UNIT – IV

SYLLABUS

Chapter 1: Preprocessor Directives: Introduction, The #define Directive, Undefining a Macro, TokenPasting and Stringizing Operators, The #include Directive, Conditional Compilation, The #ifndef Directive.

Chapter 2: Structure and Union

Chapter 3: Files

Programming Exercises for Unit - IV:

1. Operations on complex numbers
2. matrix operations with the matrix and the size of the matrix as a structure
3. sorting a list of student records on register number using array of pointers
4. To read an input file of marks and generate a result file
5. Sorting a list of names using command line arguments.

CHAPTER 1: PREPROCESSOR DIRECTIVES

- The C program execution needs following steps
    1. Creation of program
    2. Execution of **preprocessor** program
    3. Compilation and linking of a program
    4. Execution the program
- The preprocessor is program that process the source code before it passes to the compiler.
- C language offers the preprocessor directives and it initialized at the beginning of the program before main( ) by using symbol **#(hash).**

THE #define DIRECTIVE

syntax:

#define identifier substitute

(OR)

#define identifier (argument1 …..argument N) substitute

example: #define pi 3.14

This statement defines "pi" as macro templates and 3.14 as macro substitute.

PROGRAM: Use the identifier for 3.14 as pi and write a program to find the area of circle using it.

```
#define  pi  3.14
void main()
{
        float r,area;
        printf("\n Enter radius of the circle in cm:");
        scanf("%f",&r);
        area=pi*r*r;
        printf("\n Area of the circle=%.2f cm^2",area);
}
```

OUTPUT:

```
        Enter radius of the circle in cm :7
        Area of the circle=153.86 cm^2
```

UNDEFINING A MACRO

A macro defined with **#define** directives can be undefined with **#undef** directive.

syntax:

#undef macro_templatesubstitute

*It is useful, when we do not want to allow the use of macros in any portion of program.

PROGRAM: write a program to undefine a macro.

```
#define  wait   getche()
void main()
{
        int k;
        #undef   wait()   getche
        for(k=1;k<=5;k++)
                printf("%d\t",k);
```

```
        wait( );
}
```

TOKEN PASTING AND STRINGIZING OPERATORS:

String zing operation: In this operation macro argument is converted to a string. The sign '#' carries the operation.It is placed before the argument.

PROGRAM:Write a program to carry out stringizing operation.

```
#define  say(m)  printf(#m)
void main()
{
        say(Hello);
        getche();
}
```

OUTPUT:

```
        Hello
```

THE #include DIRECTIVE:

- The #include directive is used to load the file to our current program.
- There are two ways to include #include directive. The syntax is given below:

  (a)#include "file name"

  (b)#include <file name>

Example:

```
#include "stdio.h"
#include<conio.h>
```

(a)The file name is included in the double quotations marks, which indicates that the search for the file is made in the current directory and in the standard directories.

(b)When the file name is included within the angle brackets,the search for file is made only in the standard directories.

PROGRAM: Write a program to call the function defined in 'udf.c' file.

```
#include "und.c"
void main()
{
clrscr();
display();
}
```

OUTPUT:

```
        Function called
```

CONTENTS of udf.c file:

```
int display();
display()
{
        printf("\n Function called");
        return 0;
}
```

CONDITIONAL COMPILATION:

- In preprocess, it provide conditional compilation directives in the programs.
- The most frequently used conditional compilation directives are **#ifdef, #else, #endif**.
- These directives are allow the programmer to include the portion of the code based on the conditions.
- The compiler compiles selected portion of the source code based on conditions.

THE #ifdef DIRECTIVE:

If identifier is defined then #if block is compiled and executed otherwise #else block is executed.

SYNTAX:

```
#ifndef  identifier
{
        Statement 1;
        Statement 2;
}
#else
{
        Statement 3;
        Statement 4;
}
#endif
```

PROGRAM: Write a program to check conditional compilation whether the identifier is defined or not.

```
# define  LINE  1
void main()
{
        #ifndef  LINE
                printf("\n this is line number one.");
        #else
                printf("\n this is line number two.");
        #endif
}
```

OUTPUT:

```
this is line number one.
```

THE #ifndef DIRECTIVE:

- The #infndef works exactly opposite to that of #ifdef.
- If identifier is defined then #else block is compiled and executed and the compiler ignores #if block even if errors are intentionally made.
- If identifier is not defined then #if block is executed.

SYNTAX:

```
#ifndef<identifier>
{
    Statement 1;
    Statement 2;
}
#else
{
    Statement 3;
    Statement 4;
}
#endif
```

PROGRAM: Write a program to check conditional compilation whether the identifier is defined or not.

```
# define  LINE  1
void main()
{
    #ifndef  LINE
        printf("\n this is line number one.");
    #else
        printf("\n this is line number two.");
    #endif
}
```

OUTPUT:

```
this is line number two.
```

CHAPTER 2: STRUCTURES AND UNIONS

STRUCTURE:

- Structure is a collection of one or more variables of different data types, grouped together under a single name.
- It is a user defined data types because the user can decide the data types to be added in the body of structure.
- The total size of structure is equal to the number of bytes required for the all members.

DECLARATION OF STRUCTURE: Structure declaration starts with **struct** keyword

Syntax:

```
struct   structure_type
{
        type   variable1;
        type   variable2;
        ……————————————————————————
        ……————————————————————————
};
```

Example:

```
struct   student
{
        char name[20];
        int rollno;
        float per;
};
```

- Struct_type is known as ***tag*** or s***tructure nam.***
- Structure variables are declared as follows

  Syntax:

  ```
  struct  structure_type    var1, var2, var3;
  ```

  Example

  ```
  struct  student  suresh, ramesh;
  ```

Note: the memory allocation takes place for structure is only when variables are declared.

INITIALIZATION OF STRUCTURE:

- Structure can be initialised in two ways.
- Initialize the all values at a time to the structure members.

  Example: struct student suresh = { "suresh", 25,78.9};
- Another way of initializing isassighn the values individually to the structure members.
- Structure members can be accessed by using dot (**.**) operator

  Syntax:

  ```
  structure_valiable . member = value;
  ```

  Example:

  ```
  suresh.name = "suresh";
  suresh.name = 25;
  suresh.per = 78.9;
  ```

Example program for initializing and accessing of structure .

```
#include<stdio.h>
struct student
{
        char name[30];
        int rollno;
        float per;
};
void main( )
{
        struct student  suresh = { "suresh", 25, 95.4};
        printf("Student Name : %s\n", suresh.name);
        printf("Student  roll no : %d\n", suresh.rollno);
        printf("Student  percentage : %f\n", suresh.per);
}
```

Output:

```
Student Name : suresh
Student  roll no : 25
Student  percentage :95.4
```

POINTER TO STRUCTURE:

- We know that the pointer is a memory variable that stores address of another data variable. The variable may be int, char etc.
- We can also define the pointer to structure. The starting address of member variable can be accessed. Such a pointer is called structure pointer.
- We can access the structure pointer using arrow (->) operator.

Syntax:

```
struct  struct_tag
{
        type variable1;
        type variable2;
        -----------------
        -----------------
};
struct   struct_tag  *pointer_variable;
```

Example:

```
struct   student
{
        char name[20];
        int rollno;
        float per;
};
struct  student  *eee;
```

Example program for initializing and accessing of structure pointer.

```
#include<stdio.h>
struct   student
{
        char name[20];
        int rollno;
        float per;
};
void main( )
{
        struct student  suresh = { "suresh", 25, 95.4};
        struct  student  *ptr;
         ptr  = &suresh;
        printf("Student Name : %s\n", ptr -> name);
        printf("Student rollno : %d\n", ptr  -> pages);
        printf("Student percentage  : %f\n", ptr  -> price);
}
```

output:

```
        Student Name : suresh
        Student  roll no : 25
        Student  percentage :95.4
```

<u>Write a c program to display the total sizes of structure members</u>

```
#include<stdio.h>
  #include<stdio.h>
  struct student
  {
        char name[30];
        int rollno;
        float per;
  };
  void main( )
  {
        struct  student  suresh ;
        printf("Student Name : %d bytes \n", sizeof(suresh.name));
        printf("Student  roll no : %d bytes \n", sizeof(suresh.rollno));
        printf("Student  percentage : %d bytes \n", sizeof(suresh.per));
        printf("Student  total size : %d bytes\n", sizeof(suresh.per));
  }
```

Output:

```
        Student Name : 20 bytes
        Student  roll no : 2 bytes
        Student  percentage : 4 bytes
        Student  total size : 26 bytes
```

Features of structures:

- In arrays, it is not possible to copy all elements at a time but in structure it's possible by using assignment (=) operator because the structure elements are stored in successive memory location.
- Comparison between arrays and structure

S.No	points of comparison	Arrays	Structure
1	Collection of data	Same data type	Different data type
2	Keyword	It is not a keyword	It is a keyword
3	Declaration and definition of data types	Only declaration	Declaration and definition
4	Bit field	Does not have a bit fields	May contain bit fields

- Nesting of structure is possible, i.e. placing one structure into another structure. It used for handle comple data type.
- It is also possible to create structure pointer like pointer to int and pointer to float.
- "->" operator is used to access the structure pointer variables.

ARRAY OF STRUCTURES:

- We can declare the structure as an array. In this all the structure array elements having the same members.
- We can access the array of structure members using period (.) sign.

Syntax:

```
struct  struct_tag
{
        type variable1;
        type variable2;
        -----------------
        -----------------
};
struct   struct_tag   array_of_stucture[size];
```

Example:

```
struct   student
{
          char name[20];
          int rollno;
          float per;
};
struct  student  eee[50];
```

Example: write a c program to student details and display them

```
#include<stdio.h>
struct   student
{
      char name[20];
      int rollno;
      float per;
};
```

```
void main()
{
        struct  student  eee[50];
        int n, i;
        printf("\nEnter no.of students  :");
        scanf("%d", &n);
        printf("Enter %d students information\n", n);
        for(i=1; i<=n; i++)
        {
                printf("Enter student %d details\n",i);
                scanf("%s %d %f", &eee[i].name, &eee[i].rollno, &eee[i].per);
        }
        for(i=1; i<=n; i++)
        {
                printf("\nStudent %d details\n",i);
                printf("%s\t %d\t %f\n", eee[i].name, eee[i].rollno, eee[i].per);
        }
}
```

Output:

```
Enter no.of students  :  2
Enter student 1 details
Ramesh  31  87.4
Enter student 2 details
Suresh  23  75.4
Student 1 details
Ramesh  31  87.4
Student 2 details
Suresh  23  75.4
```

STRUCTURE AND FUNCTIONS:

- Like as standard data type variables, structure variables can be passed to the function by value or address.
- Whenever a structure element requires to pass any other function, then the structure can be declared outside of the main i.e., global.

Syntax

```
struct   student
{
        char name[20];
        int rollno;
        float per;
};
   void main()
   {
           struct  student  s1;
           ----------------------
```

```
                    ----------------------
                    show(s1)
          }
          void show( struct  student  s)
          {
                    --------------------
                    --------------------
          }
```

W A P to pass structure variable and display the content(call by value mechanish).

```
#include<stdio.h>
struct   student
{
          char name[20];
          int rollno;
          float per;
};
void show(struct  student  s);
void main( )
{
          struct student  suresh = { "suresh", 25, 95.4};
          show(suresh);
}
void show(struct student s)
{
          printf("Student Name : %s\n", suresh.name);
          printf("Student rollno : %d\n", suresh. pages);
          printf("Student percentage  : %f\n", suresh. price);
}
output:
          Student Name : suresh
          Student  roll no : 25
          Student  percentage :95.4
```

W A P to pass structure variable and display the content(call by reference mechanish).

```
#include<stdio.h>
struct   student
{
          char name[20];
          int rollno;
          float per;
};
void main( )
{
          struct student  suresh = { "suresh", 25, 95.4};
          show(&suresh);
  }
```

```
void show(struct   student    *ptr)
{
        printf("Student Name : %s\n", ptr -> name);
        printf("Student rollno : %d\n", ptr  -> pages);
        printf("Student percentage  : %f\n", ptr  -> price);
}
```

Output:

```
Student Name : suresh
Student  roll no : 25
Student  percentage :95.4
```

UNION

- Union is a variable, which is similar to the structure.
- It contains the number of member of members like structure but it holds only one object at a time.
- In the structure each member has its own memory location where as the members of union have the same memory location.
- The total size of union is equal to the number of bytes required for the largest members.

  For example union contains char, integer, and float then the union size is 4 bytes.

DECLARATION OF UNION: union declaration starts with ***union*** keyword.

Syntax:

```
union   union_type
{
        type   variable1;
        type   variable2;
        ……................
        ……................
};
```

union type is known as ***tag*** or union ***name***.

union variables are declared as follows

```
union      union_type    var1, var2, var3;
```

Here var1, var2, and var3 are the variable of union union_type.

Example:

```
union   student
{
        char name[20];
        int rollno;
        float per;
};
union  student   ravi;
```

INITIALIZATION OF UNION:

- union can be initialised in two ways.
- Initialize the all values at a time to the union members.

  Example: union student suresh = { “suresh”, 25,78.9};
- Another way of initializing isassighn the values individually to the structure members.

- Structure members can be accessed by using dot (**.**) operator

Syntax:

union _valiable **.** member = value;

Example:

suresh.name = “suresh”;

suresh.name = 25;

suresh.per = 78.9;

Example program for initializing and accessing of union.

```
#include<stdio.h>
union student
{
        char name[30];
        int rollno;
        float per;
};
void main( )
{
        union student  suresh = { "suresh", 25, 95.4};
        printf("Student Name : %s\n", suresh.name);
        printf("Student  roll no : %d\n", suresh.rollno);
        printf("Student  percentage : %f\n", suresh.per);
}
```

Output:

Student Name : suresh

Student roll no : 25

Student percentage :95.4

Example program to find the size of union and the number of bytes reserved for it

```
void main( )
{
        union result1
        {
                int marks;
                char grade;
        };
        struct  result2
        {
                int marks;
                char grade;
        };
        union result1 rs1;
        struct result2 rs2;
        printf(“\nTotal size of union  :%d”, sizeof(rs1));
        printf(“\nTotal size of structure  :%d”, sizeof(rs2));
}
```

Output:-
Total size of union :2
Total size of structure :3

NESTED STRUCTURES:

A structure is placed into another structure is called nested structure.

DECLARATION:

Syntax:

```
struct   struct_type1
{
        type   variable1;
        type   variable2;
        ……————————————————————————
        ……————————————————————————
};

struct   struct_type2
{
        type   variable1;
        type   variable2;
        ……————————————————————————
        ……————————————————————————
        struct  struct_typ1 variable;
};
```

Example:

```
struct  student
{
        char sname[30];
        int rollno;
        float per;
};
struct  college
{
        char cname[30];
        long int contact;
        struct  student  ravi;
};
struct   college   bec;
```

- Here, student ravi is a member of the structure collage.
- When you create a memory for structure college then it also create memory for student structure.

INITIALIZATION OF NESTED STRUCTURE:

- union can be initialised in two ways.
- Initialize the all values at a time to the union members.

  Example: struct college bec = {“bec”, 1234,{ “suresh”, 25,78.9}};
- Another way of initializing isassighn the values individually to the structure members.
- Structure members can be accessed by using dot (**.**) operator

  Syntax:

  struct _valiable **.** member = value;

  Example:

  bec.cname= bec;

  bec.contact=1234;

```
bec.ravi.sname = “suresh”;
bec.ravi.name = 25;
bec.ravi.per = 78.9;
```

Example program for initializing and accessing of nested structure.

```
#include<stdio.h>
struct student
{
        char name[30];
        int rollno;
        float per;
};
struct  college
{
        char cname[30];
        long int contact;
        struct  student  ravi;
};
void main(  )
{
        struct   college   bec={“bec”,1234,{"suresh", 25, 95.4}};
        printf("College Name : %s\n", bec.cname);
        printf("College Contact : %ld\n", bec.contact);
        printf("Student Name : %s\n", bec.ravi.name);
        printf("Student  roll no : %d\n", bec.ravi.rollno);
        printf("Student  percentage : %f\n", bec.ravi.per);
}
```

Output:

```
College Name : bec
College Contact : 1234
Student Name : suresh
Student  roll no : 25
Student  percentage :95.4
```

typedef:

- The statement ***typedef*** is to be used for defining the new data type or duplicate name for the data types.
- **Syntax:**

  typedef <datatype> data_name;

  here data_name is the user-defined name for the data type.
- **Example:**

```
typedef  int  hours;
typedef   struct   student   s;
```

here, hours is the another name for the int and s another name for struct student.

Example program:

```
void main( )
{
        typedef  int hours;
        hours hrs;
        printf(“Enter hours :”);
```

```
        scanf("%d", &hrs);
        printf(" \nhours  :%d", hrs);
        printf("\Minutes  :%d", hrs * 60);
        printf("\nSeconds : %d", hrs * 60 * 60);
}
Output:-
        Enter hours :2
        hours  :2
        Minutes  : 120
        Seconds :720
```

BIT FIELD:

- Bit field provides extract amount of bits required for storage of value.
- For example
- ➢ the values 0 or 1 need only single bit;
- ➢ the values 2 and 3 need 2 bits but it takes more bits to store a values.
- The number of bits required for a variable is specified by non-negative integer followed by colon.
- Syntax:

  <data type> <variable_name> : <number_of_bits> ;
- Example:

  int x : 3;

  Here x integer variable and it occupy only 3 bits memory.

ENUMERATED DATA TYPE:

- The enumerated data type is used to create his/her own data type or user-defined data type and define what values the variables of these data types can hold.
- The enumerated data type is created by using ***enum*** keyword.
- Example:

  enum month { jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
- here, ***month*** is user defined data type and ***jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec are*** identifiers.
- The identifier value starts from 0 and if we want to change starting value then it's also posible.

CHAPTER 3: FILES

What is a file?

"A file is a collection of data that is stored permanently on disk".

(or)

"A file is a stream of bytes of arbitrary length, which is used to hold data".

Why do we need files?

We need files in order

1. To store data permanently. i.e., the content in the file can never be lost even after program termination. It can be recalled as and when it is needed.
2. To maintain large amount of data. i.e., it is easy to handle voluminous data during execution.
3. To make disk user-friendly. i.e., the complex interface of disk can be converted to user-friendly interface by using file so that user can interact with it.
4. To have its storage capacity more. i.e., a file can have its content till there is no space in storage device.

Streams:

Stream is a file by which reading and writing is done. The file object contains all the information about stream like current position, pointer to any buffer, error and end of the file(EOF).

Types of files:

1. **Sequential file:** In this type of file, data are kept sequentially. If we want to read the last record of the file it is expected to read all the records before it. It takes more time for accessing the records.
2. **Random access files:** In this type data can be read and modified randomly. If the user desires to read the last record of the file, directly the same records can be read. It takes less time.

BASIC OPERATIONS ON FILES

File I/O is always done in a program in the following sequence:

1. Open the file.
2. Write to the file and Read from the file.
3. Close the file.

1. Opening a file:

Before performing any file I/O, the file must be opened. While opening the file, the following are to be specified:

- ✓ The name of the file and
- ✓ The mode in which it should be opened (i.e., for reading, writing, …etc).

The function **fopen()** is used to open a file. It accepts two strings as arguments with the following form:

Syntax: FILE *fp;

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

Example:

```
fp = fopen("eee.txt" , "r");
```

- Each file that we open has its own FILE structure. The information that is there in the file may be its current size, and its location in memory.
- fp is pointer variable that contains the address of the structure FILE that has been defined in the header file **stdio.h**. It is necessary to write **FILE** in uppercase.

File Opening Mode	Meaning
r (read-only)	Open an existing file for reading only.
w(write-only)	Open a new file for writing only. If a file with specified *filename* currently exists, it will be destroyed and a new file with the same name will be created in its place.
a (append-only)	Open an existing file for appending (i.e., for adding new information at the end of file). A new file will be created if the file with the specified *filename* does not exist.
r+ (read & write)	Open an existing file for update (i.e., for both reading and writing).
w+ (write & read)	Open a new file for both writing and reading. If a file with the specified *filename* currently exists, it will be destroyed and a new file will be created in its place.
a+(append & read)	Open an existing file for both appending and reading. A new file will be created if the file with specified *filename* does not exist.

<u>Return value:</u>

- ✓ On success, fopen() returns a pointer to the predefined structure **FILE**, whose definition is available in **stdio.h**.
- ✗ On failure, it returns NULL.

Example:

```
FILE *fp;   //statement-1
fp=fopen("sample.txt","w"); //statement-2
if(fp==NULL) //statement-3
{
        printf("\n Unable to open a file");
        exit(0);
}
```

2. Writing to a file

- To write to a file, the functions **fprintf(), fputs()** and **fputc()** are used write the content to file.
- All these functions accept a file pointer as one of the parameters along with other parameters required by the standard output functions.
- Syntax for **fprintf()**:

  fprintf(fp, "control string ", var1, var2,…var n);

Reading from a file

- To read from a file, the functions **fscanf(), fgets()** and **fgetc()** are used read content from the file.
- All these functions accept a FILE pointer as one of the parameters along with other parameters required by the standard input functions.
- Syntax for **fscanf()**:

  fscanf(fp, "control string", var_list);

3. Closing a file

During a write to a file, the data written to it is not put on the disk immediately. It is stored in a buffer (a temporary storage area in primary memory). When the buffer is full or the program gets ended, all the content in buffer is actually written back to the disk. The process of emptying the buffer by writing its content back to disk is called ***flushing the buffer.***

Closing a file flushes the buffer and releases the space taken by FILE structure which is returned by fopen(). If we want to close a file, pass its file pointer to the function **fclose()** that has the following form:

```
int fclose(pointer_to_FILE);
```

- fclose() function takes one argument: the file pointer to which we want to close.
- On success, it returns 0. on failure, it returns EOF.

Ex:
```
FILE *fp;
fp=fopen("sample.txt","w");
…….
fclose(fp); //closes the file sample.txt
```

Write a program to create a file, write some text to it and print that text on monitor.

```
#include<stdio.h>
main()
{
    char filename[15], ch =' ';
    FILE *fp;
    printf("\n Enter the file name to be created:");
    scanf("%s",filename);
    fp =f open(filename,"w");  //first, open file in write mode
    if(fp==NULL)
    {
```

```
            printf("\n unable to open");
            exit(0);
    }
    printf("\n Enter some text to file ('.' to end):");
    while(ch!=EOF)   // check that the entered character is not ctrl+d
    {
            ch=getchar(); //again, read a character from key board
            fputc(ch,fp);  //write that character to file
    }
    fclose(fp);       // close the file
    fp=fopen(filename , "r");  //open the same file in read mode
    if(fp==NULL)
    {
            printf("\n unable to open");
            exit(0);
    }
    printf("\n The content of  file %s is:",filename);
    while(ch!=EOF) // check the character is not EOF
    {
            ch=fgetc(fp);
            printf("%c",ch);  //print that character on monitor
    }
    fclose(fp); //close the file
}
```

Output:

Enter the file name to be created: master.txt

Enter some text to file ('.' to end):

A

B

C

D

The content of file master.txt is:

A

B

C

D

What is the purpose feof()?

The feof() is a pre-defined function that indicator for an end-of-file.

It has the following form:

```
int feof(pointer_to_FILE);
```

From this prototype, it is clear that:

- feof() function takes one argument: the file pointer to which we want to search for EOF character.
- On success, it returns non-zero if an end-of-file character is detected. If end-of-file is not reached, then it returns 0.

Write a program to count number of characters, alphabets, digits, white spaces and special symbols in an existing file.

```
#include<stdio.h>
main()
{
    char filename[15],ch;
    FILE *fp;
    int ccount,acount,dcount,scount,sscount;
    printf("\n Enter an existing file name to be read:");
    scanf("%s",filename);
    fp=fopen(filename,"r"); //open an existing file in read mode
    if(fp==NULL)
    {
        printf("\n Unable to open file");
        exit(0);
    }
    ccount=acount=dcount=scount=sscount=0; //initially, all counts are 0
    ch=fgetc(fp); //read a character from file
    while(ch!=EOF)
    {
        ccount++; //increment character count by 1
        if(toupper(ch)>='A' &&toupper(ch)<='Z')  //if it is alphabet
            acount++; //increment alphabet count by 1
        else if(ch>='0' && ch<='9') //if it is digit
            dcount++; //increment alphabet count by 1
        else if(ch==' ' || ch=='\t' || ch=='\n') //if it is white space
            scount++; //increment space count by 1
```

```
            else
                    sscount++; //increment special symbol count by 1
            ch=fgetc(fp); //again, read a character from file
    }
    fclose(fp); //close the file
    printf("\n No.of characters=%d",ccount);
    printf("\n No.of Alphabets=%d",acount);
    printf("\n No.of digits=%d",dcount);
    printf("\n No.of spaces=%d",scount);
    printf("\n No.of special symbols=%d",ccount);
}
```

Output:

```
Enter an existing filename to be read: master.txt
No.of characters=160
No.of Alphabets=122
No.of digits=2
No.of spaces=30
No.of special symbols=6
```

Write a program to write student records to a sequential file and access them

```
#include<stdio.h>
struct student
{
    int rno, rank;
    char name[15];
    float fees;
    struct date_of_birth
    {
            int day,month,year;
    }dob;
}s1;
main()
{
    FILE *fp;
    char ch;
    fp=fopen("sturecords","w"); //create a new file
    if(fp==NULL)
```

```
    {
            printf("\n Unable to open file");
            exit(0);
    }
do
{
            printf("\nEnter student details (rno,name,rank,fees and date of birth):");
    scanf("%d%s%d%f%d%d%d",&s1.rno,s1.name,&s1.rank,&s1.fees,&s1.dob.day,&s1.dob.month,&s1.dob.year);  //read the student record from keyboard
            fprintf(fp,"%d\t%s\t%d\t%f\t%d/%d/%d",s1.rno,s1.name,s1.rank,s1.fees,s1.dob.day,s1.dob.month,s1.dob.year);  //write the student record to file
            printf("\n Do u want to add one more record (y/n):");
            scanf("\n%c",&ch);
}while(toupper(ch)!='N');
fclose(fp);


    fp=fopen("sturecords","r"); //open the file in read mode
    if(fp==NULL)
    {
            printf("\n Unable to open file");
            exit(0);
    }
    printf("\n content of sturec file:");
    while(fscanf(fp,"%d\t%s\t%d\t%f\t%d/%d/%d",&s1.rno,&s1.name,&s1.rank,&s1.fees,&s1.dob.day,&s1.dob.month,&s1.dob.year)!=EOF) //read a record from file
{
printf("\n%d\t%s\t%d\t%f\t%d/%d/%d",s1.rno,s1.name,s1.rank,s1.fees,s1.dob.day,s1.dob.month,s1.dob.year); //print the record onto monitor
}
fclose(fp);
}
```

Output:

Enter student details (rno, name ,rank ,fees and date of birth):1

Mercylin

193

27500.00

29

2

1990

Do you want to add one more record (y/n):n

Content of sturec file:

1 Mercylin 193 27500.00 29/2/1990

FILE I/O:

We have different File I/O function for handling the file.

1) **fputc( ):** fputc( ) is used to write a single character to a file.
   - Open the existing file in “w”,“w+”,”a” or “a+” mode using fopen() and handle the file pointer that is returned.
   - Read the character from keyboard using getchar() or scanf().
   - Input that character as an argument to the function fputc() along with the file pointer as follows:

```
int fputc(character, FILE_pointer);
```

   - fputc() takes two arguments: a character that is to be written to file and pointer to the structure FILE.
   - On success, it returns an unsigned integer. Otherwise, it returns EOF.

Example:

```
FILE *fp;
char ch;
fp=fopen(“sample.txt”,”w”); //step-1:open file in writing mode
printf(“\n Enter the text to the file (ctrl+d to end):”);
ch=getchar();                    //step-2: reads a character from keyboard
while(ch!=EOF)        //write character-by-character till the end-of-file is reached
{
        fputc(ch,fp); //step-3
        ch=getchar();
}
```

2) **fputs ( ):** fputs( ) is used to write a string to the file.
   - Open the existing file in “w”,“w+”,”a” or “a+” mode using fopen() and handle the file pointer that is returned.
   - Read the string from keyboard using gets() or scanf().
   - Input that string as an argument to the function fgets() along with the file pointer as follows:

```
int fputs(string, pointer_to_FILE);
```

   - fputs() takes two arguments: first argument is a string that is to be written to the file and second argument is file pointer.
   - fputs() writes the string to the specified file. On success, it returns an unsigned integer. On failure, fputs() returns EOF.

Example:

```
FILE *fp;
char str[25];
fp=fopen("normal.txt","w");                    //step-1 : Open file in write mode
printf("\n Enter any string to write to the file:");
scanf("%s",str);                               //step-2: read a string from keyboard
fputs(str,fp);                                 //step-3 : write that string to file
```

3) **fprintf( ):** fprintf( ) is used to to write formatted data from a file.
   - Open the existing file in "w","w+","a" or "a+" mode using fopen() and handle the file pointer that is returned.
   - Read the formatted data from keyboard using scanf().
   - Input that formatted data to function fscanf() along with the file pointer as follows:

fprintf(pointer_to_FILE, <format string>, < variables>);

   - fprintf() is same as printf(), except that it accepts a file pointer as first argument.
   - On success, it returns the number of bytes output. Otherwise, it returns EOF.

Ex:

```
struct student
{
        int rno,rank;
    float fees;
    char name[15];
}s1;
main()
{   FILE *fp;
    fp=fopen("student.dat","w"); //step-1:open file in write mode
    printf("\n Enter student details( rno,name,rank and fees):");
scanf("%d%s%d%f",&s1.rno,s1.name,&s1.rank,&s1.fees); //step-2: read content from keyboard
 fprintf(fp,"%d\t%s\t%d\t%f",s1.rno,s1.name,s1.rank,s1.fees); //step-3: write content to file
}
```

4. **fgetc():** fgetc() is used to read a character from a file, do the following:
   - Open the existing file in "r" or "r+" mode using fopen() and handle the file pointer that is returned.
   - Use the function **fgetc()** that is used to read a character from file which has the following prototype:

int fgetc(pointer_to_FILE);

- fgetc() takes an argument which is a pointer to the structure FILE and returns an unsigned integer value that should be converted to a character. We should place that file pointer which is already returned by using fopen().
- When end-of-file is reached or there is an error, the predefined character **EOF** is returned.

Ex:

```
FILE *fp;
char ch;
fp=fopen("sample.txt","r"); //step-a: Open file in read mode
ch=fgetc(fp);  //step-b : read a character from file
```

```
while(ch!=EOF) //read character-by-character till the end-of-file is reached
{
        printf("%c",ch); // step-c: display that character on monitor
        ch=fgetc(fp);
}
```

5. **fgets():** fgets( ) is used to to read a string from the file, do the following:

a. Open the existing file in "r" or "r+" mode using fopen() and handle the file pointer that is returned.
b. Use the function **fgets()** that is used to read a string of specified length from file which has the following form:

```
char * fgets(string,size,pointer_to_FILE);
```

- fgets() takes three arguments: first argument is a string, second argument is used to specify the size of string that is to be read and third argument file pointer.
- fgets() reads the next line from the specified file. On success, the string of specified size that is read will be stored in the first argument. On failure, fgets() returns NULL.

```
FILE *fp;
char *str="";           //initialize with empty string
fp=fopen("sample.txt","r");  //step-a: open file in read mode.
fgets(str,10,fp);        //step-b: str holds the string that is read from file
printf("%s",str);        //prints the string to monitor.
```

6. **fscanf():** fscanf( ) is used to to o read formatted data from a file do the following:

a. Open the existing file in "r" or "r+" mode using fopen() and handle the file pointer that is returned.
b. Use the function **fscanf()** that is used to read formatted data (i.e., different forms of data at a time) from file which has the following prototype:

```
fscanf(pointer_to_FILE,<format string>,<addresses of variables>);
```

- fscanf() is same as scanf(), except that it accepts a file pointer as first argument.
- On success, it returns the number fields successfully scanned, converted and stored. On failure, it returns 0, if no fields are stored.

Example:

```
struct student
{
        int rno,rank;
        float fees;
        char name[15];
}s1;
FILE *fp;
fp=fopen("student.dat","r"); //step-a:open file in read mode
fscanf(fp,"%d%s%d%f",&s1.rno,s1.name,&s1.rank,&s1.fees);
//step-b: read  formatted content from file
printf("\nRoll number=%d",s1.rno);// step-c: display content on  monitor
```

```
printf("\n Name=%s",s1.name);
printf("\n Rank=%d",s1.rank);
printf("\n Fees=%.2f",s1.fees);
```

STRUCTURE READ AND WRITE:

- The function **fwrite()** is used to write entire structure block to the file. It has the following form:

```
size_t  fwrite(ptr, size , n, File_Pointer);
```

- fwrite() takes four arguments: **ptr** is a pointer to the data that is to be written. **Size** is a value of type size_t, an unsigned integer, used to hold the size of data that is to be written. **n** is a value of type size_t, an unsigned integer, used to hold number of data items to be appended**. Pointer_to_FILE** is the pointer to the file in which the data is to be written.
- On success, it returns the number of items (not bytes) actually written. on failure, it returns a short count (possibly zero).

  Ex: struct student s1;

```
fwrite(&s1,sizeof(s1),1,fp);
```

- The function **fread()** is used to read structure block from the file. It has the following form:

```
size_t  fread(ptr,size,n,pointer_to_FILE);
```

- fread() takes four arguments: **ptr** is a pointer to the block into which data is read. **Size** is a value of type size_t, an unsigned integer, used to hold the length of each item read, in bytes. **n** is a value of type size_t, an unsigned integer, used to hold number of data items to read**. Pointer_to_FILE** is the pointer to the file in which the data is to be read.
- On success, it returns the number of items (not bytes) actually read. On end-of-file or error, it returns 0.

  Ex: struct student s1;

```
fread(&s1,sizeof(s1),1,fp);
```

By using fread() and fwrite() functions, fixed-length records are written or read. By using fprintf() and fscanf() functions, variable-length records are written or read.

INTEGER READ AND WRITE:

- The function **getw( )** is used to read integer data from the file. It has the following form:

  Syntax:

```
getw(fp);
```

- The function **putw( )** is used to write integer data to the file. It has the following form:

  Syntax:

```
putw(fp);
```

putw( ) and getw( ) both functions perform operation on integer data only.

Random-access files:

UNIT – IV

Random-access files are the stream-oriented files in which the accessing of data is done at any position. That is the data should be written to or read from any position in the specified file. A *position pointer* is always associated with a file while reading or writing. Read and write operations are done at the position pointed by this pointer. In order to perform random-access while reading a file, the functions fseek(), ftell() and rewind() functions are very helpful:

1. **fseek()**- places the position pointer at the specified position.
2. **rewind()**- places the position pointer to the beginning of the file.
3. **ftell()**- tells the current position of the position pointer in the file.

fseek() function: The fseek() function is used to place the position pointer at the specified location in the file. It has the following prototype:

```
int fseek(pointer_to_FILE,offset,location);
```

- fseek() function takes three arguments: the first argument is the file pointer that points to the file in which the position pointer should be placed. The second argument, a long integer value, is the position at which the pointer should be placed. If it is positive value, the file position pointer will be moved forward. If it is negative number, the file position pointer will be moved back. The third argument takes one of the following three:

Constant	value	File location
SEEK_SET	0	eks from beginning of file.
SEEK_CUR	1	eks from current position.
SEEK_END	2	eks from end of file.

- On success, it returns 0. On failure, it returns a non-zero value. fseek() returns an error code only when an unopened file or when a non-existing device is accessed.

rewind() function: The function rewind() is used to place the position pointer to the beginning of the file. It has the following prototype:

```
void rewind(pointer_to_FILE);
```

- rewind() function takes one argument: a pointer to the FILE in which we want to place the position pointer at the beginning of the file.

The same operation can also be done using the function call:

```
fseek(pointer_to_FILE,0,0);
```

ftell() function: The function ftell() is used to get the current file position pointer. It has the following prototype:

```
long ftell(pointer_to_FILE);
```

- ftell() function takes one argument: file pointer.
- on success, it returns the current file pointer position. On error, it returns -1 and sets errno to a positive value.

Write a program to demonstrate fseek(), ftell() and rewind() functions

#include<stdio.h>

```
main()
{
    FILE *fp;
    char ch;
    fp=fopen("stud.txt","w+");
    if(fp==NULL)
    {
        printf("\n Unable to open file:");
        exit(0);
    }
    printf("\n Enter the text into file (end with ctrl+d):");
    ch=getchar();
    while(ch!=EOF)
    {
        fputc(ch,fp);
        ch=getchar();
    }
    ch=fgetc(fp);
    while(ch!=EOF)
    {
        printf("%c",ch);
        ch=fgetc(fp);
    }
    fseek(fp,5,SEEK_SET);                          //place the cursor at the position 5
    printf("\n The cursor is at %d position",ftell(fp));   //tells the cursor position as 5
    printf("\n Enter the character to insert:");
    scanf("\n%c",&ch);
    fputc(ch,fp);                                  //inserts the character at specified position
    ch=fgetc(fp);
    while(ch!=EOF)          //prints all the characters from specified position
    {
        printf("%c",ch);
        ch=fgetc(fp);
    }
    rewind(fp);  //places the cursor to the beginning of file
    printf("\n The cursor is at %d position",ftell(fp));
```

```
        fclose(fp);

}
```

The content of stud.txt

pradKep is a good boy.

Output:

Enter the text into file (end with ctrl+d):

pradeep is a good boy.

The cursor is at 5 position

Enter the character to insert: K ep is a good boy.

The cursor is at 0 position

Write a program to search for particular word in a file

```
#include<stdio.h>
#include<string.h>
main()
{
        FILE *fp;
        char ch,temp[15],word[15],filename[15];
        int count=0;

        printf("\n Enter file name:");
        scanf("%s",filename);
        fp=fopen(filename,"w");
        printf("\n Enter the text (end with ctrl+d):");
        ch=getchar();
        while(ch!=EOF)
        {
                fputc(ch,fp);
                ch=getchar();
        }
        fclose(fp);
        printf("\n Enter the word to be searched:");
        scanf("%s",word);
        fp=fopen(filename,"r");
        //fscanf(fp,"%s",temp);
        while(!feof(fp))
        {
```

```
        fscanf(fp,"%s",temp);
        if(strcmp(temp,word)==0)
        count++;
    }
    fclose(fp);
    printf("\n The word %s is occured %d times",word,count);
}
```

Output:

Enter the file name: sample.txt

Enter the text into file (end with ctrl+d):

Mater minds are not made.

Master minds are born.

Enter the word to be searched: minds

The word minds is occurred 2 times.

ERROR-HANDLING

While processing files, we usually assume that:

- The file that is opened exists.
- There is enough space on the disk for the file.
- The functions fseek(), fread(), fwrite()…etc carry out the requested job successfully.

It is important to note that these assumptions are not always true. There is possibility of the following errors in manipulating files:

Error	Reason
File not found	Wrong file name used (spelling mistake, error in the path specified) or file actually not present.
Disk full	When writing to a disk with no more free sectors.
Disk write protected	The disk has its write protect tag put on or some software has disabled writes to the disk.
Unexpected EOF	File length is zero of file length is less than that expected.
Sector not found	Reading and writing to two different streams, one as read only and the other write only, but the file used for these streams is the same. The disk head is not aligned on the physical sectors.

The error-handling functions in C are:

perror() function: The perror() function is used to print the nature of the error. It has the following prototype:

```
void perror(char *msg);
```

The argument msg points to an optional user-defined message. This message is printed first, followed by a colon and the implementation-defined message that describes the most recent error. If you call perror() when no error has occurred, the message displayed is no error.

Write a program to demonstrate perror() function

```
#include <stdio.h>
#include <stdlib.h>
main()
{
  FILE *fp;
  char filename[80];

  printf("Enter filename: ");
  gets(filename);

if (( fp = fopen(filename, "r")) == NULL)
{
      perror("You made a mistake");
      exit(1);
}
else
{
   puts("File opened for reading.");
 fclose(fp);
}
}
```

Output:

Enter file name: master.xxx

You made a mistake: No such file or directory.

What is the purpose ferror()?

The ferror() is a macro that tests the given file pointer for a read or write error. It has the following prototype:

```
int ferror(pointer_to_FILE);
```

- ferror() function takes one argument: the file pointer .
- On success, it returns non-zero if an error is detected on the named file pointer.

Write a program to demonstrate ferror()

```
#include<stdio.h>
main()
{
	char filename[15],ch;
	FILE *fp;
	printf("\n Enter file name:");
	scanf("%s",filename);
	fp=fopen(filename,"r");
	if(fp==NULL)
	{
		printf("\n Unable to open file");
		exit(0);
	}
	fputc(ch,fp);
	if(ferror(fp)>0)
	{
		perror("error");
	}
	fclose(fp);
}
```

Output:

Enter file name: master.txt

error: Bad file descriptor.