UNIT-III

Functions:

A function is a self contained block of statements that perform a coherent task of some kind. The general form of a function is as follows:

```
return type function-name (Parameter list)
{
        // body of the function
}
```

The **return type** specifies the type of the data that the function returns. A function may return any type of data except an array.

The **parameter list** is a comma separated list of variable names and their associated types.

The parameters receive the value of the arguments when the function is called. A function can be without parameters, in which case the parameter list is empty. An empty parameter list can be explicitly specified as such by placing the keyword 'void' inside the parenthesis.

In variable declarations, you can declare several variables to be of the same type by using a comma separated list of variable names. In contrast, all functions parameters must be declared individually, each including both the type and name. That is, the parameter declaration list for a function takes this general form:

function-name (type var_name1, type var_name2,…. type var_nameN)

Ex:- f(int i, int j, int k) /*correct*/

f(int i, k, float j) /*incorrect, k must have its own type spcifier */

/*program with functions*/

```
#include<stdio.h>
void message( );
void demo( );/*function prototype declaration*/
void main( )
{
  message( );
  demo( ); /*function call*/
  printf("ROM");
}
void message( ) /*function definition*/
{
  printf("motherboard\n");
}
void demo( )
{
   printf("RAM\n");
}
```

O/P:-

```
motherboard
RAM
ROM
```

A number of conclusions on functions:

- A C program is a collection of one more functions.
- If a C program contains more than one function, it must be main( ).
- If a C program contains more than one function, then one of these functions must be main(0, because program execution always begins with main( ).
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in main( ).
- After each function has done its thing, control returns to main( ). When main( ) runs out of statements and function calls, the program ends.

  One function can call another function that it has already called but has in the meantime left temporarily in order to call a third function which will sometime later call the function that has called it.

Ex:

```
#include<stdio.h>

void fun1( );
void fun2( );
void fun3( );
void main( )
{
   printf("i am in main\n");
   fun1( );
   printf("i am finally back in main\n");
}
void fun1( )
{
   printf("i am in function1\n");
   fun2( );
   printf("i am back in function1\n");
}
void fun2( )
{
   printf("i am in function2\n");
   fun3( );      }
```

```
        void fun3( )
        {
          printf("i am in function3\n");
        }
o/p:    i am in main
        i am in function1
        i am in function2
        i am in function3
        i am back in function1
        i am back in main.
```

Use of functions: -

- Writing functions avoids rewriting the same code over and over. Suppose we have a section of code in our program that calculates area of a triangle. If later in the program we want to calculate the area of a different triangle, we won't like it if you are required to write the same instructions all over again. Instead, we would prefer to jump back to the place from we left off. This section of code is nothing but a function.
- By using functions functions it becomes easier to write programs and keep track of what they are doing. Separating the code into modular functions also makes the program easier to design and understand.

Passing values between functions: -

Consider the following program, In this program in main ( ) we receive the values of a, b through the input device and then output the swapping of a and b. However, the mechanism of swapping is done in a different function called swap ( ).

If Swapping is done in swap ( ) and values a, b are received in main ( ), then we must pass on these values to swap ( ), and once swap ( ) process the swapping. We must return it from swap ( ) back to main ( ).

```
/* Program for Swapping of two values*/

#include<stdio.h>
void swap(int x, int y);
void main( )
{
  int a,b;
  printf("enter a,b values : \n");
  scanf("%d%d",&a,&b);
  printf("the values before swapping are ");
  scanf("%d%d",a,b);
  swap(a,b);
  printf("values after swapping are");
  printf("%d%d",a,b);
  getch( );     }
```

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

```
O/P : enter a,b values : 2 5
      Values before swapping are   a=2   b=5
      values after swapping are   a=5   b=2
```

- In the above program, from the function main ( ), the values of a and b are passed to the function swap ( ) by making a call to the function swap and mentioning a and b in the parentheses.

  swap (a, b);

  In the swap ( ) function these values get collected in variables x and y.

  void swap (int x,int y)

- The variables a and b are called as 'actual arguments' where as the variables x and y are called 'formal arguments'. Any no. Of arguments can be passed to a function being called. However, the type, order and no. Of the actual arguments and formal arguments must always be same.

  Instead of using different variable names x and y we could have used the same variable names a and b. But the compiler would still treat them as different variables since they are in different functions.

