Chapter-7

Function

7.1 What is a Function?

A `function` is defined as self-contained block of statements that performs a particular specified job in a program. Actually, this is a logical unit composed of a number of statements grouped into a single unit. It is often defined as a section of a program performing a specific job. A single program in C may include one or more functions defined within it. Thus, every C program can be thought of as a collection of these functions. The function `main()` is always present in each program which is executed first and other functions are optional (i.e. a program may have other user-defined functions or not. It depends upon the nature of the program).

Let us suppose a program wherein a set of operations has to be repeated often, though not continuously, `n` times or so. If they had to be repeated continuously, loops could be used. Instead of inserting the program statements for these operations at so many places, a separate program segment is written and compiled it separately. As many times as it is needed, the segment program is called. The separate program segment is called a function and the program that calls is called the main program. For example: to calculate combination to two numbers `n` and `r`, we have formula: ${}^{n}c_{r} = \frac{n!}{(n-r)!\ r!}$ i.e. we have to calculate factorial of `n`, `n-r` and `r`. If we try to solve this problem without using function, we have to repeat same logic of finding factorial three times. Instead, we can make a general function which calculates factorial of a given number and this function is called three times by passing different numbers to the function as argument (i.e. `n` one time, `n-r` second time and then `r`) to calculate factorial of `n`, `n-r` and `r`.

> A function is a self-contained block of statements that performs a particular specified task.

7.2 Advantages of Functions

a) **Manageability:** The use of function to perform specific job in a program makes easier to write small programs (i.e. small program blocks) and keep track of what they do. It makes programs significantly easier to understand and maintain by breaking them up into easily manageable chunks. The code for each job or activity is placed in individual function such that testing and debugging becomes easy and efficient. Thus, the use of functions in a program helps to manage the code.

b) **Code Reusability:** A single function can be used (i.e. called) multiple times in a single program from different places (i.e. the same code of function is used again and again–reused). It avoids re-writing the same code again and again such that it reduces extra effort. Thus, it helps to reuse the code once written and tested. The C standard library is an example of functions being reused.

c) **Non-redundant (non-repeated) programming (i.e. avoids redundant programming):** A function written once is called multiple times in the program when needed to perform the specified task. The particular job or activity that is to be accessed repeatedly multiple times from several different places within or outside the program is written within function. If the function is not used in such situations, the code for same activity is to be re-written every time. Thus, the use of function avoids the need of redundant programming for the task.

d) **Logical Clarity:** When a single program is decomposed into various well-defined functions, the main program consists of a series of function calls rather than countless lines of code so that the size of main program seems small and program becomes logically clear to understand. It helps to make program easier to write, understand, debug and test.

e) **Easy to divide the work among programmers:** The number of programmers working on one large project can divide the workload on the basis of functions (i.e. one may write one function and other may write another function.) Finally, they can be integrated in the program.

Let us consider an example to compute combination i.e. ${}^{n}C_{r}=n!/((n-r)!\ r!)$. If we are solving this problem without using function, the program will be as below.

Example 7.1

Write a program to compute combination of numbers ${}_{n}C_{r}$ without using function.

```
#include<stdio.h>
#include<conio.h>
int main()
  {
    long f1=1,f2=1,f3=1,comb;
    int n,c,r,i;
    printf("\nEnter n and r!!\t");
    scanf("%d%d",&n,&r);
    for(i=1;i<=n;i++)
      f1*=i;  /*to calculate n!*/
    for(i=1;i<=(n-r);i++)
      f2*=i;     /*to calcualte n-r!*/
    for(i=1;i<=r;i++)
      f3*=i;   /*to calculate r! */
    comb=f1/(f2*f3);
    printf("\nThe combination is %ld",comb);
    getch();
    return 0;
    }
```

Output

```
i.  Enter n and r!! 6 2
    The combination is 15
ii. Enter n and r!! 5 3
    The combination is 10
```

Explanation: Here, to compute combination, we have to calculate factorial of `n`, `n-r` and `r`. The logic and code is similar in each case. We are writing code to calculate factorial of a number three times, one for `n!`, the second one for `n-r!` and the third for `r!`. Thus, this program has repeated code which is not good and efficient. We can make a function to calculate factorial of a number and call this as required multiple times instead of repeating same logic.

