UNIT 8 FUNCTIONS

Structure

8.0 INTRODUCTION

To make programming simple and easy to debug, we break a larger program into smaller *subprograms* which perform '*well defined tasks*'. These subprograms are called *functions.* So far we have defined a single function *main ( ).*

After reading this unit you will be able to define many other functions and the *main( )* function can call up these functions from several different places within the program, to carry out the required processing.

Functions are very important tools for ***Modular Programming***, where we break large programs into small subprograms or modules (functions in case of C). The use of functions reduces complexity and makes programming simple and easy to understand.

In this unit, we will discuss how functions are defined and how are they accessed from the main program? We will also discuss various types of functions and how to invoke them. And finally you will learn an interesting and important programming technique known as *Recursion*, in which a function calls within itself.

8.1 OBJECTIVES

After going through this unit, you will learn:

- the need of functions in the programming;
- how to define and declare functions in 'C' Language;
- different types of functions and their purpose;
- how the functions are called from other functions;
- how data is transferred through parameter passing, to functions and the Return statement;
- recursive functions; and
- the concept of 'C*all by Value*' and its drawbacks.

8.2 DEFINTION OF A FUNCTION

A ***function*** is a self- contained block of executable code that can be called from any other function .In many programs, a set of statements are to be executed repeatedly at various places in the program and may with different sets of data, the idea of functions comes in mind. You keep those repeating statements in a function and call them as and when required. When a function is called, the control transfers to the called function, which will be executed, and then transfers the control back to the calling function (to the statement following the function call). Let us see an example as shown below:

Example 8.1

```
/* Program to illustrate a function*/

#include <stdio.h>
main ()
{
void sample( );
printf("\n You are in main");
}

void sample( )
 {
  printf("\n You are in sample");
  }
```

OUTPUT

```
You are in sample
You are in main
```

Here we are calling a function ***sample ( )*** through ***main( )*** i.e. control of execution transfers from ***main( )*** to ***sample( )*** , which means ***main( )*** is suspended for some time and ***sample( )*** is executed. After its execution the control returns back to ***main( )***, at the statement following function call and the execution of ***main( )*** is resumed.

The syntax of a function is:

```
return  data type  function_name (list of arguments)
{
        datatype declaration of the arguments;
        executable statements;
        return (expression);
}
```

where,

- return data type is the same as the data type of the variable that is returned by the function using return statement.
- a function_name is formed in the same way as variable names / identifiers are formed.
- the list of arguments or parameters are valid variable names as shown below, separated by commas: (data type1 var1,data type2 var2,…….. data type n var n) for example *(int x, float y, char z)*.
- arguments give the values which are passed from the calling function.

- the body of function contains executable statements.
- the return statement returns a ***single*** value to the calling function.

Example 8.2

Let us write a simple function that calculates the square of an integer.

```
/*Program to calculate the square of a given integer*/

/* square( ) function */
{
  int square (int no)                  /*passing of argument */
  int result ;                 /* local variable to function square */
  result = no*no;
  return (result);                     /* returns an integer value */
          }

  /*It will be called from main()as follows */
main( )
{
int n ,sq;                     /* local variable to function main */
printf (“Enter a number to calculate square value”);
scanf(“%d”,&n);
sq=square(n);                /* function call with parameter passing */
printf (”\nSquare of  the number is : %d”, sq);
 }  /* program  ends */
```

OUTPUT

```
Enter a number to calculate square value : 5
Square of the number is : 25
```

8.3 DECLARATION OF A FUNCTION

As we have mentioned in the previous section, every function has its declaration and function definition. When we talk of declaration only, it means only the function name, its argument list and return type are specified and the function body or definition is not attached to it. The *syntax* of a function declaration is:

return data type function_name(list of arguments);

For example,

```
int square(int no);
float temperature(float c, float f);
```

We will discuss the use of function declaration in the next section.

8.4 FUNCTION PROTOTYPES

In Example 8.1 for calculating square of a given number, we have declared function ***square( )*** before ***main( )*** function; this means before coming to ***main( )***, the compiler knows about ***square( )***, as the compilation process starts with the first statement of

any program. Now suppose, we reverse the sequence of functions in this program i.e., writing the ***main( )*** *function* and later on writing the ***square( )*** function, *what happens* ? The "C" compiler will give an error. Here the introduction of concept of *"function prototypes"* solves the above problem.