User-defined functions:-

The function declaration which needs to be done before the function call gives the whole picture of the function that's needs to be defined later. The declaration mentions the name of the function, return type and type and order of the parameters.

The function definition contains the code needed to complete the task. The declaration uses only the header of the function definition ended in a semicolon where the formal parameter names are optional.

Basic Function Designs:-

A function name is used three times: for declaration, in a call and for definition. We classify the basic function designs by their return values and their parameter lists. Functions either return a value is known as void functions. Some functions have parameters and some don't. Combining return values and parameter lists results in four basic designs: void functions with no parameters, void functions with parameter functions that return a value but have no parameters, and functions that return a value and have parameters.

Void functions without parameters: -

A void function can be written with no parameters. The message function in the following program receives nothing and returns nothing. It has only a side effect displaying the message, and is called only for that side effect.

Because a void function does not have a value, it can be used only as a statement, it cannot be used in an expression. Examine the call to the message function in program, this call stands alone as a statement. Including this call in an expression, as shown below would be an error.

```
Result=message ( );   //error. Void function.
```

/*program using functions without parameters*/

```
#include<stdio.h>
void message(void);
int main(void)
{
  message( );
  return 0;
}
void message(void)
{
  printf("hello");
  return;
}
```

O/P: hello

Void functions with parameters: -

Now let's call a function that has parameters but still returns void. The show one as in the following program, receives on integer parameter. Since this function returns nothing to the calling function, main its return type is void. However, while show one returns no values, it does have a side effect. The parameter value is printed to the monitor.

In the following program, note that the name of the variable in main (i.e. a) and name of the parameter in show one (x) do not have to be the same if that makes it easier to understand the code. In program we use show one to demonstrate that a function can be called multiple times.

Ex:
```
#include<stdio.h>
void showone (int x)
int main (void)
{
   int a;
  a=5;
  showone(a);
  a=25;
```

```
        showone(a);
     }
     void showone(int x)
     {
        printf("%d\n", x);
        return;
     }
O/P:- 5
      25
```

Non void functions without parameters: -

Some functions return a value but don't have any parameters. The most common use for this design reads data from the key board or a file and returns the data to the calling program. In the example shown below, note that the called function contains all of the code read the data. In this simple example only a prompt and a read are required.

```
Ex:  //function declaration

     int getquantity(void);

     int main(void)
     {
        int amt;
        //statements
        amt=getquqlity( );
        . . . . . . . . . .
        return 0;
     }
     int getquqlity(void)
     {
        //local declarations
        int qty;
        //statements
        printf("enter quantity :");
        scanf("%d", &qty);
        return qty;
     }
```

Non-void functions with parameters: -

Example program contains a function that passes parameters and returns a value – in this case the square of the value returned, the value on the right side of the assignment expression is the returned value which is then assigned to b.

```
     //Function declaration
     int sqr(int x);
     int main(void)
     {
```

```
    int a,b;
    printf("enter number :");
    scanf("%d",&a);
    b=sqr(a);
    printf("square of a is %d",b);
    return 0;
}
int sqr(int x)
{
    return (x*x);
}
```

Scope: -

Scope determines the region of the program in which a defined object is visible, i.e. the part of the program in which we can use the object's name. Scope pertains to any object that can be declared, such as a variable or a function declaration.

Figure: Scope (Global and Block areas)

```
#include< stdio.h >
int fun (int a, int b);          //Global area

int main(void)
{
    int a:
    int b;                  //main's area
   float y;
   . . . . . . . .
   {           //Beginning of nested block.
      float a=y/2;
     float y;                        //nested block area
     float z;
     . . . . . . . .
     z= a * b;
   }//End of nested block
}//End of main

int fun(int i, int j)
{
    int a;                              //functions area
    int y;
    . . . . . . . .
}//fun
```

To know about the scope, we need to review two concepts. First, a block is zero or more statements enclosed in a set of braces. Recall that a function's body is enclosed in a set of braces, thus a body is also a block. A block has declarations section and allows each block to be an independent group of statements section. This concept gives us the ability to nest blocks within the body of a function and allows each block to be an independent group of statements with its own isolated definitions.

Second, the global area of our program consists of all statements that are outside functions. Figure is a graphical representation of the concept of global area and blocks.

In object's scope extends from its declaration until the end of its block. A variable to the statement being examined. Variables are in scope from their print of declaration until the end of their block.

