Functions

Function : Functions in C programming are blocks of code that perform a specific task.
A function is a self contained program segment that carries out a particular task & well defined tasks.

In fact, the concept of functions, which were originally a subset of a concept called sub-program. As many times as it is needed, keep 'calling' the segment to get the result. The separate program segment is called a function and the program that calls it is called the 'main program'.

```
main()
{
statement-1;
statement-2;
        ---
        ---
statement-n;
}
```

(i) Program not using Function

```
main()                    deposit()
{                         {
deposit();                ---
---                       ---
deposit();                ---
---                       ---
}                         }
```

(ii) Program using one Function many times

```
main()          deposit()          withdraw()
{               {                  {
deposit();      ---                ---
 ---            ---                ---
withdraw();     ---                ---
---             ---                ---
}               }                  }
```

(iii) Program using Multiple Functions

```
main()          fun1()          fun2()
{               {               {
---             ---             ---
fun1();         fun2();         ---
---             ---              ---
---           return;           return;
}               }               }
```

(iv) A Program where One Function calls another function

Every 'C' program contains one or more functions, but these one or more functions can be called a single main() function. These one or more function definitions must be appear in any order. But the calling function controls function calls and their order.

Uses of functions :

Function supports the concept of Modular Programming**.** When your programming is too complicated(large), the function declaration divided into modules (or) logics to simplify the complexity of program. Hence this process is called as Modular Programming**.**

For this reason functions are implemented.

a) Reusability: Code can be used multiple times without repetition.
b) Modularity: Breaks down complex programs into smaller, manageable units.
c) Readability: Improves code organization and clarity.
d) Maintainability: Easier to modify and debug specific parts of the program.
e) Flexible debugging
f) Code sharing
g) Data protection

Types Of Functions :

C functions can be classified into mainly two categories:

1) Predefined Functions

Standard Library Functions:

- These are built-in-functions(predefined functions) provided by the C Standard Library and can be directly used in C programs without the need for explicit declaration or definition.
- Examples include printf() , scanf() , malloc() , free() , etc.
- They are typically declared in standard header files like , <stdio.h>, <stdlib.h> , etc.

External Library Functions:

- Functions that are part of external libraries and are linked to the program during compilation.
- Examples include math functions like sin() , cos() , etc., found in the <math.h> library.

2) User-Defined Functions

- Functions created by the programmer to perform specific tasks within a program.
- These functions enhance code modularity, readability, and reusability.
- Examples include functions like add() , multiply() , etc., created by the programmer.

User-Defined Function Declaration and Definition:

In C, a function is declared before it is used. The declaration typically includes the function name, return type, and parameter types (if any).

The function is then defined elsewhere in the code.

Syntax for Function Declaration/Function prototype :

```
return_type function_name(parameter_type1 parameter1, parameter_type2 parameter2, ...);
```

Syntax for Definition:

```
return_type   function_name(parameter_type1 parameter1, parameter_type2 parameter2, ...) {
        // Function body or implementation
        // Statements to perform the desired task
        // Optionally, use the return statement to return the result
```

```
  return result;
}
```

Let's break down the components of a function definition:

return_type: Specifies the type of value the function will return to the calling code. If the function doesn't return a value, the return type is void.

function_name: The name assigned to the function. This name is used to call the function from other parts of the program.

Parameters or arguments : Variables declared inside the parentheses, representing values passed into the function when it is called. If a function doesn't take any parameters, the parentheses are left empty.

The argument list (or) parameter list can be divided into 2 types such as

a) Formal Parameters (or) Formal Arguments
b) Actual Parameters (or) Actual Arguments

a) **Formal Arguments**: - The arguments used in Function definition are called as Formal Arguments. These are also called as dummy arguments. These arguments are defined in the brackets in function definition after the function–name in the called function. It can be only variables but not expressions, constants.
b) **Actual Arguments**: - The arguments used in Function call are called Actual Arguments. These arguments are defined in the brackets in function-call after the function-name in the calling function. It can be variables, expressions, (or) constants.

When the function is called the actual arguments values are copied into formal arguments.

