Unit III

CONTENTS

Lesson 11
Pointers

Lesson 12
Functions

Lesson 13
Function Parameters

Lesson 14
Pass by Value / Address

Lesson 15
Recursion

Lesson 11

Pointers

Structure

47.0. Introduction

47.1. What is a Pointer?

47.2. How to Initialize a Pointer?

47.3. How to Dereference a Pointer?

47.4. Pointer Arithmetic

47.5. Pointer to Pointer

47.6. Summary

47.7. Model Questions

11.0 Introduction

Variable name is a name given to some memory location that will contain some data. The variable name has associated to it data type which dictates the type of operation that can be performed on data it holds, type of data it holds and the size of data. This variable name is used in C programs to access the data. However, internally these variable names are resolved to some memory location because data is within memory and memory locations have unique addresses. It is sometimes required that the memory address of that variable be known. As an example, scanf() function takes memory address of a variable. In C language, the programs can be written efficiently by the use of a Pointer variable that points to memory locations. There are innumerous benefits of using pointers in a C program. Indeed there are certain disadvantages of pointers too; however the advantages supersede disadvantages. It should be noted that one the biggest strengths of C language is support for pointers.

In this lesson, we will discuss what actually a pointer variable is, how to declare it, initialize it and other aspects related to pointers.

11.1 What is a Pointer?

Pointers are an extremely powerful programming tool. They can make some things much easier, help improve program's efficiency, and even allow us to handle unlimited amounts of data. For example, it is also possible to use pointers to dynamically allocate memory, which means that we can write programs that can handle nearly unlimited amounts of data. As a consequence, we need not to know, when we write the program, how much memory we need. This is the biggest flaw of arrays. So what exactly is a pointer?

A pointer is a variable whose value is the memory address of another variable, i.e. it contains a value which is actually address of some memory location that has been labeled with some variable name. That is why it is named as 'Pointer'; the one which points towards something else. Like any other variable, you must declare a pointer variable before you can use it to store any variable's address. The general syntax of a pointer variable declaration is as follows:

```
<data-type> * <pointer-name>;
```

Look at the syntax, it is almost similar to the syntax for declaration of a variable. However, there is an asterisk (*) between data type and variable/pointer name. This asterisk tells our compiler that the variable is not a simple variable but a pointer variable. The declaration of pointer variable only declares it and it needs to be initialized to appropriate value; value which is actually the memory address of some memory location or variable. On 16-bit machines and compilers, the memory address is 16-bit wide. Thus, the width of a pointer variable is always 16-bits on 16-bit platform. However, as obvious from the syntax it also has a data type and different data types have different sizes. However, in context of pointers the data type only specifies the data type of variable whose address will be stored in this pointer. Consider an example below.

```
int * myPtr1;
```

This statement creates a pointer variable named myPtr1 which takes up 2 bytes of memory and can hold memory addresses of only those variables whose data type is integer. Consider another example below.

```
 char * myPtr2;
```

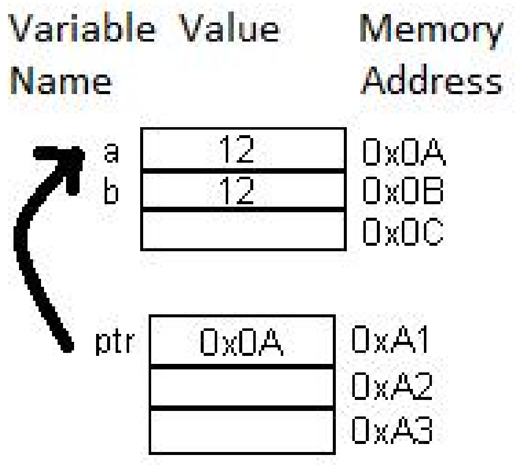
This statement creates a pointer variable named myPtr2 which also takes up 2 bytes of memory and can hold memory addresses of only those variables whose data type is character. Now this is clear that whatever be the data type of a pointer variable, it will always occupy 16 bits of memory. However, its data type will restrict it to store address of only those variables that have same data type.

11.2 How to Initialize a Pointer?

After a pointer variable is declared, 16 bit memory is allocated to the pointer variable and thus before initialization it contains bogus value. Such a pointer which contains bogus value is called wild pointer. To properly initialize a pointer means assigning it memory address of some variable of same data type. Recall when we discussed scanf() function which takes address of a variable as a parameter. In that case we prepended the variable name with ampersand & sign to yield its address. Same applies here. Below is a code snippet that declares an integer pointer which is assigned address of some other integer variable.

```
int a;

int * ptr;

ptr = &a;
/* Now, ptr contains address of variable x*/
```

This relationship can be shown pictorially as follows.



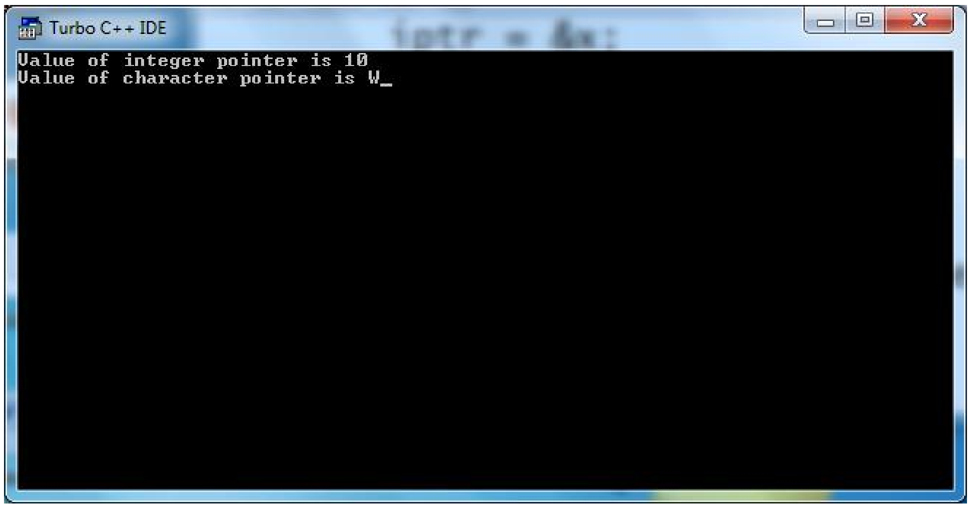
11.3 How to Dereference a Pointer?

Assigning a pointer variable address of a variable of same type is not enough. The real power of pointers lies in accessing the contents of that variable to which it points. This is called de-referencing a pointer variable. In order to deference a pointer variable, an asterisk * should be prepended to the pointer variable whenever we want to dereference it in any expression. It should be noted that differencing does not change the contents of pointer variable rather during execution time when a dereferencing statement is encountered the program fetches the contents at the memory location pointed to by the pointer. This means every time we have to access the contents of variable to which a pointer points, we to dereference it. Program 11.1 shows how a pointer can be used to access contents of variables it points to. The program is followed by the screenshot of the output.

```
#include <stdio.h>

void main()
     {

     int x = 10;
     int * iptr;

     char c = ‘W’;
```

```
    char * cptr;


    /* Initialize the pointers*/

    iptr = &x;
    cptr = &c;

    printf("Value of integer pointer is %d\n", *iptr);

    printf("Value of character pointer is %c", *cptr);

    }
```

Program 11.1: Program shows how to use pointers



```
Value of integer pointer is 10
Value of character pointer is W_
```

It should be noted that pointer dereferencing can not only be used to get values but also to set values. The crux of the story is that the dereferenced pointer can be used in any expression the same way as the variable to which it is pointed can be. The following code snippet explains this.

```
    int x, *ptr;

    ptr = &x;

    *ptr = 10;
    x = 10;
```

The two statements above have same affect. This means we can use any of the two names of some memory location to set a value or get a value.

Furthermore, a pointer can be indirectly assigned the address of some variable via another pointer. This means, if some pointer ptr1 points to a variable, then another pointer variable ptr2 can be made to point to same variable by assigning contents of ptr1 to ptr2 as follows

```
    int x, *ptr1, *ptr2;

    ptr1 = &x;

    ptr2 = ptr1;
```

It should be noted that all the objects should have same data type.

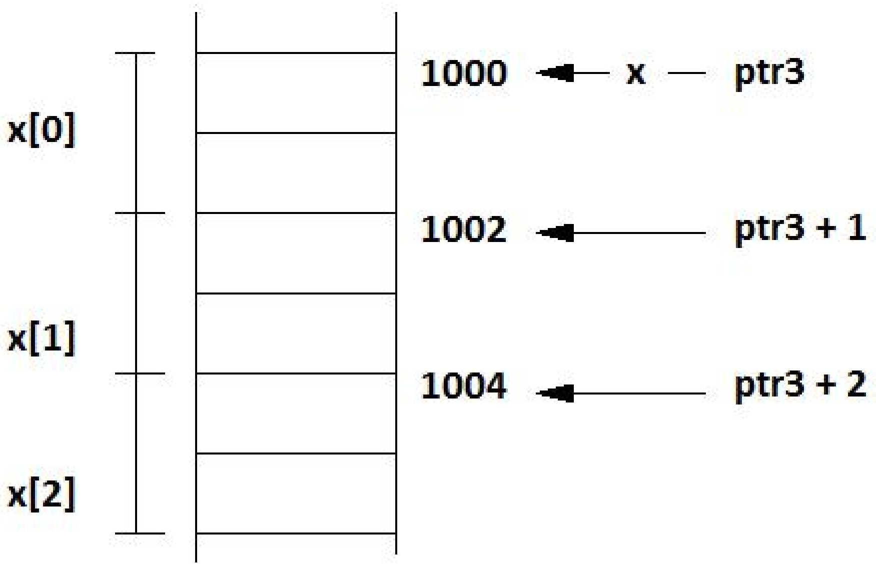
11.4 Pointer Arithmetic

Pointers do not have to point to single variables. They can also point at the cells of an array. In C, arrays have a strong relationship to pointers. For example, we can write

```
    int x[10], *ptr1, *ptr2, *ptr3;

    ptr1 = &x[2];

    ptr2 = &x[0];

    ptr3 = x;
```

This means ptr1 is pointing at the 3rd cell of the array x. Now, as we mentioned before, the variable/element can also be accessed *ptr1. Similarly, ptr2 points to the first element of array and can be accessed either as x[0] or *ptr2. However, what about the last statement? Remember that the name of the array is the base address of the array and all elements of array are placed sequentially starting from this base address. This means ptr3 contains the base address of array x. And, at the base address of an array lies the first element of array. This means, ptr3 is pointing towards the first element of array x which is x[0]. Now, x[0], *ptr2 and *ptr3 all are pointing towards same location.

Nevertheless, we can access all elements of array x using the same ptr pointer. How? Because elements of array are placed sequentially in memory: So if we increment the ptr pointer it will point towards the second element of the array x. But the question is by how much should the pointer value be incremented? The answer is simple. A pointer variable has a data type; so if we increment it by 1 it will skip by the number of bytes equal to the data type it is pointing towards. Hence, in case of integer array x, if we increment ptr3 by 1, it will skip 2 bytes and hence will point to next integer element of array x. Same applies to character, float and other user-defined arrays. This is shown below pictorially.



Pointer Arithmetic not only involves incrementation but also decrementation of pointer variable value. However, other arithmetic operations are prohibited. The program 11.2 shows how to read all elements of an array using a pointer.

```
#include <stdio.h>

void main()
    {

    int i, x[10] = {0,1,2,3,4,5,6,7,8,9};
    int * iptr;

    /* Initialize the pointer*/

    iptr = &x;

    for(i=0; i<10; i++)
        {
        printf("Array Element %d has Value
            %d\n", i, *iptr);
        iptr ++;
        }
    /* Here iptr will be pointing past
    the array*/
    }
```

Program 11.2: Program shows how to use pointer to read elements of an array

11.5 Pointer to Pointer

When a pointer can point to primary data types like int, float, etc., derived data types like arrays, user-defined data types like structures and unions; can it point to other pointers also? The answer is yes. This is called Pointer to Pointer. In pointer to pointer, the first pointer stores the address of second pointer. Logically, there is no limit on the level of indirection that can be achieved in pointer to pointer relationship. The general syntax to declare a pointer to pointer relationship with first level is as follows.

```

<data-type> ** <pointer1>;

<same-data-type> * <pointer2>;
```

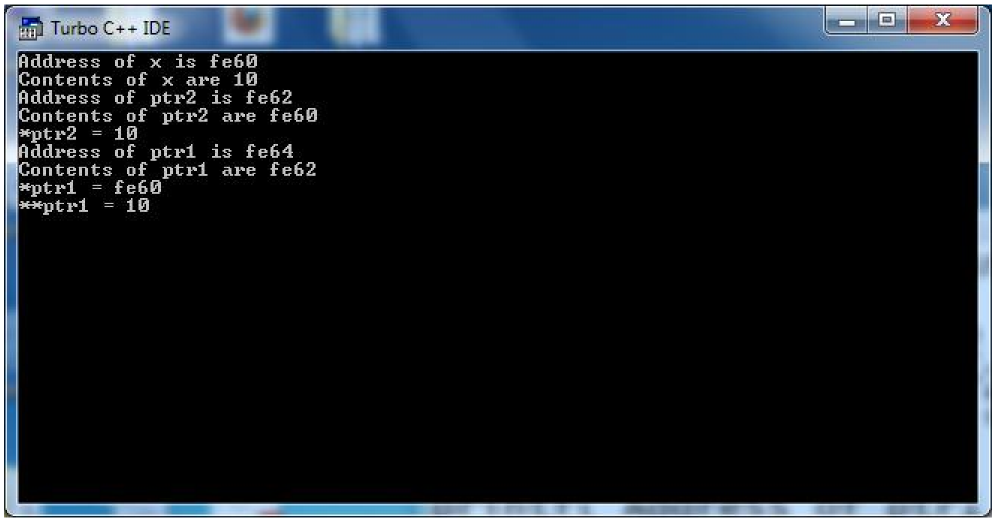
```
/*
Here <pointer1> points to <pointer2>
*/
```

It is clear that the pointer that points to another pointer is declared by having 2 asterisks between data type and pointer name. Actually, the first asterisk tells us that the variable is a pointer and second asterisk tells us that it will point to a pointer variable only of same data type. Now, dereferencing the <pointer1> with a single asterisk will give us contents of <pointer2> which actually is address of some variable. However, if we dereference it with 2 asterisk it will then give us the contents of the variable to which <pointer1> is pointing. It should be noted that to dereference a pointer to pointer variable we should use same number of asterisks as used during declaration, i.e., dereference it to same level as to which it is pointing. The program 11.3 explains the concept of pointer to pointer. This is followed by screenshot of the output.

```
#include <stdio.h>

void main()
     {

     int ** ptr1, *ptr2, x = 10;

     /* Initialize the pointers*/

     ptr2 = &x;

     ptr1 = &ptr2;

     printf(“Address of x is %x\n”, &x);
     printf(“Contents of x are %d\n”, x);

     printf(“Address of ptr2 is %x\n”, &ptr2);
     printf(“Contents of ptr2 are %x\n”, ptr2);
     printf(“*ptr2 = %d\n”, *ptr2);

     printf(“Address of ptr1 is %x\n”, &ptr1);
```

```
    printf(“Contents of ptr1 are %x\n”, ptr1);
    printf(“*ptr1 = %x\n”, *ptr1);
    printf(“**ptr1 = %d\n”, **ptr1);


    }
```

Program 11.2: Program shows how to use pointer to read elements of an array



```
Address of x is fe60
Contents of x are 10
Address of ptr2 is fe62
Contents of ptr2 are fe60
*ptr2 = 10
Address of ptr1 is fe64
Contents of ptr1 are fe62
*ptr1 = fe60
**ptr1 = 10
```

In order to use second level of indirection in pointer to pointer we have to add another asterisk as follows.

```
<data-type> *** <pointer1>;

<same-data-type> ** <pointer2>;

<same-data-type> * <pointer3>;

/*
Here <pointer1> points to <pointer2> which points to
<pointer3>
*/
```

11.6 Summary

- A pointer is a variable whose value is the memory address of another variable, i.e. it contains a value which is actually an address of some memory location that has been labeled with some variable name.
- The asterisk in declaration of pointer tells our compiler that the variable is not a simple variable but a pointer variable.
- The declaration of pointer variable only declares it and it needs to be initialized to appropriate value; value which is actually the memory address of some memory location or variable.
- The width of a pointer variable is always 16-bits on 16-bit platform.
- In context of pointers the data type only specifies the data type of variable whose address will be stored in this pointer.
- A variable name is prepended with ampersand & sign to yield its address.
- The real power of pointers lies in accessing the contents of that variable to which it points. This is called de-referencing a pointer variable.
- In order to deference a pointer variable, an asterisk * should be prepended to the pointer variable whenever we want to dereference it in any expression.
- Pointer dereferencing can not only be used to get values but also to set values.
- Pointers do not have to point to single variables. They can also point at the cells of an array.
- Pointer Arithmetic not only involves incrementation but also decrementation of pointer variable value. However, other arithmetic operations are prohibited.
- A pointer can point to primary data types like int, float, etc., derived data types like arrays, user-defined data types like structures and unions.
- It can also point to other pointers. This is called Pointer to Pointer. In pointer to pointer, the first pointer stores the address of second pointer.
- Logically, there is no limit on the level of indirection that can be achieved in pointer to pointer relationship.