Global Scope: - The global is easily defined. Any object defined in the global area of a program is visible from its definition until the end of the program is visible from its definition until the end of the program. Referring to figure, the function declaration for 'fun' is a global definition. It is visible everywhere in the program.

Local Scope: - Variables defined within a block have local scope. They exist only from the point of their declaration until the end of the block in which they are declared. Outside the block they are invisible.

In figure, we see two blocks in main. The first block is all of main(). Since the second block is nested within main (). All definitions within main () are visible to the block unless local variables with an identical name are defined. In the inner block a local version of the variable 'a' has been defined; its type if float. The integer value of the 'a' in main () is visible from its declaration until the declaration of the floating point variable 'a' in nested area. At that point main ()'s 'a' can no longer be referenced in the nested block. Any statement in the block that references 'a' will get the float version. Once we reach the end of the block, the float 'a' is no longer in scope and the integer 'a' becomes visible again.

Variables are in scope from declaration until the end of their block.

Standard Library Functions:

C provides rich collection of standard functions whose definitions have been written and are ready to be used in our program. To use these functions, we must include their function declarations. The function declarations for these functions are grouped together and collected in several header files. Instead of adding the individual declarations of each function, therefore, we simply include the heading at the top of our programs.

Ex: - #include<stdio.h>

The include statement causes the library header file for standard input and output (stdio.h) to be copied into our program. It contains the declarations for printf()

and scanf(). Then when the program is linked, the object code for these functions is combined with our code to build the complete program.

Math functions: -

Many important library functions are available for mathematical functions. Most of the function declarations for these functions are in either the math header file (math.h) or standard library (stdlib.h). The integer functions are found in stdlib.h.

Absolute value functions:-

An absolute value is the positive rendering of the value regardless of its sign. There are two integer functions and two real functions.

The integer functions are abs, labs and for abs the parameter must be an int and it returns an int. For labs the parameter must be along int, and it returns a long int.

```
int abs (int number);
long labs (long number);
```

The real functions are fabs, fabsf. For fabs the parameter is a double and it returns a value of double.

```
double fabs (double number);
float fabsf (float number);
```

Ex: - abs (3)=3

fabs (-3.4)=3.4

Complex Number functions: -

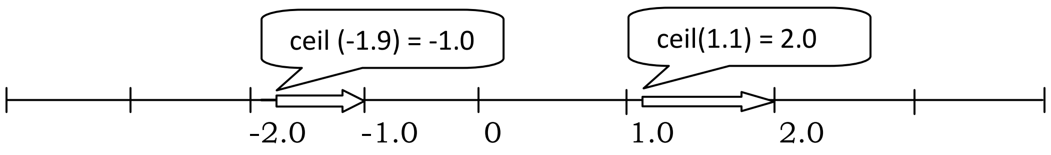
The functions for manipulating complex numbers are collected in the complex.h header file. Some of the complex number functions are shown below.

```
double cabs (double complex number);     //absolute
float cabsf (float complex number);
double carg (double complex number);     //argument
float cargf(float complex number);
 double creal (double real number);      // real
float crealf(float real number);
double cimag (double complex number);    // imaginary
float cimagf (float complex number);
```

Ceiling functions: -

A ceiling is the smallest integral value greater than or equal to a number. For example the ceiling of 3.00001 is 4. If we consider all numbers as a continuous range from minus infinity to plus infinity (in figure 1), this function moves the number right to an integral value.

Fig 1: ceiling function



Although the ceiling function determines an integral value, the return type is defined as a real value that corresponds to the argument. The ceiling function declarations are,

double ceil(double number);

float ceil(float number);

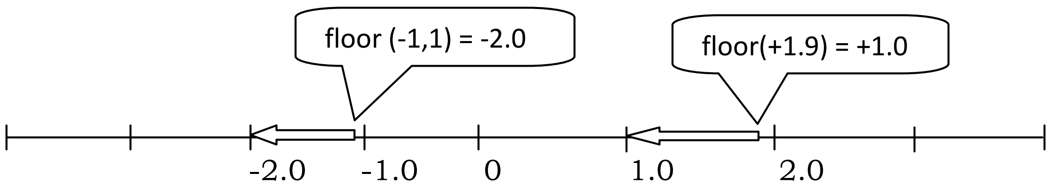
Ex: ceil(-1.9)= -1.0

ceil(1.1)= 2.0

Floor function:

A floor is the largest integral value that is equal to or less than a number. The floor of 3.99999 is 3.0. Looking at numbers as a continuous, this function moves the number left to an integral value.