Function body: The block of code enclosed within curly braces {} . This is where the actual work of the function is performed.

return statement: Optional, used to return a value to the calling code. The return statement is not required for functions with a return type of void .

Function Call :- Function-Name refers to the name given to the function which is used in the main program to call that particular function

In Function Call statements, The function name followed by either without or with arguments enclosed within parenthesis & terminated by ;.

The syntaxes of Function-Calls :

1) Calling a function without arguments and without return value

```
Function-name();
```

Ex:- large();

2) Calling function with arguments and without return value

```
Function-name(argument-list);
```

Ex:- large(a,b);

3) Calling function with arguments and with return value

```
Variable-name = Function-name(argument-list);
```

Ex:- c = large(a,b);

4) Calling function without arguments and with return value

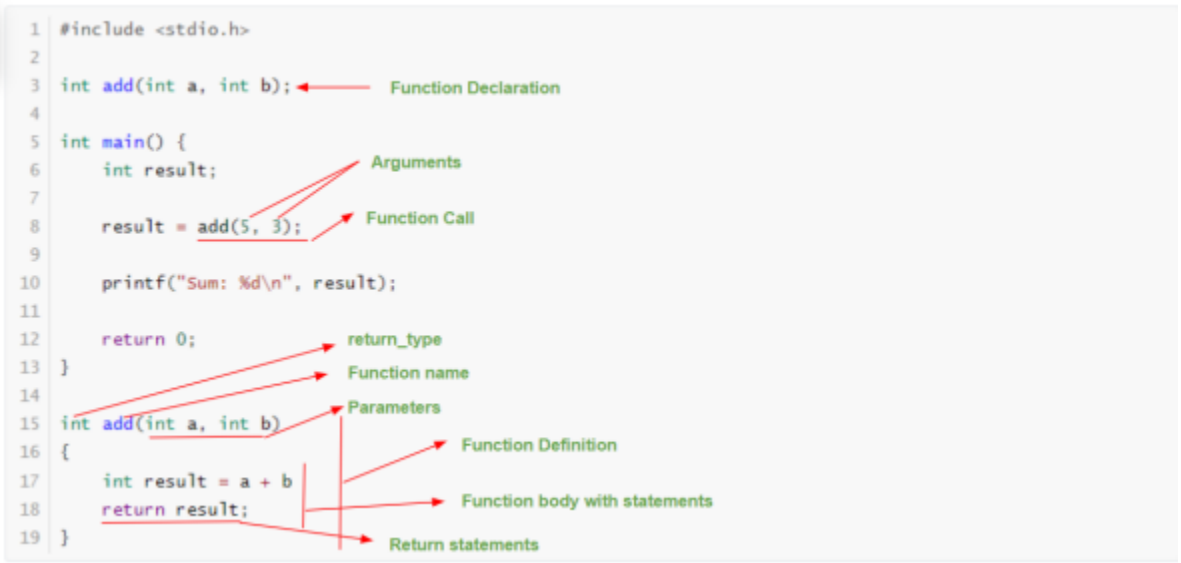
```
Variable-name = Function-name();
```

Ex:- c = large();

The Rules that must be followed during function call are

a) The number of actual arguments are equal to the number of formal arguments.
b) The type of actual and formal arguments must be same.
c) The sequence of actual and formal arguments must be same.

Example : Function two add two integers



Example 2: Function two find the square of a number

```c
#include <stdio.h>

// Function declaration
int square(int num);

int main() {
    int result;

    // Function call
    result = square(4);

    printf("Square: %d\n", result);

    return 0;
}

// Function definition
int square(int num) {
    return num * num;
}
```