Function Prototypes require that every function which is to be accessed should be declared in the calling function. The function declaration, that will be discussed earlier, will be included for every function in its calling function . Example 8.2 may be modified using the function prototype as follows:

Example 8.3

```
/*Program to calculate the square of a given integer using the function prototype*/
#include <stdio.h>
main ( )
{
int n , sq ;
int square (int ) ;                    /* function prototype */
printf ("Enter a number to calculate square value");
scanf("%d",&n);
sq = square(n);              /* function call with parameter passing */
printf ("\nSsquare of  the number is : %d", sq);
}

/* square function */
int square (int no)                    /*passing of argument */
{
  int result ;               /* local variable to function square */
  result = no*no;
  return (result);                     /* returns an integer value */
}
```

OUTPUT

```
Enter a number to calculate square value :  5
Square of the number is:  25
```

Points to remember:

- *Function prototype* requires that the function declaration must include the return type of function as well as the type and number of arguments or parameters passed.
- The variable names of arguments need not be declared in prototype.
- The major reason to use this concept is that they enable the compiler to check if there is any mismatch between function declaration and function call.

Check Your Progress 1

(1) Write a function to multiply two integers and display the product.

……

……

(2) Modify the above program, by introducing function prototype in the main function.

……

……

8.5 THE *return* STATEMENT

If a function has to return a value to the calling function, it is done through the ***return*** statement. It may be possible that a function does not return any value; only the control is transferred to the calling function. The syntax for the *return* statement is:

return (expression);

We have seen in the *square( )* function, the *return* statement, which returns an integer value.

Points to remember:

- You can pass any number of arguments to a function but can return only one value at a time.

  For example, the following are the valid *return* statements

  (a) return (5);
  (b) return (x*y);

  For example, the following are the invalid *return* statements
  (c) return (2, 3);
  (d) return (x, y);

- If a function does not return anything, ***void*** specifier is used in the function declaration.

  For example:

```
void square (int no)
 {
 int sq;
 sq = no*no;
 printf ("square is %d", sq);
 }
```

- All the function's return type is by default is ***"int"***, i.e. a function returns an integer value, if no type specifier is used in the function declaration.

  Some examples are:

  (i) square (int no); /* will return an integer value */

  (ii) int square (int no); /* will return an integer value */

  (iii) void square (int no); /* will not return anything */

- What happens if a function has to return some value other than integer? The answer is very simple: use the particular type specifier in the function declaration.

For example consider the code fragments of function definitions below:

1) **Code Fragment - 1**

```
char func_char( …….. )
 {
      char c;
```

```
…….
…….
……
}
```

2) **Code Fragment - 2**

```
float func_float (……..)
{
  float f;
       ……..
       ……..
       ……..
       return(f);
}
```

Thus from the above examples, we see that you can return all the data types from a function, the only condition being that the value returned using return statement and the type specifier used in function declaration should match.

- A function can have many *return* statements. This thing happens when some condition based returns are required.

  For example,

```
/*Function to find greater of two numbers*/
int greater (int x, int y)
 {
     if (x>y)
         return (x);
      else
         return (y);
  }
```

- And finally, with the execution of return statement, the control is transferred to the calling function with the value associated with it.

  In the above example if we take x = 5 and y = 3, then the control will be transferred to the calling function when the first return statement will be encountered, as the condition (x > y) will be satisfied. All the remaining executable statements in the function will not be executed after this returning.

Check Your Progress 2

1. Which of the following are valid return statements?

   a) return (a);
   b) return (z,13);
   c) return (22.44);
   d) return;
   e) return (x*x, y*y);

……

……

……

8.6 TYPES OF VARIABLES AND STORAGE CLASSES

In a program consisting of a number of functions a number of different types of variables can be found.

Global vs. Static variables: Global variables are recognized through out the program whereas local valuables are recognized only within the function where they are defined.

Static vs. Dynamic variables: Retention of value by a local variable means, that in static, retention of the variable value is lost once the function is completely executed whereas in certain conditions the value of the variable has to be retained from the earlier execution and the execution retained.

The variables can be characterized by their ***data type*** and by their ***storage class***. One way to classify a variable is according to its data type and the other can be through its storage class. ***Data type*** refers to the type of value represented by a variable whereas ***storage*** class refers to the ***permanence*** of a variable and its scope within the program i.e. portion of the program over which variable is recognized.

Storage Classes

There are four different storage classes specified in C:

1.	Auto (matic)	2.	Extern (al)
3.	Static	4.	Register

The storage class associated with a variable can sometimes be established by the location of the variable declaration within the program or by prefixing keywords to variables declarations.

For example:
```
auto    int     a, b;
static  int     a, b;
extern  float   f;
```

8.6.1 Automatic Variables

The variables local to a function are automatic i.e., declared within the function. The scope of lies within the function itself. The automatic defined in different functions, even if they have same name, are treated as different. It is the default storage class for variables declared in a function.

Points to remember:

- The auto is optional therefore there is no need to write it.
- All the formal arguments also have the auto storage class.
- The initialization of the auto-variables can be done:
    - in declarations
    - using assignment expression in a function
- If not initialized the unpredictable value is defined.
- The value is not retained after exit from the program.

Let us study these variables by a sample program given below:

Example 8.4

/* To print the value of automatic variables */

```
# include <stdio.h>
main ( int argc, char * argv[ ])
{
int    a, b;
double  d;
printf("%d",   argc);
a = 10;
b = 5;
d = (b * b) – (a/2);
printf("%d, %d, %f", a, b, d);
}
```

All the variables a, b, d, argc and argv [ ] have automatic storage class.

8.6.2 External (Global) Variables

These are not confined to a single function. Their scope ranges from the point of declaration to the entire remaining program. Therefore, their scope may be the entire program or two or more functions depending upon where they are declared.

Points to remember:

- These are global and can be accessed by any function within its scope. Therefore value may be assigned in one and can be written in another.
- There is difference in external variable definition and declaration.
- External Definition is the same as any variable declaration:
    - Usually lies outside or before the function accessing it.
- It allocates storage space required.
- Initial values can be assigned.
- The external specifier is not required in external variable definition.
- A declaration is required if the external variable definition comes after the function definition.
- A declaration begins with an external specifier.
- Only when external variable is defined is the storage space allocated.
- External variables can be assigned initial values as a part of variable definitions, but the values must be constants rather than expressions.
- If initial value is not included then it is automatically assigned a value of zero.

Let us study these variables by a sample program given below:

Example 8.5

/* Program to illustrate the use of global variables*/

```
#  include <stdio.h>
int gv;                                    /*global variable*/
main  (  )
{
void function1();                    /*function declaration*/
gv = 10;
printf  ("%d is the value of gv before function call\n", gv);
function1( );
printf  ("%d is the value of gv after function call\n", gv);
}
```

```
void function1 ( )
{
gv = 15: }
```

OUTPUT

10 is the value of gv before function call
15 is the value of gv after function call

8.6.3 Static Variables

In case of single file programs static variables are defined within functions and individually have the same scope as automatic variables. But static variables retain their values throughout the execution of program within their previous values.

Points to remember:

- The specifier precedes the declaration. Static and the value cannot be accessed outside of their defining function.
- The static variables may have same name as that of external variables but the local variables take precedence in the function. Therefore external variables maintain their independence with locally defined auto and static variables.
- Initial value is expressed as the constant and not expression.
- Zeros are assigned to all variables whose declarations do not include explicit initial values. Hence they always have assigned values.
- Initialization is done only is the first execution.

Let us study this sample program to print value of a static variable:

Example 8.6

```
/* Program to illustrate the use of static variable*/

#include <stdio.h>

main()
{
int call_static();
int  i,j;
i=j=0;
j = call_static();
printf("%d\n",j);
j = call_static ();
printf("%d\n",j);
j = call_static();
printf("%d\n",j);
}

int    call_static()
{
static  int  i=1;
int   j;
j  =  i;
i++;
return(j);
}
```

OUTPUT

1
2
3

This is because ***i*** is a static variable and retains its previous value in next execution of function call_static( ). To remind you ***j*** is having auto storage class. Both functions main and call_static have the same local variable ***i*** and ***j*** but their values never get mixed.

8.6.4 Register Variables

Besides three storage class specifications namely, Automatic, External and Static, there is a *register* storage class. *Registers* are special storage areas within a computer's CPU. All the arithmetic and logical operations are carried out with these registers.

For the same program, the execution time can be reduced if certain values can be stored in registers rather than memory. These programs are smaller in size (as few instructions are required) and few data transfers are required. The reduction is there in machine code and not in source code. They are declared by the proceeding declaration by register reserved word as follows:

register int m;

Points to remember:

- These variables are stored in registers of computers. If the registers are not available they are put in memory.
- Usually 2 or 3 register variables are there in the program.
- Scope is same as automatic variable, local to a function in which they are declared.
- Address operator '&' cannot be applied to a register variable.
- If the register is not available the variable is though to be like the automatic variable.
- Usually associated integer variable but with other types it is allowed having same size (short or unsigned).
- Can be formal arguments in functions.
- Pointers to register variables are not allowed.
- These variables can be used for loop indices also to increase efficiency.

8.7 TYPES OF FUNCTION INVOKING

We categorize a function's invoking (calling) depending on arguments or parameters and their returning a value. In simple words we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.

The various types of invoking functions are:

- With no arguments and with no return value.
- With no arguments and with return value
- With arguments and with no return value
- With arguments and with return value.

Let us discuss each category with some examples:

TYPE 1: With no arguments and have no return value

As the name suggests, any function which ***has no arguments and does not return any values to the calling function***, falls in this category. These type of functions are confined to themselves i.e. neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function are only program control will be transferred.

Example 8.7

/* Program for illustration of the function with no arguments and no return value*/

/* Function with no arguments and no return value*/

```
#include <stdio.h>
main()
 {
void message();
printf("Control is in main\n");
message();                    /* Type 1 Function */
printf("Control is again in main\n");
 }

void message()
{
  printf("Control is in message function\n");
 }                            /* does not return anything */
```

OUTPUT

```
Control is in main
Control is in message function
Control is again in main
```

TYPE 2: With no arguments and with return value

Suppose if a function does not receive any data from calling function but does send some value to the calling function, then it falls in this category.

Example 8.8

Write a program to find the sum of the first ten natural numbers.

/* Program to find sum of first ten natural numbers */

```
#include <stdio.h>

int cal_sum()
 {
int i, s=0;
for (i=0; i<=10; i++)
s=s + i;
return(s);                    /* function returning sum of first ten natural numbers */
}

main()
 {
int sum;
```

```
sum = cal_sum();
printf("Sum of first ten natural numbers is % d\n", sum);
}
```

OUTPUT

Sum of first ten natural numbers is 55

TYPE 3: With Arguments and have no return value

If a function ***includes arguments but does not return anything***, it falls in this category. One way communication takes place between the calling and the called function.

Before proceeding further, first we discuss the *type of arguments or parameters* here. There are two types of arguments:

- Actual arguments
- Formal arguments

Let us take an example to make this concept clear:

Example 8.9

Write a program to calculate sum of any three given numbers.

```
#include <stdio.h>

main()
{
int a1, a2, a3;
void sum(int, int, int);
printf("Enter three numbers: ");
scanf ("%d%d%d",&a1,&a2,&a3);
sum (a1,a2,a3);    /*  Type 3 function */
}


/* function to calculate sum of three numbers */
void sum (int f1, int f2, int f3)
{
int s;
 s = f1+ f2+ f3;
printf ("\nThe sum of the three numbers is %d\n",s);
}
```

OUTPUT

Enter three numbers: 23 34 45
The sum of the three numbers is 102

Here f1, f2, f3 are ***formal arguments*** and a1, a2, a3 are ***actual arguments***.
Thus we see in the function declaration, the arguments are formal arguments, but when values are passed to the function during function call, they are actual arguments.

Note: The actual and formal arguments should match in type, order and number

TYPE 4: With arguments function and with return value

In this category two-way communication takes place between the calling and called function i.e. a function returns a value and also arguments are passed to it. We modify above Example according to this category.

Example 8.10

Write a program to calculate sum of three numbers.

```
/*Program to calculate the sum of three numbers*/

#include <stdio.h>
main ( )
 {
int a1, a2, a3, result;
int sum(int, int, int);
printf("Please enter any 3 numbers:\n");
scanf ("%d %d %d", & a1, &a2, &a3);
result = sum (a1,a2,a3);   /* function call */
printf ("Sum of the given numbers is : %d\n", result);
}

/* Function to calculate the sum of three numbers */
int sum (int f1, int f2, int f3)
 {
  return(f1+ f2 + f3);  /* function returns a value */
 }
```

OUTPUT

```
Please enter any 3 numbers:
3 4 5
Sum of the given numbers is: 12
```

8.8 CALL BY VALUE

So far we have seen many functions and also passed arguments to them, but if we observe carefully, we will see that we have always created new variables for arguments in the function and then passed the values of actual arguments to them. Such function calls are called ***"call by value".***

Let us illustrate the above concept in more detail by taking a simple function of multiplying two numbers:

Example 8.11

Write a program to multiply the two given numbers

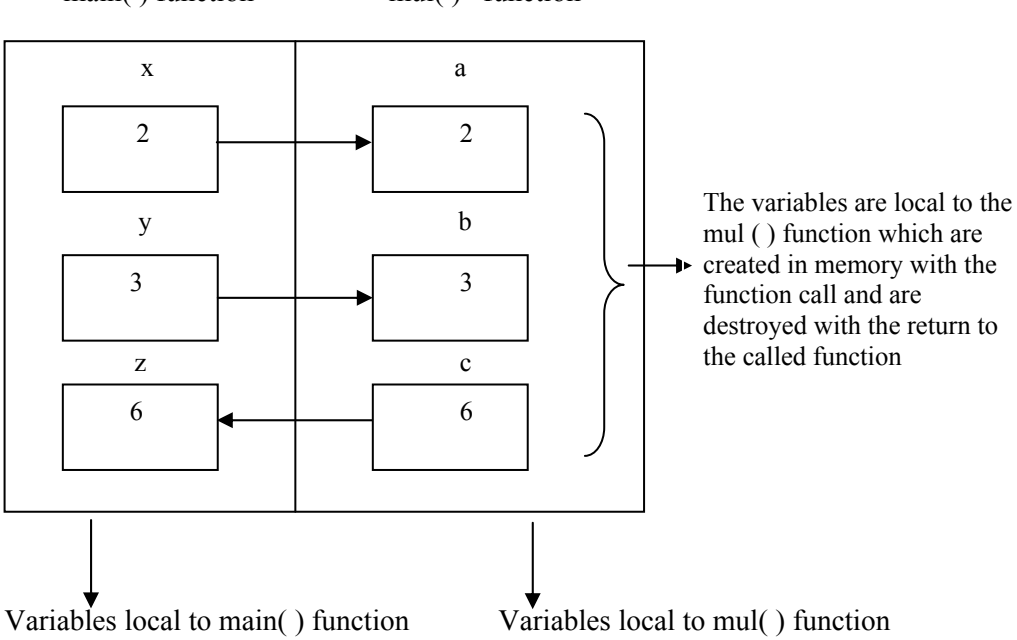
```
#include <stdio.h>
main()
{
int x, y, z;
int mul(int, int);
printf ("Enter two numbers: \n");
scanf ("%d %d",&x,&y);
z= mul(x, y);                /* function call by value */
printf ("\n The product of the two numbers is : %d", z);
}
```

```
/* Function to multiply two numbers */
int mul(int a, int b)
{
int c;
c =a*b;
return(c);    }
```

OUTPUT

```
Enter two numbers:
23  2
The product of two numbers is: 46
```

Now let us see what happens to the actual and formal arguments in memory.



What are meant by local variables? The answer is *local variables are those which can be used only by that function.*

Advantages of Call by value:

The only advantage is that this mechanism is simple and it reduces confusion and complexity.

Disadvantages of Call by value:

As you have seen in the above example, there is separate memory allocation for each of the variable, so unnecessary utilization of memory takes place.

The second disadvantage, which is very important from programming point of view, is that any changes made in the arguments are not reflected to the calling function, as these arguments are local to the called function and are destroyed with function return.

Let us discuss the second disadvantage more clearly using one example:

Example 8.12

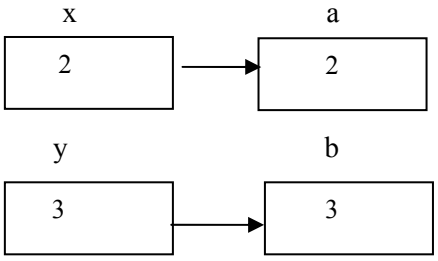
Write a program to swap two values.

```
/*Program to swap two values*/

#include <stdio.h>
main ( )
{
int x = 2, y = 3;
void swap(int, int);

printf ("\n Values before swapping are %d %d", x, y);
swap (x, y);
printf ("\n Values after swapping are %d %d", x, y);
}

/* Function to swap(interchange) two values */
void swap( int a, int b )
{
int t;
t = a;
a = b;
b = t;
}
```

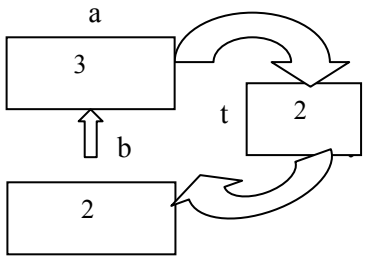
OUTPUT

Values before swap are 2 3
Values after swap are 2 3

But the output should have been 3 2. So what happened?



Values passing from main ( ) to swap() function



Variables in swap ( ) function

Here we observe that the changes which takes place in argument variables are not reflected in the main() function; as these variables namely a, b and t will be destroyed with function return.

- All these disadvantages will be removed by using *"call by reference"*, which will be discussed with the introduction of pointers in UNIT 11.

Check Your Progress 3

1. Write a function to print Fibonacci series upto 'n' terms 1,1,2,3,…..n
   ……
   ……
2. Write a function power (a, b) to calculate a^b
   ……
   ……
   ……

8.9 RECURSION

Within a function body, if the function calls itself, the mechanism is known as **'Recursion'** and the function is known as **'Recursive function'**. Now let us study this mechanism in detail and understand how it works.

- As we see in this mechanism, a chaining of function calls occurs, so it is necessary for a recursive function to stop somewhere or it will result into infinite callings. So the most important thing to remember in this mechanism is that every "recursive function" should have a terminating condition.

- Let us take a very simple example of calculating factorial of a number, which we all know is computed using this formula 5! = 5*4*3*2*1

- First we will write non – recursive or iterative function for this.

Example 8.13

Write a program to find factorial of a number

```
#include <stdio.h>
main ()
{
int n, factorial;
int fact(int);
printf ("Enter any number:\n" );
scanf ("%d", &n);
factorial = fact ( n);   /* function call  */
 printf ("Factorial is %d\n", factorial);
}

/* Non recursive function of factorial */

int fact (int n)
{
int res = 1, i;
for (i = n; i >= 1; i--)
res = res * i;
return (res);
}
```

OUTPUT

```
Enter any number: 5
Factorial is 120
```

How it works?

Suppose we call this function with n = 5

Iterations:

1. i= 5 res = 1*5 = 5
2. i= 4 res = 5*4 = 20
3. i= 3 res = 20*4 = 60
4. i= 2 res = 60*2 = 120
5. i= 1 res = 120*1 = 120

Now let us write this function **recursively**. Before writing any function recursively, we first have to examine the problem, that it can be implemented through recursion.

For instance, we know n! = n* (n – 1)! (Mathematical formula)

Or fact (n) = n*fact (n-1)
Or fact (5) = 5*fact (4)

That means this function calls itself but with value of argument *decreased by '1'*.