Fig 2: floor function



Declarations:

double floor (double number);

float floorf (float number);

Ex: floor (-1.1) returns -2.0

floor (1.9) returns 1.0

Truncate functions: -

This function returns the integral in the direction of 0. They are the same as the floor functions for positive numbers, and the same as ceiling function for negative numbers.

Declarations:

double trunc(double number);

float truncf (float number);

Ex: trunc (-1.1) = -1.0

trunc (1.9) = 1.0

Round functions: -

The round function returns the nearest integral value. Their function declarations are,

double round (double number);

float roundf (float number);

In addition to the real round functions, C provides another type that returns a long int. Note that there is not a round function that returns an int.

Declarations:

long int lround (double number);

long int lroundf (float number);

Ex: round (-1.1) returns -1.0

round (1.9) returns 2.0

round (-1.5) returns -2.0

Power functions: -

The power (pow) function returns the value of the x raised to the power y that is x^y. An error occurs if the base (x) is negative and the exponent (y) is not an integer or if the base is the zero and the exponent is not positive.

Declarations:

double pow (double n1, double n2);

float powf (float n1, float n2);

Ex: pow(3.0, 4) returns 81.0

pow(3,5) returns 243

Square Root Functions: -

The square root (sqrt) functions return the non-negative square root of a number. An error occurs if the numbers is negative. The square root functions declarations are,

double sqrt (double n1);

double sqrtf (float n1);

Ex: sqrt (25) = 5.0

Call by value Call by reference: -

In a computer language there are two ways that arguments can be passed to a function. The first is *call by value*. This method copies the value of an argument into the formal parameter of the function. In this case, changes made to the parameter have no effect on the argument.

C uses *call by value* to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the following program:

```
#include<stdio.h>
   int sqr(int x);
   int main(void)
   {
       int t=10;
       printf("%d%d", sqr(t),t );
       return 0;
 }
 int sqr(int x)
{
      x= x*x;
      return x;
}
```

In this program, the value of the argument to sqr( ), 10, is copied into the parameter x. When the assignment x= x*x takes place, only the local variable x is modified. The variable t, is used to call sqr( ), till has the value 10. Hence the output is 100 10. It is a copy of the value of the argument that is that passed into a function.

Call by reference is the second way of passing arguments to a function. In this method the address of an argument is copied into the parameter. Inside the function the address is used to access the actual arguments used in the call. This means that changes made to the parameter affect the argument. We can create a call by reference by passing a pointer to an argument, instead of passing the argument itself. The swap() function in the following program is able to exchange the values of the two variables pointed to by x and y because their addresses (not their values) are passed. Within the function the contents of the variables are accessed using standard pointer operations, and their values are swapped. The swap( ) (or any function that uses the pointer parameters) must be called with the address of the arguments.

```
#include<stdio.h>
void swap(int *x, int *y);  // function declaration
 int main(void)
 {
        int i=10, j=20;
        printf("i & j before swapping : %d %d",i,j);
        swap(&i & j); /*pass the addresses of i & j*/
```

```
         printf("i & j after swapping : %d %d",i,j);
        return 0;
}
void swap(int *x, int *y)  // function definition
{
        int temp;
        temp=*x;     //save the value at address x
        *x = *y;       //put y into x
        *y= temp;    //put x into y
}
```