11.7 Model Questions

Q 106. Explain the concept of a pointer.

Q 107. With a help of a program explain declaration, initialization and dereferencing of a pointer variable in C language.

Q 108. Write short notes on

a. Pointer Declaration

b. Pointer Initialization

c. Pointer Arithmetic

d. Pointer Dereferencing

Q 109. Write a program in C language to explain the concept of pointer arithmetic.

Q 110. Write a program to show how a pointer can be sued to set and retrieve elements of an array.

Q 111. Arrays are just like Pointers in C programming language. Justify.

Q 112. Write a program in C language to explain the concept of pointer to pointer.

Q 113. Discuss the need and significance of Flow Control statements.

Q 114. Classy the Flow Control statements in C programming language.

Lesson 12

Functions

Structure

57.0. Introduction

57.1. What are Functions?

57.2. How Functions Work?

57.3. Function Declaration

57.4. Function Definition

57.5. Summary

57.6. Model Questions

12.0 Introduction

In our day to day life we come across the concept in which a larger problem is broken down into smaller ones to solve it effectively and efficiently. Consider this learning material as an example. The book has been divided into Units and each unit has been divided into lessons. This way we were able to write this text effectively (at least to our best) and hope that you will understand it with same effectiveness what we expect. Throughout this text whenever we feel like the topic needs to be read in context of some other topic, or more information on the topic is somewhere else, we simply mention the lesson number. This way it is easy for you to locate and jump to that topic, read it and come back to what you were reading. However, if this text wouldn’t have been compartmentalized, then the reference text needed to be repeated every time it is required. This approach has several problems. First, it will like re-inventing the wheel several times. Second, repetition of text is boring and time consuming. Third, it will occupy more number of pages. Fourth, the context of current text will be lost, and many more. However, if quality and price of a book would be rated as per number of pages it has, then it would be a better approach.

In computer programming, we solve problems using programs. Sometimes, rather most of times, a larger problem can be broken into smaller ones whose solution can be used again and again in different permutations and combinations to solve the actual problem. Let me explain. Consider that you are asked to write a program that will have 3 variables which need to be checked for being prime. How will you do that? Simple, declare 3 variables, assign some value to them and use the logic thrice to do the job. What about 1000 different variables (which are not elements of an array)? Wouldn't it be better to group those statements which actually check whether a number is prime or not, and somehow pass our numbers to them for validity? Yes. This kind of functionality is provided by C language in the form of functions.

In this lesson, we will discuss what functions are, how to declare them, how to define them, how to use them and much more.

12.1 What are functions?

A Function in C language is a block of program code which deals with a particular task. In other words, it a mechanism to divide a program into independent blocks of code which when used together achieves the solution. We have been using functions before this lesson. The most prominent one is the main() function that is present in every C program. This is the point within the program where from execution begins. Also, recall printf(), scanf(), and others are functions defined in stdio.h header file. The printf() function is used to display on screen anything passed to it. If there wouldn't have been the concept of a function, then we would have to write the code required to display something on the screen every time in every program that uses printf(). This means the program code would have been large and the probability of bugs would have been high. These functions have been created by other developers and because they are standard functions needed by everybody, they are packaged into the library of the compiler. They are called Library/Built-in Functions. That is why we mention #include <stdio.h> in all our programs that use these functions. Nevertheless, C allows programmers to create user-defined functions within the main program. The name of such function, the list and type of parameters required, and actual body of these functions are all defined by the programmer himself. Such functions are called User-Defined Functions. Whatever type the function be, they serve two purposes.

1. First, they allow a programmer to specify a piece of code that will stand by itself and does a specific job.
2. Second, they make a block of code reusable since a function can be reused in many different contexts without repeating the actual program code.

12.2 How functions work?

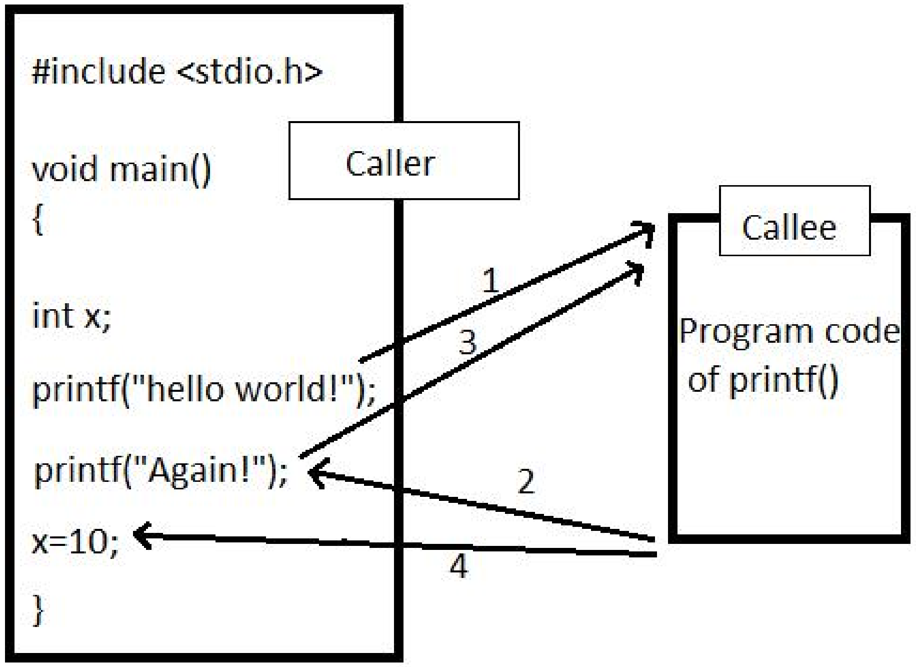
A Function in C language is a block of program code that has a specific name which is used to invoke it whenever and wherever required. In C language there are few steps to create and use a function:

1. Declare the prototype of function,
2. Define the body of function, and
3. Invoke the function.

We will be discussing all these steps one by one but before that let us look into how it works.

After the declaration and definition of a function, the function can be called or invoked from anywhere within the program code. Because every C program has atleast one function, main(), this means the function will be called from within the body of another function. The function that calls another function is called caller while as the function that is called is called callee. The caller simply calls the function by appending function name with a pair of opening and closing braces '()'. Recall how we have been using printf() function. Certain functions may require that caller passes some data to it. These are called Parameters or Arguments.

printf("hello world!"); is a classic example of a function call. The function name is printf and caller (at least main()) calls it by appending braces '()' to it followed by ';'. However, this function also requires certain parameters. In this case a string of characters enclosed in pair of double quotes is passed to it. Now, when the function is called, the control of execution is transferred to the callee (printf() in this case), which executes some code and returns control back to the caller. When it returns, the caller continues its execution from the statement immediately after the call to callee. This is pictorially explained below.



```
#include <stdio.h>

void main()
{

int x;
printf("hello world!");
printf("Again!");
x=10;
}
```

The beauty of the concept lies in the fact that this function can be called multiple times, however there will be single copy of the program code of the callee.

12.3 Function Declaration

A Function needs to be declared before defining and using it as is the case with variables. The definition of a function includes its name, return type, and optional parameter list. The function definition is a one line statement that can exist before main() but after file includes, within main() before use or within any other function before use. This one-liner is called the signature or prototype of a function. Whenever a caller calls a callee, this prototype or signature is used to locate appropriate function and validate the list of parameters passed to it by the caller. The general syntax of function declaration is as follows:

```
<return-data-type> <function-name>
(
<optinal-parameter-list>
);
```

The return-data-type of the function declaration specifies the type of data that will return. The data can be any primary, derived or user-defined data type. Accordingly, the it may be replaced with int, float, etc. or int [], float [], etc. or some user defined type like struct myStruct, etc. However, if a function does not return any value then the return data type should be void. Check previous programs, the main() function is prepended with void keyword. This means that it does not return any value to its caller which is Operating system. Further, if nothing is placed in front of the function name, then the default return type is integer.

The function-name can be any valid identifier name.

The parameter-list is optional and will be discussed in next lesson. However, if the list is not present the default value is void which means nothing is required by caller to pass it to callee. Again, what is the parameter list of main() function.

12.4 Function Definition

A Function after being declared needs to be defined. By defining a function we mean writing the body of the function. The body of the function contains usual C statements. Recall that main() itself is a function and whatever we write inside the curly braces of the main() function constitutes its body. Same is true with other functions. It should be noted that the function can't be defined within the body of another function unlike its declaration. Thus, the function definition goes either above main() function or below it. It is worth mentioning here that if the definition of some function precedes its usage then, we don't need to declare its prototype.

Having that said, the general syntax of a function definition is as follows,

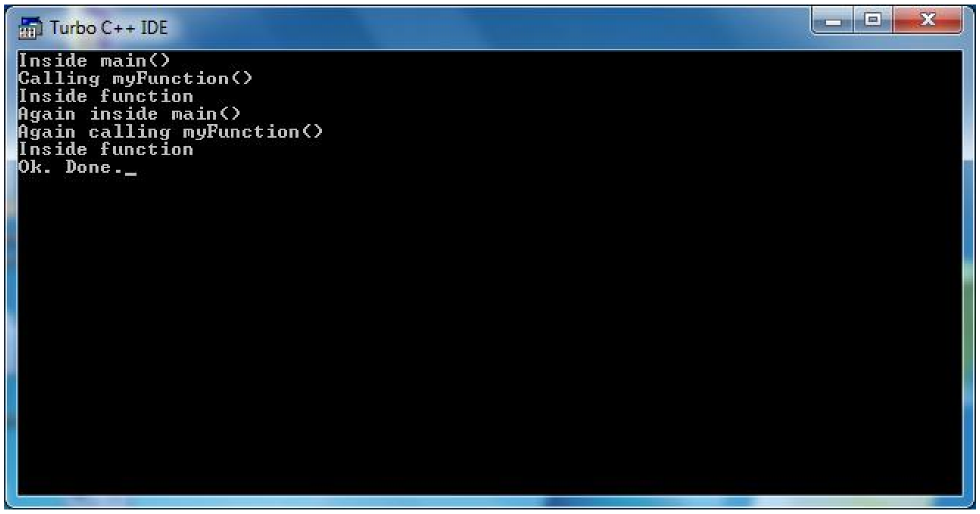
```
<return-data-type> <function-name>
(
<optinal-parameter-list>
)
{
/* Body of function*/
statements;

}
```

It is obvious that the declaration and definition of a function is almost same expect in declaration we don’t specify the body.

Let us create a simple function which will display “Inside function” whenever it is called. Program 12.1 shows how to do it. It is followed by the screen shot of the output.

```

#include <stdio.h>

/*Function Prototype/Signature/Declaration*/
void myFunc();

void main()
    {

    printf(“Inside main()\n”);

    printf(“Calling myFunction()…\n”);

    myFunc();//function call

    printf(“Again inside main()\n”);

    printf(“Again calling myFunction()…\n”);

    myFunc();

    printf(“Ok. Done.”);
    }

/* Function Definition */
void myFunc()
    {

    printf("Inside function\n");

    }
```

Program 12.1: Program that shows function declaration, definition, and calling.



```
Inside main()
Calling myFunction()
Inside function
Again inside main()
Again calling myFunction()
Inside function
Ok. Done._
```

The program can be written without function declaration by moving the definition before main() as follows.

```
#include <stdio.h>

/* Function Definition */
void myFunc()
    {

    printf("Inside function\n");

    }
```

```
void main()
    {

    printf(“Inside main()\n”);

    printf(“Calling myFunction()…\n”);

    myFunc();

    printf(“Again inside main()\n”);

    printf(“Again calling myFunction()…\n”);

    myFunc();

    printf(“Ok. Done.”);
    }
```

12.5 Summary

- A larger problem is broken down into smaller ones to solve it effectively and efficiently.
- A Function in C language is a block of program code which deals with a particular task.
- It a mechanism to divide a program into independent blocks of code which when used together achieves the solution.
- The most prominent function is the main() function that is present in every C program. This is the point within the program where from execution begins.
- printf(), scanf(), and others are functions defined in stdio.h header file.
- The functions that have been created by other developers and are frequently used by others are packaged into the library of the compiler. They are called Library/Built-in Functions.
- C allows programmers to create user-defined functions within the main program.
- In C language there are few steps to create and use a function:
    - Declare the prototype of function,
    - Define the body of function, and

    - Invoke the function.

- The function that calls another function is called caller while as the function that is called is called callee.
- The caller simply calls the function by appending function name with a pair of opening and closing braces '()'.
- Certain functions may require that caller to pass some data to it. These are called parameters.
- When the function is called, the control of execution is transferred to the callee, which executes some code and returns control back to the caller.
- When a callee returns, the caller continues its execution from the statement immediately after the call to callee.
- A callee can be called and executed multiple times, however there will be a single copy of the program code of the callee.
- A Function needs to be declared before defining and using it as is the case with variables.
- The definition of a function includes its name, return type, and optional parameter list.
- The function definition is a one line statement that can exist before main() but after file includes, within main() before use or within any other function before use. This one-liner is called the signature or prototype of a function.
- The return-data-type of the function declaration specifies the type of data that will return. The data can be any primary, derived or user-defined data type.
- The function-name can be any valid identifier name.
- A Function after being declared needs to be defined which means adding program code to its body.
- The body of the function contains usual C statements.
- The function can't be defined within the body of another function unlike its declaration.

12.6 Model Questions

Q 115. Explain the significance of functions in C programming language.

Q 116. Explain the steps to create and use functions in C programming language.

Q 117. Differentiate between library and user-defined functions.

Q 118. Explain the general syntax of a function declaration in C language.

Q 119. What is default return type and parameter of a function declaration?

Q 120. Explain the general syntax of function definition in C programming language.

Q 121. Write a program to explain the declaration, definition and invocation of a function.

Lesson 13

Function Parameters

Structure

68.0. Introduction

68.1. return Statement

68.2. Formal Parameters

68.3. Actual Parameters

68.4. Array as a Parameter

68.5. Structure as a Parameter

68.6. Union as a Parameter

68.7. Summary

68.8. Model Questions

13.0 Introduction

In previous lesson, we discussed the declaration, definition and invocation of a user-defined function. However, we didn't discuss the optional parameters that can be passed to a function. Parameters are simply data (constants, variables, expressions) which can be passed by a caller to the callee for processing. However, if a callee expects parameters to be passed to it, then it should specify this in its declaration (if declaration exists) and definition. The specification includes the number of parameters, data type of each parameter and sequence of these. The parameter-list in general syntax of a function declaration and definition is a comma delimited list of typed-variables. Therefore, the complete syntax of function declaration is as follows:

```
<return-data-type> <function-name>
(

<data-type> <variable-name1>,
<data-type> <variable-name2>,
.
.

<data-type> <variable-name3>,
);
```

This means that if there are 3 parameters to be expected by a callee (say myFunc) which returns nothing and expects first parameter as int, second as float and third as char, then this should be specified as follows.

```
void myFunc( int, float, char);
```

The parameters can also be specified alongwith the variable names as follows

```
void myFunc( int a, float b, char c);
```

The same syntax is valid for function definition as shown below.

```

void myFunc( int a, float b, char c)
    {
    /* Body of function */
    }
```

It should be noted that during definition it is required to specify the variables names alongwith the data type. This means following definition is invalid.

```

/* Invalid function definition */
void myFunc( int, float, char)
    {
    /* Body of function */
    }
```

This is because of the fact that during definition, within the body of the function these parameters will be processed. So, to locate the data, the statements need to know the variable name. However, in case of declaration, we only need to know the number, sequence and type of parameters to create the signature or prototype of a function which will be checked during a function call for validity of function name and parameter list.

Having said that, there are two types of parameters:

1. Formal Parameters, and
2. Actual Parameters

In this lesson, we will discuss formal parameters and actual parameters. In addition, we will discuss how arrays, structures and unions can be passed to and returned from a function.