Example 8.14

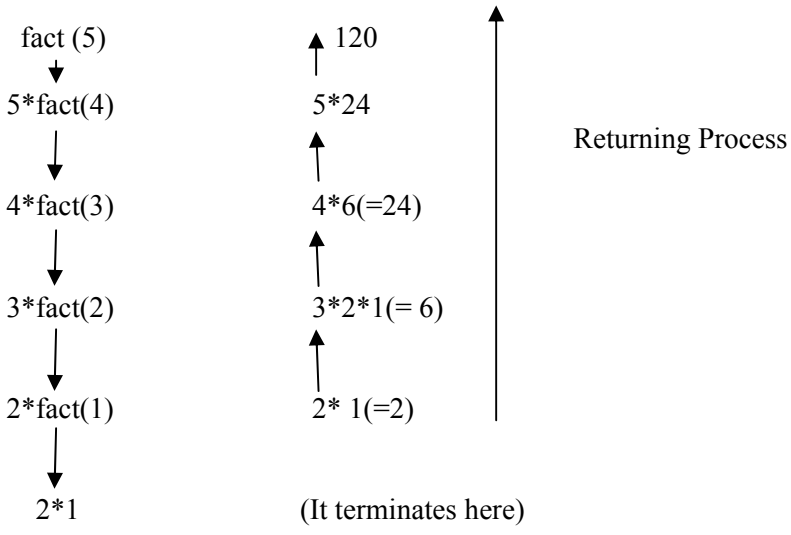
Modify the program 8 using recursion.

```
/*Program to find factorial using recursion*/
#include<stdio.h>
main()
{
int n, factorial;
int fact(int);
printf("Enter any number:  \n" );
scanf("%d",&n);
factorial = fact(n);          /*Function call  */
printf ("Factorial is %d\n", factorial);        }

/* Recursive function of factorial */
int fact(int n)
{
int res;
      if(n == 1)              /* Terminating condition  */
            return(1);
      else
             res = n*fact(n-1);       /* Recursive call */
            return(res);  }
```

OUTPUT
Enter any number: 5
Factorial is 120

How it works?

Suppose we will call this function with n = 5



Thus a recursive function first proceeds towards the innermost condition, which is the termination condition, and then returns with the value to the outermost call and produces result with the values from the previous return.

Note: This mechanism applies only to those problems, which repeats itself. These types of problems can be implemented either through loops or recursive functions, which one is better understood to you.

Check Your Progress 4

1. Write recursive functions for calculating power of a number 'a' raised by another number 'b' i.e. a^b

   ……

   ……

   ……

   …….....

8.10 SUMMARY

In this unit, we learnt about "Functions": definition, declaration, prototypes, types, function calls datatypes and storage classes, types function invoking and lastly Recursion. All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function. We have seen that the functions, which do not return any value, must be declared as *"void"*, return type. A function can return only one value at a time, although it can have many return statements. A function can return any of the data type specified in 'C'.

Any variable declared in functions are local to it and are created with function call and destroyed with function return. The actual and formal arguments should match in type, order and number. A recursive function should have a terminating condition i.e. function should return a value instead of a repetitive function call.

8.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1.
```
/* Function to multiply two integers */
int mul( int a, int b)
{
      int c;
       c = a*b;
      return( c );
}
```

2.
```
#include <stdio.h>
main ( )
{
 int x, y, z;
int mul (int, int);     /* function prototype */
printf ("Enter two numbers");
scanf ("%d %d", &x, &y);
z = mul (x, y);                  /* function call  */
printf ("result is %d", z);      }
```

Check Your Progress 2

1. (a) Valid
   (b) In valid
   (c) Valid
   (d) Valid
   (e) Invalid

Check Your Progress 3

1. /* Function to print Fibonacci Series */

```
void fib(int n)
   {
      int curr_term, int count = 0;
      int first = 1;
      int second = 1;
      print ("%d %d", curr_term);
      count = 2;
      while(count < = n)
        {  curr_term = first + second;
           printf ("%d", curr_term);
           first = second;
           second = curr_term;
           count++;
        }
    }
```

2. /* Non Recursive Power function i.e. pow(a, b) */

```
int pow( int a, int b)
    {
 int i, p = 1;
 for (i = 1; i < = b; i++)
 p = p*a;
 return (p);
        }
```

Check Your Progress 4

1. /* Recursive Power Function */

```
int pow ( int a, int b )
  {       if ( b = = 0 )
             return (1);
        else
             return (a* pow (a, b-1 ));     /*  Recursive call  */
   }

/*  Main Function  */
main ( )
      {
           int a, b, p;
           printf (" Enter two numbers");
           scanf ( "%d %d", &a, &b );
           p = pow (a, b);      /* Function call */
printf ( " The result is %d", p);
        }
```

8.12 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI
2. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill, 2002.
3. Computer Programming in C, *Raja Raman. V*, 2002, PHI.
5. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, TMGH,2002.