<u>O/P:</u>

```
i & j before swapping: 10 20
i & j after swapping :   20 10
```

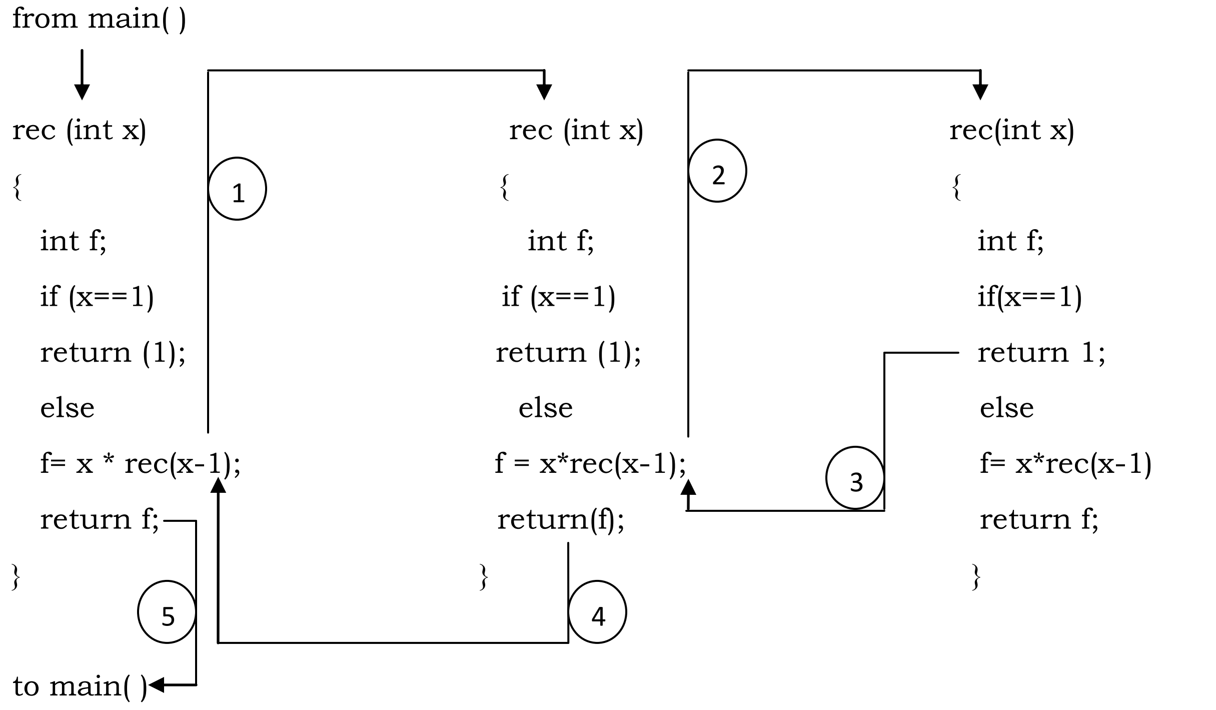
Recursion: -

A function is called 'recursive' if a statement within the body of a function calls the same function. Sometimes called 'circular definition' recursion is thus the process of defining something in terms of itself.

```
/* Finding factorial value using recursion */

      #include<stdio.h>
      int rec (int);
      void main( )
     {
        int a, fact;
        printf("enter any number : ");
        scanf("%d", &a);
        fact = rec (a);
        printf("factorial value : %d", fact);
    }
    int rec (int x)

   {

       int f;

       if (x==1)

       return (1);

      else

      f= x* rec(x-1);

      return (f);

  }
```

O/P: -

```
enter any number : 5

factorial value : 120
```



```
from main( )

rec (int x)
{
  int f;
  if (x==1)
  return (1);
  else
  f= x * rec(x-1);
  return f;
}

to main( )
```

```
rec (int x)
{
  int f;
 if (x==1)
return (1);
  else
f = x*rec(x-1);
return(f);
}
```

```
rec(int x)
{
  int f;
  if(x==1)
  return 1;
  else
  f= x*rec(x-1)
  return f;
}
```

Fig: Factorial Program Execution

Assume that the number entered through scanf( ) is 3. Using figure let's visualize what exactly happens when the recursive function gets called. The first time when rec( ) is called from main( ), x collects 3. From here, since x is not equal to 1, the if block is skipped and rec( ) is called again with the argument(x-1), i.e. 2. This is a recursive call. Since x is not equal to 1, rec( ) is called yet another time, with argument(2-1). This time as x is 1, control goes back to previous rec( ) with the value 1, and f is evaluated as 2.

Storage classes: -

To fully define a variable, one needs to mention not only its 'type' but also its 'storage class'. In other words, not only do all variables have a data type, they also have a storage class.

Storage classes have defaults, if we don't specify the storage class of a variable in its declaration; the compiler will assume a storage class depending on the context in which the variable is used. Thus, variables have certain default storage classes. From C compiler's point of view, a variable name identifies some physical location within the computer where the string of bits representing the variables value is stored. There are basically two kinds of locations in a computer where such a value may be kept- memory and CPU registers. It is the variable's storage class that determines in which of these two types of locations, the value is stored.

