Unit - III: Pointers, Preprocessor and File handling in C:

Pointers: Idea of pointers, Defining pointers, Pointers to Arrays and Structures, Use of Pointers in self-referential structures, usage of self referential structures in linked list (no implementation).

Pre processor: Commonly used Pre processor commands like include, define, undef, if, ifdef, ifndef

Files: Text and Binary files, Creating and Reading and writing text and binary files, Appending data to existing files **,** Random access using fseek, ftell and rewind functions.

C Pointers

- The pointer in C language is a variable which stores the address of another variable.
- This variable can be of type int, char, array, function, or any other pointer.
- The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte, in 64-bit architecture the size of a pointer is 4 byte.
- A Pointer in C is used to allocate memory dynamically i.e. at run time.

Syntax :

data_type *var_name;

where

- data_type any datatype like int,char, float..
- var_name any user defined name.

Example : int *p; char *p;

Where, * is used to denote that "p" is pointer variable and not a normal variable.

Consider the following example to define a pointer which stores the address of an integer.

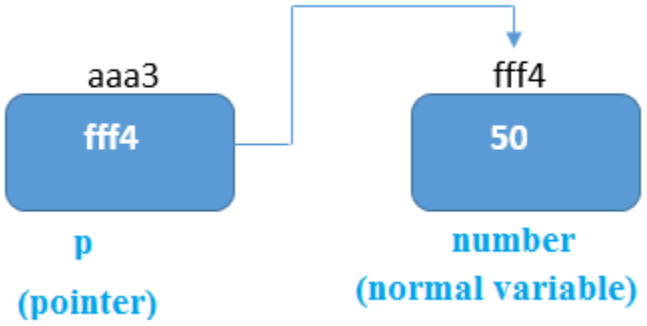
int n = 10;

int* p = &n;

// Variable p of type pointer is pointing to the address of the variable n of type integer.

Pointer Example

An example of using pointers to print the address and value is given below.



int n=50;

int *p;

p=& n;

p=address of n, *p=50(i.e *(&n)=50)

int a[50], p=a p=&a[0];

As you can see in the above figure, pointer variable stores the address of number variable, i.e., fff4.

The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (indirection operator), we can print the value of pointer variable p.

KEY POINTS TO REMEMBER ABOUT POINTERS

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null,
  i.e. int *p = NULL;
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.

Example of pointer

```
#include<stdio.h>
void main()
{
int n=50;
int *p;
p=&n; //stores the address of number variable
printf("\n values of n=%d",n);
printf("\n Address of n=%u",&n);
printf("\n Address of n variable is %u \n",p); //p=&n
printf("\n Value of n variable is %d \n",*p);// p=*(&n)
}
```

Output

values of n=50

Address of n=fff4

Address of n variable is fff4

value of n variable is 50

Advantage of pointer

1. **Pointer reduces the code and improves the performance;** it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

2. We can return multiple values from a function using the pointer.

3. It makes you able to access any memory location in the computer's memory.

Usage of pointer (applications)

There are many applications of pointers in c language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

2) Arrays, Functions, and Structures

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

Compatible in pointers

Two pointer types with the same type qualifiers are compatible .

Example:

```
    float  s;
float  *ps;
/* ... */
ps = &s; //compatible
```

The next example shows incompatible

```
int  a;
float  *p;

p=&a   /* error/ incompatible */
```

TYPES OF POINTERS

1. NULL Pointer

- A pointer that is not assigned any value but NULL is known as the NULL pointer.
- If you don't have any address to be specified in the pointer at the time of declaration, you can assign NULL value. It will provide a better approach.

int *p= NULL; //initialize the pointer as null.

In the most libraries, the value of the pointer is 0 (zero).

// **the example NULL pointer.**

```
#include <stdio.h>
int main()
{
    // Null Pointer
    int *ptr = NULL;
   printf("\n The value of ptr is %p", ptr);
    return 0;
}
```

Output:

The value of ptr is

2. Wild Pointer

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer.

int *p; // by default garbage address

Program:

```
int main()
{
   int *p;  /* wild pointer */

   int x = 10;
   // p is not a wild pointer now
   p = &x;

   return 0;
}
```

Note: NULL vs Void Pointer – Null pointer is a value, while void pointer is a type

3. Dangling Pointers in C

Dangling pointer is a pointer pointing to a memory location that has been free (or deleted).

4. Void pointer

- Void pointer is a pointer which is not associate with any data types.
- It points to some data location in storage means points to the address of variables. It is also called general purpose/universal pointer.

Syntax:

```
void  *variable_name;
 Ex: void *p;
```

Program:

```
#include<stdlib.h>
void main()
{
   int x = 4;
   float y = 5.5;
   void  *ptr;      //A void pointer
   ptr = &x;
   printf("\n Integer variable is = %d\n", *( (int*) ptr) );
   ptr = &y;      // void pointer is now float
   printf("\nFloat variable is= %f\n", *( (float*) ptr) );

}
```

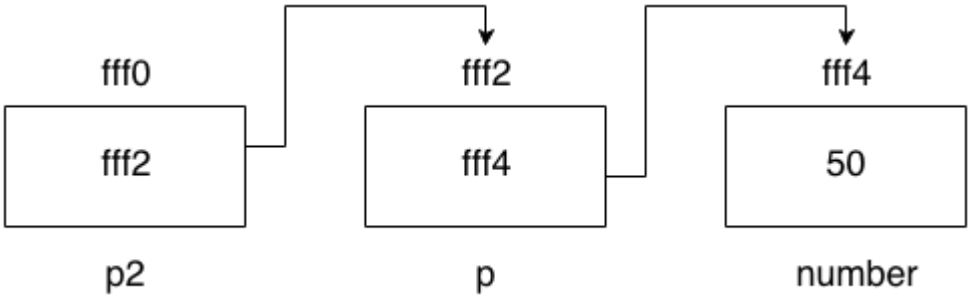
Output:

```
Integer variable is = 4
Float variable is= 5.50000
```

Pointer to Pointer VARIABLE

The syntax of declaring a double pointer is given below.

int **p2; // pointer to a pointer which is pointing to an integer.




```
#include<stdio.h>
void main()
{
int  number=50;
int  *p;//pointer to int
int  **p2;//pointer to pointer
p=&number;//stores the address of number variable
p2=&p;
printf("Address of number variable is %u \n",&number);
printf("Address of number variable is %x \n",p);
printf("Value of  number variable is %d \n",*p);
printf("Address of p variable is %x \n",p2);
printf("Value of p variable is %d \n",**p2);
}
```

Output:

Address of number variable is fff4

Address of number variable is fff4

Value of number variable is 50

Address of p2 variable is fff2

Value of number variable is 50

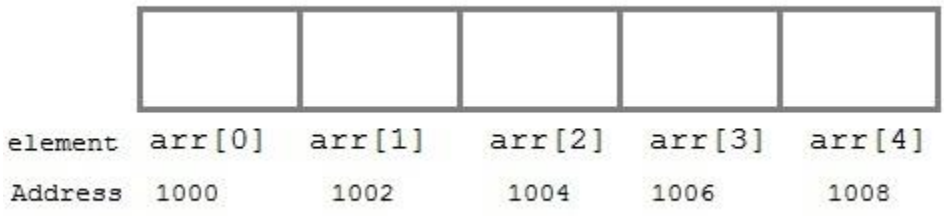
Pointer to Arrays in C

- Pointer to an array is also known as array pointer.

