**UNIT-12**

**UNDERSTAND BASICS OF FILE MANAGEMENT AND PREPROCESSOR DIRECTIVES**

**12.1: DEFINE FILE:**

**A file is a place on the disk, where a group of related data is stored.** C supports a no.of functions that the ability to perform basic file operations, like:

* Naming a file.
* Opening a file.
* Reading data from a file.
* Writing data to a file.
* Closing a file.

There are 2 ways to perform file operations in C. they are:

1. Using low-level I/O and uses UNIX system calls.
2. Using high-level I/O functions.

|  |  |
| --- | --- |
| **function name** | **operation** |
| fopen() | * creates a new file for use. * opens an existing file for use. |
| fclose() | * closes a file which has been opened for use. |
| getc() | * reads a character from a file. |
| putc() | * writes a character to a file. |
| fprintf() | * writes a set of data values to a file. |
| fscanf() | * reads a set of data values from a file. |
| getw() | * reads an integer from a file |
| putw() | * writes an integer to a file |
| fseek() | * sets the position to a desired point in the file. |
| ftell() | * gives the current position in the file |
| rewind() | * sets the position to the beginning of the file |

When a C program begins execution , the following three streams are already open:

**1.stdin (Standard Input):** stdin is the file from which input is received and normally from the keyboard which is the defaut input device.

**2 .stdout(Standard Output):** stdout is the default output file.

**3 . stderr(Standard Error):** The purpose of stderr is to display the error messages to the standard device i.e., to the screen.

**12.2 know how to declare file pointer to a file.**

To perform any operation on a file we must specify following things they are:

1. **File name**: is a string of characters that make up a valid filename. It contains two parts , a primary name and an optional period with the extension.
2. **Data structure**: is defined as FILE in library of standard I/O function
3. **Purpose**: specifies what we want to do with the file.

Following is the general format of declaring a File

|  |
| --- |
| **FILE \*fp;** |

🡪The above statement declares the variables **fp** as a “**pointer to the data type FILE”. FILE is a** structure that is defined in the I/O library. The “fp” pointer , which contains all the information about the file, is used as communication link between the system and program.

**12.3 Illustrate the concept of file opening in various modes.**

When we open a file , we must specify what we want to do with file . The general format for opening a File is :

**fp=fopen(“filename”,”mode”);**

The above statement opens the file named filename and assigns to the **FILE** type pointer fp.

The statement also specifies the purpose of opening this file.The modes can be of the following:

|  |  |
| --- | --- |
| **r** | open the file for reading only. |
| **w** | open the file for writing only. |
| **a** | open the file for appending only. |
| **r+** | the existing file is opened to the beginning for both reading and writing. |
| **w+** | same as **w** except both for reading and writing. |
| **a+** | same as **a** except both for reading and writing. |

When trying to open a file, one of the following.

1. If mode is “**writing**” file is created if the file does not exist. The contents are deleted, if the file already exists.
2. If mode is “**reading**” and if it is exists, then file is opened with the current contents safe otherwise an error occurs.
3. If mode is “**appending**”, file is opened with the current contents safe. A file with specified name is created if the file does not exist,

Consider the following statements:

**file \*fp1,\*fp2;**

**fp1=fopen(“data”, “r”);**

**fp2=fopen(“result”,”w”);**

The file **data** is opened for reading and **results** is opened for writing.

**12.4 Illustrate the concept of closing a file:**

A file must be closed as soon as all operations on it have been completed for following reasons

* Any accidental misuse of the file will be prevented.
* Sometimes we have to close a file when we want to reopen the same file in a different mode.

The I/O library supports a function to do this for us. It takes the following segment:

|  |
| --- |
| **fclose(file\_pointer);** |

e.g:

**file \*fp1,\*fp2;**

**fp1=fopen (“data”, “r”);**

**fp2=fopen (“result”, ”w”);**

**………….**

**………….**

**fclose(fp1);**

**fclose(fp2);**

**……….**

**……….**

This program opens two files and closes them after all operations on them are completed.

**12.5 Illustrate the concept of input/output operations on a file:**

Once a file is opened , reading from and writing data to it is accomplished using the standard I/O functions:

1. **The getc() and putc() Functions:**

The simplest file I/O functions are getc() and putc(). These are similar to getchar() and putchar() for handling one character at a time.

Assume that a file is opened with **w** and file pointer **fp1** , then the statement

**putc (c,fp1);**

Writes the character contained in the character variable **c** to the file associated with file pointer **fp1.**

Similarly, **getc** used to read a character from a file that has been opened in read mode.for example the statement:

**c=getc(fp2);**

would read a character form the file whose file pointer is fp2.

Note:

1. The file pointer moves by one character position for every operation of **getc** or **putc.**
2. The getc will return an end-of-file **EOF,** when end of the file has been reached. Therefore, the reading should be terminated when EOF is encountered.

|  |
| --- |
| #include<string.h>  #include<stdio.h>  main()  {  FILE \*fp;  char c;  fp=fopen("data","w");  printf("enter the data into data file\n");  while((c=getchar())!=EOF)  {  putc(c,fp);  }  fclose(fp); /\*closing the file data\*/  printf("\n output data in data file\n");  fp=fopen("data","r"); /\*re-opening data file\*/  while((c=getc(fp))!=EOF) /\*reading data from file and putting onto screen\*/  {  printf("%c",c);  }  fclose(fp); /\*closing the opended file\*/  } |

2.**The getw and putw functions:**

* The **getw** and **putw** are integer oriented functions.
* They are similar to the **getc** and **putc** functions and used to read and write integer values.
* The general form of **getw** and **putw** are:

|  |
| --- |
| **putw(integer,fp);**  **integer=getw(fp);** |
| Example 2: reading digits and placing them into a file, and writing even numbers,odd numbers among them into other files.  #include<string.h>  #include<stdio.h>  main()  {  FILE \*f1,\*f2,\*f3;  int number,i;  f1=fopen("data1","w"); /\*opening the file data1 with filepointer f1\*/  printf("enter the data into data file\n");  for(i=1;i<=30;i++)  {  scanf("%d",&number);  if(number==-1)  break;  putw(number,f1); /\*writing a digit to data1 file\*/  }  fclose(f1); /\*closing the file data1\*/  f1=fopen("data1","r");  f2=fopen("even","w");  f3=fopen("odd","w");  printf("\n read data from data1 file\n");  while((number=getw(f1))!=EOF)  {  if(number%2 == 0)  putw(number,f2); /\*writing to even file\*/  else  putw(number,f3); /\*wrinting to odd file\*/  }  fclose(f1); /\*closing the opended file data1\*/  fclose(f2); /\*closing the opended file even\*/  fclose(f3); /\*closing the opended file odd\*/  f2=fopen("even","r"); /\*opening the file even in read mode\*/  f3=fopen("odd","r"); /\*opening the file odd in read mode\*/  printf("\n contents of file even\n");  while((number=getw(f2))!=EOF)  {  printf("%d\t",number);  }  printf("\n contents of file odd\n");  while((number=getw(f3))!=EOF)  {  printf("%d\t",number);  }    fclose(f2);  fclose(f3);  } | | |

**3.using fprintf and fscanf functions:**

* The functions **fprintf** and **fscanf** perform I/O operations that are similar to the functions **printf** and **scanf ,** except that they **work on files.**

The general form of **fprintf** is:

|  |
| --- |
| **fprintf(fp, “control string”, list);** |

* fp-is the file pointer
* the control string contains output specifications for the items in the list.
* The list include variables or constants or strings.

e.g.: **fprintf(fp, “%d %s %f”, age, name, 7.5);**

The general form of **fscanf** is:

|  |
| --- |
| **fscanf(fp, “control string”, list);** |

* this statement would read the items in the list form the file specified by the file pointer **fp.**

e.g.: **fscanf(fp, “%d %s %f”,& age, name, &price);**

|  |
| --- |
| #include<stdio.h>  main()  {  FILE \*fp;  char item[10];  int number,quantity,i;  float price,value;  fp=fopen("inventory","w");  printf("itemname number price Quantity\n");  for(i=1;i<=3;i++)  {  scanf(“%s%d%f%d",item,&number,&price,&quantity);  fprintf(fp,"%s%d%f%d",item,number,price,quantity);  }  fclose(fp); /\*closing the file data\*/  printf("\n \n");  fp=fopen("inventory","r"); /\*re-opening data file\*/  printf("itemname number price Quantity\n");  for(i=1;i<=3;i++)  {  fscanf(fp,"%s%d%f%d",item,&number,&price,&quantity);  value=price\*quantity;  fprintf(fp,"%s %d %f %d %f",item,number,price,quantity,value);  }  fclose(fp); /\*closing the opended file\*/  } |

**12.6: illustrate the concept of random access to files.**

* When we are interested in accessing only a particular part of a file and not in reading the other parts, then this can be achieved with the help of the functions **fseek, ftell,** and **rewind.**

**ftell():** Takes a file pointer and returns the current position. This function is useful in saving the current position of a file, which can be used later in the program.

It takes the following form:

**n=ftell(fp);**

**rewind():** Take a file pointer and resets the position to the start of the file. For e.g: the statement

**rewind(fp);**

**n=ftell(fp);**

Would assign **0** to n because the file position has been set to the start of the file by rewind.

**fseek():** is used to move the file position to a desired location within the file . It takes the following form:

**fseek(filepointer, offset, position);**

* Filepointer is a pointer to the file ,
* Offset is a number or variable of type long –specifies number of positions to be moved from the location specified by position.
* Position is an integer number . and it has one of the following 3-values:

|  |  |
| --- | --- |
| **Value** | **Meaning** |
| 0 | Beginning of file |
| 1 | Current position |
| 2 | end of the file |

|  |
| --- |
| Example program on random access of a file:  #include<stdio.h>  main()  {  FILE \*fp;  char c;  long n;  fp=fopen("random","w");  printf("enter the data into the RANDOM file\n");  while((c=getchar())!=EOF)  {  putc(c,fp);  }  printf("\n no.of characters entered=%ld \n",ftell(fp));  fclose(fp); /\*closing the file random\*/  fp=fopen("random","r"); /\*re-opening random file\*/  n=0L;  while(feof(fp)==0) /\*reading data from file and putting onto screen\*/  {  fseek(fp,n,0); /\*position to starting of file\*/  printf("position of %c is %ld \n",getc(fp),ftell(fp));  n=n+5L;  }  } |

**12.7 State the need of preprocessor directives:**

Another unique feature of C language is the preprocessor . The C preprocessor provides several tools that are available in other language .

* Easy to modify.
* Easy to read.
* Portable.
* More efficient.
* It is used transport form one machine to another.
* Programs easier to develop.

**12.8 Explain preprocessor directives**.

**Preprocessor** : is a **program that processes the source code before it passes through the compiler**. It operates under the control of **preprocessor command line** or **directives.**

**Directives** are placed in the source program before the main line, and appropriate actions are taken and then the source program is handed over to the compiler.

All preprocessor commands(Directives) should

* Begin with pound symbol(#).
* Begin in first column.
* Be placed before main()

The list of C preprocessor directives are given below

|  |  |
| --- | --- |
| **Directive** | **Function** |
| #define | Defines a macro substitution. |
| #undef | Un defines a macro |
| #include | Specifies the files to be included. |
| #ifdef | Test for a macro definition. |
| #if | Test a compile-time condition. |
| #else | Specifies alternatives when #if test fails. |
| #endif | Specifies the end of #if |
| #ifndef | Tests whether a macro is not defined. |

Preprocessor directives can be divided into 3-categories:

**1.Macro substitution directives**: Used to define and undefined user defined macros.

**2. File inclusion directives:** Used to include header files and other C files in a C program .

**3.Compiler control directives :** Used to provide conditional compilation and to change the natural flow of control in macro substitution and handle errors.

12.9 **Explain macro substitution using #define with an example**:

Macro substitution is a process where an identifier in a program is replace by a string composed of one or more tokens using **#define** statement. It takes the following form:

**#define identifier string**

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the **identifier** in the source code by the string.

There are different forms of macro substitutions and are:

1. Simple macro substitution.
2. Argumented macro substitution.
3. Nested macro substitution.

* **Simple macro substitution:** examples are

**#define COUNT 100**

**#define FALSE 0**

**#define SUBJECTS 6**

**#define Pi 3.14**

Example :

**#define M 5**

**……**

**…..**

**Total=M+200; /\*5 will be substituted in M\*/**

**Printf(“%d”,M); /\* 5 will be displayed on screen\*/**

Note:

1. It can include expressions as well.

e.g.: **#define area 5\*12.46**

**#define size sizeof(int)\*4**

**#define D (45-32)**

1. We can also use macro to define almost anything.

e.g: **#define TEST if(x>y)**

**#define AND**

**#define PRINT printf(“x is big”);**

to build statement as follows:

**TEST AND PRINT.**

This is equivalent to

If(x>y) printf(“x is big”);

1. Some more useful macro substitutions are:

**#define EQUALS ==**

**#define AND &&**

**#define OR ||**

* **Argumented macro substitution:** It takes the form:

**#define identifier(x1,2,……….x3) string**

e.g.:

#define cube(x) x\*x\*x

If the following statement appears later in the program:

**Volume=cube(side);**

Then the preprocessor would expand this statement to:

**Volume=(side\*side\*side);**

* **Nested macro substitution:** by using this we can use one macro in the definition of another one.

e.g.: **#define M 5**

**#define N M+1**

1. **File inclusion directives:**

An external file containing functions, variables or macro definitions can be includes as apart of our program.

The #include directive is used to inform the preprocessor to include or attach the specified header to the source code that we write.

The #include can be used in two ways

**#include<filename>:** include the system header files

**#include “filename”**: include the user defined header files

**Compiler control directives :**

**#ifdef:**

#ifdef is the simplest form of conditional preprocessor directive and is used to check for the existence of macro definition.

**Syntax:**

#ifdef MACRO

CONTROLLED TEXT

#endif

**#ifndef:**

It checks whether the macro has not been defined.

**Syntax:**

#ifndef MACRO

controlled text

#endif

**#if:**

The if directive is used to control the comp[ilation of portions of a source file.

If the specified condition has a non-zero value,the controlled text immediately following the #if directive is retained in the translation unit.

**Syntax:**

#if condition

controlled text

#endif

**#else:**

The #else directive can be used within the controlled text of a #if directive to provide alternative text to be used if the condition is false.

**Syntax:**

#if condition

Controlled text 1

#else

Controlled text 2

#endif

**#elif:**

The #elif directive is used when there are more than two possible alternatives.

The #elif directive is like #else directive embedded within the #if directive.

**Syntax:**

#if condition

Controlled text 1

#elif new-condition

Controlled text 2

#endif

**#endif:**

The #endif directive is used to end the conditional compilation directive.

**Syntax:**

#endif