Moreover, a variable's storage class tells us:

i. Where the variable would be stored.

ii. What will be the initial value of the variable, if initial value is not specifically assigned. (i.e. the default initial value.)

iii. What is the scope of the variable: i.e. in which functions the value of the variable would be available.

iv. What is the life of the variable: i.e. how long would be the variable exists.

There are four stage classes in C:

a) Automatic storage class

b) Register storage class

c) Static storage class

d) External storage class

Automatic storage class: - The features of a variable defined to have an automatic storage class are as under:

Storage	- Memory
Default initial value	- An unpredictable value, which is often called a garbage value.
Scope	- Local to the block in which the variable is defined.
Life	- Till the control remains within the block in which the variable is defined.

Following program shows how an automatic storage class variable is declared, and the fact is that if the variable is not initialized, it contains a garbage value.

```
#include<stdio.h>

void main( )

{

  auto int i,j;

  printf("\n %d%d",i,j);

}
```

```
O/P:  1211   221
```

Where 1211, 221 are garbage values of i and j. When we run the program, we may get different values, since garbage values are unpredictable.

The keyword for this storage class is 'auto'. Scope and life of an automatic variable is illustrated in the following program.

```
#include<stdiio.h>
void main( )
{
  auto int i=1;
{
  {
    {
       printf("%d".i);
     }
     printf("%d",i);
    }
    printf("%d",i);
   }
}
```

O/P: - 1   1   1

This is because, all printf( ) statements occur within the outermost block in which I has been defined. It means the scope of i is local to the block in which it is defined. The moment the control comes out of the block in which the variable is defined.

Let us consider another program.

```
   #include<stdiio.h>
   void main( )
   {
     auto int i=1;
   {
     auto int i=2;
     {
     auto int i=3;
       {
          printf("%d".i);
        }
        printf("%d",i);
       }
       printf("%d",i);
      }
   }
```

O/P: - 3   2   1

That the compiler treats the three i's as totally different variables, since they are defined in different blocks. Once the control comes out of the innermost block, the variable i with value 3 is lost, and hence the i in the second printf ( ) refers to i with value 2. Similarly when the control comes out of the next innermost block, the third printf ( ) refers to the i with value 1.

Register storage class: - The features of variables defined to be of register storage class are:

Storage	-	CPU registers

Default initial value	-	Garbage value
Scope	-	Local to the block in which the variables is defined.
Life	-	Till the control remains within the block in which the variable is defined.

A value stored in a CPU register can always be accessed faster than the one that is stored in memory. If a variable is used at many places in a program, it is better to declare its storage class as register. A good example of frequently used variables is loop counters. We can name their storage class as register.

```
#Include <stdio.h>
void main ( )
{
  register int i;
  for (i=1; i<=0; i++)
  printf (“%d”, i);
}
```

Not every type of variable can be stored in a CPU register. For example, if the microprocessor has 16-bit register then they cannot hold a float value or a double value, which require 4 and 8 bytes respectively. However, if we use the register storage class for a float or a double variable, we can’t get any error messages. All that would happen is the compiler would treat the variables to be of auto storage class. The keyword for this storage class is ‘register’.

Static storage class: - The features of a variable defined to have a static storage class are:

Storage	-	memory
Default initial value	-	zero
Scope	-	Local to the block in which the variable is defined.
Life	-	Value of the variable persists between different function calls.

To understand the difference between static and automatic storage classes consider the following two programs and their outputs.

```
#include<stdio.h>
void increment( );
void main( )
{
  increment ( );
  increment ( );
  increment ( );
}
```

```
#include<stdio.h>
void increment( );
void main( )
{
  increment ( );
  increment ( );
  increment ( );
}
```

```
        void increment ( )
        {
          auto int i=1;
          printf("%d\n",i);
          i = i+1;
        }
O/P:    1       1       1
```

```
        void increment ( )
        {
          static int i=1;
          printf("%d\n",i);
          i = i+1;
        }
O/P:    1       2       3
```

The above programs consists of two functions main ( ) and increment ( ). The function increment gets called from main three times. Each time it prints the value of I and then increments it. The only difference in the two programs is that one uses an auto storage class for variable I, where as the other uses static storage class.

Like auto variables, static variables are also local to the block in which they are declared. The difference between them is that static variables don't disappear when the function is no longer active. Their values persist. If the control comes back to the same function again, the static variables have the same values they had last time around.