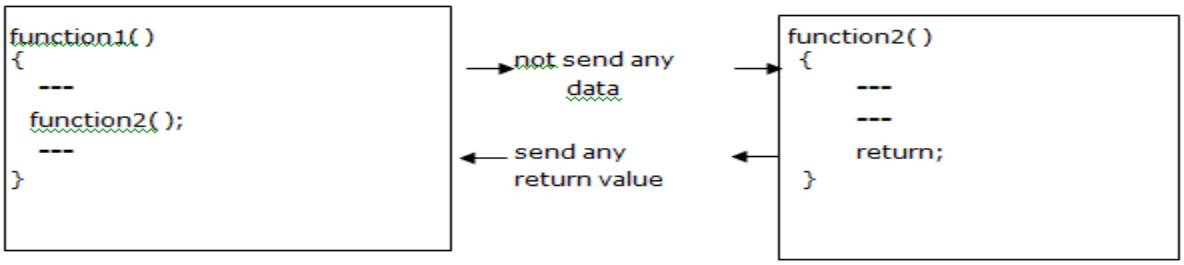
Categories of user defined functions:-

The user defined functions can be categorized into 4 types
1. Without arguments & Without return values
2. With arguments & Without return values
3. With arguments & With return values
4. Without arguments & With return values

1. Without arguments & Without return values :-

When a function has no arguments i.e., function name should be followed by empty parenthesis. Hence the calling function does not send any data to the called function where as the called function does not contain return value i.e., the called function does not send any return value to the calling function. In effect there is no data transfer in between calling and called function.

Calling function Called function



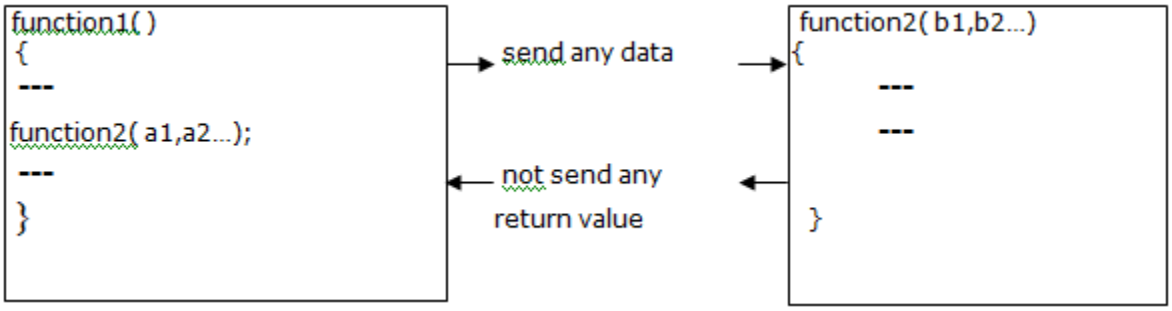
Example : ***** Write a program to find the subtraction of numbers.

```
main()
{
   sub();
}

sub()
{
          int a,b,c,d;
          scanf (“%d%d%d”, &a,&b,&c);
          d=a-b-c;
          printf (“%d”,d);
}
```

2. <u>With arguments & Without return values</u> :-

When a function has arguments then the calling function sending any data to the called function where as the called function does not contain return value i.e., the called function does not send any return value to the calling function.



Here a1,a2 are actual arguments & b1,b2 are formal arguments

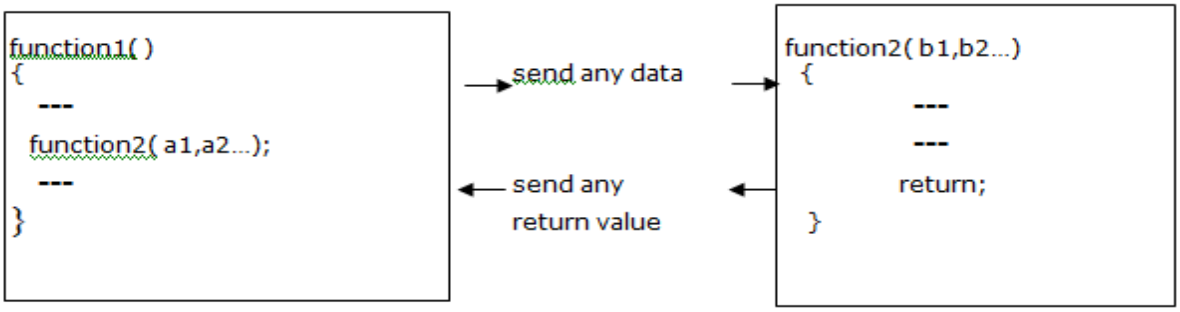
Example : ***** Write a program to find subtraction three numbers

```
main()
{
      int a,b,c,;
      scanf(“%d%d%d”,&a,&b,&c);
       sub(a,b,c);
 }
```

```
  sub(int a, int b, int c)
{
     int d;
     d=a-b-c;
    printf("%d",d);
 }
```

