opening the file for writing data to it
- **a** opening the file for appending data to it

Note that both **filename** and **mode** are specified as string. So, both of them must be enclosed within double quotes. Some compilers support the following additional modes:

- **r+** Open the existing file for both reading and writing
- **w+** Open the file for both reading and writing
- **a+** Open the file for both appending and reading

Note: *When a file is opened in **r** mode, the compiler searches for the file and if the file does not exist, nothing happens or some compilers might generate an error. When a file is opened in **w** mode, the compiler searches for the file and if the file does not exist, it creates a new file with the specified name. If the file already exists, the file is opened with the all the previous data in the file erased.*

Closing a File

A file must be closed as soon as all the operations on it have been finished. This ensures that all information associated with the file is flushed out from the buffers and all links to the file with the program are broken. It also prevents the accidental misuse of the file. Another case in which we might want to close the connection with the file is, when we want to reopen the same file in a different mode. The syntax for closing a file is as shown below:

```
fclose(file-pointer);
```

Input/Output Operations on Files

getc and putc Functions

The simplest I/O functions are **getc** and **putc**. These are analogous to **getchar** and **putchar** functions and handle one character at a time. The **putc** function writes a character to the file associated with a file pointer. The syntax is as shown below:

```
putc(ch, file-pointer);
```

Similarly the function **getc** is used to read a character from a file associated with a file-pointer. The syntax is as shown below:

```
getc(file-pointer)
```

The file pointer moves by one character position for every operation of **getc** or **putc**. The **getc** will return an end-of-file marker EOF, when end of the file has been reached.

getw and putw Functions

The **getw** and **putw** are integer oriented functions. They are similar to **getc** and **putc** functions and are used to read and write integers to and from files. These functions would be useful when the user is dealing with integer data. The syntax for these functions is as shown below:

```
putw(integer, file-pointer);

getw(file-pointer);
```

fprintf and fscanf Functions

When the user need to work with mixed data, C provides two functions namely: **fprintf** and **fscanf.** These functions are used to read and write mixed data to and from files. These two functions are similar to **printf** and **scanf** except these two functions work on files. The syntax for these functions is as shown below:

```
fprintf(fp, “control strings”, var list);

fscanf(fp, “control strings”, var list);
```

Error Handling During I/O Operations

While writing programs which involve accessing files, certain errors might occur while performing I/O operations on a file. Some of the error situations include the following:

1. Trying to read beyond the end-of-file mark.
2. Device overflow.
3. Trying to use a file that has not been opened.
4. Trying to perform an operation on a file, when the file is being use by another application.
5. Opening a file with invalid name.
6. Attempting to write to write-protected file.

If such errors are unchecked, it may lead to abnormal termination of the program or may lead to incorrect output. C library provides two functions namely **feof** and **ferror** for handling the above mentioned situations.

The **feof** function is used to test for the end-of-file condition. It takes a file-pointer as a parameter and returns a **non-zero** integer value if all of the data from the specified file has been read, and returns **zero** otherwise. The syntax is as shown below:

```
Int feof(file-pointer)
```

The **ferror** function reports the status of the file. It takes a file-pointer as its argument and returns a **non-zero** integer if an error has been detected up to that point, or returns **zero** otherwise. The syntax is as shown below:

```
int ferror(file-pointer)
```

Whenever a file is opened using **fopen** function, a file pointer is returned. If the file cannot be opened for some reason, the function returns a **NULL** pointer. This can be used to test whether the file has been opened successfully or not. It can be used as shown below:

```
if(fp == NULL)

printf("File cannot be opened!");
```

Random Access to Files

All the functions that we have seen so far are useful for reading and writing data sequentially to and from a file. Sometimes the user might want to access data at random locations from a file. For this purpose, C library provides functions namely: **ftell**, **fseek** and **rewind.**

The **ftell** function lets the user to know the current location of the file pointer. It takes a file-pointer as a parameter and returns a long integer that corresponds to the current position of the pointer. Syntax is a shown below:

```
long ftell(file-pointer)
```

The **rewind** function lets the user to move the file pointer again to the beginning of the file. This function accepts a file-pointer as a parameter and resets the position to the start of the file. This helps us in reading a file more than once, without having to close and reopen the file. Syntax is as follows:

```
rewind(file-pointer)
```

The **fseek** function is used to move the pointer to any position within the file. This function takes three parameters: *file-pointer, offset and position.* When the operation is successful, **fseek** returns a non-zero value. If we attempt to move the file pointer beyond the file boundaries, an error occurs and **fseek** returns a -1. Syntax is as follows:

```
fseek(file-pointer, offset, position)
```

The *offset* is a number of the type long, and position is an integer number. The *offset* represents the number of positions to be moved from the location specified by *position.* The *position* can take one of the following values:

Value	Meaning
0	Beginning of the file
1	Current Position
2	End of file

A positive value for the *offset* specifies that the pointer moves forward and a negative value for the *offset* specifies that the pointer moves backward from the current position. Following are some of the example usages of the **fseek** function:

Statement	Meaning
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position
fseek(fp,0L,2);	Go to the end of file
fseek(fp,-m,1);	Move backward by m bytes from current position
fseek(fp,m,1);	Move forward by m bytes

Command Line Arguments

The parameters passed to the program when the program is invoked are known as command line arguments. These parameters are the additional information like filename or other kind of input to the program. By passing command line arguments there is no need for the user to provide the input while executing the program.

