

Module II

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 innumerable 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?

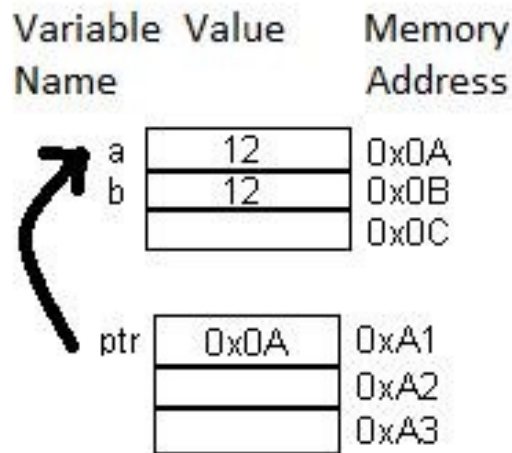
After a pointer variable is declared, 16 bit memory is allocated to the pointer variable and thus before initialization it contains bogus value. Such a pointer which contains bogus value is called **wild pointer**. To properly initialize a pointer means assigning it memory address of some variable of 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 dereference 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 dereferencing 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



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;

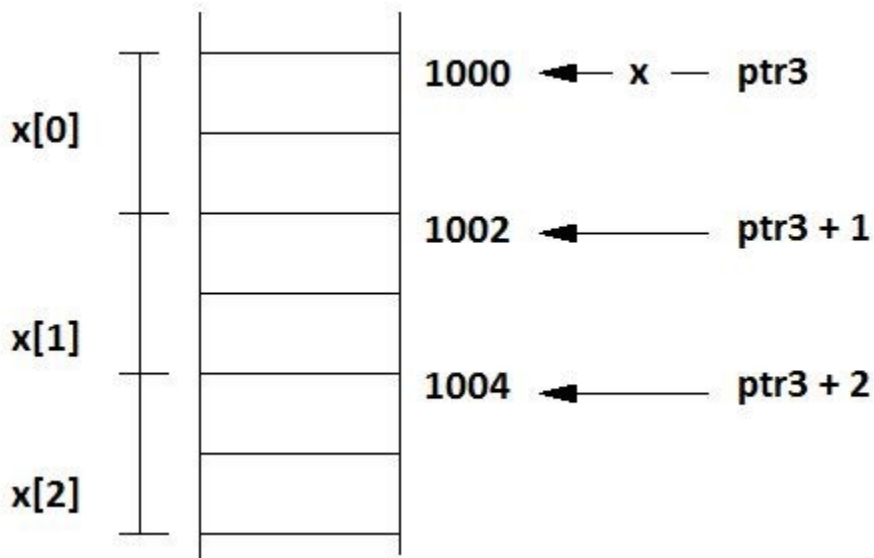
ptr1 = &x[2];

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);

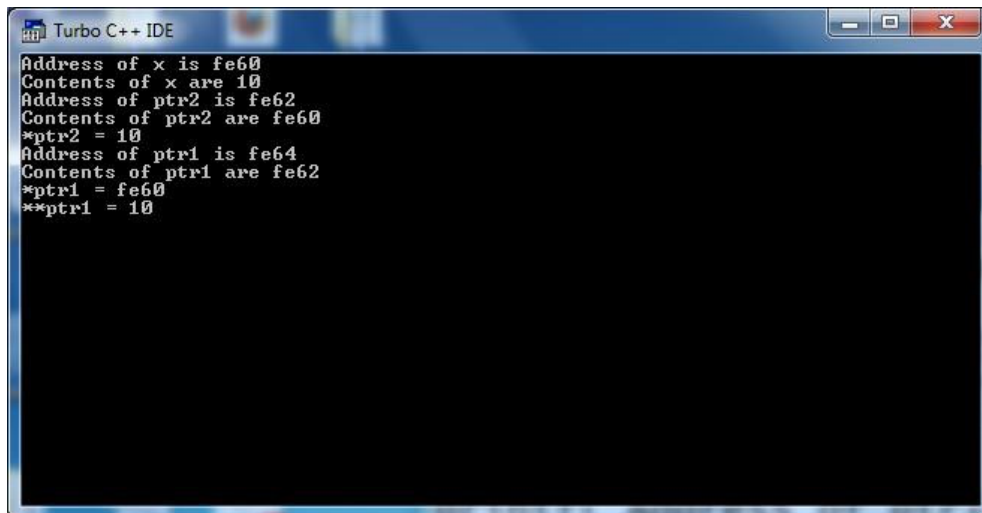
    \
    printf("Address of ptr1 is %x n", &ptr1);

    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



```
Turbo C++ IDE
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 used 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. Classify 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 is 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:

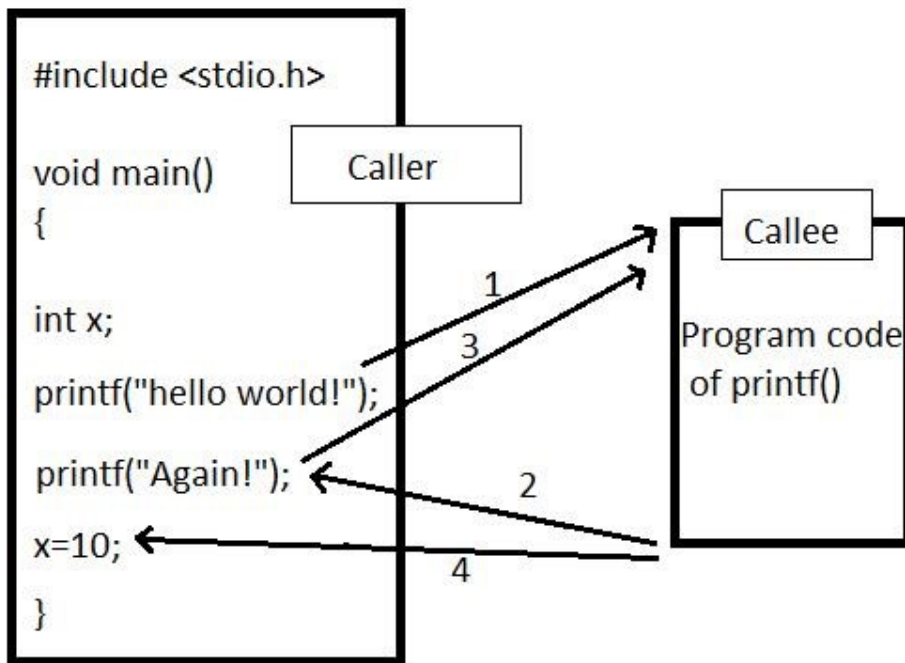
1. Declare the prototype of function, 2. Define the body of function, and
3. Invoke the function.

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.



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>
(
<optional-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.");

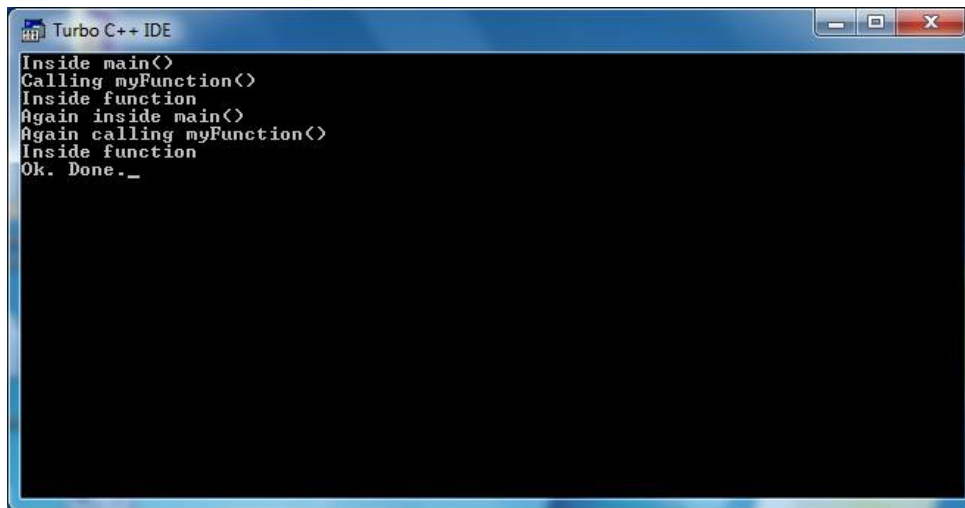
}

/* Function Definition */
void myFunc()
{

    printf("Inside function\n");

}
```

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

A screenshot of the Turbo C++ IDE window. The title bar reads "Turbo C++ IDE". The output window contains the following text:

```
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 is 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 oneliner 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-datatype* of the function specified in function declaration and definition. Furthermore, this data when returned to the caller needs to be 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*/
}

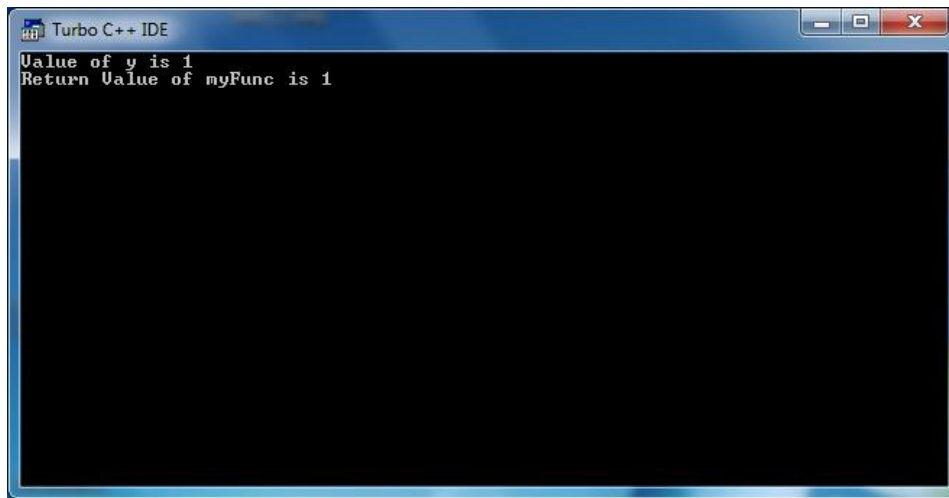
void main()
{
    int y;

    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



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};

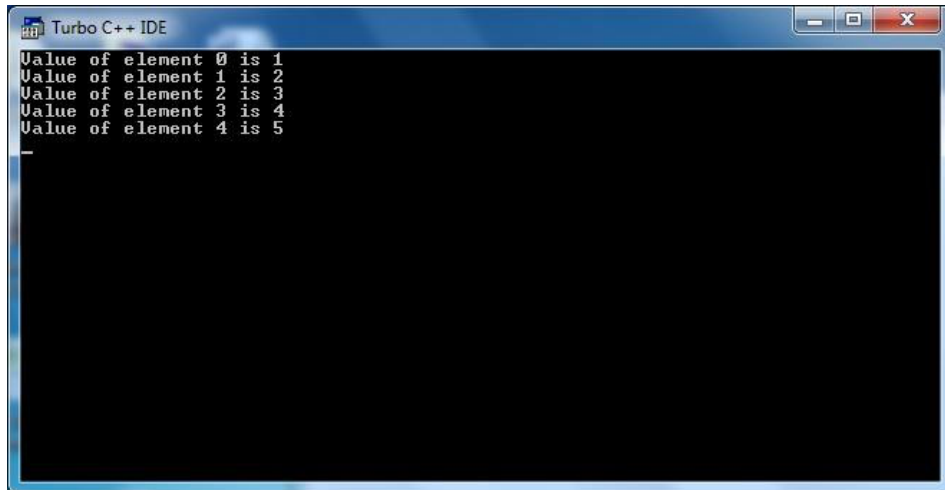
    listArray(myArray);
}

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.



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;

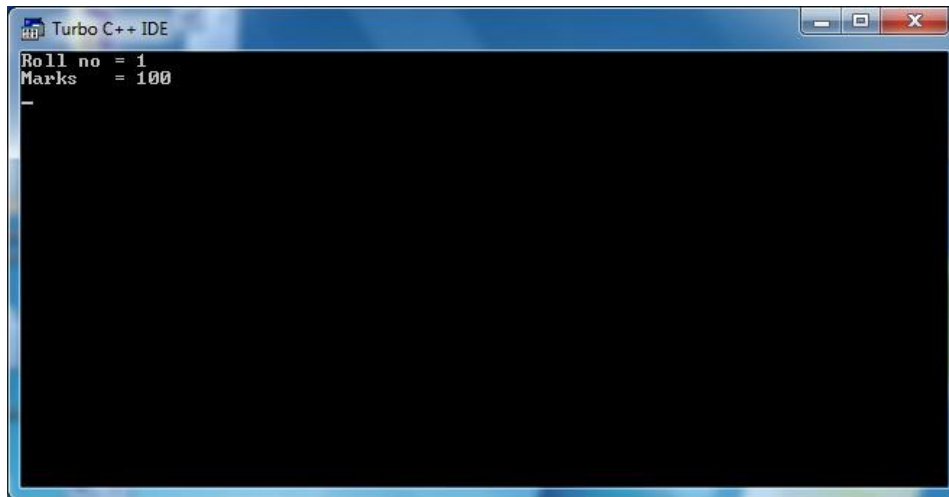
    listStructure(std1);

}

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



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);

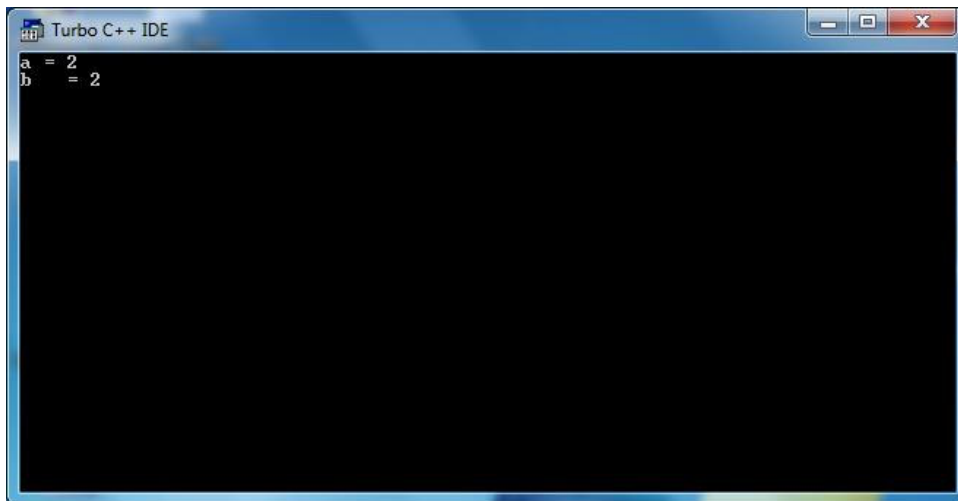
}

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



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-byvalue*, 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



```
Turbo C++ IDE
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 from actual to formal during function call in *pass-byvalue*. 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-byvalue*? 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-byaddress*. 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



```

Turbo C++ IDE
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()
{

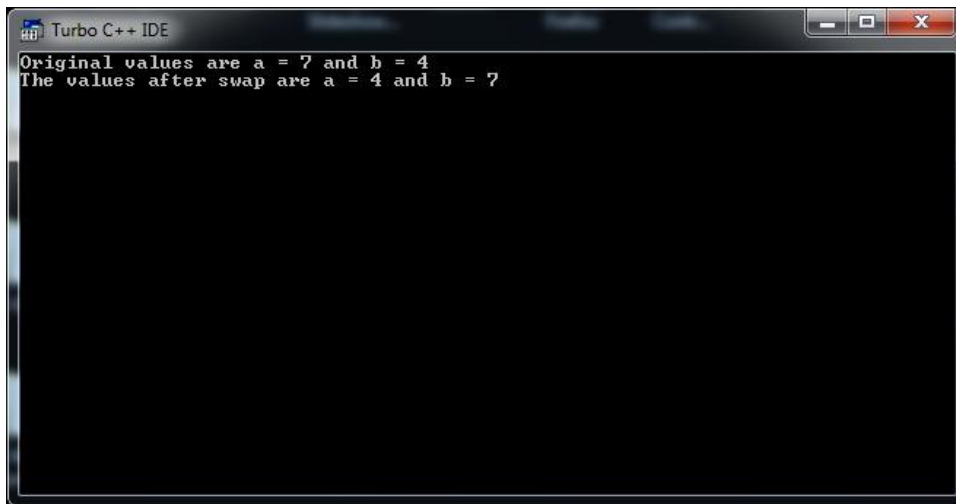
    int a = 7, b = 4;

    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



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';

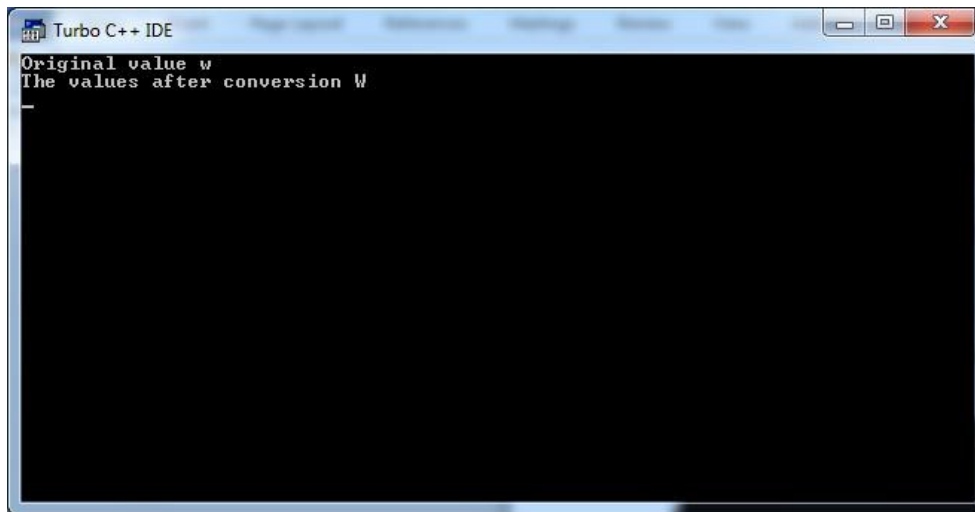
    printf("Original value %c\n", a);

    convert(&a);
```



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

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



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 nullterminated 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

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",`

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
<code>strcpy(s1, s2);</code>	Copies string s2 into string s1
<code>strcat(s1, s2);</code>	Concatenates string s2 onto the end of string s1
<code>strlen(s1);</code>	Returns the length of string s1
<code>strcmp(s1, s2);</code>	Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2
<code>strchr(s1, ch);</code>	Returns a pointer to the first occurrence of character ch in string s1
<code>strstr(s1, s2);</code>	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 userdefined 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, $5! = 5 \times 4 \times 3 \times 2 \times 1$. However, in other words, the number is multiplied by the factorial of the number one less than the actual number, which means $5! = 5 \times 4!$. 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.

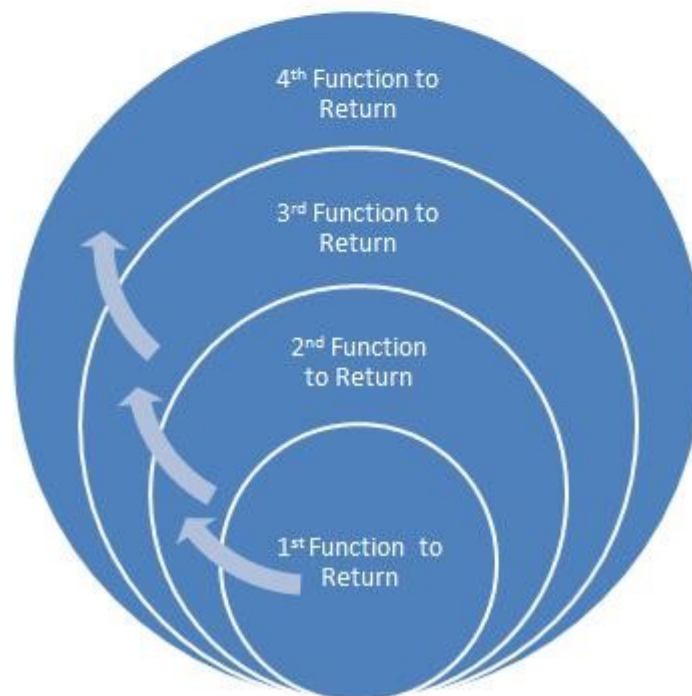
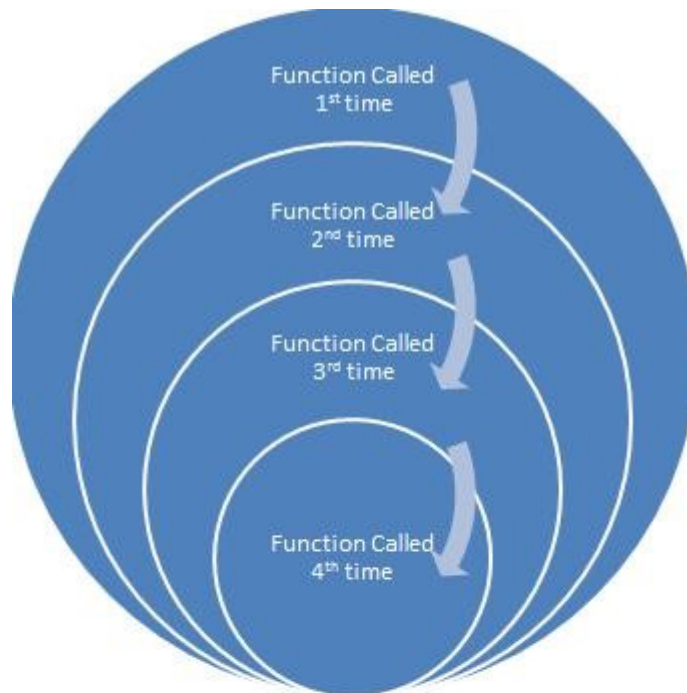
```
void myFunc1()  
{  
  
    myFunc2();  
    /* Recursive Function Call*/  
  
}  
  
void myFunc2()  
{  
  
    myFunc1();  
    /* Recursive Function Call*/  
  
}
```

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 series 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.