Example 7.2

Write a program to calculate combination of two numbers. Use function to calculate factorial of a number.

```
#include<stdio.h>
#include<conio.h>
long factorial(int n)
{
long fact=1;
int i;
for(i=1;i<=n;i++)
    fact*=i;
 return fact;
}

int main()
{
long f1=1,f2=1,f3=1,comb;
int n,c,r;
printf("\nEnter n and r!!\t");
scanf("%d%d",&n,&r);
f1=factorial(n); /* call function factorial() to calculate n!*/
f2=factorial(n-r);/*call again factorial() to calculate n-r!*/
f3=factorial(r); /* call again factorial() to calculate r!*/
comb=f1/(f2*f3);
printf("\nThe combination is %ld",comb);
getch();
return*0;
}
```

Output

```
Enter n and r!! 6 2
The combination is 15
```

Explanation: In this example, a function `factorial()` is defined once and it is called three times to calculate the factorial of `n` , `n-r` and `r`. The detail of defining function will be discussed later of this chapter.

`Calling and Called function:` The `main()` function is entry point for every program and it is called from operating system. Generally, the `main()` function calls other functions. Here, the `main()` function is treated as `calling function` and other functions are known as `called functions`. One user-defined function can call other library function or user-defined function also.

7.3 User-Defined and Library Functions

C functions can be classified into two categories: `user-defined` and `library functions`.

7.3.1 Library Functions (Built-in Functions)

These are the functions which are already written, compiled and placed in C Library and they are not required to be written by a programmer again. The function's name, its return type, their argument number and types have been already defined. We can use these functions as required by just calling them. For example: `printf()`, `scanf()`, `sqrt()`, `getch()` are examples of library functions. The library functions are also known as built-in functions and they are defined within a particular header file.

7.3.2 User-defined Functions

These are the functions which are defined by user at the time of writing a program. The user has choice to choose its name, return type, arguments and their types. In the case of above example 7.1, the function `factorial()` is user-defined function. The task of each user-defined function is as defined by the user (i.e. not fixed as in library functions). A complex C program can be divided into a number of user-defined functions.

What is main() function?

The function `main()` is a user-defined function. The only difference from other user-defined function is that the name of function is defined or fixed by the language. But, the return type, arguments and body of the function can be defined by the programmer as his requirement. This function is executed first when the program starts execution. Thus, the `main()` function is entry point for every C program.

7.4 Components of a Function

7.4.1 Function Definition

One or more statements that describe the specific task to be done by a function is called a function definition. It consists of function header, which defines function's name, its return type & its argument list and a function body, which is a block of code enclosed in parenthesis.

The syntax for function definition is:

```
return_type function_name(data_type variable1, . . . . ., data_type variable n)
        {
         statements;
        return value;
        }
```

If a function doesn't return any value, then it's `return _type` is `void`.

```
void function_name(data_type variable1, . . . . . ,  data_type variable n)
        {
        statements;
        . . .
        }
```

The first line of the function definition is known as function `declarator or header`. This is followed by function body which is composed of the statements that make up the function, delimited by braces.

Some Examples of Function definitions	

`int add(int a, int b)  /* function header */`
`{ /*function body starts from here */`
`int sum;  /*function body  */`
`sum=a+b;  /*function body */`
`retun(sum); /*function body */`
`}  /*function body ends here*/`
	The function `add()` takes two integers as arguments and returns sum of them in integer form.

`float areaOfCircle(float radius)`
`{`
`return (3.1428*radius*radius);`
`}`
	The function definition of `areaOfCircle()` which takes radius in `float` and returns area of circle in `float`.

`void display(int a)`
`{`
`printf("\nThe value is %d",a);`
`}`
	The function definition of `display()` function which takes an `integer` and displays it to the screen. It does not return anything.

The return type in function definition is optional. The default value is `int`. Thus the function header `int add(int a, int b)` is same as `add(int a, int b)`. The arguments `variable1, variable2, variable3,……., variablen` present in function definition are called `formal arguments` or `parameters` or `formal parameters`, because they represent the names of the data item that are transferred into the function from the calling portion of the program. These are the `local` variables for the function.