3. <u>With arguments & With return values</u> :-

When a function has arguments then the calling function sending any data to the called function where as the called function contain return value then the called function send any return value to the calling function.

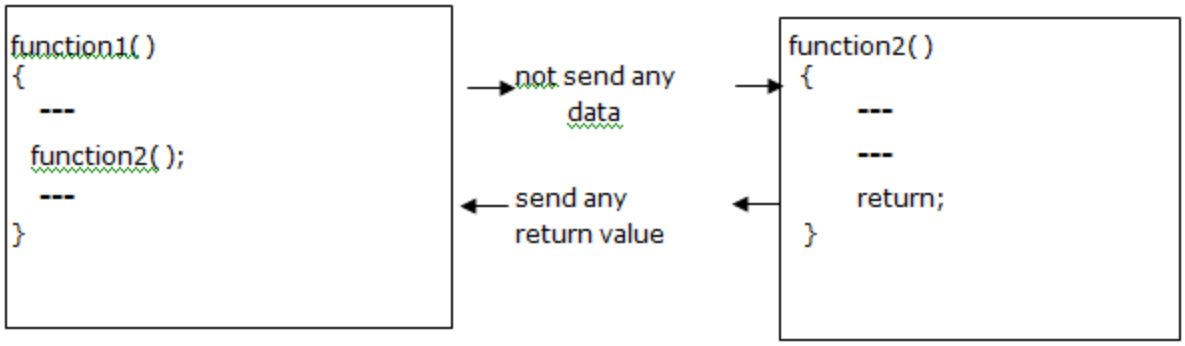


<u>Example</u> ***** Write a program to find the subtraction of three numbers

```
main()
{
int a,b,c,d;
scanf("%d%d%d",&a,&b,&c);
d= sub(a,b,c);
printf("%d",d);
}
sub(int a,int b,int c)
{
int d;
d=a-b-c;
return(d);
}
```

4. <u>Without arguments & With return values</u> :-

When a function has arguments then the calling function sending any data to the called function where as the called function contain return value then the called function send any return value to the calling function.



Example : ***** Write a program to find the subtraction of three numbers

```
main()
{
int d;
d= sub();
printf("%d",d);
}
sub()
{
int a,b,c,d;
scanf("%d%d%d",&a,&b,&c);
d=a-b-c;
return(d);
}
```

Different ways to pass parameters to functions

In C programming we can pass value to a function in two ways.

1. Call by value
2. Call by reference

1. Call by value:

- The actual value of the variable is passed to the function.
- Changes made to the parameter inside the function do not affect the original variable.
- uitable for small-sized data types.

Example :

```
#include<stdio.h>
void increment(int x);
int main() {
    int num = 5;
    printf("\nbefore calling increment() function num = %d",num);
    increment(num);
    printf("\nafter calling increment() function num = %d",num);
   return 0;
```

```
}

 void increment(int x)
 {
   x++;
   printf("\nInside function  num =  %d", x);
 }
```

Output :

```
before calling increment() function num = 5
Inside function  num =  6
after calling increment() function num = 5
```

2. Call by Reference (using Pointers):

- The address of the variable is passed to the function using pointers.
- Changes made to the parameter inside the function affect the original variable.
- Suitable for large-sized data types or when modification of the original value is required.

```
#include<stdio.h>
void increment(int *x);
int main() {
     int num = 5;
     printf("\nbefore calling increment() function num = %d",num);
     increment(&num);
     printf("\nafter calling increment() function num = %d",num);
    return 0;
}

 void increment(int *x)
 {
   (*x)++;
   printf("\nInside function  num =  %d", *x);
 }
```