Suppose we declare an array arr,

int arr[5] = { 1, 2, 3, 4, 5 };

Assuming that the base address of arr is 1000 and each integer requires two bytes, The five elements will be stored as follows:



- Here variable arr will give the base address, which is a constant pointer pointing to the first element of the array, arr[0].
- Hence arr contains the address of arr[0] i.e 1000.
- In short, arr has two purpose - it is the name of the array and it acts as a pointer pointing towards the first element in the array.

Note: arr is equal to &arr[0] by default

We can also declare a pointer of type int to point to the array arr.

```
int *p; // pointer
int arr[10]; // array
p = arr;
// or,
p = &arr[0];    //both the statements are equivalent.
```

Now we can access every element of the array arr using p++ to move from one element to another

Example:

```
#include <stdio.h>
void main()
{
   int i;
   int a[5] = {1, 2, 3, 4, 5};
   int *p = a;     // same as int*p = &a[0]
   for (i = 0; i < 5; i++)
   {
      printf("%d\n", *p);
      p++;
   }
   }
```

In the above program, the pointer *p will print all the values stored in the array one by one.

Pointers to Structures

Pointer is a variable which contain address of another variable

So similarly we have pointer to structure

Definition:

A variable which contains address of a structure variable is called pointer to structure

```
struct student
{
  data-type member-1;
  data-type member-2;
  data-type member-3;
  data-type member-n;
};
main()
{
  struct student *ptr,s;
ptr=&s;
```

Assessing structure members using pointer to structure variable

Two methods

1. **Using dereference operator * and (.) operator**
2. **Using selection operator (->)(minus+ greater than symbol)**

Example: ptr *.member

ptr ->member

Example: using structure pointer.

```
#include <stdio.h>
struct person
{
  char name[30];
  int age;
  float weight;
};

void main()
{
   struct person p, *p1;
   p1 = &p;   //pointer to structure

   printf("Enter name:\n ");
   scanf("%s", &p1->name);

   printf("\nEnter age: ");
   scanf("%d", & p1->age);

   printf("\n Enter weight: ");
   scanf("%f", &p1->weight);


 printf("\n Displaying person details:\n");
   printf("\nName: %s", p1->name);
   printf("\n Age: %d\n", p1->age);

   printf("\n weight: %f\n", p1->weight);


}
```

Output:

Enter Name: Amit
Enter age: 21
Enter weight:61.5

Name: Amit
Age: 21
weight:61.5

Self Referential Structures

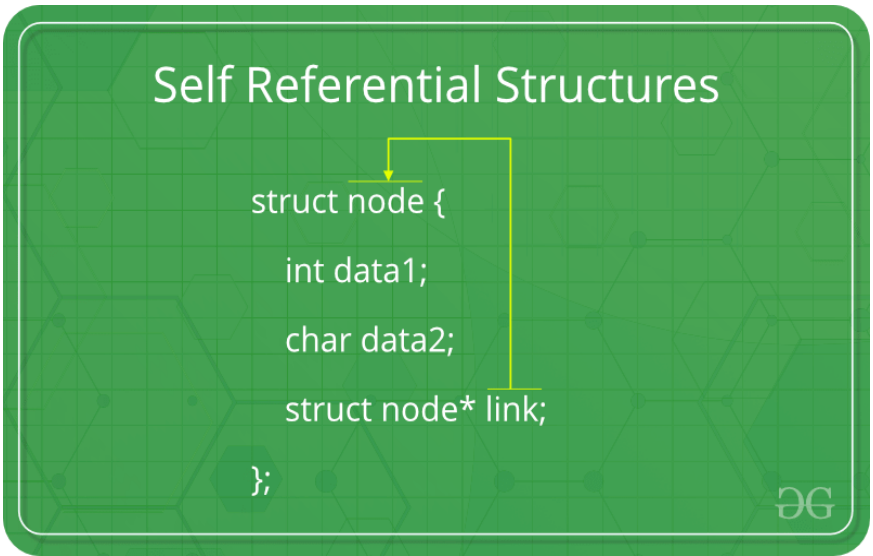
Definition:

Self referential a structure definition which includes/contain at least one member that is a pointer to the structure variable of same structure is called self referential structure

Sytax:

```
struct name
{
        member 1;
        member 2;
        . . .
        struct name *pointer;
};
```

Example:

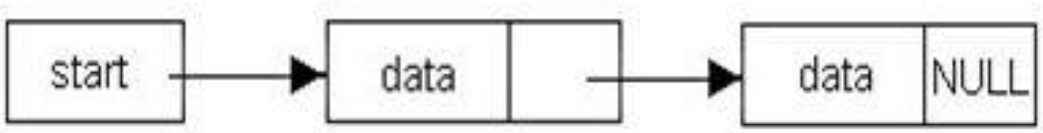


Usage of self referential structures in linked list

- Self-referential structures are very useful in applications that involve linked data structures, such as lists and trees.
- Static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array.
- Self-referential structure can dynamically be expanded or contracted.
- Operations like insertion or deletion of nodes in a self- referential structure involve simple and straight forward alteration of pointers.

Linked List (singly linked list)

A linear linked list illustration:



A linear linked list is a chain of structures where each node points to the next node to create a list.

To keep track of the starting node's address a dedicated pointer (referred as start pointer) is used.