13.1 return Statement

A function may not only return control to the caller, but some data also. In that case, the function can use keyword `return` followed by a value (constant, variable or expression) to be returned to the caller. The general syntax of return statement is as follows:

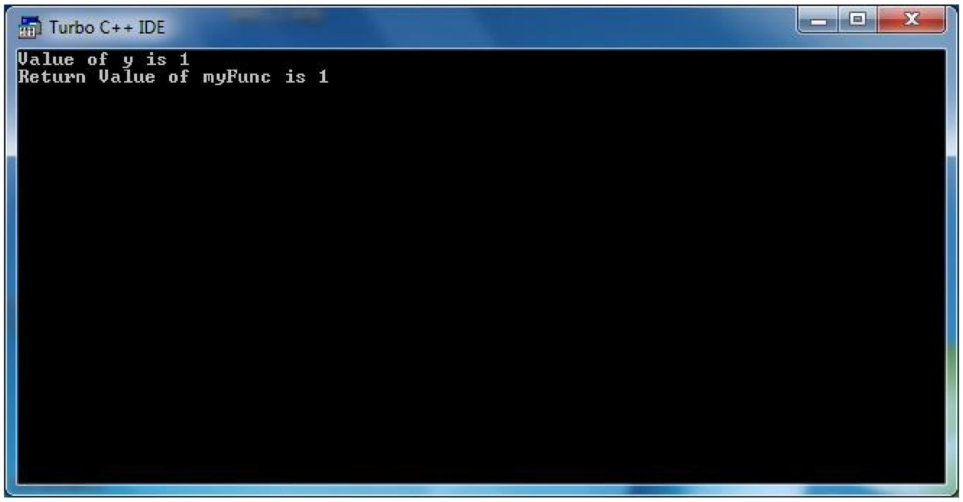
```
    return <returned-value>;
```

The data type of the returned-value should be same as that of return-data-type of the function specified in function declaration and definition. Furthermore, this data when returned to the caller needs to stored somewhere or processed immediately; otherwise it will be lost. To accomplish this task, a function call that returns some value should be invoked on right hand side of some assignment operator so that the returned value will be assigned to some variable of same data type as return-data-type or it can be an actual argument (explained later) to some function. Consider a simple program 13.1 which calls a function that returns some value. It is followed by the screenshot of the output.

```
#include <stdio.h>

int myFunc()
    {
    int x = 1;

    return x;
    /* return 1; will also work*/
    }
```

```
void main()
     {
     int y;

     y = myFunc();
     printf("Value of y is %d\n", y);

     printf("Return Value of myFunc is
     %d\n", myFunc());

     myFunc();
     /* return value is lost*/
     }
```

Program 13.1: Program shows how to capture values returned by a function



```
Value of y is 1
Return Value of myFunc is 1
```

It should be noted that when a function execution encounters a return statement it immediately returns control to the caller even if there are other statements which follow this return statement. This means that return statement can be used to conditionally or un-conditionally exit from the function without executing the rest of the function body. The code snippet below explains this

```
void myFunc()
     {
```

```
    printf(“I will be displayed\n”);

    return;

    printf(“I will not be displayed”);
    }
```

Also, the value that is appended to return statement is optional and can be skipped. In that case, the function returns bogus value. However, if the return-data-type of the function is void, return statement can be used without return-value. As an example, to exit from main() any time one can use return;

13.2 Formal Parameters

Formal Parameters are the list of comma delimited typed-variables which act as place holders within the function for the data which is passed by a caller. In other words, formal parameters are the declaration of variables within the callee which will hold the data that will be passed to it by caller. Within the body of the callee these variables will be used to retrieve and process the data passed.

13.3 Actual Parameters

Actual Parameters are the list of comma delimited constant values, variables or expressions which are passed by a caller to the callee during a function call. It should be noted that there is no relationship between the variable names of Actual parameters and Formal parameters. It is enough to support the argument that all actual parameters can be constants.

Consider a simple function add() which takes two integer parameters and returns their sum. The following is the function declaration of such a function.

```
 int sum ( int, int);
```

The code snippet below is the function definition of sum() function.

```
int sum ( int a, int b)
     {
     int c;
     c = a + b;
     return c;
     }
```

Following is the main program in which it will be called.

```

#include <stdio.h>

int sum ( int, int);

void main()
     {
     int x, y;

     x = 10;

     z = sum(x, 10);
     }
```

There are few observations worth noting:

1. The function declaration needs not to have the list of variables in parameter list; parameter data-types are enough.
2. The actual parameters need not to be all constants or all variables.
3. The formal parameters variable-names need not to be same as that of actual parameter variable-names.

13.3 Array as a Parameter

The formal and actual parameters need not be only primitive data types. Rather the parameters can be any mix of primary, derived and user-defined data types. The derived data type Array can be passed as a parameter to any function.

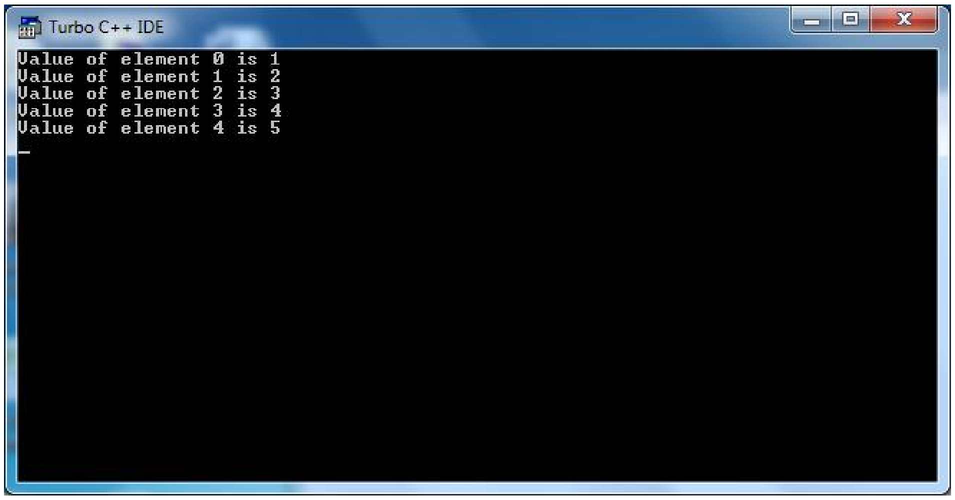
Consider a simple program in which an array is passed as a parameter. Inside this function the elements of array are retrieved and displayed. Program 13.2 lists the program which is followed by screenshot of the output.

```
#include <stdio.h>


void listArray ( int *);

void main()
     {

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

     listArray(myArray);

     }

void listArray(int * arrPtr)
     {
     int i;
     for (i=0; i<5; i++)
       printf(“Value of element %d is %d\n”,
              i, *arrPtr + i);
     }
```

Program 13.2: Program shows how to pass an array as a parameter to a function using a pointer

Recall that the name of an array is actually the base address of that array; where from the elements of the array are placed sequentially. Also, if some pointer of same data type is pointed to the base address of this array, using pointer arithmetic we can traverse through whole array. The program 13.2

exploits this to pass an array to the function by declaring a pointer as a formal parameter of the function.


Turbo C++ IDE

```
Value of element 0 is 1
Value of element 1 is 2
Value of element 2 is 3
Value of element 3 is 4
Value of element 4 is 5
_
```

Another method to pass array to a function is to declare an array as a formal parameter. There are two ways to do this as shown below.

```
void listArray ( int formalArray [5]);
```

or

```
void listArray ( int formalArray []);
```

In both cases, the body of the function should contain following code.

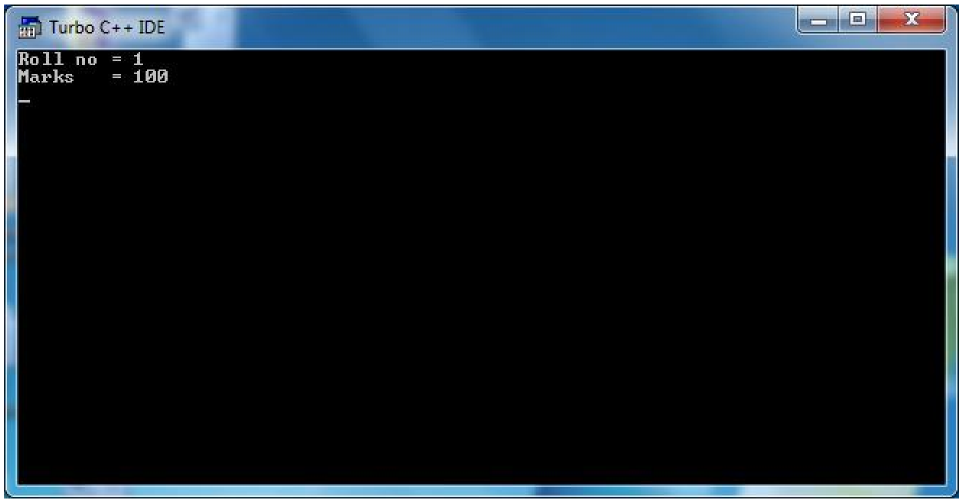
```

    int i;

    for (i=0; i<5; i++)

      printf("Value of element %d is %d\n",
             i, formalArray[i])
```

13.4 Structure as a Parameter

Consider a simple program 13.3 in which a structure variable is passed as a parameter to some function which retrieves the values and displays them. It is followed by the screenshot of the output.

```
#include <stdio.h>

struct myStruct
     {
     int rno;
     int marks;
     };

void listStructure ( struct myStruct);

void main()
     {

     struct myStruct std1;
     std1.rno = 1;
     std1.marks = 100;

     listStructure(std1);

     }

void listStructure ( struct myStruct x)
     {
     printf(“Roll no = %d\n“, x.rno);

     printf(“Marks   = %d\n“, x.marks);
     }
```

Program 13.3: Program shows how to pass a structure variable as a parameter to a function



```
Roll no  = 1
Marks    = 100
_
```

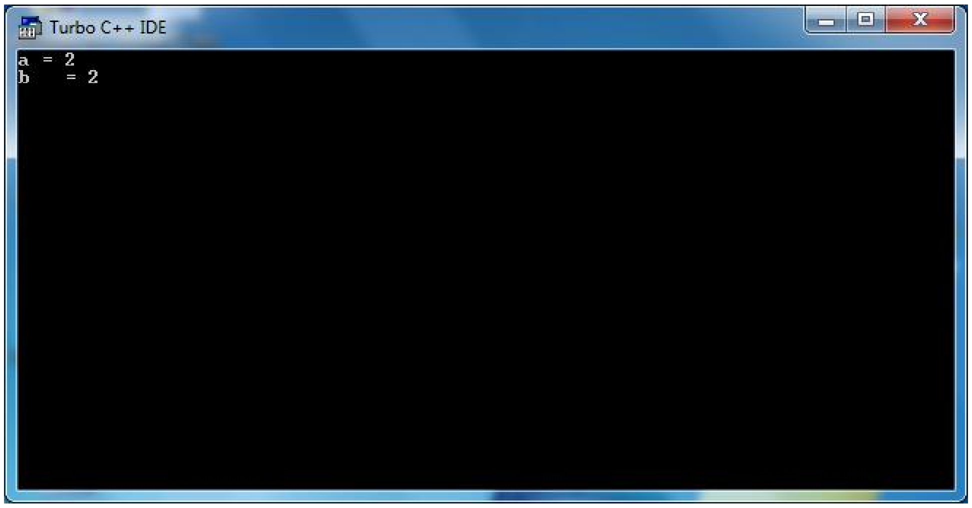
Recall that the structure is a user-defined data type. Thus, if we want the definition to be recognized by both functions; caller and callee (which is required during structure passing), the structure should be defined outside the functions above the function declaration/definition of caller and callee.

13.5 Union as a Parameter

Consider a simple program 13.4 in which a union variable is passed as a parameter to some function which retrieves the value and displays it. It is followed by the screenshot of the output.

```
#include <stdio.h>

union myUnion
    {
    int a;
    int b;
    };

void listUnion (union myUnion);

void main()
    {

    union myUnion var1;
    var1.a = 1;
    var1.b = 2;

    listUnion(var1);

    }

void listUnion (union myUnion x)
    {
    printf(“a = %d\n“, x.a);

    printf(“b   = %d\n“, x.b);
    }
```

Program 13.4: Program shows how to pass a union variable as a parameter to a function



```
a = 2
b   = 2
```

Recall that the union is a user-defined data type. Thus, if we want the definition to be recognized by both functions; caller and callee (which is required during union passing), the union should be defined outside the functions; above the function declaration/definition of both caller and callee.

13.6 Scope of Variables

Scope of a Variable in any programming is the region of the program where a defined variable can have its existence and beyond that the variable cannot be accessed. There are two places where a variable can be declared in C programming language:

1. Inside a function or a block. Such a variable is called Local Variable, and
2. Outside of all functions. Such a variable is called Global Variable.

13.6.1 Local Variables

Variables that are declared inside a function or block are called Local Variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. It should be noted that formal parameters are treated as local variables.

13.6.2 Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of a program and they can be accessed inside any of the functions defined for the program.

Recall the "Storage Classes" from lesson 4.The auto and register storage classes are used with Local Variables within a function. The former is default and later optimizes the access to variable. The static storage class forces the retention of value of local variables even after the function call returns. Finally, extern storage class is used with Global Variables to make them visible across multiple object files.

It should be noted that a function can return the value of both Local and Global variable; however it should never ever return the address of a local variable. This is because of the fact that after the return, that variable will no more exist and the pointer will become wild. Returning address of global variable has no significance.

As a consequence, formal parameters which are local to a function; their address shouldn't be returned. However, when a local array is returned it is always that its address is returned because the name actually points to the base address of array. Hence, returning arrays should be avoided.

13.7 Summary

- Placing one construct inside the body of another construct of same type is called Nesting.
- Nesting if statement means placing if statement inside another if statement within a program written in C language.
- Any variant of if statement can be nested inside another variant of if statement within a program written in C language.
- The else-if ladder in C language is actually nested if-else statement.
- Nesting loops means placing one loop inside the body of another loop.
- Any variant of loop can be nested inside any other variant of loop.
- The outer loop changes only after the inner loop is completely finished.
- The outer loop takes control of the number of complete executions of the inner loop.
- Logically there is no limit on the level of nesting carried. However, if nesting is carried out to too deep a level, code should be properly indented.

13.8 Model Questions

Q 122. Explain nesting in decision making statements in C programming language.

Q 123. Write a program to explain nested-if statement in C language.

Q 124. Explain how else-if ladder is actually nested if-else statement.

Q 125. Explain nested loops in C language.

Q 126. Using a program show how outer loop control the number of times the inner loop is executed to its completion.

Q 127. Why break is not enough in nested loops? What is the remedy?

Q 128. Write a program to create pyramid of stars of height 8 using nested loops.

Lesson 14

Pass by Value/Address

Structure

80.0. Introduction

80.1. Pass-by-value

80.2. Pass-by-address

80.3. Some Examples

80.4. Strings

80.5. Summary

80.6. Model Questions

14.0 Introduction

If a function expects arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function. The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. While calling a function, there are two ways that arguments can be passed to a function:

1. pass by value, and
2. pass by address

In this lesson, we will discuss the pass-by-value and pass-by-address; two mechanisms to pass parameters to a function. In addition, we will discuss the concept of strings in C language.

14.1 Pass-by-value

The pass-by-value (also called call-by-value) is a method of passing parameters to a function (callee) by copying the contents or value of the actual parameters into formal parameters. This means when a parameter is passed via pass-by-value, the local-formal parameter of the callee is the copy of the actual parameter. Therefore, the operation on formal parameter within the body of callee will not affect the value of actual parameter, because formal parameters contain copy and operation is done on that copy.

In addition, when the callee returns, the local-formal parameter is destroyed. As a consequence, if within the body of the callee, the formal parameter value is changed, the change will not persist after function return.

The pass-by-value is default parameter passing mechanism used in C language. The rule is simple; if a function expects an argument via pass-by-value, it has to specify this by declaring its data-type and variable name in the parameter-list. As an example, consider a function (say sum()) which expects 2 integer parameters to be passed via pass-by-value, then the definition of this function should be like this

```
 int sum (int a, int b)
    {
    return a + b;
    }
```

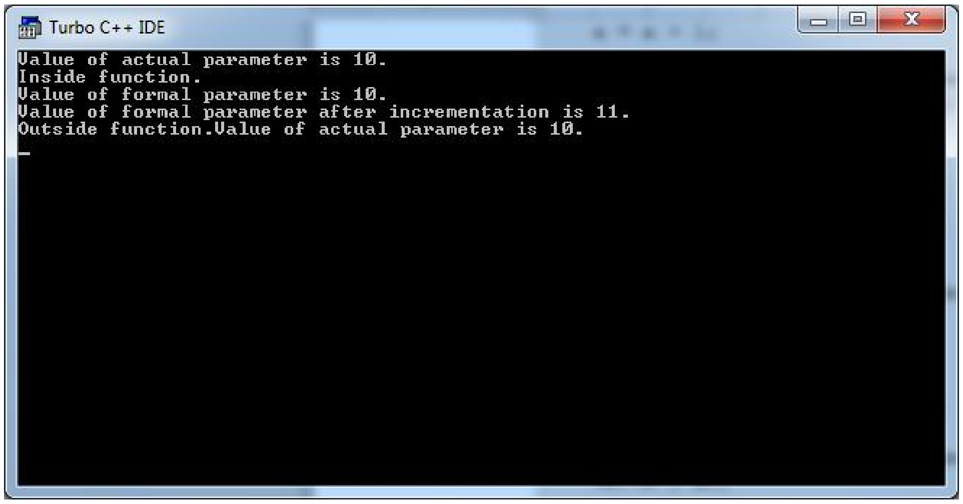
And when this function is called it should be like this

```
z = sum (x, y);
```

The program 14.1 explains this concept. In this program, a function is passed an integer argument via pass-by-value. Within the body of function, the argument is incremented and its value is displayed. After and before function call, the value of actual arguments is displayed. The program is followed by the screenshot of the output.

```

#include <stdio.h>

void myFunc(int a)
    {
    printf("Inside function.\n");

    printf("Value of formal parameter is
    %d.\n", a);

    a = a + 1;

    printf("Value of formal parameter after
    incrementation is %d.\n", a);
    }

void main()
    {

    int x = 10;

    printf("Value of actual parameter is
    %d.\n", x);

    myFunc(x);

    printf("Outside function.");

    printf("Value of actual parameter is
    %d.\n", x);

    }
```

Program 14.1: Program that shows pass-by-value



```
Value of actual parameter is 10.
Inside function.
Value of formal parameter is 10.
Value of formal parameter after incrementation is 11.
Outside function.Value of actual parameter is 10.
_
```

14.2 Pass-by-address

It is sometimes required that the function body does some operation on the actual parameter and not on its copy (i.e. formal parameter). Indeed, the formal parameter has exactly same content as that of actual parameter as the value is copied form actual to formal during function call in pass-by-value. Let us suppose we want to create a function that will swap the value of any two parameters passed to it. This means, if we have two variables ‘a’ and ‘b’, having values 1 and 2 respectively; which are passed to a function (say swap()), then when the callee returns the value of ‘a’ should be 2 while value of ‘b’ should be 1. Can we do this by passing parameters by pass-by-value? Indeed no. The reason is that formal parameters contain copy of actual parameters and even if within the body of the function we swap the value of two formal parameters, the actual parameters are not affected.

This task can be accomplished by using another parameter passing method called pass-by-address (also called call-by- address). In this method, instead of copying the contents of actual parameter into the formal parameter, the address of actual parameter is copied into formal parameter. Therefore, because address is available within the body of the function, we can dereference it to perform any operation directly on the value of actual parameter.

The pass-by-address is not the default parameter passing mechanism used in C language. To allow parameters to be passed via pass-by-address, two things need to be done:

1. The caller should pass address of the actual parameter and not its value.
2. The callee should specify within the corresponding the formal parameter as pointer.

It should be noted that, it is not necessary to pass all parameters to a function via either pass-by-value or pass-by-address. Rather, some parameters can be passed one way and some as other way. The parameter that should be passed via pass-by-value should have a normal corresponding formal variable in specification while as for pass-by-address a pointer formal variable should be used. Furthermore, the actual parameters should be passed as such in pass-by-value while their address should be passed in pass-by-address.

As an example, consider a function (say sum()) which expects 2 integer parameters to be passed via pass-by-address, then the definition of this function should be like this

```
 int sum (int* a, int* b)
     {
     return *a + *b;
     }
```

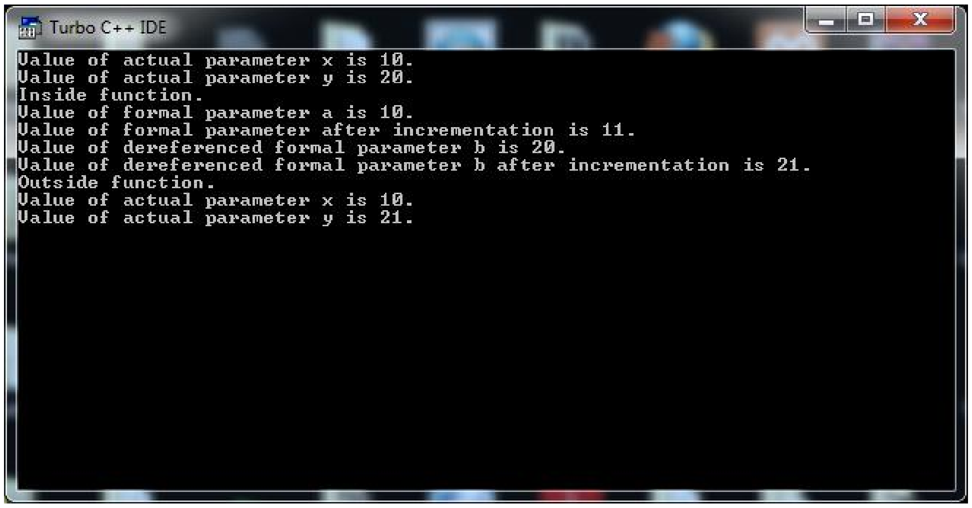
And when this function is called it should be like this

```
z = sum (&x, &y)
```

Let us modify the program 14.1 to pass another variable to it via pass-by-address. Program 14.2 lists the modified program. The program is followed by the screenshot of the output.

```

#include <stdio.h>

void myFunc(int a, int *b)
     {

     printf("Inside function.\n");

     printf("Value of formal parameter a is
     %d.\n", a);

     a = a + 1;

     printf("Value of formal parameter after
     incrementation is %d.\n", a);

     printf("Value of dereferenced formal    parameter b
is %d.\n", *b);

     *b = *b + 1;

     printf("Value of dereferenced formal
     parameter b after incrementation is
     %d.\n", *b);

     }

void main()
     {

     int x = 10, y = 20;

     printf("Value of actual parameter x is
     %d.\n", x);

     printf("Value of actual parameter y is
     %d.\n", y);

     myFunc( x, &y );

     printf("Outside function.\n");

     printf("Value of actual parameter x is
     %d.\n", x);

     printf("Value of actual parameter y is
     %d.\n", y);

     }
```

Program 14.2: Program that shows pass-by-address



```
Value of actual parameter x is 10.
Value of actual parameter y is 20.
Inside function.
Value of formal parameter a is 10.
Value of formal parameter after incrementation is 11.
Value of dereferenced formal parameter b is 20.
Value of dereferenced formal parameter b after incrementation is 21.
Outside function.
Value of actual parameter x is 10.
Value of actual parameter y is 21.
```

There are few things worth mentioning here:

1. Both methods can be used simultaneously while calling a single function.
2. The parameter which was passed via pass-by-address was successfully modified as claimed.

The power of pass-by-address can be exploited to save the memory while passing large structure variables. In case they are passed by value, the copy will be created. This means if it is 100KB large structure we will be creating another 100KB. In contrast, if passed via pass-by-address we only need to create a 16-bit pointer to hold its address.

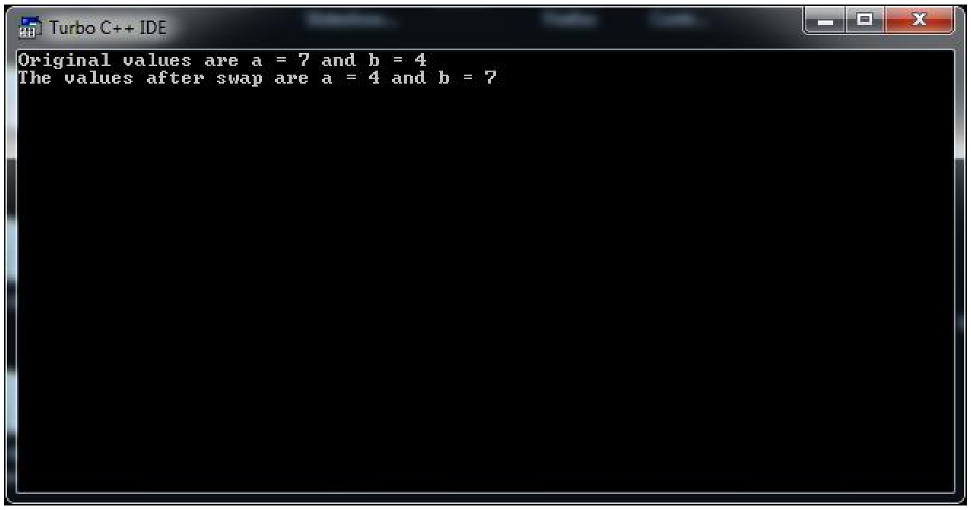
14.3 Some Examples

Program 14.3 shows how to swap the contents of two variables using a function.

```
#include <stdio.h>

void swap(int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

```
void main()
{

  int a = 7, b = 4;

  printf("Original values are a = %d and b =
    %d\n", a, b);

  swap(&a, &b);

  printf("The values after swap are a = %d
    and b = %d\n", a, b);
}
```

Program 14.3: Program that shows how to swap contents of 2 variables



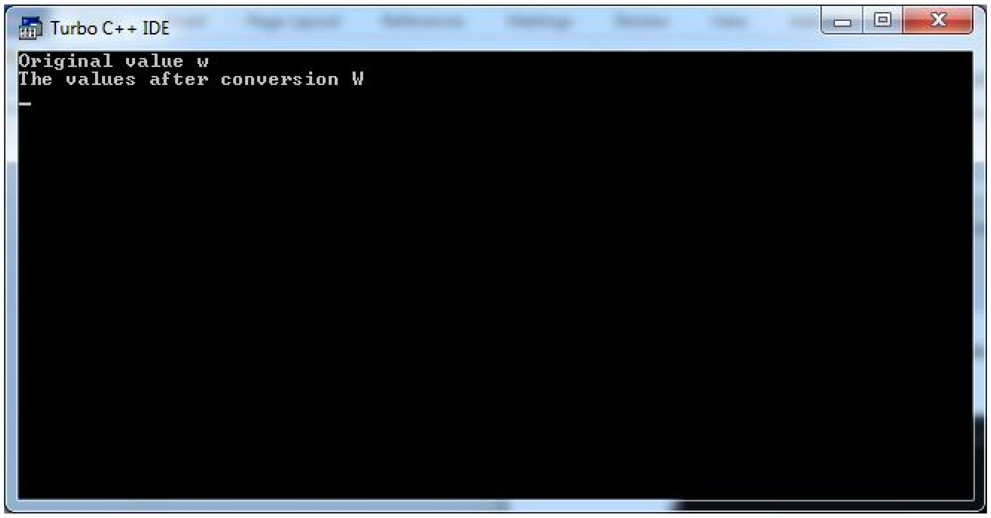
```
Original values are a = 7 and b = 4
The values after swap are a = 4 and b = 7
```

Program 14.4 shows how to invert the case of character within a variable using a function.

```
#include <stdio.h>

void convert(char *c)
{
 if (*c < 'a' )
      *c += 32;
 else
```

```
    *c -= 32;
}

void main()
{

  char a = 'w';

  printf("Original value %c\n", a);

  convert(&a);

  printf("The values after conversion %c\n", a);
}
```

Program 14.4: Program that shows how to invert case of a character variable



```
Original value w
The values after conversion W
_
```

14.4 Strings

The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'. Thus, a null-terminated string contains the characters that comprise the string followed by a null. The following declaration and initialization create a string consisting of the word "Hello".

```
char str[6] = {‘H’,‘E’,‘L’,‘L’,‘O’,‘\0’};
```

To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello".

The character array can also be initialized as follows:

```
char str[] = “HELLO”;
```

In this case we do not need the null character at the end of a string constant. The C compiler automatically places the ’\0’ at the end of the string when it initializes the array.

This string can be printed as follows

```
printf(“%s”, str);
```

The string can be populated from keyboard as follows

```
scanf(“%s”, str);
```

C supports a wide range of functions that manipulate null-terminated strings. A short description of these built-in functions is as follows

Built-in Function	Purpose
strcpy(s1, s2);	Copies string s2 into string s1
strcat(s1, s2);	Concatenates string s2 onto the end of string s1
strlen(s1);	Returns the length of string s1
strcmp(s1, s2);	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2

strchr(s1, ch);	Returns a pointer to the first occurrence of character ch in string s1
strstr(s1, s2);	Returns a pointer to the first occurrence of string s2 in string s1.

It should be noted that these functions are not part of stdio.h library. Rather, to use these built-in function another library called string.h should be included.

14.5 Summary

- While calling a function, there are two ways that arguments can be passed to a function:
  - pass by value, and
  - pass by address
- The pass-by-value (also called call-by-value) is a method of passing parameters to a function (callee) by copying the contents or value of the actual parameters into formal parameters.
- The pass-by-value is default parameter passing mechanism used in C language.
- If a function expects an argument via pass-by-value, it has to specify this by declaring its data-type and variable name in the parameter-list.
- In pass-by-value, the operation on formal parameter within the body of callee will not affect the value of actual parameter, because formal parameters contain copy and operation is done on that copy.
- In pass-by-address, instead of copying the contents of actual parameter into the formal parameter, the address of actual parameter is copied into formal parameter.
- To allow parameters to be passed via pass-by-address, two things need to be done:
  - The caller should pass address of the actual parameter and not its value.
  - The callee should specify within the corresponding the formal parameter as pointer.
- Both methods can be used simultaneously while calling a single function.

- The string in C programming language is actually a one-dimensional array of characters which is terminated by a null character '\0'.

14.6 Model Questions

Q 129. Explain the difference between pass-by-value and pass-by-address.

Q 130. Explain with help of an example the significance of pass-by-address.

Q 131. Write a program to explain pass-by-value in C language.

Q 132. Write a program to explain pass-by-address in C language.

Q 133. Write a program to swap the contents of actual parameters passed to a function.

Q 134. Write a program to create a function that will invert the case of any character variable passed to it.

Q 135. Explain the concept of strings in C programming language.

Lesson 15

Recursion

Structure

93.0. Introduction

93.1. What is Recursion?

93.2. Recursion in C language

93.3. Steps of Recursion

93.4. Summary

93.5. Model Questions

15.0 Introduction

Functions are used to break a complex and larger problem into simple and smaller problems. These functions individually achieve the solution of these problems and in aggregate solve the actual problem efficiently. Most of the times, a function will be using the services of another function. All programs that we have coded are examples of this scenario. In our programs, `main()` function has been using the services of `printf()`, `scanf()` and other user-defined functions by calling them. The `printf()` function is a rich function which can display wide variety of data types. If we were asked to write a function like `printf()`, we would break down this task into other functions; each specialized in displaying a specific data type. These sub-functions will be called accordingly within our `printf()` as per the type of data passed.

In general, in computer programs it is common that a function calls another function. Even the operating system calls one function of the program to execute it. In case of C programs, that function is `main()`. Sometimes, it is required that a function calls itself. As an example, consider a function that calculates the factorial of a number. This can be done iteratively as follows.

```
    /* calculates factorial of number */
    fact = 1;
    for ( i=number; i > 1; i--)
         {
              fact = fact * i;
         }
```

The logic behind calculating factorial of an integer number is to multiply it by all positive integer numbers less than the number. This means, 5! = 5 x 4 x 3 x 2 x1. However, in other words, the number is multiplied by the factorial of the number one less than the actual number, which means 5! = 5 x 4! Hence, we can create a simple function which takes a number and calls itself recursively to calculate the factorial.

In this lesson, we will discuss recursion and will look at some examples that explain the concept.

15.1 What is Recursion?

Recursion is the process of repeating items in a self-similar way. Same applies in programming languages as well. In C language, a function call is said to be recursive if it calls itself. This means, if within the body of a function, there exists a statement, that is actually a function call to itself. As an example, consider the following code snippet in which myFunc() calls itself.

```
    void myFunc()
    {
    /* Other Statements*/

    myFunc();
    /* Recursive Function Call*/

    /* Other Statements*/
    }
```

The type of recursive call as shown above is called Direct Recursion. However, there can be a case that a function calls another function that in turn calls the former. The following code snippet explains this.

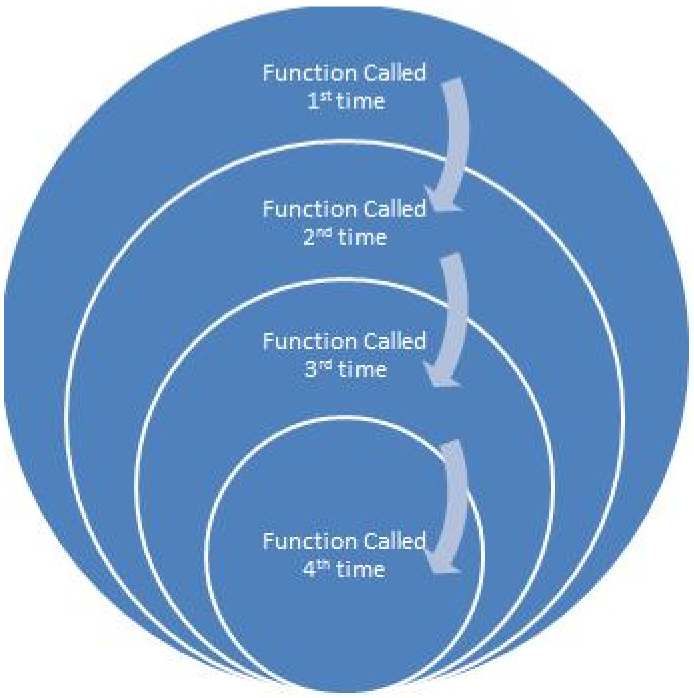
```
void myFunc1()
{

myFunc2();
/* Recursive Function Call*/

}

void myFunc2()
{

myFunc1();
/* Recursive Function Call*/

}
```

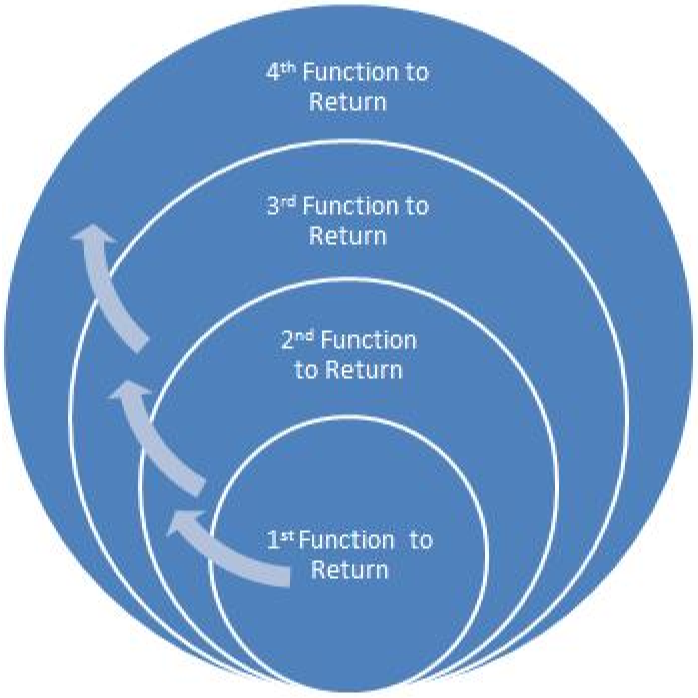
The type of recursive call as shown above is called Indirect Recursion.

15.2 Recursion in C language

Recursion in C language is one of the greatest strengths of the language. In C language, whenever a function is called the arguments to the function are passed via stack. Also, to save the context of caller other necessary data like return address, state of processor, and so on are stored on stack. After the callee returns, the stack is cleaned because it is no more needed. What is Stack? In simple words, it is that part of main memory (RAM) which is set aside for this purpose. This means when a function is called some memory is allocated and when it returns that memory is set free. In case of recursive calls, when the function calls itself which in-turn will call itself again and so on, the stack will be continuously allocated. If this recursive calling is not stopped somewhere, the system will run out of stack for this process as memory is finite. Such a situation is commonly called Stack Over-flow.

The point where this recursive calling is stopped will be the last recursive function to execute but first recursive function to return. Similarly, the recursive function to execute before last one will be second recursive function to return. This concept is explained pictorially below.





15.3 Steps of Recursion

In C language, to have a logically correct and reliable direct or indirect recursive call, certain steps should be kept in view. These include:

1. When a function recursive calls itself, the callee contains the same number and type of variables as the caller.
2. The contents of a variable (say ‘a’) within the caller are different as compared to the same variable in callee.
3. The contents of a variable (say ‘a’) within the caller are retained as long as the caller doesn’t return.
4. There should be a stopping statement that will be executed based on some condition. This is avoid an infinite recursion.

15.4 Some Examples

Consider the program 15.1 which recursively calculates the factorial of a number.

```
#include <stdio.h>

long factorial(long num)
   {

   long n;

   /* Stopping Condition*/
   if ( num == 1 )
       return 1;

   n = factorial(num – 1);

   return num * n;
```

```
    }

    void main()
        {

        long number, fact;

        printf(“Enter a number? ”);
        scanf(“%ld”, &number);

        fact = factorial (number);

        printf(“\nThe factorial of the
        number %ld is %ld”, number, fact);

        }
```

Program 15.1: Program to calculate the factorial of a number recursively**.**

Consider the program 15.2 which recursively generates the Fibonacci series upto n terms.

```
#include <stdio.h>

int Fibonacci (int n)
    {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return
       (Fibonacci(n-1) + Fibonacci(n-2));

    }

void main()
    {
```

```
    int n, i;

    printf("Please enter no. of terms? ");
    scanf("%d", &n);
    printf("\nFibonacci Series\n");

    for (i=0; i<n; i++)
        printf("%d\t", Fibonacci(i));

    }
```

Program 15.2: Program to generate the Fibonacci seried upto n terms**.**

15.5 Summary

- In computer programs it is common that a function calls another function. Even the operating system calls one function of the program to execute it. In case of C programs, that function is main().
- Recursion is the process of repeating items in a self-similar way.
- In C language, a function call is said to be recursive if it calls itself. This means, if within the body of a function there exists a statement that is actually a function call to itself.
- Recursion can be direct or indirect.
- In case of recursive calls, when the function calls itself which in-turn will call itself again and so on, the stack will be continuously allocated.
- If recursive calling is not stopped somewhere, the system will run out of stack for a process as memory is finite. Such a situation is commonly called Stack Over-flow.

15.6 Model Questions

Q 136. Explain the concept of Recursion.

Q 137. Explain with the help of a program the difference between Direct & Indirect Recursion.

Q 138. Explain the concept of Stack-Overflow in infinite recursion.

Q 139. Write a program to calculate the factorial of a number using recursive function calls.

Q 140. Write a program to generate Fibonacci series up to n terms using recursive function calls.

Unit IV

CONTENTS

Lesson 16

Introduction to Header Files

Lesson 17

C Pre-Processor

Lesson 18

Macros

Lesson 19

File Processing in C

Lesson 16

Introduction to Header Files

Structure

107.0. Introduction

107.1. Libraries and Header Files

107.2. Standard C Library & Header Files

107.3. Adding Functions to Library

107.4. Creating User-defined Library

107.5. Summary

107.6. Model Questions

16.0 Introduction

By now we have discussed almost all keywords available in C programming language. These keywords are the pre-defined tokens of the C programming language. Also, we have discussed how to create and use user-defined tokens like variables, functions, and so on. The C programming language tokens provide basic barebones functionality to solve a problem. However, in general more is needed. As an example, in all our programs we have been reading input required by a program from keyboard and displaying the output on the screen. C programming language at its core does not provide any functionality to do so. However, we have been using scanf() and printf() functions to do the task. These functions are not actually part of C programming language specification but are provided as a plugin that can be used in any C program on demand. This is the beauty of C programming language to create, distribute and use re-usable plugins. It should be noted that these plugins do not add to the keyword set of the C language. In fact, these plugins are available as a set of library functions that are included in a C program with the help of header files.

In this lesson, we will discuss the concept of library and header files in C language. In addition, we will discuss the standard C library, creation of user-defined library and adding functions to exiting libraries.

16.1 Library and Header Files

The core of C language is small and simple. However, additional and required functionality in C language is provided in the form of libraries of ready-made functions. A Library in C language is a group of functions in the form of compiled code that can be used by other programs. These libraries are merged with a C program during linking so that the functions embedded within them are available for use.

Why are libraries precompiled? First, since libraries rarely change, they do not need to be recompiled. It would be a waste of time to compile them every time when we write a program that uses them. Second, because precompiled objects are in machine language, it prevents people from accessing or changing the source code, which is important to businesses or people who don't want to make their source code available for intellectual property reasons.

Is there any limit on the number of libraries included in a program? Although there is logically no limit to the number of libraries which can be included in a program, there may be a practical limit: namely memory, since every library adds to the size of program. Libraries also add to the time it takes to compile a program. In addition, a particular operating system might require its own special library for certain operations such as using a mouse or for opening windows in a GUI environment.

How does a programmer specify which library to be included? Each library comes with a number of associated Header Files that contains declarations for everything the library wishes to expose to programmers. Bear in mind that header files typically only contain declarations. They do not define how something is implemented. The definitions are within the libraries. However, header files may define macros, data types and external data to be used in conjunction with the libraries.

Why are these libraries not just included automatically? Because it would be wastage of time and space to add lots of code from every library when only a few functions from few libraries are required. When some library functions are used in a program, the appropriate library code is included by the compiler with the help of a header file, making the resulting code small.

How to include a header file? To include a header file in a C program so that the functions within the library to which the header file is pointing can be used within the program, the programmers at the beginning of the program specify the header file as follows

```
#include <header-filename>
```

We have been using this statement in our previous programs to include the functionality of standard input/output into our program by including stdio.h header file. It should be noted that, once a header file has been included, all the declarations within it are now part of the program. Thus, re-defining the names of functions or macros declared in header files can cause error.

On different platforms, these libraries have different names and file extensions. On Windows platform, the most common extensions include ‘.lib’ and ‘.dll’. However, header file names and extensions are consistent across all platforms. The header files in C language have extension ‘.h’. The Figure 16.1 shows the 2 phases of program compilation along with the necessary information required.

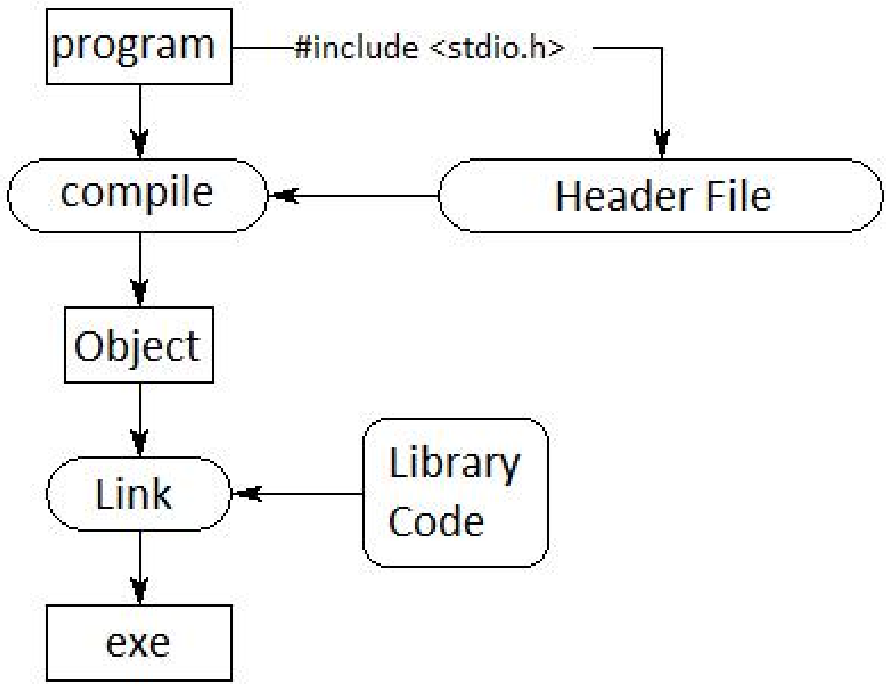


Figure 16.1: The figure shows 2 phases of a program compilation that uses header files and libraries.

In Compile phase, the contents of header file are included in the program to create an object file. Now, during Link phase, based on the declarations within the header file included and the invocations of functions within the program that are declared in these header files, the appropriate library code is embedded with the final executable code.

16.2 Standard C Library & Header Files

Over time, programmers shared ideas and implementations to provide pluggable functionality to C language. These ideas became common, and were eventually incorporated into the definition of the standardized C programming language. These are now called the Standard C Library.

The Standard C Library is the standard library for the C programming language, as specified in the ANSI C standard. Since ANSI C was adopted by the International Organization for Standardization, the C standard library is also called the ISO C library. The C standard library provides macros, type definitions, and functions for tasks like string handling, mathematical computations, input/output processing, memory allocation and several other operating system services. The ANSI C standard library consists of 24 C header files which can be included into a program.

The standardization of C library has gone through various phases. During the 1970s, the C programming language became increasingly popular, with many universities and organizations beginning to create their own variations of the language for their own projects. By the start of the 1980s compatibility problems between the various C implementations became apparent. In 1983, the American National Standards Institute (ANSI) formed a committee to establish a standard specification of C known as "ANSI C". This work culminated in the creation of the so-called C89 standard in 1989. Part of the resulting standard was a set of software libraries called the ANSI C Standard Library. Later revisions of the C standard have added several new required header files to the library. Support for these new extensions varies between implementations. The headers <iso646.h>, <wchar.h>, and <wctype.h> were added with Normative Addendum 1 (NA1), an addition to the C Standard ratified in 1995. The headers <complex.h>, <fenv.h>, <inttypes.h>, <stdbool.h>, <stdint.h>, and <tgmath.h> were added with C99, a revision to the C Standard published in 1999.

Table 16.1 enumerates these header files along with a short description of the functionalities it provides.

Header File	**Description**
<assert.h>	Contains the assert macro, used to assist with detecting logical errors and other types of bug in debugging versions of a program.

<complex.h>	A set of functions for manipulating complex numbers. (New with **C99**)
<ctype.h>	This header file contains functions used to classify characters by their types or to convert between upper and lower case in a way that is independent of the used character set (typically ASCII or one of its extensions, although implementations utilizing EBCDIC are also known).
<errno.h>	For testing error codes reported by library functions.
<fenv.h>	For controlling floating-point environment. (New with **C99**)
<float.h>	Contains defined constants specifying the implementation-specific properties of the floating-point library, such as the minimum difference between two different floating-point numbers (_EPSILON), the maximum number of digits of accuracy (_DIG) and the range of numbers which can be represented (_MIN, _MAX).
<inttypes.h>	For precise conversion between integer types. (New with **C99**)
<iso646.h>	For programming in ISO 646 variant character sets. (New with **NA1**)

<limits.h>	Contains defined constants specifying the implementation-specific properties of the integer types, such as the range of numbers which can be represented (_MIN, _MAX).
<locale.h>	For setlocale() and related constants. This is used to choose an appropriate locale.
<math.h>	For computing common mathematical functions
<setjmp.h>	setjmp and longjmp, which are used for non-local exits
<signal.h>	For controlling various exceptional conditions
<stdarg.h>	For accessing a varying number of arguments passed to functions.
<stdbool.h>	For a boolean data type. (New with **C99**)
<stdint.h>	For defining various integer types. (New with **C99**)
<stddef.h>	For defining several useful types and macros.
<stdio.h>	Provides the core input and output capabilities of the C language. This file includes the venerable printf function.

<stdlib.h>	For performing a variety of operations, including conversion, pseudo-random numbers, memory allocation, process control, environment, signalling, searching, and sorting.
<string.h>	For manipulating several kinds of strings.
<tgmath.h>	For type-generic mathematical functions. (New with **C99**)
<time.h>	For converting between various time and date formats.
<wchar.h>	For manipulating wide streams and several kinds of strings using wide characters - key to supporting a range of languages. (New with **NA1**)
<wctype.h>	For classifying wide characters. (New with **NA1**)

Table 16.1: The header files for C Standard Library

It is worth mentioning here that TC3.0 is an old compiler and does not fully complaint to ANSI C Standard Library.

16.3 Adding Functions to Library

We can add new functions to any C library (Standard and Non-Standard). If there are certain functions that we use frequently in our programs, it makes sense to put them in some library and include that library in the program. One may argue that we can simply copy/paste the code of these functions in our programs. Indeed, yes. However, this way every time we compile our program, the function code also needs to be compiled. By putting function in library, the code is in machine form and hence is not compiled every time the program is compiled. Also, whenever we want to make changes to our function it needs to made at one place (library) and will get reflected in every program on re-compilation.

Different compilers provide different mechanisms to do this. In TC3.0, a utility called Turbo Librarian (tlib.exe) within the BIN folder is used to add, modify or remove a function from a library.

Let us assume that we want to add the factorial() function created in last lesson to the math standard library of TC3.0. To do so, we have to follow following steps:

1. Create a C program that contains the function definition of factorial(). Let us name this file as fact.c. The code will be something like this

```
long factorial(long num)
    {

    long n;

    /* Stopping Condition*/
    if ( num == 1 )
        return 1;

    n = factorial(num – 1);

    return num * n;

    }
```

Do not add main() function to this program and put all files in BIN folder.

2. Compile it on command prompt using tcc.exe as shown below and in figure 16.2. This will create an object file named fact.obj.

```
    tcc –c fact.c
```

3. Create a Header File (say fact.h) that will declare the factorial() function as follows

```
     long factorial(long);
```

4. Now, to add the function to math library, invoke tlib as shown below and in figure 16.2.

```
     tlib math.lib + fact.obj
```

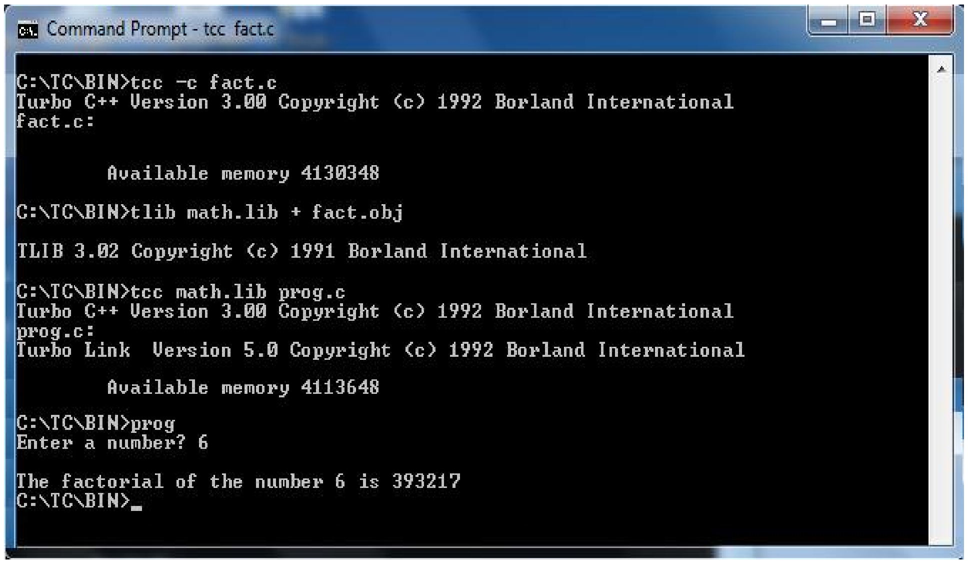
5. Now, create a simple program that will use this function. The program should include the fact.h header file. Consider the program below.

```

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

void main()
    {

    long number, fact;

    printf("Enter a number? ");
    scanf("%ld", &number);

    fact = factorial (number);

    printf("\nThe factorial of the number %ld is %ld",
number, fact);

    }

```

6. Compile it on command prompt as shown below and in figure 16.2

```
    tcc math.lib prog.c
```

7. Run it.



```
C:\TC\BIN>tcc -c fact.c
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
fact.c:

        Available memory 4130348

C:\TC\BIN>tlib math.lib + fact.obj

TLIB 3.02 Copyright (c) 1991 Borland International

C:\TC\BIN>tcc math.lib prog.c
Turbo C++ Version 3.00 Copyright (c) 1992 Borland International
prog.c:
Turbo Link  Version 5.0 Copyright (c) 1992 Borland International

        Available memory 4113648

C:\TC\BIN>prog
Enter a number? 6

The factorial of the number 6 is 393217
C:\TC\BIN>_
```

Figure 16.2: Steps to add a function to Standard Library

16.4 Creating User-Defined Library

We can also create our own library of functions. In fact, creating own library is better than modifying the Standard Library. The steps to create a library, varies from compiler to compiler.

Let us assume that we want to create a library called mymath.lib which will contain a factorial() function created in last lesson To do so, we have to follow following steps:

1. Create a C program that contains the function definition of factorial(). Let us name this file as mymath.c. The code will be something like this

```
long factorial(long num)
    {

    long n;

    /* Stopping Condition*/
    if ( num == 1 )
        return 1;

    n = factorial(num – 1);
    return num * n;

    }
```

Do not add main() function to this program and put all files in BIN folder.

2. Create a Header File (say mymath.h) that will declare the factorial() function as follows

```
     long factorial(long);
```

3. Compile it on command prompt using tcc.exe as shown below. This will create an object file named mymath.obj.

```
    tcc –c mymath.c
```

4. Create library file named mymath.lib from the object file by executing tlib as follows

```
    tlib mymath.lib + mymath.obj
```

5. Now, create a simple program that will use this function. The program should include the mymath.h header file. Consider the program below.

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

void main()
    {

    long number, fact;

    printf("Enter a number? ");
    scanf("%ld", &number);

    fact = factorial (number);

    printf("\nThe factorial of the number %ld is %ld",
number, fact);
```

```
    }
```

6. Compile it on command prompt as was the last program compiled. However, this time replace math.lib by mymath.lib.

16.5 Summary

- The C programming language tokens provide basic barebones functionality to solve a problem.
- C programming language at its core does not provide any functionality to read input required by a program from keyboard and display the output on the screen.
- Additional and required functionality in C language is provided in the form of libraries of ready-made functions.
- A Library in C language is a group of functions in the form of compiled code that can be used by other programs.
- Each library comes with a number of associated Header Files that contains declarations for everything the library wishes to expose to programmers.
- The Standard C Library is the standard library for the C programming language, as specified in the ANSI C standard.
- We can add functions to this library or can create our own library.

16.6 Model Questions

Q 141. C is small and simple yet powerful. Justify this statement.

Q 142. Explain the extensibility of a C language in terms of its ability to import functionality lying within libraries.

Q 143. What is a Standard C library? Discuss its brief history.

Q 144. With the help of a program show how a function can be added to a library in TC3.0.

Q 145. With the help of a program show how a library can be created in TC3.0.

Lesson 17

C Pre-Processor

Structure

122.0. Introduction

122.1. What is a C Pre-Processor?

122.2. Syntax of a C Pre-Processor

122.3. Advantages of a C Pre-Processor

122.4. File Inclusion Directive

122.5. Conditional Compilation Directives

122.6. Summary

122.7. Model Questions

17.0 Introduction

Pre-Processor as the name suggests is a process in which something goes through some preliminary processing before the actual processing. The immediate question that may be popping up in your mind will be "why not combine the two processing into one?" Consider two students; one who after his 10th standard is directly allowed admission into PGDCA course and another who has to finish his graduation to get into the same course. Who will understand this content more appropriately? Well this is not the answer to the question. The answer is shall we put all the content of 3 years long graduation course into the PGDCA course and hope that every 10th standard student will grab it and pass in examination? Indeed, no. Without knowing the basic mathematics how can one understand the calculus? You may argue that some people are capable of doing it and have successfully done it. Yes, off course and you can count them on your fingertips. It is always efficient to impart knowledge in an incremental fashion.

In programming languages, when a program needs to be stretched beyond limits to meet the requirements, we need to solve things by certain pre-processing mechanisms that are not supported by the language itself. Consider any simple program that is to be run on 2 different machines (say 16-bit and 32-bit). To compile such a program we can write 2 versions of the program and compile them individually one by one. However, if some changes are made in actual logic of the program, these changes need to be incorporated in every version. In contrast, we can simply write one program and using some specific statements that are understood by a pre-processor compile the program into 16-bit version. Now, to get another version, we just change the single pre-processor statement understood so that this time it will be pre-processed into 32-bit source code which on compilation will yield 32-bit version of program. The good news is that any change to the actual logic is to be made in a single source.

In this lesson, we will discuss the concept of C pre-processor and will look at various directives available.

17.1 What is a C Pre-Processor?

Pre-processing is the first step in the C program compilation stage. It is a translation phase that is applied to the source code before the compiler does actual compilation. In past, the pre-processor was a separate program, much as the compiler and linker still are. However, nowadays the pre-processor for C language is embedded within the C compiler and is called C Pre-processor.

The C Pre-processor more or less provides its own language which is a very powerful tool to the programmer. The statements written in C pre-processor's language consists of directives. These directives are instructions to the pre-processor to perform certain operations before submitting the code to the compiler. It should be noted that these statements are part of a C program. As an example, the statement #include <stdio.h> that we have been using in our programs is a C pre-processor directive that actually tell the pre-processor to replace the statement with the contents of the file stdio.h. This is enough to support the argument that include is not a keyword of C language. In general, a C Pre-processer only performs textual substitutions on the C source code. Consider the above example. What has pre-processor done?

17.2 Syntax of a C Pre-Processor

The syntax of the C pre-processor is different from the syntax of the C language in several respects.

1. First, the preprocessor is line based. This means, each of the pre-processor directives must begin at the beginning of a line and end at the end of the line.
2. Second, the C pre-processor does not know about the structure of C statements like functions, statements, or expressions. Thus it is possible to play strange tricks with the preprocessor to turn something which does not look like C into C (or vice versa).
3. Third, all directives begin with a pound sign i.e. '#' which must be the first non-blank character, and for readability should begin in first column.
4. Fourth, as they are not C statements so they do not end with a semi-colon.

17.3 Advantages of a C Pre-Processor

C Pre-processer actually performs textual substitutions on the C source code, in three ways:

1. File Inclusion: This way the C pre-processor inserts the contents of another file into the source file. Recall the example of #include <stdio.h>. Thus, when the modified source file is fed to the compiler all the statements are simply C statements. Because the header files contain declarations and other definitions, when it is placed into the actual source it should be placed before main(). Hence, as a rule it is always stated that #include should be the first statement of the C program.

2. Macro Substitution: This way the C pre-processor replaces instances of one piece of text with another. This will dealt in next lesson.
3. Conditional Compilation: This way the C pre-processor arranges the parts of source code based on certain circumstances. The arrangement is generally directed towards whether a block of code should be fed to the compiler or not. Recall the example of requirement of running a program on 16-bit and 32-bit machines.

Thus, use of the C pre-processor is advantageous since it makes:

1. Programs are easier to develop,
2. Programs are easier to read,
3. Programs are easier to modify, and
4. The C code is more portable across different machine architectures.

17.4 File Inclusion Directive

In our last lesson, we discussed the significance of having libraries of functions which can be used in any C program. However, we also mentioned that these libraries cannot be automatically embedded into any source unless the corresponding header file is included. These header files which contain function declarations, macro definitions and extern data declarations are included into a C program using File Inclusion Directive of a C pre-processor called include. The directive begins with '#' as all C pre-processor directives do. The general syntax to do so is as follows:

```
    #include <headerfile.h>
```

or

```
    #include “headerfile.h”
```

The directive takes a kind of argument which is the name of a file with whose content the complete directive within the source needs to be replaced.

The difference between the <> and “” forms is where the C pre-processor searches for headerfile.h. As a general rule, it searches for files enclosed in <> in central, standard directories, and it searches for files enclosed in “” in the directory containing the source file that’s doing the including. Therefore, “” is usually used for user-defined header files, and <> is usually used for headers which are provided with the compiler.

The extension ‘.h’ reflects the fact that #include directives should be placed at the top of source files, and should contain

1. External declarations of global variables and functions,
2. Preprocessor macro definitions,
3. Structure & Union definitions,
4. Typedef declarations, and so on.

However, it should never contain the instances of global variables and function bodies.

17.5 Conditional Compilation Directives

As mentioned earlier, sometimes it is required to compile a program into different versions according to certain circumstances. In general, the difference between the versions of a program is matter of few code blocks. Conditional Compilation directives of a C program allow placing of various blocks of code within a single program however feeds to the compiler one or more blocks as per the pre-processing results. There are many directives used to accomplish this task, however we will discuss the three most commonly and widely supported directive construct called

1. #if … #else … #endif
2. #ifdef … #else … #endif
3. #ifndef … #else … #endif

The general syntax of these directive constructs along with usage is as follows:

```
    #if constant-expression
```

```
        {
        statements1;
        }
    #else
        {
        statements2;
        }
    #endif
```

In this directive, if the constant expression evaluates to True (non-zero), then the statements immediately after #if (i.e. statements1) are fed to the compiler for compilation while as the statements immediately after #else are removed from final source. In contrast, if the constant expression evaluates to False (zero), then the statements immediately after #if (i.e. C_statements1) are not fed to the compiler for compilation while as the statements immediately after #else are.

It should be noted that this action is taken by C pre-processor before compilation and does not ever happen during run-time of a program. This means, the constant expression can be simple numeric values, expressions containing other directives like MACROS (as explained in next lesson) but not C variables.

It is worth mentioning here that the statements can be other C pre-processor directives. In this case, further pre-processing is done before the generation of final source.

```
    #ifdef some-macro
        {
        statements1;
        }
    #else
        {
        statements2;
        }
    #endif
```

In this directive, if the MACRO some-macro is defined within the program (or in header files included by the program), then the statements immediately after #ifdef (i.e. C_statements1) are fed to the compiler for compilation while as the statements immediately after #else are removed from final source and vice versa.

```
    #ifndef some-macro
        {
        statements1;
        }
    #else
        {
        statements2;
        }
    #endif
```

In this directive, if the MACRO some-macro is not defined within the program (or in header files included by the program), then the statements immediately after #ifndef (i.e. C_statements1) are fed to the compiler for compilation while as the statements immediately after #else are removed from final source and vice versa.

17.6 Summary

- Pre-Processor as the name suggests is a process in which something goes through some preliminary processing before the actual processing.
- Pre-processing is the first step in the C program compilation stage.
- It is a translation phase that is applied to the source code before the compiler does actual compilation.
- The C Pre-processor more or less provides its own language which is a very powerful tool to the programmer.
- The statements written in C pre-processor’s language consists of directives.
- These directives are instructions to the pre-processor to perform certain operations before submitting the code to the compiler.

- In general, a C Pre-processer only performs textual substitutions on the C source code.
- The syntax of the C pre-processor is different from the syntax of the C language in several respects.
    - First, the preprocessor is line based.
    - Second, the C pre-processor does not know about the structure of C statements like functions, statements, or expressions.
    - Third, all directives begin with a pound sign i.e. '#' which must be the first non-blank character, and for readability should begin in first column.
    - Fourth, as they are not C statements so they do not end with a semi-colon.
- C Pre-processer performs textual substitutions on the C source code, in three ways:
    - File Inclusion
    - Macro Substitution
    - Conditional Compilation
- In File Inclusion, the C pre-processor inserts the contents of another file into the source file.
- In Conditional Compilation, the C pre-processor arranges the parts of source code based on certain circumstances. The arrangement is generally directed towards whether a block of code should be fed to the compiler or not.
- The use of the C pre-processor is advantageous since it makes:
    - Programs are easier to develop,
    - Programs are easier to read,
    - Programs are easier to modify, and
    - The C code is more portable across different machine architectures.

17.7 Model Questions

Q 146. Explain the concept and significance of pre-processing in C programming Language.

Q 147. Enumerate few syntactical differences between the statements of C language and C pre-processor language.

Q 148. "C pre-processor actually performs textual substitution". Explain.

Q 149. What are directives?

Q 150. Explain the File Inclusion Directive of C pre-processor.

Q 151. With the help of a program explain the include directive of C pre-processor.

Q 152. What is the significance of File Inclusion Directive of C pre-processor?

Q 153. Explain the Conditional Compilation Directive of C pre-processor.

Q 154. With the help of an example, explain the #if…#else…#endif directive.

Q 155. What is the significance of Conditional Compilation Directive of C pre-processor?

Lesson 18

MACROS

Structure

138.0. Introduction

138.1. Macro & Macro Substitution

138.2. Simple Macro Substitution

138.3. Macros with Arguments

138.4. Nesting of Macros

138.5. Undefining a Macro

138.6. Summary

138.7. Model Questions

18.0 Introduction

C Pre-Processor in simple words is just a text substitution tool. In last lesson, we saw how an include directive can substitute the directive with the contents of the file which is passed to it. In #if…#else…#endif we saw how a particular block of code is fed to compiler while another is simply not, depending upon some condition. In both cases, the C pre-processor is doing text substitution. There is another directive supported by C pre-processor that does text substitution.

In this lesson, we will discuss the concept of C pre-processor's most powerful feature called Macro. Specifically, we will discuss how to define Macros, use them and undefine them.

18.1 Macro & Macro Substitution

A Macro is an identifier which when encountered within the program by the C pre-processor substitutes it by the pre-defined string composed of one or more tokens. Conversely, a Macro-substitution is a process where an identifier in a program called Macro is replaced by a predefined string composed of one or more tokens. This means, there are two steps in Macro substitution:

1. Defining a Macro, and
2. Using a Macro

The first is to define a Macro. By defining a Macro we mean assigning it some text so that when this Macro is used anywhere in the program the C pre-processor replaces the Macro with that text. To accomplish this task we have a directive called #define. The general syntax of defining a Macro is as follows:

```
    #define MACRO_NAME SUBSTITUTION_TEXT
```

The MACRO_NAME follows the same rules as ordinary C identifiers. It can contain only letters, digits, and underscores, and may not begin with a digit. Since Macros behave quite differently from normal variables (or functions), it is customary to give them names which all are capital letters (or at least which begin with a capital letter). Also, the SUBSTITUTION_TEXT can be absolutely anything. It is not restricted to numbers, simple strings, etc. but full-fledged C statements and other C pre-processor directives can be used.

It should be noted that the directive does not end with a semi-colon. Also, there are different forms of Macro Substitution. The most common forms are:

1. Simple Macro Substitution
2. Argumented Macro Substitution
3. Nested Macro Substitution

18.2 Simple Macro Substitution

In Simple Macro Substitution, the Macro is generally used to define a literal constant like number, string, etc. and the Macro is used within the program wherein these constants are to be used. The advantage of this is that whenever the constant needs to be changed, we only need to make a change at one place. Furthermore, this way the code is more readable as constants can be given meaningful names. The general syntax of a simple Macro is as follows:

```
     #define MACRO_NAME Literal_Constant
```

The program 18.1 shows how to define and use a simple macro.

```
    #include <stdio.h>

    #define PI 3.14

    void main()
         {
         float radius = 12.2;
         float area;

         area = PI * radius * radius;

         printf(“Area = %f”, area);
    }
```

Program 18.1: Program shows how to define and use a simple Macro

The simple Macro can also be used to define mathematical equations and C statements as shown in program 18.2.

```
#include <stdio.h>

/* Mathematical Equation */
#define PI (3 + .14)

/* C Statements */
#define DEF_VAR1 float radius = 12.2;

#define DEF_VAR2 float area;
```

```
#define CALC area = PI * radius * radius;

#define PRINT printf(“Area = %f”, area);


    void main()
        {

        DEF_VAR1
        DEF_VAR2

        CALC

        PRINT
    }
```

Program 18.2: Program shows various forms of Simple Macro Definition

18.3 Macros with Arguments

The C Pre-processer also allows defining and using a complex Macro which takes arguments. The general syntax of this Macro definition is as follows:

```
#define MACRO_NAME(A1,… An) SubstitutionText
```

The identifiers A1 through An, are called Formal Arguments of a Macro and are analogous to the formal arguments of a function. It should be noted that there is no blank space between the left parentheses and the Macro name.

As analogous to functions, during the Macro Call within the program, the actual arguments replace the formal arguments in the substitution text and then the call is replace by the substitution text. To explain this more clearly, let us consider the above program. Instead of defining the Simple Macro CALC as follows

```
#define CALC area = PI * radius * radius;
```

we can define an argumented Macro CALC(r) as

```
#define CALC(r) area = PI * r * r;
```

And within the program, instead of

```
         CALC
```

we call our Macro as

```
         CALC(radius)
```

The C pre-processor will first replace the formal argument ‘r’ within the substitution text by actual argument ‘radius’. Thus, the substitution text will become area = PI * radius * radius; and then will replace the call by this substitution text. The program 18.3 shows how this can be done

```
#include <stdio.h>

/* Mathematical Equation */
#define PI (3 + .14)

/* C Statements */
#define DEF_VAR1 float radius = 12.2;

#define DEF_VAR2 float area;

#define PRINT printf(“Area = %f”, area);

/* Macro with arguments */
#define CALC(r) area = PI * r * r;


    void main()
        {

        DEF_VAR1
        DEF_VAR2

        /* Macro Call */
        CALC(radius)
        PRINT
    }
```

Program 18.3: Program shows how to define and use Macros with arguments

It should be noted that all programs (18.1, 18.2 and 18.3) will produce same output.

18.4 Nesting of Macros

As decision making statements of the C language can be nested, so can Macros be. This means a Macro can be used within the substitution text of another Macro provided the former is defined before its usage. In program 18.2 and 18.3 we have used the Macro PI within the substitution text of Macro CALC. Now, consider the following Macro definitions below

```
#define PI 3.14

#define AREA(r) (4*PI*r*r);

#define VOLUME(r) (AREA(r)*r/3)
```

Let us consider a sphere whose area and volume is to be calculated. Using above Macro definitions, we can write a simple program as follows.

```
#include <stdio.h>

#define PI 3.14

#define AREA(r) (4*PI*r*r)

#define VOLUME(r) (AREA(r)*r/3)

void main()
    {
```

```
    float radius;

    printf(“Enter radius of Sphere?”);

    scanf(“%f”, &radius);

    printf(“Area of Sphere is ”);

    printf(“%f\n”, AREA(radius));

    printf(“Volume of Sphere is ”);

    printf(“%f\n”, VOLUME(radius));
    }
```

Program 18.4: Program shows how to use nested Macros

In this program, the argumented Macro VOLUME uses another argumented Macro AREA. In this case, when VOLUME is used within the program the Macro expansion of Macro VOLUME

```
    VOLUME(radius)
```

will be something like this

```
    (AREA(radius)*radius/3)
```

This substitution text also contains another argumented Macro AREA. Hence, it will further be expanded as follows

```
    ((4*PI*radius*radius)*radius/3)
```

This expansion also contains another simple Macro PI. Thus final expansion will be like this

```
    ((4*3.14*radius*radius)*radius/3)
```

This means the source which will be fed to the compiler for compilation into machine code will be

```
#include <stdio.h>

void main()
    {
    float radius;

    printf(“Enter radius of Sphere?”);

    scanf(“%f”, &radius);

    printf(“Area of Sphere is ”);

    printf(“%f\n”, (4*3.14*radius*radius));

    printf(“Volume of Sphere is ”);

    printf(“%f\n”,
     ((4*3.14*radius*radius)*radius/3));

    }
```

18.5 Undefining a Macro

Sometimes it is required that a Macro be undefined and removed from the program. To accomplish this task C pre-processor provides a directive called #undef. The general syntax to undefine an already defined Macro is as follows

```
    #undef MACRO_NAME
```

18.6 Summary

- A Macro is an identifier which when encountered within the program by the C pre-processor substitutes it by the pre-defined string composed of one or more tokens.
- A Macro-substitution is a process where an identifier in a program called Macro is replaced by a predefined string composed of one or more tokens.
- There are two steps in Macro substitution:

- Defining a Macro, and
- Using a Macro

- By defining a Macro we mean assigning it some text so that when this Macro is used anywhere in the program the C pre-processor replaces the Macro with that text.
- To accomplish this task we have a directive called #define.
- The Macro name follows the same rules as ordinary C identifiers.
- It can contain only letters, digits, and underscores, and may not begin with a digit.
- Since Macros behave quite differently from normal variables (or functions), it is customary to give them names which all are capital letters (or at least which begin with a capital letter).
- Also, the substitution text can be absolutely anything. It is not restricted to numbers, simple strings, etc. but full-fledged C statements and other C pre-processor directives can be used.
- There are different forms of Macro Substitution. The most common forms are:
    - Simple Macro Substitution
    - Argumented Macro Substitution
    - Nested Macro Substitution
- In Simple Macro Substitution, the Macro is generally used to define a literal constant like number, string, etc. and the Macro is used within the program wherein these constants are to be used.
- The simple Macro can also be used to define mathematical equations and C statements.
- The C Pre-processer also allows defining and using a complex Macro which takes arguments.
- As analogous to functions, during the Macro Call within the program, the actual arguments replace the formal arguments in the substitution text and then the call is replace by the substitution text.
- As decision making statements of the C language can be nested, so can Macros be.
- A Macro can be used within the substitution text of another Macro provided the former is defined before its usage.
- Sometimes it is required that a Macro be undefined and removed from the program.

- To accomplish this task C pre-processor provides a directive called #undef.

18.7 Model Questions

Q 156. What is a Macro and how is it different from a variable in C language?

Q 157. What are the advantages of using Macro in C programming language?

Q 158. The value of a Macro can’t be changed during running the program. Explain.

Q 159. Explain the steps of Macro Substitution in C programming language.

Q 160. Classify Macro Substitution in C programming language.

Q 161. Explain with the help of a program the Simple Macro Substitution.

Q 162. Explain with the help of a program the Macro Substitution with arguments.

Q 163. Explain with the help of a program the Nested Macro Substitution.

Lesson 19

File Processing in C

Structure

155.0. Introduction

155.1. What is a File?

155.2. General Steps for File Processing

155.3. C Functions for Opening & Closing a File

155.4. C Functions for Reading a File

155.5. C Functions for Writing a File

155.6. Command Line Parameters

155.7. Summary

155.8. Model Questions

19.0 Introduction

The digital technologies have penetrated into every aspect of our day to day life. This penetration is continuously creating voluminous amount of digital data. The data so created needs to be processed, stored, and later retrieved. However, the main memory is small, expensive and volatile. This means, in such a scenario, to store digital data one should declare an array of characters in some C program to hold it in main memory and expect to run this program for next 100 years without crashing it. If this would have been the remedy then we would have kept our machine running 24x7 while drafting this text. But we know this is not. Computer systems are equipped with secondary storage devices which are finite but have large capacity, are slow but inexpensive, and are nasty but not volatile. Whenever some digital data needs to persist for a long time we store this data on secondary storage device like magnetic disk (Hard Disk), flash disk (Pen Drive), and so on. However, the unit storage in these devices is not byte but a sector (1 sector = 512 bytes) and micro-processors cannot address these locations the way it locates memory locations. To put it in simple words, we can't declare a variable in C program whose address will be some location on hard disk. This means to store and retrieve data onto hard disk we need some different abstraction. However, when we have to process this data we need to bring it into main memory (as content of variables), process it and store it back onto the secondary storage device. The abstraction that is used to store and retrieve data onto a secondary storage device like magnetic disk is called File.

In this lesson, we will discuss the File Processing in C. Specifically; we will discuss how to create a file, write data to it, and read data from it using a C program.

19.1 What is a File?

A File is a collection of bytes stored on a secondary storage device. The collection of bytes may be interpreted, for example, as characters, words, lines, paragraphs and pages from a textual document, fields and records belonging to a database, or pixels from a graphical image. Every file contains bytes however the meaning of these bytes is determined entirely by the data structures and operations used by a program to process the file. Essentially there are two kinds of files that programmers deal with; text files and binary files.

19.1.1 Text Files

A Text File is a collection of printable characters that a computer can process. As the name suggests, it only contains printable textual characters. These characters can be retrieved one-by-one, or in terms of words, or lines and so on, by a program. The ASCII table contains a list of printable characters supported by C language. However, it also contains non printable characters. Among these non-printable characters some characters like NEWLINE, etc. are also part of text files. These characters are used by a text processing program to take specific actions like starting a new line, etc.

19.1.2 Binary Files

A Binary File is like a Text File with no restriction on the set of characters. This means, every text file is essentially a binary file but every binary file is not a text file. It should be noted that special characters of ASCII table are interpreted as special instructions for functions that read a file in text mode. However, no such transformation is done when a file is read in binary mode.

19.2 General Steps for File Processing

Whatever be the programming language in which a program is written to process a file, there are certain general steps followed. These include:

1. Opening a File,
2. Reading/Writing a File, and
3. Closing a File

19.2.1 Opening a File

Before some data is read or written to a file, we need to specify the location of file on the secondary storage device to the operating system. This step is accomplished by opening a File. The program specifies the location of file by using the path of file within the device. As an example, let us assume that a file named def.txt exists in C drive under directory abc. Thus the complete path of the file is C:\abc\def.txt. This path is called Absolute Path. In case the program is also within the abc directory, we can specify Relative Path of the file as def.txt.

Now, if the file exists, the program is returned some pointer to a structure that holds the information about the file like its size, location where next read/write should go, and so on. However, if the file doesn't exist the pointer is set to NULL value which indicates its non-existence.

Some languages support creating (and then opening) of a file in case the file does not exist while opening a file.

Also, this is the moment where a programmer specifies where he is going to read or write the file and whether the read/write will be done in Text mode or Binary mode. Thus Opening a File accomplishes following tasks:

1. Specifying the location of file on device,
2. Specifying whether the file be created if it does not exist,
3. Specifying whether the program will read or write to the file, and
4. Specifying whether the file will be accessed in Text or Binary mode.

19.2.2 Read / Write a File

Now, after the file is successfully opened (or created and then opened) the program should read or write data in to the file.

Reading a file means getting some data from the secondary storage into main storage. The functions which accomplish this task read the data into variables declared by the program. Depending upon whether the function reads one byte or more at a time, the variable can be accordingly used. Similarly, writing a file means putting some data from the main memory into the secondary storage. Here we also need some variable to hold data that will be put into the file.

The question is where within the file will a function read or write some data? To answer this question let us consider an example of a blank notebook and pencil where notebook pertains to a file. When we don't have the notebook we will get one – case of file creation when a file doesn't exist. Then we open it – case of normal file opening. As the notebook is blank, there is nothing to read. Now, we want to write some data. Where from shall we begin? The logical answer is from the first line of first blank page. Ok, we wrote the first alphabet – case of file writing. Now, where should the second alphabet go? Next to the first alphabet. This means as soon as an alphabet occupies a space, we move to the next free space for next alphabet write. Finally, when we are done with writing we will close the notebook – case of file closing.

Now, let us assume we want to read it. So, we will open it first – case of normal file opening as it exists. Now, we want to read some data. Where from shall we begin? The logical answer is from the first alphabet of first line of first page. Ok, we read the first alphabet – case of file reading. Now, where from should the second alphabet be read? Next to the first alphabet. This means as soon as an alphabet is read, we move to the next space for next alphabet read and continue until there is nothing to reed. Finally, when we are done with reading we will close the notebook – case of file closing.

The procedure explained above is called sequential file access. However, files also support random access. This means, we can use a function to move to any specific location within the file to read or write there. In files, every location is given a logical address called offset. The first location is located at offset 0, the next location is at offset 1, and so on.

19.2.3 Closing a File

When we are done with processing a file, we need to free the resources. This is accomplished by closing the file.

19.3 C Functions for Opening and Closing a File

The C language as mentioned in previous lessons is small and compact. To perform additional functions, we make use of library function. Thus, to perform I/O with secondary storage device we need a function. Fortunately, the secondary storage device is a standard device and all the functions required to open, read/write and close a file are provided by stdio.h header file. The functions needed by a program to open and close a file are as follows:

19.3.1 fopen

fopen() function is used to open a file, specify the operation to be performed, and mode of access. It takes 3 arguments:

1. The first argument is the path of the file to be opened. The path should be a string wherein the special character '\' is escaped. This

means the path in previous example should be as follows "C:\\abc\\def.txt"

2. The second argument is the mode of the file in which it is opened. The mode corresponds to 3 things:
   a. Whether to create a file or not, if it doesn't exist,
   b. Whether a file is opened in read, write or append mode.
   c. Whether a file is to be accessed in Text mode or Binary mode.

The mode is also specified as string which can have following values:

Mode String	Description
"r"	Open for Read-Only. The reading will begin from offset=0.
"w"	Open for Write-Only. If a file already exists, it will be overwritten. Writing will begin from offset=0. If the file doesn't exist, it will be created. Writing will begin from offset=0.
"a"	Open for Append at the end of file. Writing will begin from offset=file size. If the file doesn't exist, it will be created. Writing will begin from offset=0.
"r+"	Open file for both Reading and writing. However, it will begin from offset = 0.
"w+"	Create a new file for both Reading and writing. If a file already exists, it will be overwritten. In both cases, it will begin from offset=0.

“a+”	Open file for both Reading and writing. However, it will begin from offset = size. If the file doesn’t exist, it will be created. Writing will begin from offset=0.
“rt”	Same as “r” and text mode.
“wt”	Same as “w” and text mode.
“at”	Same as “a” and text mode.
“r+t”	Same as “r+” and text mode.
“w+t”	Same as “w+” and text mode.
“a+t”	Same as “a+” and text mode.
“rb”	Same as “r” and binary mode.
“wb”	Same as “w” and binary mode.
“ab”	Same as “a” and binary mode.
“r+b”	Same as “r+” and binary mode.
“w+b”	Same as “w+” and binary mode.
“a+b”	Same as “a+” and binary mode.

It should be noted that those mode strings in which the access mode (Text or Binary) is not specified open the file in default access mode which is Text Mode.

The general syntax of fopen() function is as follows:

```
<filepointer> fopen(<filepath>,<mode>)
```

It should be noted that the fopen() returns a pointer of type struct FILE. This pointer, if not NULL, points to a structure in memory that holds the information about the opened file. However, if it is NULL then the call was unsuccessful. This pointer is used in file reading, writing and closing functions to identify the opened up file. Thus this pointer must be saved in some variable for further usage. As an example, the file opening construct of a C program should be like this

```
FILE *fileptr;

fileptr = fopen(“c:\\abc\\def.txt”, “r”);

if (fileptr == NULL)

   printf(“Error in file opening!”);
```

Look at the code snippet. The file path is a string with ‘\’ escaped. The mode is a string which specifies opening of a file in read mode and accessing contents of a file in default text mode. The function returns a pointer, which if NULL means the call was unsuccessful.

19.3.2 fclose

fclose() function is used to close an opened file. This essentially means to free and flush all the resources held by the program due to the opened file. Specifically, the FILE structure to which the file pointer returned by the fopen() call is pointing, is freed and file pointer is set to NULL.

The general syntax of fclose() function is as follows:

```
<returncode> fclose(<filepointer>)
```

It should be noted that fclose() returns an integer code whose value if zero (0) means the file was successfully closed. In other case, there was some error.

19.4 C Functions for Reading a File

Indeed we need to open a file before processing it and close it after we are finished. However, between these two function calls for a particular file, we will be reading, writing or both, the file. The functions in stdio.h to read a file are as follows:

19.4.1 fgetc()

fgetc() is a function provided by stdio.h to read a single character at a time from a file. The general syntax is as follows

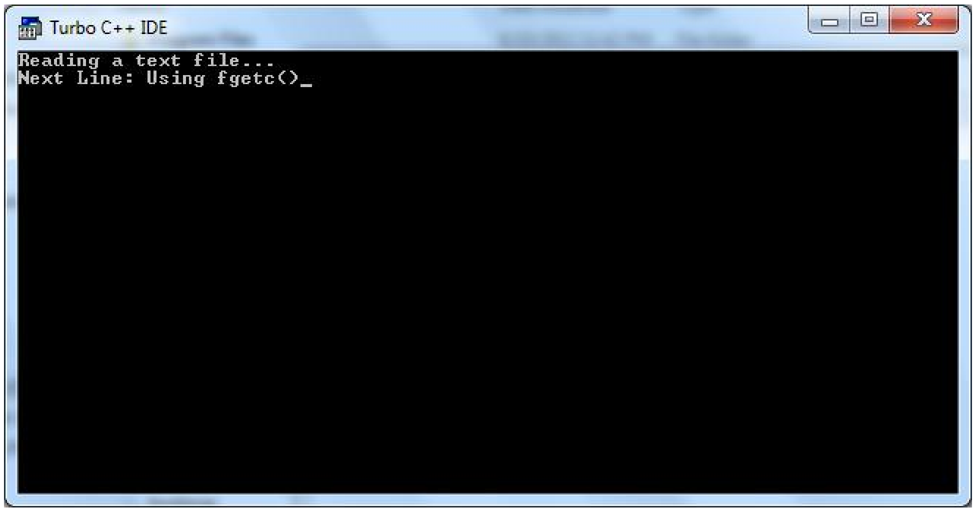
```
int fgetc (FILE *);
```

This means the function takes a single argument which is the file pointer of the file already opened and returns the ASCII code of the character read at the current position of the file offset. After this, the offset is incremented by one to point to the next character. In case, the read character has ASCII value 26, it means the End of File (EOF) has reached and there is no more data in the file. The program 19.1 shows how to read and display an entire text file using fgetc() function which is followed by the screen shot of the output.

```
#include <stdio.h>

void main()
   {
   FILE * fp;
   int c;

   fp = fopen (“c:\\abc.txt”, “rt”);

   if (fp == NULL)
        printf(“File opening error!”);
   else
        {
        while( (c=fgetc(fp)) != EOF)
             putchar(c);
        }
```

```
    fclose(fp);
    }
```

Program 19.1: Reading a text file using fgetc()



```
Reading a text file...
Next Line: Using fgetc()_
```

19.4.2 fgets()

fgets() is another function that reads a file in terms of fixed length strings of characters. The general syntax is as follows

```
 char* fgets ( char *, int, FILE *);
```

This means the function takes 3 arguments; the first argument is the address of a string into which the string read from the file will be copied, the second argument is the maximum string length to be read at a time and the last argument is the file pointer of the file opened. The maximum string length parameter is used to avoid overflow of destination character array.

Because fgets() reads strings and strings have a terminating NULL character; therefore the function reads one less than what is specified in maximum string length and appends a NULL character (‘\0’) at the end of the character string so that it is properly initialized. Thus, fgets() function reads n-1 characters from a file if the second parameter specifies n as maximum string length.

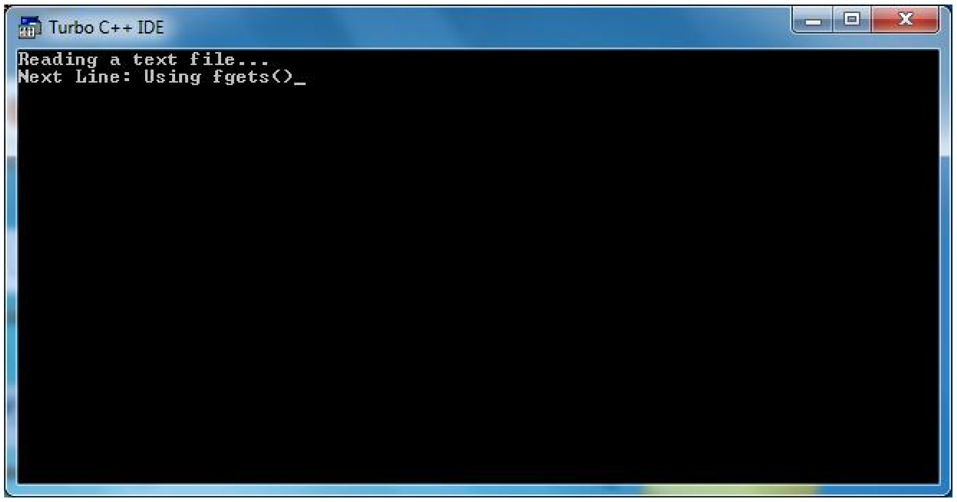
After every read, the file offset is incremented by the number of characters read (n-1) and function returns the read string terminated with ‘\0’. However, during reading, if the file contains lesser characters i.e. EOF has reached, the reading stops immediately and the function returns the string read. Next time, when fgets() tries to read, it fails and returns NULL signaling EOF.

It should be noted that when fgets() encounters new line character (‘\n’), it stops reading immediately and the function returns the string read. Next time, when fgets() tries to read, it reads from the character next to the new line character. In other word, fgets() delimits strings on the basis of size and new line character.

The program 19.2 shows how to read and display an entire text file using fgets() function which is followed by the screenshot of the output.

```
#include <stdio.h>

void main()
   {
   FILE * fp;
   int str[5];

   fp = fopen (“c:\\abc.txt”, “rt”);

   if (fp == NULL)
       printf(“File opening error!”);
   else
       {
       while( fgets(str, 5, fp) != NULL)
           printf(“%s”, str);
       }
   fclose(fp);
   }
```

Program 19.2: Reading a text file using fgets()



```
Reading a text file...
Next Line: Using fgets()_
```

It should be noted that in both fgetc() and fgets(), the new line character ('\n') is read, interpreted and transformed properly, as it should be. This is why we are getting output in two lines the way abc.txt was written using notepad.

19.4.3 fscanf()

fscanf() is used to read a file in a formatted manner just like scanf() reads the data from keyboard in a formatted manner. The argument list, format specifiers, etc. of a fscanf() are same as that of scanf() function. However, there are two differences,

1. The fscanf() function has extra argument that specifies the file. This argument is a file pointer and is the first argument of fscanf().
2. The fscanf() reads from any file while scanf() reads from a standard input file (keyboard) only.

The general syntax of fscanf() function is as follows:

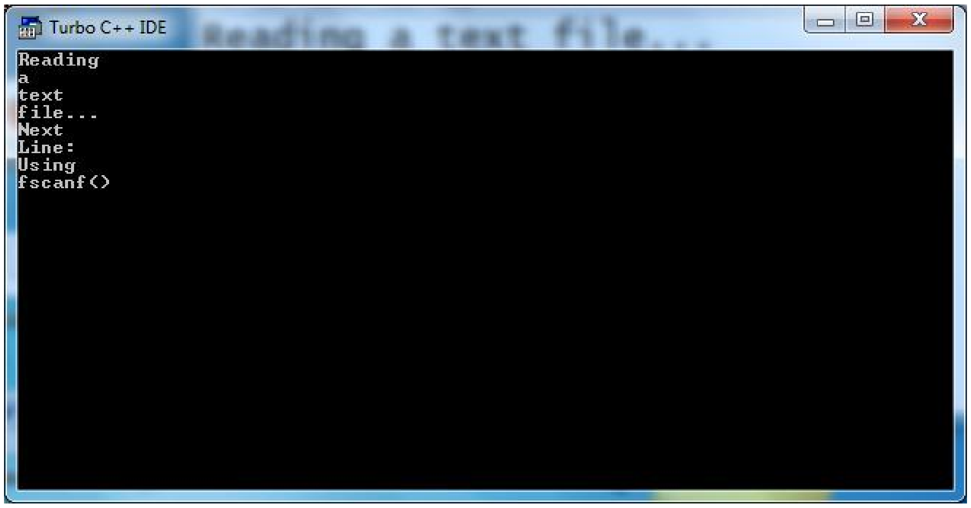
```
int fscanf
    (
    FILE *,
    <format-specifier>,
    <Argument_list>
    );
```

The function can read in terms of characters, integers, strings, and so on. It can do conversions of numeric-characters into numeric value by using specifiers. However, it should be noted that while reading in terms of strings, it delimits the strings on the basis of blank space and new line character. It ignores interpretation of spaces and new line characters and saves none in the string read. Though the string is properly initialized by appending a NULL character at the end.

The program 19.3 shows how to read and display an entire text file using fscanf() function followed by the screenshot of the output.

```
#include <stdio.h>

void main()
    {
    FILE * fp;
    int str[15];

    fp = fopen (“c:\\abc.txt”, “rt”);

    if (fp == NULL)
         printf(“File opening error!”);
    else
         {

         while( fscanf(fp,”%s”,str) != EOF)
              printf(“%s”, str);

         }
    fclose(fp);
    }
```

Program 19.3: Reading a text file using fscanf()



```
Reading
a
text
file...
Next
Line:
Using
fscanf()
```

19.4.4 fread()

All the functions that we have discussed so far are good for reading text files. Check the programs, all have the default file opening mode; text. However, when a file has been written in binary these functions may fail. The reason behind this is that there are 3 main differences in text and binary files. These include:

1. Handling of new line character,
2. Representation of EOF, and
3. Storage of numeric data.

In binary mode, no interpretation and conversion of new line character takes place, file doesn't contain EOF, and numeric data is stored as binary and not text. So, to read data from binary files, we have a function called fread(). The general syntax of the function is as follows:

```
int fread
     (
     void *,  /* Buffer Address */
     int,     /* Size of Data Item */
     int,     /* No. of Items */
     FILE *   /* File pointer */
     );
```

The function takes first argument as the address of the buffer where the read data is to be placed. The second argument specifies the size of data while third specifies the number of these data items. Finally, the last argument specifies the file pointer of file opened. The function returns the number of items read; if return value is zero it means EOF has reached.

It should be noted that as fread() is used with binary files, the file should be opened in binary mode. Also, the buffer data type should be same as that of data item which is supposed to be read. This means, if we are reading integers then buffer should be address of some integer variable or array element or structure member and so on. Also, the size should be size of integer i.e. 2 bytes. However, if we are reading one item at a time then number of items should be one. In case, we are reading multiple items the buffer should be address of array of integers. The program 19.4 shows how to read records from a binary file using fread().

```
#include <stdio.h>

struct record
    {
    int rno;
    char name[15];
    };

void main()
    {
    FILE * fp;
    struct record std;

    fp = fopen (“c:\\abc.txt”, “rb”);

    if (fp == NULL)
         printf(“File opening error!”);
    else
         {
         while( fread(
                   &std,
                   sizeof(std),
                   1,fp) == 1)
              {
              printf(“Rollno: %d ”, std.rno);
```

```
             printf(“Name: %s\n”, std.name);
             }
         }

     fclose(fp);
     }
```

Program 19.4: Reading a binary file using fread()

19.5 C Functions for Writing a File

The functions in stdio.h to write a file are as follows:

19.5.1 fputc()

fputc() is a function provided by stdio.h to write a single character at a time to a file. The general syntax is as follows

```
int fputc (char, FILE *);
```

This means the function takes two arguments. The first argument is the character to be written and the second argument is the file pointer of the file already opened. On success, the function returns the ASCII code of the character written at the current position of the file offset. After this, the offset is incremented by one to point to the next location. In case, there is an error, the function returns EOF. The program 19.5 shows how to copy a text file into another text file using fputc() function.

```
#include <stdio.h>

void main()
   {
   FILE * fp1, * fp2;
   int c;

   fp1 = fopen (“c:\\abc.txt”, “rt”);
   fp2 = fopen (“c:\\def.txt”, “wt”);

   if (fp1 == NULL)
```

```
        {
        printf(“File opening error!”);

        return;
        }
    if (fp2 == NULL)
        {
        printf(“File opening error!”);

        fclose(fp1);

        return;
        }
    while( (c=fgetc(fp1)) != EOF)
        fputc(c, fp2);

    fclose(fp1);

    fclose(fp2);

    }
```

Program 19.5: Copying one text file into another text file using fputc()

19.5.2 fputs()

fputs() is another function that writes a file in terms of strings of characters. The general syntax is as follows

```
 int fputs ( char *, FILE *);
```

This means the function takes 2 arguments; the first argument is the address of a string which will be copied into the file, and the second argument is the file pointer of the file opened. The function on success returns the ASCII code of last character written to the file while as on error it returns EOF.

Because fputs() writes strings and strings have a terminating NULL character; therefore the function always writes without NULL character to maintain the continuity in text of the file. Also, it does not append the new line character to the string as puts() does.

The program 19.6 shows how to copy a file using fputs() function.

```
#include <stdio.h>

void main()
    {
    FILE * fp1, * fp2;
    char str[5];

    fp1 = fopen (“c:\\abc.txt”, “rt”);
    fp2 = fopen (“c:\\def.txt”, “wt”);

    if (fp1 == NULL)
         {
         printf(“File opening error!”);

         return;
         }
    if (fp2 == NULL)
         {
         printf(“File opening error!”);

         fclose(fp1);

         return;
         }

    while( fgets(str, 5, fp1) != NULL)
              fputs(str, fp2);

    fclose(fp1);

    fclose(fp2);

}
```

Program 19.6: Copying a file using fputs()

19.5.3 fprintf()

fprintf() is used to write a file in a formatted manner just like printf() writes the data on screen in a formatted manner. The argument list, format specifiers, etc. of a fprintf() are same as that of printf() function. However, there are two differences,

3. The fprintf() function has extra argument that specifies the file. This argument is a file pointer and is the first argument of fprintf().
4. The fprintf() writes to any file while printf() writes only to the standard output file (screen) only.

The general syntax of fprintf() function is as follows:

```
int fprintf
    (
    FILE *,
    <format-specifier>,
    <Argument_list>
    );
```

The function can write in terms of characters, integers, strings, and so on. It can do conversions of numeric values into numeric-characters by using specifiers. Also, it does not append NULL character to the string written.

The program 19.7 shows how to copy a file using fprintf() function.

```
#include <stdio.h>

void main()
   {
   FILE * fp1, * fp2;
   int c;

   fp1 = fopen (“c:\\abc.txt”, “rt”);

   fp2 = fopen (“c:\\def.txt”, “wt”);

   if (fp1 == NULL)
       {
        printf(“File opening error!”);

        return;
        }
    if (fp2 == NULL)
        {
        printf(“File opening error!”);

        fclose(fp1);

        return;
        }

    while( fgets(str, 5, fp1) != NULL)
            fprintf(fp2, “%s”, str);

    fclose(fp1);

    fclose(fp2);

}
```

Program 19.7: Copy a file using using fprintf()

19.5.4 fwrite()

All the functions that we have discussed so far are good for writing text files. To write data into binary files, we have a function called fwrite(). The general syntax of the function is as follows:

```
int fwrite
    (
    void *,  /* Buffer Address */
    int,     /* Size of Data Item */
    int,     /* No. of Items */
    FILE *   /* File pointer */
    );
```

The function takes first argument as the address of the buffer where from the data is to be read. The second argument specifies the size of data while third specifies the number of these data items. Finally, the last argument specifies the file pointer of file opened. The function returns the number of items written; if return value is zero it means some error has occurred.

It should be noted that as fwrite() is used with binary files, the file should be opened in binary mode. Also, the buffer data type should be same as that of data item which is supposed to be written. The program 19.8 shows how to write records into a binary file using fwrite().

```
#include <stdio.h>

struct record
   {
   int rno;
   char name[15];
   };

void main()
   {
   FILE * fp;
   char c;
   struct record std;

   fp = fopen (“c:\\abc.txt”, “wb”);

   if (fp == NULL)
        printf(“File opening error!”);
   else
        {
        while( 1)
        {
        printf(“Enter a record (Y/N)? ”);

        scanf(“%c”, &c);

        if (c==’Y’ || c==’y’)
            {
            printf(“Enter Name?”);

            scanf(“%s”, std.name);
```

```
               printf(“Enter Rollno?”);
               scanf(“%d”, &std.rno);

               fwrite(&std,sizeof(std),1,fp);
               }
          else
               {
               fclose(fp);
               break;
               }
          fflush(stdin);
          }
          }
}
```

Program 19.8: Writing a binary file using fwrite()

19.6 Command Line Parameters

Now is the time that we should introduce the actual prototype of main() function. The function actually accepts two arguments and returns an integer value. The main() function returns 0 to the operating system in case the program executed without crash. Also, the first argument to main() is an integer that contains the number of arguments passed to the function while the second argument is an array of character pointers where each array element is a pointer to the argument string. Therefore the actual main() function is something like this

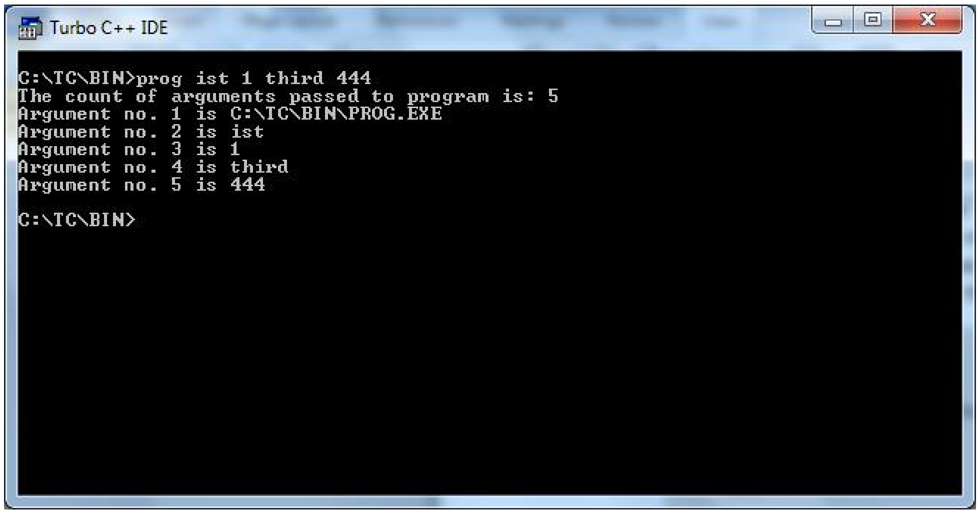
```
     int main ( int argc, char *argv[])
          {
          .
          .
          .
          return 0;
          }
```

How can arguments be passed to the main() function? The arguments can be passed to the main() function on command line when the program is executed. The procedure includes typing the name of program executable on the terminal followed by the list of arguments delimited by blank spaces. It should be noted that all arguments are passed as strings, however we need not to enclose them within the pair of double quotes in the terminal. It is worth mentioning here that the syntax of main() function that we have been following is still correct. But if arguments are passed to such a program via command line, the program knows nothing about the parameters and can't process them. However, there will be no error. This means if we expect some arguments to our program, we should follow the above listed syntax.

How to process them? The way every function processes its arguments. The variables are local to the function and hence available within the body of function. Program 19.9 shows a program that lists the number and contents of the command line arguments passed to a program. It is followed by the screenshot of the output.

```
#include <stdio.h>

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

   int i;

   printf(“The count of arguments passed
           to program is: ”);

   printf(“%d\n”, argc);

   for( i=0; i<argc; i++)
       {
       printf(“Argument no. %d is ”, i+1);
       printf(“%s\n”, argv[i]);
       }
   return 0;
   }
```

Program 19.9: Program shows how to process command line arguments



```
C:\TC\BIN>prog ist 1 third 444
The count of arguments passed to program is: 5
Argument no. 1 is C:\TC\BIN\PROG.EXE
Argument no. 2 is ist
Argument no. 3 is 1
Argument no. 4 is third
Argument no. 5 is 444

C:\TC\BIN>
```

19.7 Summary

- The abstraction that is used to store and retrieve data onto a secondary storage device like magnetic disk is called File.
- Essentially there are two kinds of files that programmers deal with text files and binary files.
- There are certain general steps to be followed while processing a file. These include:
    - Opening a File,
    - Reading/Writing a File, and
    - Closing a File
- Opening a File accomplishes following tasks:
    - Specifying the location of file on device,
    - Specifying whether the file be created if it does not exist,
    - Specifying whether the program will read or write to the file, and
    - Specifying whether the file will be accessed in Text or Binary mode
- The arguments can be passed to the main() function on command line when the program is execute.

19.8 Model Questions

Q 164. Explain the difference between Text and Binary Files?

Q 165. Write a short note on following

a. fopen()

b. fclose()

Q 166. With the help of a program explain fgetc() and fputc() functions in C language.

Q 167. With the help of a program explain fgets() and fputs() functions in C language.

Q 168. With the help of a program explain fscanf() and fprintf() functions in C language.

Q 169. With the help of a program explain fread() and fwrite() functions in C language.

Q 170. Write a program in C language to copy a text file into another text file.

Q 171. Write a program in C language to read and write student records into a binary file.

Q 172. With the help of a program explain command line parameters in C language.