These command line arguments can be processed by using the arguments available in the **main** function. The **main** allows two parameters namely: **argc** and **argv.** The **argc** represents the argument counter which contains the number of arguments passed in the command line. The **argv** is a character pointer array which points to the arguments passed in the command line.

To know the number of command line arguments, we can use the **argc** parameter of the **main** function and to access the individual arguments, we can use the **argv** array. The first element in the **argv** array is always the **program name.** So the first argument can be accessed by using **argv[1]** and so on. The main function will be as shown below:

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

{

        -----

        -----

}
```

Programs

```
/* C program to open and close a file */
#include<stdio.h>
#include<conio.h>
main()
{
   FILE *fp;
   fp = fopen("C:\\xyz.txt", "w");
   fclose(fp);
   getch();
}
```

```
/* C program to read data from a file */
#include<stdio.h>
#include<conio.h>
main()
{
   FILE *fp;
   char ch;
   fp = fopen("C:\\xyz.txt", "r");
   while((ch=getc(fp)) != EOF)
   {
      printf("%c",ch);
   }
   fclose(fp);
   getch();
}
```

```
/* C program to write data to a file */
#include<stdio.h>
#include<conio.h>
main()
{
   FILE *fp;
   char ch;
   fp = fopen("C:\\xyz.txt", "w");
   while((ch = getchar()) != '\n')
   {
      putc(ch, fp);
   }
   fclose(fp);
   getch();
}
```

```
/* C program to append data to a file */
#include<stdio.h>
#include<conio.h>
main()
{
   FILE *fp;
   char ch;
   fp = fopen("C:\\xyz.txt", "a");
   while((ch = getchar()) != '\n')
   {
      putc(ch, fp);
   }
   fclose(fp);
   getch();
}
```

```
/* C program to read numbers and write numbers to a file */
#include<stdio.h>
#include<conio.h>
main()
{
   FILE *fp;
   int i;
   clrscr();
   fp = fopen("abc.txt","w");
   for(i = 1; i <= 10; i++)
   {
      putw(i, fp);
   }
   fclose(fp);
   fp = fopen("abc.txt","r");
   while((i = getw(fp)) != EOF)
   {
      printf("%d ",i);
   }
   fclose(fp);
   getch();
}
```

```
/* C program to read and write mixed type data into files */
#include<stdio.h>
#include<conio.h>
main()
{
   char name[20], grade;
   int age, i;
   FILE *fp;
   clrscr();
   fp = fopen("abc.txt","w");
   printf("Enter Student Details: \n");
   printf("Name\t Age\t Grade\n");
   for(i = 0; i < 3; i++)
   {
      fscanf(stdin, "%s %d %c", name, &age, &grade);
      fprintf(fp, "%s %d %c", name, age, grade);
   }
   fclose(fp);
   fp = fopen("abc.txt","r");
   printf("Student details are: \n");
   for(i = 0; i < 3; i++)
   {
      fscanf(fp, "%s %d %c", name, &age, &grade);
      fprintf(stdout, "%s %d %c\n", name, age, grade);
   }
   fclose(fp);
   getch();
}
```

```
/* C program to demonstrate ftell, fseek and rewind functions */
#include<stdio.h>
#include<conio.h>
main()
{
   FILE *fp;
   long offset;
   char ch;
   clrscr();
   fp = fopen("abc.txt","w");
   printf("Enter some characters: ");
   while((ch=getchar())!= '\n')
   {
      putc(ch, fp);
   }
   printf("Number of characters entered is: %ld\n\n",ftell(fp));
   rewind(fp);
   printf("After rewind file pointer is at location: %ld\n\n",ftell(fp));
   fclose(fp);
   offset = 0L;
   fp = fopen("abc.txt","r");
   while(feof(fp) == 0)
   {
      fseek(fp, offset, 0);
      printf("Position of %c is %ld\n", getc(fp), ftell(fp));
      offset++;
   }
   fseek(fp, -1L, 2);
   printf("\nFile contents in reverse: \n");
   do
   {
      putchar(getc(fp));
   }
   while(!fseek(fp, -2L, 1));
   fclose(fp);
   getch();
}
```

```
/* C program to demonstrate command line arguments */
#include<stdio.h>
#include<conio.h>
main(int argc, char *argv[])
{
   FILE *fp;
   int i;
   printf("Number of arguments is: %d\n",argc);
   fp = fopen("C:\\xyz.txt", "w");
   for(i = 1; i < argc; i++)
   {
        fprintf(fp, "%s ", argv[i]);
   }
   fclose(fp);
   printf("Data Saved!");
   getch();
}
```

Running the program:
C:/tc > tcc file8.c
C:/tc > file8 aaa bbb ccc ddd

aaa, bbb, ccc and ddd are the command line arguments.