7.4.2 Function Declaration or Prototype

The function declaration or prototype is model or blueprint of a function. If a function is used before it is defined in the program, then function declaration or prototype is needed to provide the following information to the compiler.

i. The name of the function
ii. The type of the value returned by the function
iii. The number and the type of arguments that must be supplied while calling the function

When a user-defined function is defined before its use (i.e. function call statement), there is no need for function declaration.The syntax for function declaration is:

```
return_type function_name(type1, type2, type3,….,typen);
```

Here, `return_type` specifies the data type of the value returned by the function. A function can return value of any data type. If there is no return value, the keyword `void` is used. The function declaration and declarator or header in function definition must use the same function name, number of arguments, argument types and return types. Some examples of function prototype are:

```
i.   int add(int a, int b);  or int add(int , int );
ii.  float areaOfCircle(float radius);
iii. void display(int a);
iv.  int max(int n1, int n2);
v.   double power(int m, int n);
vi.  multiply(int,int);
```

The `function prototype` and the first line of function definition (i.e. `function header`) must be similar in term of function name, return type, type of arguments and number of arguments. But the function prototype may not have argument name (i.e. only argument type is sufficient) while it is mandatory in function header. Again, the function prototype has `semicolon` at the end. It is just a declaration and information to compiler.

7.4.3 Return Statement

The function defined with its return type must return a value of that type. For example: if a function has return type `int`, it returns an integer value from the function to the calling program. The "`return`" is a keyboard used at the end of every function body to return the specified value from the function, is called a `return` statement. If a function has `void` as return type, there is no need of `return` statement.

The main functions of `return` statement in a program are:

i. It immediately transfers the control back to the calling program after executing the `return` statement (i.e. no statement within function body is executed after the `return` statement.)
ii. It returns the value to the calling function.

The syntax for return is:

```
return (expression);
```

Where `expression` must evaluate to a value of the type specified in the function header for the return value. The `expression` can be any desired expression as long as it ends up with a value of required type. The value of `expression` is returned to the calling portion of the program. The `expression` is optional. If the `expression` is omitted, the return statement simply causes the control to revert back to the calling portion of the program without any transfer of information. A function definition can include multiple `return` statements, each containing a different expression. But, a function can return only one value to the calling portion of the program via return statement.

A function may or may not return any value. If a function doesn't return a value, the `return type` in the function definition and declaration is specified as `void`. Otherwise, the `return type` is specified as a valid data type. A `return` statement at the end is optional for functions that don't return values. Some examples of `return` statements:

i. `return 0;`
ii. `return;`
iii. `return 10;`
iv. `return a+b;`
v. `return (a+b);`

7.4.4 Accessing/Calling a function

A function is called or accessed by specifying its name, followed by a list of arguments enclosed in parentheses and separated by commas. For example, the function `add()` with two arguments is called by `add(a,b)` to add two numbers. If function call doesn't require any arguments, an empty pair of parentheses must follow the name of function. The arguments appearing in the function call are referred as `actual arguments`. In function call, there will be one argument for each formal argument. The value of each argument is transferred into the function and assigned to the corresponding formal argument. If a function returns a value, the returned value can be assigned to a variable of type same as return type of the function. When a function is called, the program control is passed to the function. Once the function completes its task, the program control is passed back to the calling function. The general form of the function call statements are:

i. if function has parameters but it does not return value
```
function_name(variable1, variable2,.....);
```
ii. if function has no arguments and it does not return value
```
function_name();
```
iii. If function has paramenters and it returns value
```
variable_name=function_name(variable1, varaible2,.....);
```
iv. If function has no parameters but returns value
```
variable_name=function_name();
```

The following points to be taken into consideration when function call statements are used.

i. The `function name` and the `type & number of the variables` (or arguments) listed in the function call statement must match with that of the function declaration statement and the header of the function definition.

ii. The names of the variables in the function declaration, function call statement and that in the header of the function definition may be different.

iii. By default, arguments (or variables) are always passed by value in C function call. This means that local copies of the values of the arguments are passed to the function.

iv. Arguments (or variables) present in the form of expressions are evaluated and converted to the type of the formal parameters at the beginning of the body of the function.

v. A variable may be assigned the value returned by the function, after it is executed, using the function call statement, provided the returned type is not a void.

Example 7.3

Define a function `int greater (int, int)` to find the greatest number among two numbers. Write a program that uses this function to find the greatest number among three numbers.

```
#include<stdio.h>
#include<conio.h>
int greater(int x, int y)/* function header*/
        {
         if(x>y)
            return (x);    /*return statement*/
         else
            return (y);   /* return statement */
        }
int main()
    {
     int a, b,c,d,e;
     printf("Enter three numbers");
     scanf("%d%d%d",&a,&b,&c);
     d=greater(a,b);    /* function call*/
     e=greater(d,c);  /* same function is called again*/
     printf("The greatest number is:%d",e);
     getch();
    }
```

Output

```
Enter three numbers:34 56 10
The greatest number is:56
```

Explanation: Let us consider `34, 56, 10` are three numbers of which we have to find out the greatest number. The calling mechanism is illustrated by the following sequences while running the program. Each line of the program can be traced into (i.e. debug line by line) by pressing F7 in Turbo C. The number at the end of statement specifies the number of steps in which the statement is executed.

```
int greater(int x, int y)----------6--10
{
if(x>y)--------------------7--11
return (x);----------------12
else
return (y);------------8
}
void main()  ----------------------------------------1
{
int a, b,c,d,e;
clrscr();---------------------------------2
printf("Enter three numbers");--------3
scanf("%d%d%d",&a,&b,&c);-------4
/* let a=34, b=56 & c=10 */
d=greater(a,b);---------------5
e=greater(d,c);---------------9
printf("The greatest number is%d",e);--13
getch();--------------------------------14
}
```

Here, execution of the program starts from the function `main()`. When the function call statement `d=greater(a,b)` is encountered, the control goes to function definition. The value of actual arguments are copied into formal arguments `x` and `y` in function definition. The appropriate return statement is executed within function definition depending upon the value of `x` and `y` and then control comes back to calling program, returned value is assigned to a variable `d`. Thus, `d` stores greater number of `a` and `b`. Then, the function `greater()` is again called in statement `e=greater(d,c)`. The control goes to function definition again. The values of `d` and `c` are copied into formal arguments `x` and `y` in function definition. When the function returns greater value, it is assigned to the variable `e`. which will be greatest number among three numbers `a,b` and `c`.

When a function is specified (coded) before the main program in programming environment, no prototyping is required. But when the function is coded below the main program, a function prototyping is required. For example the above program can also be written as:

```
#include<stdio.h>
#include<conio.h>
//function prototype
int greater(int,int);
void main(void)
{
//main body code here
...
//function call
d=greater(a,b);
...
}
```

```
//function
int greater (int x, int y)
{
//function code here
//return value
}
```

7.4.5 Function Parameters (Arguments)

Function parameters are the means for communication between the calling and the called functions. They can be classified into `formal parameters` and `actual parameters`. The formal parameters are the parameters given in the function declaration and function definition. The actual arguments or parameters are specified in the function call. In above example (Example 7.3), the parameters `a` and `b` are actual arguments which are used in calling function while parameters `x` and `y` are formal arguments or parameters which are defined in function definition. The name of formal and actual arguments need not be same but data type and number of them must match.

The formal arguments used in a function are generally local variables for the function. Thus, they can not be used in calling functions (like `main()`). Again, the variables defined within calling function can not be used in the called function. In other words, the calling function and called functions are two different worlds. If we have to supply values from calling function to called function, the values are supplied as actual parameter which are copied to formal arguments of called function. Thus, actual arguments act as sender and formal arguments act as receiver for the data values. Actually, the arugments act as communication medium between calling and called function.

7.5 Category of Functions according to Return Values & Arguments

According to arguments and return values present in functions, we can categorize the functions in three categories.

Category 1: Functions with no arguments and no return values

Category 2: Function with arguments and no return values

Category 3: Function with arguments and return values

7.5.1 Functions with no Arguments and no Return Values

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it doesn't return a value, the calling function does not receive any data from the called function. Thus, in such type of functions, there is no data transfer between the calling function and the called function. This type of function is defined as

```
void  function_name()
   {
     /* body of function */
   }
```

The keyword `void` means the function does not return any value. There is no arguments within parenthesis which implies function has no argument and it does not receive any data from the called function.

Example 7.4

Write a program to illustrate the "functions with no arguments and no return values"

```
#include<stdio.h>
#include<conio.h>
void addition()
    {
    int a,b,sum;
    printf("\nEnter two numbers:\t");
    scanf("%d%d",&a,&b);
    sum=a+b;
    printf("\nThe sum is %d",sum);
    }
int main()
    {
    addition();
    getch();
    return 0;
    }
```

Output

```
Enter two numbers:      45 90
The sum is 135
```

Explanation: In this example, the function `addition()` has neither arguments nor return values. There is no data transfer between function `addition()` and `main()`. The function itself reads data from the keyboard and display in the screen. As it does not return any value, there is no `return` statement within the function.

7.5.2 Functions with Arguments but no Return Value

The function of this category has arguments and receives the data from the calling function. The function completes the task and does not return any values to the calling function. Such type of functions are defined as

```
void function_name(argument list)
    {
    body of function
    }
```

Example 7.5

Write a program to illustrate the "functions with arguments but no return values".

```
#include<stdio.h>
#include<conio.h>
void addition(int a, int b)
    {
    int sum;
    sum=a+b;
    printf("\nThe sum is %d",sum);
    }
```

```
int main()
    {
    int a,b;
    printf("\nEnter two numbers:\t");
    scanf("%d%d",&a,&b);
    addition(a,b);
    getch();
    return 0;
    }
```

Output

```
Enter two numbers:      50  40
The sum is 90
```

Explanation: In this example, the function `addition()` does not read data from keyboard directly. It receives the data from `main()` function i.e. calling function. It adds these two numbers and displays the sum in the screen but it does not return any value to the calling function. Thus, there is no return statement within the function `addition()`.

7.5.3 Function with arguments and return values

The function of this category has arguments and receives the data from the calling function. After completing its task, it returns the result to the calling function through `return` statement. Thus, there is data transfer between called function and calling function using return values and arguments. These type of functions are defined as

```
return_ type function_name(argument list)
    {
    body of the function;
    return value;
    }
```

Example 7.6

Write a program to illustrate the "functions with arguments and return values".

```
#include<stdio.h>
#include<conio.h>
int addition(int a, int b)
    {
    int sum;
    sum=a+b;
    return sum;
    }
int main()
    {
    int a,b,sum;
    printf("\nEnter two numbers:\t");
    scanf("%d%d",&a,&b);
    sum=addition(a,b);
    printf("\nThe sum is\t%d",sum);
```

```
    getch();
    return 0;
    }
```

Output

```
Enter two numbers:    78  56
The sum is      134
```

Explanation: In this example, the function `addition()` has return type `int` and it has two arguments of type `int`. It receives two numbers from the calling function and adds them. The sum is returned to the calling function (i.e. `main()`) which assigns the returned value to some variable and displays to the screen.

7.6 Different Types of Function Calls

The arguments in function can be passed in two ways: pass arguments by value and pass arguments by address or reference or pointers.

7.6.1 Function call by value (or Pass arguments by value)

When values of actual arguments are passed to a function as arguments, it is known as function call by value. In this type of call, the value of each actual argument is copied into corresponding formal argument of the function definition. The content of the arguments in the calling function are not altered, even if they are changed in the called function. The function is called using the syntax:

```
function_name(value_of_argument1, value_of_argument2,.....);
```

Examples: Let us consider a function with prototype `void Addition(int, int)` which receives two arguments and doesnot return any value. The function can be called passing values as:

```
Addition(20,30);
Addition(a,b);//where a and b are two int variables
```

Example 7.7

Write a program to illustrate function call by value.

```
#include<stdio.h>
#include<conio.h>
void swap(int , int); /* function prototype */
int main()
   {
    int a, b;
    a=99; b=89;
    printf("\nBefore function calling, a and b are: %d\t%d",a,b);
    swap(a,b); /* function call by value */
    printf("\nAfter function calling, a and b are: %d\t%d",a,b);
    getch();
```

```
    return 0;
    }
void swap(int x, int y)
    {
    int temp;
    temp=x;
    x=y;
    y=temp;
    printf("\nThe values within functions are:%d\t%d",x,y);
    }
```

Output:

```
    Before function calling, a and b are:    99    89
    The values within functions are:         89    99
    After function calling, a and b are:     99    89
```

Explanation: In this example, the values of `a` and `b` are passed in function `swap()` by value. The value of `a` is copied into formal argument `x` and the value of `b` is copied into formal argument `y`. The copy of values of `a` and `b` to `x` and `y` means the original values of `a` and `b` remain same and these values are copied into the variables `x` and `y` also. Thus, when `x` and `y` are changed within the function, the values of the variables `x` and `y` are changed but the original values of `a` and `b` are not changed.

7.6.2 Function Call by Reference (Pass Argument by Address)

In this type of function call, the address of a variable or argument is passed to a function as argument instead of actual value of variable. A function can be called by passing address in its arugment as:

```
function_name(address_of_argument1, address_of_argument2, …..);
```

Examples: Let us consider a function with prototype `void Addition(int*, int*)` which receives addresses of two arguments and doesn't return any value. The prototype of the function has formal argument in the form of `int*` that denotes pointer type argument. As such type of function receives address from calling function, the formal argument must be pointer type. It is because only the pointer can receive or assign address of another variables. The function can be called passing address as:

`Addition(&a,&b);//`where `a` and `b` are `int` variables and `&a` and `&b` are addresses of them.

For clear understanding of this, concept of pointer is necessary which we will study later in detail. The overview of pointer is given below to understanding the call by reference.

What is Pointer?

A pointer is a special type of variable. It is special because a normal variable assigns data value but the pointer variable assigns a memory address of another variable. A pointer can have any name that is legal for other normal variable and it is declared in the same fashion like other variables but is always preceded by ' *' (asterisk) operator. Thus, the pointer variables are declared as:

i. `int *a`

ii. `float *b;`

iii. `char *c;`

Here, `a` , `b` and `c` are pointer variables which store address of `int`, `float` and `char` variables. Thus, the following assignments are valid.

```
int x,
float y;
char c;
```

i. `a=&x;` /* the address of `x` is assigned to pointer variable `a` */

ii. `b=&y;` /* the address of `float` variable `y` is stored in pointer variable `b` of type `float` */

iii. `c=&c;` /* the address of `char` variable `c` is assigned to pointer variable `c` of type `char` */

Example 7.8

Write a program to swap the values of two variables using call by reference.

```
#include<stdio.h>
#include<conio.h>
void swap(int * , int *);   /* function prototype */
   /* here formal arguments must be of type pointer as they receive address of variables*/
int main()
    {
     int a, b;
     a=99; b=89;
     printf("\nBefore swap, a and b are: %d\t%d",a,b);
     swap(&a,&b);     /*  function call by reference*/
     /* the address of a and b is passed instead of values*/
     printf("\nAfter swap, a and b are: %d\t%d",a,b);
     getch();
     return 0;
    }
void swap(int *x, int *y)
     {
       int temp;
      temp=*x;
     /*Here, *x represent the value at address pointed by variable x
        [here x points a hence *x is equivalent to a] */
      *x=*y;
      *y=temp;
       }
```

Output:

```
Before swap, a and b are: 99     89
After swap, a and b are:  89     99
```

Explanation: In this example, the address of variables (i.e. `&a` and `&b`) are passed to the function. These addresses are received by corresponding formal arguments in function definition. To receive addresses, the formal arguments must be of type pointers which store address of variables. The address of `a` and `b` are copied to the pointer variables `x` and `y` in function definition. When the values in addresses pointed by these pointer variables are altered, the values of original variables are also changed as the content of their addresses have been changed. Thus, while arguments are passed in function by reference, the original values are changed if they are changed within function.

7.7 Recursive Function

A recursive function is one that calls to itself to solve a smaller version of its task until a final call which does not require a self-call. Thus, a function is known as recursive function if it calls to itself. The recursion in programming is a technique for defining a problem in terms of one or more smaller versions of the same problem. The solution of the problem is built on the results from the smaller versions. The recursion is used for repeatitive computations in which each action is stated in term of previous result. Many iterative or repeatitive problems can be written in recursive form. But it is not necessary that all problems can be solved using recursion. To solve a problem using recursive method, the following two conditions must be satisfied.

i. Problem could be written or defined in term of its previous result.

ii. Problem statement must include a stopping condition (i.e. we must have an `if` statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.)

Example 7.9

Write a program to find the factorial of a number using recursive method.

```
#include<stdio.h>
#include<conio.h>
long int factorial(int n)
    {
     if(n==1)
        return(1);
      else
         return (n*factorial(n-1));
    }
int main()
      {
       int num;
       printf("Enter a number:");
       scanf("%d",&num);
       printf("The factorial is %ld", factorial(num));
       getch();
       return 0;
      }
```

Output

```
Enter a number:5
The factorial is 120
```

Explanation: Here, when 5 is passed in function factorial(), the control goes to function definition. Within body of function definition of factorial(), the same function is again called by using the statement n*factorial(n-1) i.e. to find the factorial of 5, it is expressed in terms of 5*4!. Its logic is as

```
5!
5*4!
5*4*3!
5*4*3*2!
5*4*3*2*1!
5*4*3*2*1
```

When factorial() receives argument 1, it returns 1 such that the value of 2! is calculated as 2*1!= 2*1=2. When the factorial of 2 is calculated, then factorial of 3 is calculated as 3*2!=3*2=6. Then 4!=4*3!=4*6=24 and finally 5!=5*4!=5*24=120 is calculated.

Here, the arguments are passed in order 5, 4, 3, 2,1 in function factorial() but the calculation is done in reverse order i.e. 1! is computed first and 5! is computed at last. This process Last In First Out is called stack operation. Thus, recursive functions use stack operations.

Example 7.10

Write a program to generate the following Fibonacci series using recursive method. [The Fibonacii series is: 0 1 1 2 3 5 8 13 21 34..... **to** n **terms (i.e. firs term is** 0, **second term is** 1, **third term is sum of** first **and** second, **fourth term is sum of** second **and** third **and so on)]**

```
#include<stdio.h>
#include<conio.h>
int fib(int n)
 {
   if(n==1)
      return 0;
   else if(n==2)
      return 1;
   else
    return(fib(n-1)+fib(n-2));
 }
int main()
 {
  int terms,i;
  printf("\nHow many terms do we need?\t");
  scanf("%d",&terms);
  for(i=1;i<=terms;i++)
```

```
        printf("\t%d",fib(i));
    getch();
    return 0;
}
```

Output

```
How many terms do we need?  10
0   1   1   2   3   5   8   13   21   34
```

Explanation: In Fibonacci series, first two terms are `0` and `1`. Then, sum of `0` and `1` (i.e. `1`) is third term. The sum of `1`(second term) and `1` (third term) will be fourth term i.e. `2`. Then, the sum of `1` and `2` will be fifth term. This process will be repeated up to `n` terms. In recursive method, we can apply the logic: `if number of term is 1 (i.e. first term), then return 0. If number of term is 2 (i.e. second term), then return 1; and in other case, return the sum of previous two terms.`

Difference between Recursion and Iteration

Recursion	Iteration
i. Without using loop, a function calls to itself until certain condition is satisfied and perform repeated task.	i. Loop is used to perform repeated task.
ii. Recursion is a top-down approach to problem solving; It divides the problem into small pieces.	ii. Iteration is like a bottom-up approach; It begins with what is known and from this it constructs the solution step by step.
iii. A function calls to itself until certain condition is satisfied.	iii. A function does not call to itself.
iv. A problem can be written or defined in term of its previous result to solve a problem using recursion.	iv. It is not necessary to define a problem in term of its previous result to solve using iteration.
v. All problems can not be solved using recursion. [Two conditions must be satisfied]	v. All problems can be solved using iteration.

7.8 Overview of Local, Global, Static and Register Variables

7.8.1 Local Variables (automatic or internal variables)

The automatic variables are always declared within a function or block and are local to the particular function or block in which they are declared. As local variables are defined within a body of the function or the block, they are accessible from the same block only (i.e. other functions can not access these variables.) The compiler shows errors in case other functions try to access the variables. The local variables are created when the function is called and destroyed automatically when the function is exited. The keyword `auto` may be used for storage class specification while declaration of variable. A variable declared inside a function without storage class specification `auto` is , by default, an automatic variable.

Default Initial value of such type of variable is an unpredictable value which is often called garbage value. The scope of it is local to the block in which the variable is defined. Again, its life is till the control remains within the block in which the variable is defined.

> Initial value: This is the value assigned to a variable by the language itself when no value is assigned to the variable explicitly by the programmer.
>
> Scope: Scope of a variable can be defined as the region over which the variable is visible or valid.
>
> Life Time: The period of time during which memory is associated with a variable is called the life time or extent of the variable.

Example 7.11

Write a program to illustrate local variable.

```
#include<stdio.h>
#include<conio.h>
long int fact(int n)
    {
     int i;
     long int f=1;
     for(i=1;i<=n;i++)
            f*=1;
      return(f);
     }
int main()
    {
    int num=5;
    printf("The factorial of 5 is%ld",fact(num));
    getch();
    return 0;
    }
```

Explanation: Here, the variables n, i , and f are local to function fact() and are unknown to main() function. Similarly, num is local variable to the function main() and unknown to function fact().

7.8.2 Global Variables (External)

Variables that are accessible, alive and active throughout the entire program are known as external variables. They are also known as global variables. The external or global variables are declared outside any block or function. Unlike local variables, global variables can be accessed by any function in the program. The default initial value for these variables is zero. The scope is global (i.e. within the program). The life time is as long as the program's execution doesn't come to an end. For example:

```
int roll;
float marks;
int main()
      {
       .....
       .....
      }

int fun1()
      {
       .....;
      }
```

Here, the variables `roll` and `marks` are declared outside the blocks. So, it is accessible within `main()` as well as `fun1()`. These two variables are `global` variables.

Example 7.12

Write a program to illustrate the global variables.

```
#include<stdio.h>
#include<conio.h>
int a=10;
void fun()
    {
     a=20;
     printf("\t%d",a++);
    }
int main()
    {
    printf("\t%d",a);
    fun();
    printf("\t%d",a);
    return 0;
   }
```

Output:

```
10      20      21
```

Explanation: Here, `a` is global variable and it is recognized within `main()` function as well as user-defined function `fun()` and can be used anywhere in the program.