The end of the list is indicated by a NULL pointer. In order to create a linked list of integers, we define each of its element (referred as node) using the following declaration.

```
struct  node_type
        {
        int data;
        struct node_type *next;
        };
struct  node_type *start = NULL;
```

Preprocessor Directives

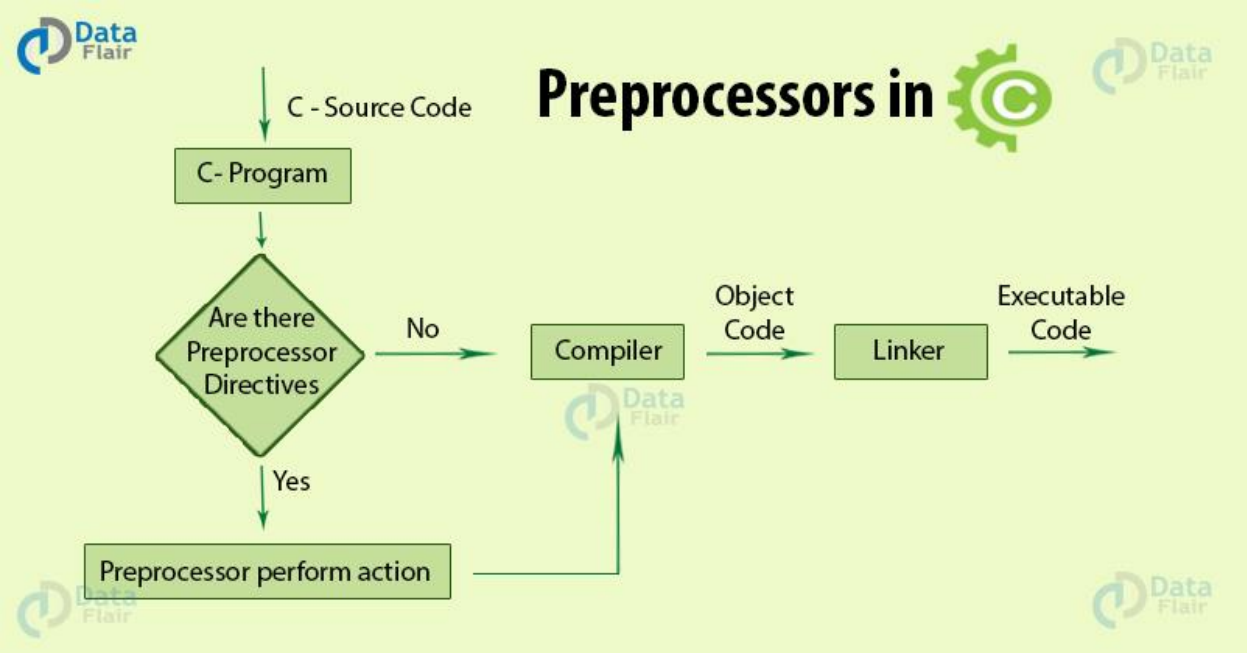
Introduction

- A program which processes the source code before it passes through the compiler is known as **preprocessor.**
- The commands of the preprocessor are known as **preprocessor directives.**
- It is placed before the main().
- It begins with a # symbol.
- They are never terminated with a semicolon.

Note: There is no semi-colon (';') at the end of preprocessor directives.

Examples of Preprocessor Directives

```
#define  PI  3.14
#include<stdio.h>
#include<math.h.>
#undef
#ifdef
```



Preprocessor Directives

The preprocessor directives are divided into four different categories which are as follows:

There are 4 main types of preprocessor directives:

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

Preprocessor	**Syntax/Description**

Header file inclusion	Syntax: #include<file_name> The source code of the file "file_name" is included in the main program at the specified place.
Macro	Syntax: #define This macro defines constant value and can be any of the basic data types.
Conditional compilation	Syntax: #ifdef, #endif, #if, #else, #ifndef Set of commands are included or excluded in source program before compilation with respect to the condition.
Other directives	Syntax: #undef, #pragma #undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

1. Macro expansion

- There are two types of macros - one which takes the argument and another which does not take any argument.
- Values are passed so that we can use the same macro for a wide range of values.

Syntax:

#define variable value

Where,

variable – it is known as the micro template.
value – it is constant.

Ex:

#define PI 3.14

Some of point on macro

- A macro name is generally written in capital letters.
- If suitable and relevant names are given macros increase the readability.
- If a macro is used in any program and we need to make some changes throughout the program we can just change the macro and the changes will be reflected everywhere in the program.

Example1: // Find area of circle

```
#define PI 3.14
void main()
{
    int r=3;
float area=0;
area=PI*r*r;
pritnf(“\area of circle=%f”,area);

    }
```

Output:

area of circle=28.26

Example2 : Simple macro

```
#define  LOWER  10
void  main()
{
    int  i;
    for (i=1;i<=LOWER; i++)
    {
        printf("\n%d", i);
    }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C Predefined Macros

ANSI C defines many predefined macros that can be used in c program.

No.	Macro	Description
1	_DATE_	Represents current date in "MMM DD YYYY" format.
2	_TIME_	Represents current time in "HH:MM:SS" format.
3	_FILE_	Represents current file name.
4	_LINE_	Represents current line number.
5	_STDC_	It is defined as 1 when compiler complies with the ANSI standard.

File: simple.c

```
#include<stdio.h>
 int main()
{
   printf("File :%s\n", __FILE__ );
   printf("Date :%s\n", __DATE__ );
   printf("Time :%s\n", __TIME__ );
   printf("Line :%d\n", __LINE__ );
   printf("STDC :%d\n", __STDC__ );
   return 0;
 }
```

Output:

```
File :simple.c
Date :Jan 6 2020
Time :12:28:46
Line :7
STDC :1
```

Note : We have saved the above code in simple.c file.

2. File inclusion

The file inclusion uses the #include.

Syntax:
#include< file_name >

- The content that is included in the filename will be replaced at the point where the directive is written.
- By using the file inclusive directive, we can include the header files in the programs.
- Macros, function declarations, declaration of the external variables can all be combined in the header file instead of repeating them in each of the program.
- The stdio.h header file contains the function declarations and all the information regarding the input and output.

There are two ways of the file inclusion statement:

1. **Header File or Standard files:**

   Syntax: #include< file_name >

   Example: #include< stdio.h >
2. **user defined files:**
   The #include "filename" tells the compiler to look in the current directory from where program is running.

   Syntax: #include “file_name”

   Example: #include “stdio.h”

- If the first way is used, the file and then the filename in the current working directory and the specified list of directories would be searched.
- If the second way, is used the file and then the filename in the specified list of directories would be searched.

3. Conditional compilation

- Conditional Compilation directives are type of directives which helps to compile a specific portion of the program or to skip compilation of some specific part of the program based on some conditions.

- It's similar to a if statement with one major difference.

- The if statement is tested during the execution time to check whether a block of code should be executed or not whereas, the conditionals are used to include (or skip) a block of code in your program before execution.

To use conditional,

#ifdef, #if, #else ,#elseif and #endif directives are used.

1. #ifdef Directive

Syntax

```
#ifdef MACRO
  // conditional codes
#endif
```

Here, the conditional codes are included in the program only if MACRO is defined.

Example program: #ifdef

```
#include <stdio.h>

#define DEBUG 1

void main()
{
        #ifdef DEBUG
                printf("Debug is ON\n");
                printf("Hi friends!\n");
        #else
                printf("Debug is OFF\n");
        #endif
}
```

Output

```
Debug is ON
Hi friends!
```

Explanation:

In this example macro DEBUG is defined so code written within the #ifdef and #endif will be executed.

2. **#ifndef**

   The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO
//code
#endif
```

Example program: #ifndef

```
#include <stdio.h>

#define DEBUG 1

void main()
{
    #ifndef DEBUG
        printf("Debug is ON\n");
        printf("Hi friends!\n");
    #else
        printf("Debug is OFF\n");
    #endif
}
```

Output

Debug is OFF

Explanation:

In this example macro DEBUG is defined so code written within the #ifndef so it will execute #else and #endif will be executed.