In the above example, when variable is auto, each time increment is called, it is re-initialized to one. When the function terminates, I vanishes and its new value of 2 is lost. The result is no matter how many times we call increment ( ), i is initialized to 1 every time.

On the other hand, if i is static, it is initialized to 1 only once. It is never initialized again. During the first call to increment ( ), i is incremented to 2. Because i is static, this value presents. The next time increment ( ) is called, i is not re-initialized to 1, and its old value 2 is still available. This current value of i (i.e. 2) gets printed and then i=i+1 adds 1 to i to get a value of 3. When increment ( ) is called the third time, the current value at I (i.e. 3) gets printed and once again i is incremented. In short, if the storage class is static, then the statement static int i=1 is executed only once, irrespective of how many times the same function is called.

External storage class: - The features of a variable whose storage class has been defined as external are:

Storage	-	Memory
Default initial value	-	Zero
Scope	-	Global
Life	-	As long as the program's execution doesn't come to an end.

External variables are declared outside all functions, yet are available to all functions that they can be using them.

```
/*Example to illustrate this */

#include<stdio.h>
void increment( );
void decrement( );
void main( )
{
  increment ( );
  increment ( );
  decrement ( );
  decrement ( );
}
void increment ( )
{
  i=i+1;
  printf("\n on incrementing i : %d",i);
}
void decrement ( )
{
  i = i-1;
  printf("("\n on decrementing i : %d",i);
}
```

```
O/P:  i = 0
      on incrementing i : 1
      on incrementing i : 2
      on decrementing i : 1
      on decrementing i : 0
```

From the above output, the value of i is available to the functions increment ( ) and decrement ( ) since i has been declared outside all functions.

```
/* Another example */
#include <stdio.h>
int x= 10;
void display ( );
void main ( )
{
   extern int y;
  printf ("\n %d %d", x, y);
}
  int y = 31;
```

Here, x and y are both global variables. Since both of them have been defined outside all the functions both having external storage classes.

```
extern int y;
```

```
        int y =31;
        Here the first statement is a declaration, where as the second is the definition.
Ex:     #include <stdio.h>
        int x = 10;
        void display ( );
        void main ( )
        {
          int x = 20;
          printf 9"\n %d", x);
          display ( );
        }
        void display ( )
        {
          printf ("\n %d", x);
        }
O/P:    20      10
```

A static variable can also be declared outside all the functions. For all practical purposes it will be treated as an extern variable. However, the scope of this variable is limited to the same file in which it is declared.

C Preprocessor: -

The C compiler is made of two functional parts: a preprocessor and a translator. The preprocessor uses programmer supplied commands to prepare to prepare the source program for compilation. The translator converts the co statements into machine code that it places in an object module.

The preprocessor can be thought of as a smart editor like a smart editor it inserts, includes, excludes and replaces text based on commands, supplied by the programmer. All preprocessor commands start with a pound sign (#). Preprocessor commands can be placed anywhere in the source program.

There are two major tasks of a preprocessor: file inclusion, macro definition.

File inclusion: -

The copying of one or more files into program is known as file inclusion. The files are usually header files that contain function & data declarations for the program, but they can contain any valid C statement.

The preprocessor command is #include and it has two different formats. The first format is used to direct the preprocessor to include header files from the system library. In this format, the name of the header file is enclosed in pointed brackets. The second format makes the preprocessor look for the header files in the user-defined directory. In this format, the name of the file path name is enclosed in double quotes. The two formats are as follows,

```
#include <filename.h>
```

```
#include "filename.h"
```

Macro definition: - The second preprocessor task is macro definition. A macro definition command associates a name and the tokens are referred to as the macro body. A macro definition has the following form.

```
#define name body
```

The body is the text is used to specify how the name is replaced in the program before it is translated.

Macros must be coded on a single line. If the macro is too long to fit on a line, we must use a continuation token. The continuation token is a backslash (\) followed immediately by a new line. If any whitespace is present between the backslash and the newline, then the backslash is not a continuation token, and it generates an error.

Ex: -
```
#define PREND\
        printf ("----------");

#define ANS = 0     /*wrong*/

#define ANS 0       /*correct*/
```

A macro definition can simulate a constant definition. The simplest application of macro is to define constant.

Ex: -
```
#define SIZE 9
```

The macro name is SIZE and the body is 9. Whenever in the program Size is encountered, it is replaced with 9.

A set of commonly used progressions and their functions are given below: