Objectives

This session will enable you to:

- Explain the concept of file and its types
- Know the basic file operations
- Perform formatted, unformatted and block file I/O operations
- Access files in both sequential and random order
- Make use of pre-processor directives

Introduction

When a large volume of data is involved, supplying data through the keyboard during the execution or displaying the output on the screen is not convenient. The input data can be stored on disks and the program may access the data from disks for processing. Similarly, the results may be stored on disks. For such applications, files are needed. A file is a place on the disk where a group of related data is stored. In C, file manipulations may be done in two ways:

- Low-level I/O using system calls
- High-level I/O using functions from standard I/O library

The files accessed through the library functions are called *Stream Oriented files* and the files accessed with system calls are known as *System Oriented files*.

Streams and Files

Streams facilitate a way to create a level of abstraction between the program and an input/output device. This allows a common method of sending and receiving data amongst the various types of devices available. There are two types of streams: *text* and *binary*.

Text streams are composed of a set of lines. Each line has zero or more characters and is terminated by a new line character. Text streams consist of printable characters, the tab character, and the new-line character. Conversions may occur on text streams during input and output. Spaces cannot appear before a newline character, a text stream removes these spaces even though implementation defines it. A text stream, on some systems, may be able to handle lines of up to 254 characters long (including the terminating new line character).

Binary streams are composed of only 0's and 1's. It is simply a long series of 0's and 1's. More generally, there need not be a one-to-one mapping between characters in the original file and the characters read from or written to a text stream. But in the binary stream there will be one-to-one mapping because no conversion exists, and all characters will be transferred as such.

When a program begins, there are three available streams:

- Standard input (stdin) is the stream where a program gets its input data
- Standard output (stdout) is the stream where a program writes its output data.
- Standard error (stderr) is another output stream typically used by programs to output error messages.

File Operations

Files are associated with streams and must be open in order to use it. The point of I/O within a file is determined by the file position. When a file is opened, the file position points to the

beginning of the file unless the file is opened for an append operation - in which case the position points to the end of the file. The file position indicates where the next operation (read/write) will occur.

When a file is closed, no more actions can be taken on it until it is opened again. Exiting from the main function causes all open files to be closed.

In C, 'FILE' is a structure that holds the description of a file and is defined in stdio.h.

Basic File operations are:

- Opening a File
- Reading from and/or writing into a File
- Closing the File

The logic is, the code must:

- define a local 'pointer' of type FILE (called file pointer)
- 'open' the file and associate it with the file pointer via fopen()
- perform the I/O operations using file I/O functions (ex. fscanf() and fprintf())
- disconnect the file from the task using fclose()

General form:

```
FILE *fp;
fp = fopen("name", "mode");
fscanf(fp, "format string", variable list);
fprintf(fp, "format string", variable list);
fclose(fp );
```

Where:

- The 'fp' is a file pointer or file handler.
- The 'name' is to represent filename and it is a string of characters. (Extensions can be specified like test.c, details.dat etc)
- The 'mode' argument in the fopen() specifies, the purpose/positioning of opening the file. It is a string enclosed within double quotes.

The 'mode' can be any of the following:

- r read text mode
- w write text mode (truncates file to zero length if it already exits or creates new file)
- a append text mode for writing (opens or creates file and sets file pointer to the end-of-file)
- rb read binary mode
- wb write binary mode (truncates file to zero length if it already exits or creates new file)
- ab append binary mode for writing (opens or creates file and sets file pointer to the end-of-file)
- r+ read and write text mode

- w+ read and write text mode (truncates file to zero length if it already exists or creates new file)
- a+ read and write text mode (opens or creates file and sets file pointer to the end-of-file)
- r+b or rb+ read and write binary mode
- w+b or wb+ read and write binary mode (truncates file to zero length if it already exists or creates new file)
- a+b or ab+ read and write binary mode (opens or creates file and sets file pointer to the end-of-file)