3. **#if**

Syntax:

```
#if expression
  // conditional codes
#endif
```

Here, expression is an expression of integer type (can be integers, characters, arithmetic expression, macros and so on).

Example program: #if

```
#include <stdio.h>
#define NUMBER 0
void main()
 {
#if (NUMBER==0)
printf("Value of Number is: %d",NUMBER);
#endif
}
```

Output:

Value of Number is: 0

4. #else

The optional #else directive can be used with #if directive.

Syntax:

```
#if expression
  conditional codes if expression is non-zero
#else
  conditional if expression is 0
#endif
```

Example program: #else

```
#include<stdio.h>
    #define MAX 45
    void main()
    {
        #if MAX > 40
            printf("Yes, MAX is greater than 40.");
        #else
            printf("No, MAX is not greater than 40.");
        #endif
    }
```

Output :

Yes, MAX is Greater than 40.

5. **#elif**

You can also add nested conditional to your #if...#else using #elif

Syntax:

```
#if expression
   // conditional codes if expression is non-zero
#elif expression1
   // conditional codes if expression is non-zero
#elif expression2
   // conditional codes if expression is non-zero
#else
   // conditional if all expressions are 0
#endif
```

Example program: #elif

```
#include<stdio.h>
#define NUM 10
void main()
{
#if(NUM == 0)
    printf("\nNumber is Zero");
#elif(NUM > 0)
    printf("\nNumber is Positive");
#else
    printf("\nNumber is Negative");
#endif
}
```

Output:

Number is Positive

4. Other directives (Miscellaneous directive)

There are some directives which do not fall in any of the above mentioned categories.

There are two directives:

1. **#undef :** This directive is used in relation to the #define directive. It is used to undefine a defined macro.
2. **#pragma :** It is a specialized and rarely used directive. They are used for turning on and off certain features.

#undef

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax:

#undef token

Example

```
#define WIDTH 80
#define ADD ( X, Y ) ((X) + (Y)) // Expression
.
.
.
#undef WIDTH
#unde ADD
```

```
/* program define, undefine and redefine a macro*/
#include <stdio.h>
void main()
{
#define MAXBUFF 100
   printf("\nMAXBUFF is : %d", MAXBUFF);
#undef MAXBUFF /*Un-define MAXBUFF*/
#define MAXBUFF 200 /*Redefine MAXBUFF*/
   printf("\nMAXBUFF is : %d", MAXBUFF);
#undef MAXBUFF
}
```

#pragma Directive in C

- This directive is a special purpose directive and is used to turn on or off some features.
- This type of directives are compiler-specific i.e., they vary from compiler to compiler.

Syntax:

```
#pragma token_name
```

Some of the #pragma directives are discussed below:

Sr.No.	#pragma Directives & Description
1	**#pragma startup** Before the execution of main(), the function specified in pragma is needed to run.
2	**#pragma exit** Before the end of program, the function specified in pragma is needed to run.
3	**#pragma warn** Used to hide the warning messages.

Example program on #pragma

```
#include<stdio.h>
void func1();
void func2();

#pragma startup func1
#pragma exit func2

void func1()
{
   printf("Inside func1()\n");
}

void func2()
{
   printf("Inside func2()\n");
}

int main()
{
   printf("Inside main()\n");
```

```
    return 0;
}
```

Output

Inside func1()

Inside main()

Inside func2()

Explanation: program on #pragma

First it will execute func1 next it will execute main function next it will execute func2.

The #error Preprocessor Directive

The #error preprocessor directive is used to force the compile to stop compiling the source code. In other words we can manually force the compiler to stop compiling and give fatal error.

Example of #error preprocessor directive 1:

```
#include<stdio.h>
    void main()
    {
        #ifndef MAX
            printf("\nMAX is not defined.");        //Statement   1
        #else
            printf("\nMAX is defined.");            //Statement   2
         #endif
         printf("\nEnd of program.");               //Statement   3
     }
```

Output:

MAX is not defined.

End of program.

Example of #error preprocessor directive 2:

```
#include<stdio.h>
    void main()
    {
        #ifndef MAX
            #error MAX is not defined.          //Statement   1
        #else
            printf("\nMAX is defined.");          //Statement   2
         #endif

         printf("\nEnd of program.");             //Statement   3
     }
```

Explanation:

The only difference between example 1 and 2 is the statement 1. In example 1, statement 1 will display message and continue to compile the rest of the code. In example 2, statement 1 will force the compiler to give fatal error and stop compiling the rest of the code.

Summary of preprocessor directives

Following table will show you various directives that we have studied in this chapter:

Directives	Description
#define	It substitutes a preprocessor macro.
#include	It inserts a particular header file from another file.
#undef	A preprocessor macro is undefined.
#ifdef	It returns true if the macro is defined.
#ifndef	It returns true if the macro is not defined.
#if	It tests if the compile time condition is true.
#else	It is an alternative for #if.
#elif	It has #else and #if in one statement.
#endif	The conditional preprocessor is ended.
#error	It prints the error message on stderr.
#pragma	It issues special commands to the compiler by using a standardized method.

Difference between Macros and functions

Macros	Functions
Macro calls are replaced with macro expression.	In function call, the control is passed to a function definition along with arguments, and definition is processed and value may be returned to call.
Macros run programs faster but increase the program size.	Functions run programs slower but decrease the program size and compact.
It is better to use Macros, when definition is very small in size.	It is better to use Functions, when definition is bigger in size.

(FILES)

Files: Text and Binary files, Creating and Reading and writing text and binary files, Appending data to existing files, Writing and reading structures using binary files, Random access using fseek, ftell and rewind functions.

Introduction

- A collection of relational data which is stored permanently on a secondary device like a hard disk is known as a file.
- A file is generally used as real-life applications that contain a large amount of data.

There are two problems with such applications:

1. Handling the huge amount of data,
2. To storing the input output data on a permanent device.

Types of Files

When dealing with files, there are two types of files you should know about:

1. Text files
2. Binary files

1. Text files

- Text files are the normal .txt files. You can easily create text files using any simple text editors such as Notepad.
- When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.
- They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.
  Example: one.txt, data.txt….

2. Binary files

- Binary files are mostly the .bin files in your computer.

- Instead of storing data in plain text, they store it in the binary form (0's and 1's).

- They can hold a higher amount of data, are not readable easily, and provides better security than text files.

  Ex: one.bin,data.exe,calc.exe,prog.java

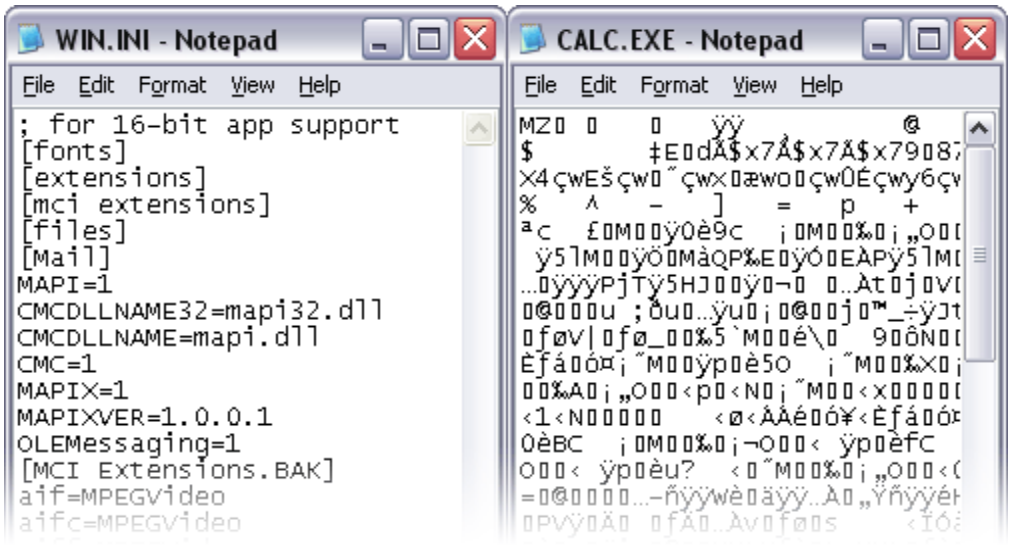


Fig: Text file and Binary file

Difference between TEXT and BINARY files:

	TEXT FILE	BINARY FILE
i)	Data are human readable characters.	Data is in the form of sequence of bytes.
ii)	Each line ends with a newline character.	There are no lines or newline character
iii)	Ctrl+z or Ctrl+d is end of file character.	An EOF marker is used.

iv)	Data is read in forward direction only.	Data may be read in any direction.
v)	Data is converted into the internal format before being stored in memory	Data stored in file are in same format that they are stored in memory

Files have three important points

1. **Naming of file**
2. **Data structure of a file**
3. **Purpose of file**

1. **Name of file**

- Valid name of file(group of characters)
- Should have two parts

Name and extension

Ex: data.c, one .txt

2. **Data structure of a file**

- FILE is the data type defined for a file

Syntax:

FILE pointer type;

Ex: FILE *fp;

Where: fp is user defined pointer

- File pointer are used communication between the file and the program.
- File pointer act a link between the operating system and a program

3. **Purpose of file**

Should defined whether the file is to be read, written or appended by defining the modes such as

'r' , 'w', or 'a' respectively.

File Operations (input /output functions)

In C, you can perform four major operations on files, either text or binary:

1. Opening an existing file
2. Closing a file
3. Reading from and writing information to a file

Opening a file - for creation and edit

- A file needs to be opened when it is to be used.
- file is opened by fopen() function defined in the stdio.h header file.
- fopen() function have two parameters
    1. file name
    2. file open mode

Syntax

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

Example:

```
fopen("program.txt","w");

fopen(“program.bin","rb");
```

Note: fopen() must be used with file pointer

```
Ex: FILE *fp;

 fp= fopen("program.txt","w");
```

Closing a File

- The file (both text and binary) should be closed after reading/writing.

Closing a file is performed using the fclose() function.

Syntax: fclose(fp);

Reading /writing a file:

- Once the file is opened (i.e it is loaded into the main memory) the file pointer points the staring character of file contents
  1. **fgetc() or getc()**
     function used to read character from file

Syntax:

ch= fgetc(fp)

2. **fputc() or putc()**

   function is write character on the file

Syntax:

fputc(ch,fp)

Here, fp is a file pointer associated with the file to be closed.

File Mode Operations (Text files)

1. Read (r)
2. Write (w)
3. Append (a)
4. Read/write(r+)
5. Write/read (w+)
6. Append/read (a+)

File Mode operations (Biary files)

1. Read (rb)

2. Write (wb)
3. Append (ab)
4. Read/write(rb+)
5. Write/read (wb+)
6. Append/read (ab+)

Mode	Meaning of Mode	During In existence of file
r	Open for reading.	If the file does not exist, fopen()/fp returns NULL. If the file exists, the content file read purpose
w	Open for writing	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append.	If the file does not exist, it will be created. If file exist, Data is added to the end of the file.
r+	Open for both reading and writing.	If the file does not exist, fopen()/fp returns NULL. If the file exists, the content file read/write purpose
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created. File open for write and read purpose
a+	Open for both reading and appending.	If the file does not exist, it will be created. Data is added to the end of the file. File open for append and read purpose

Binary files		
rb	Open for reading in binary mode.	If the file does not exist, fopen()/fp returns NULL If the file exists, the content file read purpose
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
ab	Open for append in binary mode.	If the file does not exist, it will be created. Data is added to the end of the file.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL. If the file exists, the content file read/write purpose
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created. File open for write and read purpose
ab+	Open for both reading and appending in binary mode.	If the file does not exist, it will be created. Data is added to the end of the file. File open for append and read purpose

Reading Writing characters using fgetc() and fputc() function

The fputc() function

- The fputc() function is used to write a single character specified by the first argument to a text file pointed by the fp pointer.
- After writing a character to the text file, it increments the internal position pointer

Syntax of fputc() function

fputc(char ch, FILE *fp); or fputc(ch, fp);

Where:

Ch: is character to be written to the file

fp: fp is file pointer

The fgetc() function.

- This function is complementary to fputc() function
- The fgetc()function reads a single character from the file and increments the file position pointer.
- To use this function file must be opened in read mode.
- It gets a character from the stream. It returns EOF at the end of file.

Syntax of fgetc() function

```
char ch;
ch=fgetc(FILE *fp); or ch=fgetc(fp);
```

where:

fp: is file pointer

ch: is character read from the file

The fgetc() function takes the file pointer indicates the file to read from and returns the character read from the file or returns the end-of-file character if it has reached the end of file.

// write a c program read the content of file display it

(fputc () and fgetc() function)

```
#include<stdio.h>
void main()
{
   char ch;
        FILE *fp;
   fp = fopen("one.txt", "w");
   printf("\n Enter data...press ctrl+z or ctrl+D for Linux:\n");
   while( (ch = getchar()) != EOF) // reading characters
        {
      fputc(ch, fp);
   }
   fclose(fp);
printf(“\n The  content of file is:\n”);
   fp = fopen("one.txt", "r");
   while( (ch = fgetc(fp))! = EOF)
   printf("%c",ch);
   fclose(fp);  //   closing the file pointer
   }
```

Output:

```
Enter data...press ctrl+z :
Hello cmrtc
The  content of file is:
Hello cmrtc
```

Appending data to existing files,

Step by step descriptive logic to append the one file to second file.

- Open both source files in r (read) and destination file in w (write) mode.
- Copy file contents from both source file one by one to destination file.
- Close all files to save and release all resources.

Let the given two files be file1.txt and file2.txt. The following are steps to merge.

1) Open file1.txt in read mode.
2) Open file2.txt in write mode.
3) Run a loop to one by one copy characters of file1.txt to file2.txt.
4) Close all files.

Example

Source file content (file1.txt)(contain)

It is very useful language.

Destination file (file 2.txt)(contain)

This is c programming.

Output file content after append (file 2.txt)

This is c programming.
It is very useful language.

Program

```
#include <stdio.h>
#include<stdlib.h>  // for exit(0)

        void  main()
        {

         FILE *fp1 = fopen("file1.txt", "r");

// Open file to store the result

        FILE *fp2 = fopen("file2.txt", "a");
```

```
            char ch;

        if (fp1 == NULL || fp2 == NULL )
        {
        printf("\n Could not open files\n");
        exit(0);
        }
```

// Copy contents of first file to file2.txt

```
        while ((ch = fgetc(fp1)) != EOF)
          fputc(ch, fp2);

        fclose(fp2);

        fp2=fopen("file2.txt",  "r");

printf("\n Append the  file2.txt with file1.txt\n ")

        printf("\n The second file content is:\n);
        while((ch=getc(fp2))!=EOF)
        printf("%c",ch);

        fclose(fp1);
        fclose(fp2);
        }
```

Output:

Append the file2.txt with file1.txt

This is c programming.
It is very useful language.

The Reading writing string using fputs() and fgets() functions

The fputs() function

The fputs() function is used to write string(array of characters) to the file.

Syntax of fputs() function

fputs(char str[], FILE *fp);

Example: fputs("hello c programming",fp);

The fputs() function takes two arguments,

- str: str is the string to be written to the file
- fp: fp is the file pointer where the string will be written.

The fgets() function

The fgets() function is used to read string(array of characters) from the file.

Syntax of fgets() function

fgets(char str[],int n,FILE *fp);

Example : fgets(str,200,fp)

Where: str is character array

The fgets() function takes three arguments, first is the string read from the file, second is size of string(character array) and third is the file pointer from where the string will be read.

Note: The fgets() function will return NULL value when it reads EOF(end-of-file).

Example: The fputs( ) and fgets( ) function

```
#include<stdio.h>
void main()
{
FILE *fp;
```

```
char str[50];
fp=fopen("data.txt","w");  // Statement 1
printf(“\n file witting\n” );
fputs("hello c programming",fp);  // fputs write data on file
fclose(fp);
fp=fopen("data.txt","r");  // Statement 2
printf(“\n file reading\n” );
while((fgets(str,50,fp))!=EOF) // Statement 3
{
printf("\n %s",str);
}
fclose(fp);
 }
```

Output:

```
file witting
hello c programming
file reading
hello c programming
```

In the above example, Statement 1 will CREATE new file data.txt in write mode Statement 2 will open an existing file data.txt in read mode and statement 3 will read all the string until the EOF(end-of-file) reached.

Reading writing integer using putw() and getw() functions

They are similar to getc and putc functions and are used to read and write integer values. These functions would be useful when we deal with only integer data.

1. **The putw() function**

The putw() function is used to write integers to the file.

Syntax of putw() function

putw(int n, fp);

Where,

- n is the integer we want to write in a file
- fp is a file pointer.

2. The getw() function

The getw() function is used to read integer value form the file.

and returns the end-of-file(EOF) if it has reached the end of file.

Syntax of getw() function

int n=getw(fp);

Where, fp is a file pointer.

Example programe on putw() and getw() functions

```
#include<stdio.h>
void main()
{
  FILE *fp;
  int n;
    fp=fopen("data.txt","w"); // Statement 1
    putw(97,fp); // value 97 write in file1
    fclose(fp);
```

```
  fp=fopen("data.txt","r"); // Statement 2
while((n = getw(fp))!=EOF)  //Statement   3
        {                       // The value 97 read from file1
  printf("\nThe value in data file is %d", n);
        }
  fclose(fp);
}
```

Output:

The value in data file is 97

In the above example, Statement 1 will CREATE new file data.txt in write mode Statement 2 will open an existing file data.txt in read mode and statement 3 will read all the integer values upto EOF(end-of-file) reached.

The fprintf() fscanf() Functions in C

- So far we have seen writing of characters, strings and integers in different files.
- This is not enough if we need to write characters, strings and integers in one single file. For that purpose we use fprintf() function.

The fprintf() function

The fprintf() function is used to write mixed type in the file.

Syntax of fprintf() function

```
fprintf(FILE *fp,"format-string",var-list);
```

Example : fprintf(fp,"%d%s%f",roll,name,marks)

The fprintf() function is similar to printf() function except the first argument which is a **file pointer that specifies the file to be written.**

Note: fprintf( ) write the data on the file

The fscanf() function

- The fscanf() function is used to read mixed type form the file.
- The fscanf() function is used to read set of characters from file. It reads a word from the file and **returns EOF at the end of file (\0).**

Syntax of fscanf() function

fscanf(FILE *fp,"format-string",var-list);

Example: fscanf(fp,"%d%s%f",&roll,name,&marks)

The fscanf() function is similar to scanf() function except the first argument **which is a file pointer that specifies the file to be read.**

Note: fscanf( ) read the data from the file

Example of fprintf( ) and fscanf() function

```
#include<stdio.h>

    void main()
    {
        FILE  *fp;
        int roll;
        char name[25];
        float marks;

                fp = fopen("file.txt","w");          //Statement   1

        if(fp == NULL)
        {
```

```
            printf("\nCan't open file or file doesn't exist.");
            exit(0);
        }
            printf("\nEnter Roll : ");
            scanf("%d",&roll);
            printf("\nEnter Name : ");
            scanf("%s",name);

            printf("\nEnter Marks : ");
            scanf("%f",&marks);

  fprintf(fp,"%d%s%f",roll,name,marks); //writing data into file

fclose(fp);
 fp = fopen("file.txt","r");

//statement 2

while(fscanf(fp,"%d%s%f",&roll,name,&marks))  !=EOF)
{
  printf("\n%d\t%s\t%f",roll,name,marks); // Statement 3

            }
fclose(fp);
}
```

In the above example, Statement 1 will open an existing file file.txt in read mode and statement 2 statement 2 will read all the data upto EOF(end-of-file) reached and Statement 3 print data on screen.

Output:

```
Enter Roll : 1
Enter Name : Kumar
Enter Marks : 78.53

Data in file...
1    Kumar    78.5
```

BINARY FILE INPUT AND OUTPUT
(Writing and Reading of Binary Files)

fread() and fwrite() functions are commonly used to read and write binary data to and from the file respectively. Although we can also use them with text mode files.

The fread() fwrite() function in C

The fwrite() function:

The fwrite() function is used to write records (sequence of bytes) to the file. A record may be an array or a structure.

Syntax of fwrite() function:

```
fwrite( ptr, int size, int n, FILE *fp );

Example: fwrite(&e, sizeof(e), 1, fp);
```

The fwrite() function takes four arguments.

- ptr : ptr is the reference of an array or a structure stored in memory.
- size : size is the total number of bytes to be written.
- n : n is number of times a record will be written.
- FILE* fp : fp is a file pointer to the file where data items will be written in binary mode.

The fread() function

The fread() function is used to read bytes/binary form the file.

Syntax of fread() function

```
fread( ptr, int size, int n, FILE *fp );
```

Example: fread(&e,sizeof(e),1,fp)

The fread() function takes four arguments.

- ptr : ptr is the reference of an array or a structure where data will be stored after reading.
- size : size is the total number of bytes to be read from file.
- n : n is number of times a record will be read.
- FILE* fp : fp is a file pointer the file where the records will be read.

Example: Writing and reading structures using binary files, program using (fwrite() and fwrite() function)

```
#include<stdio.h>
struct employee
{
   char name[50];
   int age;
   float salary;
} e;
 struct employee e;
void main()
{
   int n, i, chars;
   FILE *fp;
   fp = fopen("employee.txt", "wb");
   if(fp == NULL)
   {
      printf("Error opening file\n");
      exit(1);
   }


   printf("\n Enter the number of records you want to enter:\n ");
   scanf("%d", &n);

   for(i = 0; i < n; i++)
```

```
    {
        printf("\nEnter details of employee %d \n", i + 1);

        printf("\n Enter Name: ");
        gets(e.name);

        printf("\n Enter Age: ");
        scanf("%d", &e.age);

        printf("\n Enter Salary: ");
        scanf("%f", &e.salary);
         fwrite(&e, sizeof(e), 1, fp);

    }

    fclose(fp);
fp = fopen("employee.txt", "rb");
printf("\n\t Name \t age \t salary \n");
        for(i = 0; i < n; i++)
{
            fread(&e,sizeof(e),1,fp)
         printf("\n\t%s\t%d\t%f",e.name,e.age,e.salary);
}
            fclose(fp);

}
```

Output:

Enter the number of records you want to enter: 2

Enter details of employee 1

Enter Name: Bob

Enter Age: 29

Enter Salary: 34000

Enter details of employee 2

Enter Name: Jake

Enter Age: 34

Enter Salary: 56000

Name	age	salary
Bob	29	34000
Jake	34	56000

FILE HANDLING FUNCTIONS IN C PROGRAMMING LANGUAGE:

C programming language offers many other inbuilt functions for handling files.

File handling functions	Description
fopen ()	fopen () function creates a new file or opens an existing file.
fclose ()	fclose () function closes an opened file.
fgetw () or getw()	getw () function reads an integer from file.
fputw () or putw()	putw () functions writes an integer to file.
fgetc () or getc()	fgetc () function reads a character from file.
fputc () or putc()	fputc () functions write a character to file.
fgets ()	fgets () function reads string from a file, one line at a time.
fputs ()	fputs () function writes string to a file.
fprintf ()	fprintf () function writes formatted data to a file.
fscanf ()	fscanf () function reads formatted data from a file.
fread()	fwrite() Reads structured data/ binary data from the file
fwrite()	fwrite() Writes block of structured data/ records (sequence of bytes) to the file.
feof()	Detects the end of file.
ferror()	Reports error occurred while read/write operations
getch ()	getch () function reads character from keyboard and immediately return to main function.
getche ()	getche () function reads character from keyboard and echoes to o/p screen.
getchar ()	getchar () function reads character from keyboard. .(unformatted)

putchar ()	putchar () function writes a character to screen.(unformatted)
printf ()	printf () function writes formatted data to screen.
scanf ()	scanf () function reads formatted data from keyboard.
remove ()	remove () function deletes a file.
fflush ()	fflush () function flushes a file.
fseek()	Sets the pointer position anywhere in the file
ftell()	Returns the current pointer position.
rewind()	Sets the record pointer at the beginning of the file.

RANDOM ACCESS FILES IN C

RANDOM ACCESS FILES: or (File positioning functions)

Random access to file can be used, which allows to access any record directly at any position in the file.

We can access the data stored in the file in two ways.

1. Sequentially
2. Randomly

1. Sequentially access file

If we want to access the forty fourth record then first forty three records read sequentially to reach forty fourth records.

2. Randomly access file

In random access data can be accessed and processed directly there is no need to read all previous records sequentially.

Main advantages: If we want to access a particular record random access takes less time than the sequential access.

If we want to read access record in the middle of the file and if the file size is too large sequential access is not preferable. In this case, random access to file can be used, which allows to access any record directly at any position in the file.

C supports these functions for random access file.

(Or)file positioning functions in c

1. fseek( ) Function
2. ftell ( ) Function
3. rewind() Function

1. fseek()

The fseek() function to move the file position/pointer to a desired location.

Syntax:

fseek(fp, offset, position);

or

fseek(file pointer, displacement, pointer position);

Where

- file pointer /fp---- It is the pointer which points to the file.

- Offset/displacement ---- It is positive or negative value

  This is the number of bytes which are skipped backward (if negative) or forward(if positive) from the current position.

  This is attached with L because this is a long integer.

  (long integer)

- Pointer position:

This sets the pointer position in the file.

Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CURRENT	1	Current position
SEEK_END	2	End of file

fseek ()	fseek () function moves file pointer position to given location.
SEEK_SET	SEEK_SET moves file pointer position to the beginning of the file.
SEEK_CUR	SEEK_CUR moves file pointer position to given location.
SEEK_END	SEEK_END moves file pointer position to the end of file.
ftell ()	ftell () function gives current position of file pointer.
rewind ()	rewind () function moves file pointer position to the beginning of the file.

Examples:

1) fseek(fp,10L,0)

- 0 means pointer position is on beginning of the file, from this statement pointer position is skipped 10 bytes from the beginning of the file.

2) fseek(fp,5L,1)

- 1 means current position of the pointer position. From this statement pointer position is skipped 5 bytes forward from the current position.

3) fseek(fp,-5L,1)

- From this statement pointer position is skipped 5 bytes backward from the current position.

Operation	Description
fseek(fp, 0, SEEK_SET)	This will take us to the beginning of the file.
fseek(fp, 0, SEEK_END)	This will take us to the end of the file.
fseek(fp, N, 0)	This will take us to (N + 1)th bytes in the file.
fseek(fp, N, 1)	This will take us N bytes forward from the current position in the file.
fseek(fp, -N, 1)	This will take us N bytes backward from the current position in the file.
fseek(fp, -N, 2)	This will take us N bytes backward from the end position in the file.

Example program on fseek() function

```
#include <stdio.h>
void main()
{
   FILE *fp;
  fp = fopen("myfile.txt","w+");
   fputs("\n This is c programiing", fp);
    fseek( fp, 7, 0 );  //fp moves beginning and movies 7 bytes
   fputs("Dennies Ritchie", fp);
   fclose(fp);
}
```

Output

myfile.txt

This is Dennies Ritchie

2. ftell()

This function returns the value of the current pointer position in the file. The value is counted from the beginning of the file.

Syntax:

pos = ftell(fptr);

Where

fptr - is a file pointer.

pos- holds the current position i.e., total bytes read (or written).

Example:

long n;

n=ftell(fp)

n – have cursor current position in the file.

3. **rewind()**

This function is used to move the file pointer to the beginning of the given file. This function is useful when we open file for update.

Syntax:

rewind(fptr);

Where fptr is a file pointer.

Example program on ftell() and rewind() function

```
#include <stdio.h>

voidmain ()

{

  char name [20];

  int age, length;

  FILE *fp;

  fp = fopen ("test.txt","w");

  fprintf (fp, "%s %d", “Hello World”, 5); // write the file

  length = ftell(fp); // Cursor position is now at the end of file

  /* You can use fseek(fp, 0, SEEK_END); also to move the
```

```
    cursor to the end of the file */
  rewind (fp); // It will move cursor position to the beginning of the file
  fscanf (fp, "%s", &name); // read the file content
  fscanf (fp, "%d",&age);
  fclose (fp);
  printf ("\nName: %s \n Age: %d \n",name,age);
  printf ("\nTotal number of characters in file is %d", length);
}
```

Output:

Name: Hello World

Age: 5

Total number of characters in file is 11

Error-Handling functions in File

While dealing with files, It is possible that an error may occur during I/O operations on a file.

Typical error situations include:

- Reading beyond the end of file mark.
- Trying to perform operation on a file, when the file is opened for another type of operation.
- Writing to a file that is opened in the read mode.
- Opening a file with invalid filename.
- Device overflow means no space in the disk for storing data.

If we fail to check such read and write errors, a program may behave abnormally when an error occurs. An unchecked error may result in a premature termination of the program or incorrect output.

Thus, to check the status of the pointer in the file and to detect the error is the file.

C provides the two status-enquiry library functions

1. ferror ()
2. feof ()

that can help us to detect I/O errors in the files.

1. ferror()

- The ferror function reports the status of the file indicated.

  Syntax:

  ferror(fp);

  Where: fp is file pointer

ferror( ) takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise.

Example:

```
if ( ferror ( fp ) ! = 0 )
printf ( "An error has occurred.\n" ) ;
```

It prints the error message, if the reading is not successful.

- We know that whenever a file is opened using fopen( ) function, a file pointer is returned. If the file cannot be opened for some reason, then the function returns a NULL pointer. This facility can be used to test whether a file has been opened or not.

Example

```
if ( fp == NULL )
printf (" File could not be opened.\n");
```

2. **feof()**

The feof function can be used to test for an end of file condition.

Syntax:

feof(fp);

Where: fp is file pointer

It takes a FILE pointer as its only argument and returns a nonzero integer value if all the data from the specified file has been read, and returns zero otherwise. If fp is a pointer to file that has just been opened for reading.

Example:

```
if ( feof ( fp ) )
printf ( "End of data.\n" ) ;
```

It display the message "End of data." on reaching the end of file condition.

3. **clearerr ()**

The function clears the end of file and error indicator for the argument stream

Syntax:

clearerr (fp)

Where: fp is file pointer

Note: Above function are displayed system generated errors.

perror() and strerror()

To print appropriate action depending on the return value we use **perror() and strerror()**

The C programming language provides perror() and strerror() functions which can be used to display the text message associated with errno.

- The perror() function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current errno value.
- The strerror() function, which returns a pointer to the textual representation of the current errno value.

C Program to merge contents of two files into a third file

Difficulty Level : Easy

Last Updated : 07 Jul, 2017

Let the given two files be file1.txt and file2.txt. The following are steps to merge.

1) Open file1.txt and file2.txt in read mode.

2) Open file3.txt in write mode.

3) Run a loop to one by one copy characters of file1.txt to file3.txt.

4) Run a loop to one by one copy characters of file2.txt to file3.txt.

5) Close all files.

//Write a c program merge the content two files in third file

To successfully run the below program file1.txt and fil2.txt must exits in same folder.

```
#include <stdio.h>
#include <stdlib.h>
Void  main()
{
  // Open two files to be merged
  FILE *fp1 = fopen("file1.txt", "r");
  FILE *fp2 = fopen("file2.txt", "r");
  // Open file to store the result
  FILE *fp3 = fopen("file3.txt", "w");
  char c;
  if (fp1 == NULL || fp2 == NULL || fp3 == NULL)
  {
       puts("Could not open files");
       exit(0);
  }

  // Copy contents of first file to file3.txt
  while ((c = fgetc(fp1)) != EOF)
    fputc(c, fp3);

  // Copy contents of second file to file3.txt
  while ((c = fgetc(fp2)) != EOF)
    fputc(c, fp3);

  printf("Merged file1.txt and file2.txt into file3.txt");

  fclose(fp1);
  fclose(fp2);
  fclose(fp3);
  return 0;
}
```

Output:

Merged file1.txt and file2.txt into file3.txt