Once a variable has been declared as global, any function can use it and change its value. Thus, only those variables are declared as global which are to be shared among different functions. The global variable is visible only from the point of declaration to the end of the program. Thus, if external variable is to be defined after certain function or block, we can use `extern` keyword.

Examples:

```
extern int num;
int main()
{
 num = 10;
 return 0;
}
```

Explanation: This program throws error in compilation. Because `num` is declared but not defined anywhere. Essentially, the `num` isn't allocated any memory. And the program is trying to change the value to `10` of a variable that doesn't exist at all.

Another Example:

```
#include "somefile.h"
extern int num;
int main()
{
 num = 10;
 return 0;
}
```

Explanation: Supposing that `somefile.h` has the definition of variable `num`. This program will be compiled successfully.

```
#include<stdio.h>
float marks;
int main()
{
  extern float marks;
  ....;
}
func1()
{
  extern float marks;
  ------
}
```

Explanation: The `extern` declaration doesn't allocate storage space for variables. The external declaration of `marks` inside the function informs the compiler that `marks` is a `float` type external variable defined somewhere else in the program or other included file..

7.8.3 Static Variables

This is another class of local variable. A `static` variable can only be accessed from the function in which it was declared, like a local variable. The static variable is not destroyed on exit from

the function; instead its value is preserved, and becomes available again when the function is next time called. The static variables are declared as local variables, but the declaration is preceded by the word `static` e.g. `static int counter;`

The static variables can be initialized as normal; the initialization is performed only once, when the program starts up. The default `initial value` for this type of variable is `zero` if user doesn't initialize its value. Its `scope` is local to the block in which the variable is defined. Again, the `life time` is global (i.e. its value persists between different function calls).

Examples 7.13

Write a program to illustrate static variables.

```
increment()
{
int i=1;
printf("%d\n",i);
i++;
}
void main()
{
increment();
increment();
increment();
increment();
}
```

```
Output
1
1
1
1
```

```
increment()
{
static int i=1;
/* here  i is static variable*/
printf("%d\n",i);
i++;
}
void main()
{
increment();
increment();
increment();
increment();
}
```

```
Output
1
2
3
4
```

Explanation: The static variable persists its value of previous calling. Here, the value of `i` is initialized by `1` when it is called first time and its value will be increased by one (i.e. `i=2`) due to statement `i++`. When the function `increment()` is called next time, the previous value of `i` (i.e. 2) will be used (i.e. not initialized again by `1`). On the another hand, when `i` is simply declared as local variable, it is initialized by `1` every time. So, output is same in each call.

7.8.4 Register Variables

`Register` variables are special case of `automatic` variables. Automatic variables are allocated storage in the memory of the computer; however, for most computers, accessing data in memory is considerably slower than processing in the `CPU`. These computers often have small amounts of storage within the CPU itself where data can be stored and accessed quickly. These storage cells are called `registers`. Normally, the compiler determines what data is to be stored in the registers of the CPU at what times. However, the C language provides the storage class register so that the programmer can `'suggest'` to the compiler that particular automatic variables should be allocated to CPU registers, if possible. Thus, register variables provide a certain control over efficiency of program execution. Variables which are used repeatedly or whose access times are critical may be declared to be of storage class `register`.

Register variables behave in every other way just like automatic variables. They are allocated storage upon entry to a block; and the storage is freed when the block is exited. The scope of register variables is local to the block in which they are declared. Rules for initializations for register variables are the same as for automatic variables.

Example:

```
int main()
 {
 register int a=10;/* the variable is local. The difference is that it
 may allocate CPU memory.*/
  ..................
  ..................
  return 0;
}
```

Difference between local, global and static variables

Local variables	Global Variables	Static Variables
i. The local variables are declared within function or blocks. The scope is only within the function or block in which they are defined.	i. The global variables are defined outside the functions so that their scope is throughout the program.	i. The static variables are special case of local variables. Thus, static variables are also defined inside the functions or blocks.
ii. The initial value is unpredictable or garbage value.	ii. The initial value is zero	ii. The initial value is zero.
iii. The life time is till the control remains within the block or function in which the variable is defined. The variable is destroyed when function returns to the calling function or block ends.	iii. The life time is till program's execution doesn't come to an end (i.e. variables are created when program starts and destroyed. when program ends).	iii. Its value persists between different function calls. Thus life time is same as global variables.
iv. The optional keyword auto may be used to define it.	iv. The optional keyword extern may be used to define it.	iv. The mandatory keyword static is used to define static variables.