If the file does not exist and it is opened with read mode (r), the file open fails and it will return NULL to file pointer.
If the file is opened with append mode (a), all write operations occur at the end of the file regardless of the current file position.
If the file is opened in the update mode (+), output cannot be directly followed by input and input cannot be directly followed by output without an intervening fseek(), fsetpos(), rewind(), or fflush().
fopen() returns the file pointer position for successful open and returns NULL, if the file does not open or the file does not exist.
fclose() returns zero for successful close and returns EOF (end of file) when error is encountered in closing a file. By default, all the files opened are closed when the program is terminated. It is good to close all the files opened with fopen(), because files can be reopened only if they are closed.
The Standard I/O provides variety of functions to handle files. It supports the following ways of reading from and writing into file:

- Character I/O
- String I/O
- Formatted I/O
- Block I/O
- Integer I/O

Character I/O
Using character I/O, one character (byte) can be written to or read from a file at a time.

Writing in to a file
To write into a file, the file must be opened in 'w' mode The function putc() is used to write a byte to a file.

General Form:
putc(ch,fptr);

This function writes the character ch into a file pointed by the file pointer fptr. This fptr may be stdout, which represents standard output device, monitor as a file. On success, the character is returned. If an error occurs, the error indicator for the stream is set and EOF is returned.

Example 14.1: Program to create a text file (character file)
main()

```
{
FILE *fp;
char c;
if ((fp=fopen("sample.dat","w")) !=NULL)
{
while ((c=getchar()) != EOF)
putc(c,fp);
fclose(fp);
}
else
printf("Error in opening a file");
}
```

Reading from a file
The function getc() is used to read a byte from a file. This may be a macro version of fgetc.

General Form:
ch =getc (fptr);

This function reads a character from the file and it is returned to the program defined character variable. After the reading a character, the pointer is moved to the next position. The fptr may be stdin, which represents a standard input device, keyboard as a file. On success, the character is returned. If the end-of-file is encountered, EOF is returned and the end-of-file indicator is set. If an error occurs, the error indicator for the stream is set and EOF is returned. The EOF is end of file status flag, which is true if end of file is reached, otherwise false.

Example 14.2: Program to read a character data from a text file

```
main()
{
FILE *fp;
char c;
if ((fp=fopen("sample.dat","r")) !=NULL)
{
while ((c=getc(fp)) != EOF)
putchar(c);
fclose(fp);
}
else
printf("Error in opening a file");
}
```

String I/O
Using string I/O, string can be written to, or read from, a file at a time.
Writing a string in to a file
The function used is fputs(). Writes a string to the specified stream till the last character is read but does not include the null character. On success, a nonnegative value is returned. On error,

EOF is returned.

General Form:
fputs (str, fptr);

Reading a string from a file
The function used is fgets(). Reads a line from the specified stream and stores it into the string pointed to by str. It stops when (n-1) characters are read, the newline character is read, or the endof-file is reached, whichever comes first. The newline character is copied to the string. A null character is appended to the end of the string. On success, a pointer to the string is returned. On error, a null pointer is returned. If the end-of-file occurs before any characters have been read, the string remains unchanged.

General Form:
fgets(str,n,fptr);

Numeric I/O
Using numeric I/O, integers can be written to, or read from, a file at a time.

Writing integer in to a file
The function used is putw(). This function writes an integer to a file. On success, a nonnegative value is returned. On error, EOF is returned.
General Form:
putw (i, fptr);

Reading integer from a file
The function used is getw(). Reads an integer from the file and assigns it to the program defined numeric variable at the LHS.
General Form:
i = getw(fptr);

Formatted I/O
The formatted I/O functions can handle a group of data in a single call.

Writing formatted data to a file
The function fprintf() is used. This function will write the values stored in the variables into a file pointed by fptr, according to the format specifier specified in format string. On success, the number of characters printed is returned. If an error occurred, -1 is returned.
General Form:
fprintf (fptr, format-string, variable-list);
The fprintf() function takes the format string specified by the format argument and applies each following argument to the format specifiers in the string, in a left to right fashion. Each character in the format string is copied to the stream except for conversion characters which specify a format specifier.

Reading formatted data from the file

The function used is fscanf().This function will read the formatted data from the file pointed by fptr, as specified by the format specifiers in format-string and stores in the variables, whose addresses are given in addresses-list. Reading an input field (designated with a conversion specifier) ends when an incompatible character is met, or the width field is satisfied. On success, the number of input fields converted and stored is returned. If an input failure occurs, EOF is returned.

General Form:

fscanf(fptr, format-string, addresses-list);

The fscanf() function takes input in a manner that is specified by the format argument and stores each input field into the corresponding arguments, in a left to right fashion.

Each input field is specified in the format string with a conversion specifier which specifies how the input is to be stored in the appropriate variable. Other characters in the format string specify characters that must be matched from the input, but are not stored in any of the following arguments. If the input does not match, the function stops scanning and returns. A white space character may match with any white space character such as space, tab, carriage return, new line, vertical tab, or form feed, or the next incompatible character.

Example 14.3: Program using fscanf() and fprintf()

```
main()
{
FILE *fpt;
struct
{
int no;
char name[10];
int age;
}std[10], std1[10];
int i;
clrscr();
fpt = fopen("details.dat" , "w");
printf("\n\n enter the details (no , name , age )\n\n");
for(i=0; i<5 ;i++)
{
scanf("%d %s %d " , &std[i].no , std[i].name , &std[i].age);
fprintf(fpt , "%d %s %d " , std[i].no , std[i].name ,
std[i].age);
}
fclose(fpt);
fpt = fopen("details.dat" , "r");
printf("\n\n reading from file \n\n");
while(!feof(fpt))
{
fscanf(fpt , "%d %s %d " , &std1[i].no , std1[i].name
,&std1[i].age);
```

```
printf("%d %s %d \n" , std1[i].no , std1[i].name ,
std1[i].age);
i++;
}
}
```

Block I/O

Block I/O is used to read or write a specified number of bytes. The data handled by block input/output function will be in 'raw data format' (i.e. bytes of data).

Writing in to a file

The function used is fwrite().Transfers a specified number of bytes beginning at a specified location in memory to a file. Used to write a structure or an array of structures to an output file. The function writes data from the array pointed to by ptr to the given stream. It writes 'n' blocks of size 'size'. The total number of bytes written is (size*n). On success the number of elements written is returned. On error the total number of elements successfully written (which may be zero) is returned.

General Form

fwrite (ptr, size, n, fp);

Where:

ptr pointer to the data block (source)
size size of each block (number of bytes to be written)
n number of blocks to be written
fp file pointer (destination)

Reading from a file

The function used is fread(). Reads data from the given stream into the variable pointed to by ptr. It reads 'n' number of elements of size 'size'. The total number of bytes read is (size*n). On success the number of elements read is returned. On error or end-of-file, the total number of elements successfully read (which may be zero) is returned.

General Form

fread (&str, size, n, fp);

Where:

&str destination memory address
size size of each block (number of bytes to be read)
n number of blocks to be read
fp file pointer (source)

Example 14.4: Program using Block I/O

```
main()
{
FILE *fptr;
struct tag
{
char name[10];
```

```
int age ;
}stud[10] , stud1[10];
int i ;
clrscr();
fptr=fopen("ex.dat" , "w" );
for(i=0 ; i<5 ; i++)
scanf("%s %d ",stud[i].name , &stud[i].age);
fwrite(&stud , sizeof(stud[0]) , 5 , fptr);
fclose(fptr);
fptr = fopen("ex.dat" , "r" );
fread(&stud1 , sizeof(stud1[0]) , 5 , fptr);
printf(" \n\n printing the values ");
for(i=0 ; i<5 ; i++)
printf("\n %s \t %d " , stud1[i].name , stud1[i].age);
}
```

Random File Operations

The functions discussed earlier are to be used for reading and writing data sequentially. In some applications, it may be necessary to access some part of the file directly. This can be achieved by using the functions fseek(), ftell() and rewind().

ftell()

This function takes a file pointer and returns a long int, which corresponds to the current file pointer position. If it is a binary stream, then the value is the number of bytes from the beginning of the file. If it is a text stream, then the value is a value usable by the fseek() function to return the file position to the current position. On success, the current file position is returned. On error, the value -1L is returned and error number (errno) is set.

General Form:

n = ftell(fptr);

fseek()

This function sets the file position to the given offset (specified in long integer format).

General Form:

fseek(fptr, offset, from_where)

The argument *offset* signifies the number of bytes to seek from the given '*from_where'* position.

The argument *from_where* can be:

SEEK_SET Seeks from the beginning of the file. 0

SEEK_CUR Seeks from the current position. 1

SEEK_END Seeks from the end of the file. 2

On a text stream, from_where should be SEEK_SET and offset should be either zero or a value returned from ftell(). The end-of-file indicator is reset. The error indicator is NOT reset. On success, zero is returned. On error, a nonzero value is returned.

Example 14.3

fseek (fp, 0L, 0); Move the file pointer to the beginning.

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

fseek (fp, 10L, 0); Move after 10 bytes from the beginning.
fseek (fp, 10L, 1); Move after 10 bytes from the current
fseek (fp, -10L, 1); Move backward 10 bytes from the current
fseek (fp, -10L, 2); Move backward 10 bytes from the EOF.

rewind()
This function sets the file position to the beginning of the file of the given stream. The error and end-of-file indicators are reset.
General Form:
rewind(fptr);

Example : Write a program to read last 'n' characters of the file

```
#include<stdio.h>
#include<conio.h>
void main()
{
     FILE *fp;
     char ch;
     clrscr();
     fp=fopen("file1.c", "r");
     if(fp==NULL)
        printf("file cannot be opened");
     else
     {
            printf("Enter value of n  to read last 'n' characters");
            scanf("%d",&n);
            fseek(fp,-n,2);
            while((ch=fgetc(fp))!=EOF)
            {
                 printf("%c\t",ch);
            }
      }
    fclose(fp);
    getch();
}
```

PREPROCESSOR DIRECTIVES

One of C's most useful features is its preprocessor. Preprocessor directives are lines included in the code that are not program statements but directives for the preprocessor. These lines are always preceded by a pound sign (#). The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any executable code is generated for the statements.

Preprocessing is a step that takes place before compilation that lets you to:

- Replace preprocessor tokens in the current file with specified replacement tokens.
- Embed files within the current file
- Conditionally compile sections of the current file
- Generate diagnostic messages
- Remove the blank lines in the program, change the line number of the next line of source and change the file name of the current file.
- Remove comments from the source file.

A *token* is a series of characters delimited by white space. The white space allowed on a preprocessor directive may be the space, horizontal tab, vertical tab, form feed, or carriage return.

The preprocessed source program file must be a valid C program.

Preprocessor directives begin with the # token followed by a preprocessor keyword. The # token must appear as a first character. The # is not part of the directive name and can be separated from the name with white spaces.

A preprocessor directive ends at the new-line character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the new-line character as a continuation marker. The preprocessor deletes the \ (and the following new-line character) and splices the physical source lines into continuous logical lines. No semicolon (;) is expected at the end of a preprocessor directive.

Except for some #pragma directives, preprocessor directives can appear anywhere in a program.

Preprocessor Directives

Name	**Action**
#	Null directive specifying that no action be performed.
#define	Defines a preprocessor macro.
#elif	Conditionally includes source text if the previous #if, #ifdef, #ifndef, or #elif test fails.
#else	Conditionally includes source text if the previous #if, #ifdef, #ifndef, or #elif test fails.
#endif	Ends conditional text.
#error	Defines text for a compile-time error message.
#if	Conditionally includes or suppresses portions of source code, depending

	on the result of a constant expression.
#ifdef	Conditionally includes source text if a macro name is defined.
#ifndef	Conditionally includes source text if a macro name is not defined.
#include	Inserts text from another source file.
#line	Supplies a line number for compiler messages.
#pragma	Specifies implementation-defined instructions to the compiler.
#undef	Removes a preprocessor macro definition.

Preprocessing Operations:

Pre processing operations are mainly classifieds into
1) File Inclusion,
2) Macro substitution and
3) Conditional Compilation.

Preprocessing will be done before compilation, compilation process operates on the preprocessor output, which is then syntactically and semantically analyzed and translated, and then linked as necessary with other programs and libraries.

File Inclusion
The #include directive allows external files to be added in to our source file, and then processed by the compiler.
General Form:
#include <header file> OR #include "header file"
The only difference between both expressions is the places (directories) where the compiler is going to look for the included file. If the file name is enclosed between angle-brackets <>, the file is searched in the directories where the compiler is configured to look for the standard header files. Therefore, standard header files are usually included in angle-brackets, while other user specificed header files are included using quotes.
In the second case where the file name is specified between double-quotes, the file is searched first in the current working directory. In case that it is not there, the compiler searches the file in the default directories where it is configured to look for the standard header files.

Example
#include <stdio.h>
#include "stdio.h"

Preprocessor Macros:
#define preprocessor directive is used to define a macro that assigns a value to an identifier. The preprocessor replaces subsequent occurrences of that identifier with its assigned value until the identifier is undefined with the #undef preprocessor directive, or until the end of the program source is reached, whichever comes first.

There are two basic types of macro definitions that you can use to assign a value to an identifer:

Object-like Macros (Symbolic constants)	Replaces a single identifier with a specified token or constant value.
Function-like Macros	Associates a user-defined function and argument list to an identifier. When the preprocessor encounters that identifier in the program source, the defined function is inserted in place of the identifier along with any corresponding arguments.

Symbolic Constants
The preprocessing directives #define and #undef allow the definition of identifiers which hold a certain value. These identifiers can simply be constants or a macro function.

#define
General Form:
#define symbolic_varaiable name_value
Example
#define SIZE 10
#define NAME "xyz" /* good practice is to use upper case letters */

#undef:
General Form:
#undef variablename
Example
#undef SIZE

Macros:
General Form:
#define macroname(argument list) macrodefn
Example:
```
#define sqarea(a) ((a)*(a))
main()
{
areaofsquare=sqarea(a);
…..
}
```
Arguments in the macro definition are enclosed with parenthesis to avoid miscalculation. Continuation character for macro definition is \. There is no need for semicolon after the macro definition.

Example

```
#define sqarea(a) ((a)*(a))
#define sqa(b) b*b
#define add(a,b) ((a)+(b));
main()
{
      areaofsquare=sqarea(a); /* areaofsquare = (a) * (a); */
      areaofsquare=sqarea(3); /* areaofsquare = (3) *(3); */
      areaofsquare=sqarea(3+4); /* areaofsquare=(3+4)*(3+4); */
      areaofsquare=sqa(3+4); /* areaofsquare=3+4*3+4; (1) */
      addition=add(2,3); /* addition=(2)+(3);; (2) */
}
```

(1) miscalculation because of no parentheses
(2) two semicolons in macro expansion.

Conditional Compilation Directives:

A preprocessor conditional compilation directive causes the preprocessor to conditionally suppress the compilation of portions of source code. These directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be bypassed during preprocessing. The directives are:

#if
#ifdef
#ifndef
#else
#elif
#endif

The directives #ifdef and #ifndef allow conditional compiling of certain lines of code based on whether or not an identifier has been defined. For each #if, #ifdef, and #ifndef directive, there are zero or more #elif directives, zero or one #else directive, and one matching #endif directive. All the matching directives are considered to be at the same nesting level.

General Form:

```
  #if constant_expression
  #else
  #endif
```

OR

```
  #if constant_expression
  #elif constant_expression
  #endif
```

The compiler only compiles the code after the #if expression if the constant_expression evaluates to a non-zero value (true). If the value is 0 (false), then the compiler skips the lines until the next #else, #elif, or #endif. If there is a matching #else, and the constant_expression evaluated to 0 (false), then the lines between the #else and the #endif are compiled. If there is a matching #elif, and the preceding #if evaluated to false, then the constant_expression after that is evaluated and the code between the #elif and the #endif is compiled only if this expression

evaluates to a nonzero value (true).

Example
Check whether a variable is defined. If so, change the value of that variable to 1 after undefining it.

```
#if define(NUMBER)
#undef NUMBER
#define NUMBER 1
#endif
```

 # and ## operators
causes the argument to be converted as a string enclosed within quotes.
Example

```
#define name(x) #x
main()
{
…..
printf(name(xyz)); /* printf(“xyz”); */
….
}
```

concatenation operator
Example

```
#define name(x,y) x##y
main()
{
…..
printf(name(ssn,somca)); /* printf(“ssnsomca”); */
….
}
```

COMMAND LINE ARGUMENTS

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.

Example
Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly –

```
#include <stdio.h>
int main( int argc, char *argv[] )  {
  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
  }
}
```

Output
When the above code is compiled and executed with single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''.

Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes −

Example

```
#include <stdio.h>
int main( int argc, char *argv[] )  {
  printf("Program name %s\n", argv[0]);
  if( argc == 2 ) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if( argc > 2 ) {
    printf("Too many arguments supplied.\n");
  }
  else {
    printf("One argument expected.\n");
  }
}
```

Output

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$./a.out "testing1 testing2"

Programm name ./a.out
The argument supplied is testing1 testing2
```

SUMMARY

.

- In C, file manipulations may be done in two ways:
  - Low-level I/O using system calls
  - High-level I/O using functions from standard I/O library
- The files accessed through the library functions are called *Stream Oriented files* and the files accessed with system calls are known as *System Oriented files*.
- Streams facilitate a way to create a level of abstraction between the program and an input/output device. There are two types of streams: *text* and *binary*.
- Text streams are composed of a set of lines. Each line has zero or more characters and is terminated by a new line character.
- Binary streams are composed of only 0's and 1's. It is simply a long series of 0's and 1's.
- When a program begins, there are three available streams:

- Standard input (stdin) is the stream where a program gets its input data
- Standard output (stdout) is the stream where a program writes its output data.
- Standard error (stderr) is another output stream typically used by programs to output error messages.

- In C, 'FILE' is a structure that holds the description of a file and is defined in stdio.h.
- While file handling, the code must define a local 'pointer' of type FILE (called file pointer), then 'open' the file and associate it with the file pointer via fopen().
- fopen() returns the file pointer position for successful open and returns NULL, if the file does not open or the file does not exist.
- fclose() returns zero for successful close and returns EOF (end of file) when error is encountered in closing a file.
- Preprocessor directives are lines included in the code that are not program statements but directives for the preprocessor and is executed before the actual compilation of code begins.
- Except for some #pragma directives, preprocessor directives can appear anywhere in a program.
- The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments passed, and **argv[]** is a pointer array which points to each argument passed to the program.
- It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument.

Questions for Exercises

1. Write a program to read a file and display contents with its line numbers.

2. Write a program to add the contents of one file at the end of another.

3. Write a program that merges lines alternately from two files and writes the results to new file. If one file has less number of lines than the other, the remaining lines from the larger file should be simply copied into the target file.

4. In the file 'CUSTOMER.DAT' there are 100 records with the following structure:

```
struct customer
{
int accno ;
char name[30] ;
float balance ;
} ;
```

In another file 'TRANSACTIONS.DAT' there are several records with the following structure:

```
struct trans
{
int accno ,
char trans_type ;
float amount ;
} ;
```

The parameter **trans_type** contains D/W indicating deposit or withdrawal of amount. Write a program to update 'CUSTOMER.DAT' file, i.e. if the **trans_type** is 'D' then update the **balance** of 'CUSTOMER.DAT' by adding **amount** to balance for the corresponding **accno**. Similarly, if **trans_type** is 'W' then subtract the **amount** from **balance**. However, while subtracting the amount make sure that the amount should not get overdrawn, i.e. at least 100 Rs. Should remain in the account.

5. What statement performs formatted writing to text files?
6. Explain Random File Operations.
7. What do you mean by preprocessor directive? Explain in detail.
8. Write a program to accept two file names as command line argument, and copy the first file to the second.

Books for reference:

- ***The C programming Language,*** By Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall , 2004
- ***Programming in C,*** By Stephen G. Kochan, Fourth Edition, 2001
- ***Let Us C,*** By Yashavant P. Kanetkar, Fifth Edition, 2012
- ***Programming with C***, Second Edition, Byron Gottfried, Tata McGraw Hill, 2003.
- ***C. The Complete Reference***, Fourth Edition, Herbert Schildt, Tata McGraw Hill, 2002.
- ***The C Primer***, Leslie Hancock, Morris Krieger, Mc Gravy Hill, 2013.