Output :

```
before calling increment() function num = 5
Inside function  num =  6
after calling increment() function num = 6
```

Call By Value	Call By Reference
While calling a function, we pass the values of variables to it. Such functions are known as “Call By Values”.	While calling a function, instead of passing the values of variables, we pass the address of variables(location of variables) to the function known as “Call By References.
In this method, the value of each variable in the calling function is copied into corresponding dummy variables of the called function.	In this method, the address of actual variables in the calling function is copied into the dummy variables of the called function.
With this method, the changes made to the dummy variables in the called function have no effect on the values of actual variables in the calling function.	With this method, using addresses we would have access to the actual variables and hence we would be able to manipulate them.
In call-by-values, we cannot alter the values of actual variables through function calls.	In call by reference, we can alter the values of variables through function calls.
Values of variables are passed by the Simple technique.	Pointer variables are necessary to define to store the address values of variables.
This method is preferred when we have to pass some small values that should not change.	This method is preferred when we have to pass a large amount of data to the function.
Call by value is considered safer as original data is preserved	Call by reference is risky as it allows direct modification in original data

Recursion in C

Recursion is a programming technique where a function calls itself, either directly or indirectly, to solve a smaller instance of a problem. Recursive solutions break down complex problems into simpler, similar sub-problems.

Every recursion procedure should be contains terminating loop or terminating condition. When the recursion procedure does not contain terminate loop then the process repeated in-finite times. The terminating condition also called Base case is very important to ensure the repeated self invocation(calling) is to halt at some point.

Recursion have 2 types such as

a) Direct Recursion – as function calls itself.

b) Mutual(Indirect) Recursion – as a function (add()) calls another function(sub()) and the called function (sub()) calls the another function (add()).

Key Concepts:

1. Base Case:

- Every recursive function must have a base case that defines the simplest instance of the problem and provides a termination condition for the recursion.

2. Recursive Case:

- The part of the function that calls itself with a smaller or simpler input, moving towards the base case.

3. Stack Memory:

- Recursive calls are managed using the system's call stack. Each recursive call is added to the stack, and when the base case is reached, the stack is unwound.

Example: Factorial Calculation

Let's use the factorial calculation as an example to understand recursion.

```
main()
{
int n,f;
scanf("%d",&n);
f=factorial(n);
printf("%d",f);
}
int  factorial(int  n)
{
if(n= =0)
return(1);
else
return(n*factorial(n-1));
}
```

Execution Part

```
n=4
4= =0 ?base met or not
4*factorial(3)
3= =0 ?check
4*3*factorial(2)
2= =0 ?check
4*3*2*factorial(1)
1= =0 ?check
4*3*2*1*factorial(0)  = 24
```

Explanation:

1. Base Case :

- When n is 0, the function returns 1, providing a termination condition.

2. Recursive Case:

- When n is greater than or equal 1, the function calls itself with a smaller input (n-1), contributing to the recursive nature.

3. Stack Memory:

- Each recursive call is added to the stack until the base case is reached.
- The stack is then unwound, and each call contributes to the final result.

Types of Recursion

1. Direct Recursion: In direct recursion, a function directly calls itself. The example below demonstrates direct recursion in a function calculating the sum of natural numbers.

```
#include <stdio.h>
int sumDirect(int n) {
  if (n == 0)
```

```
    return 0;
  else
    return n + sumDirect(n - 1);
}

int main() {
  int num = 5;
  int result = sumDirect(num);
  printf("Sum of first %d natural numbers: %d\n", num, result);
 return 0;
}
```

In this example, sumDirect directly calls itself to calculate the sum of natural numbers.

2. Indirect Recursion: In indirect recursion, two or more functions call each other in a circular manner. Here's an example involving two functions, even and odd , checking if a number is even or odd using indirect recursion.

```
#include <stdio.h>
int even(int n);
int odd(int n);
   int even(int n) {
        if (n == 0)
          return 1; // 0 is even
        else
          return odd(n - 1);
   }

   int odd(int n) {
        if (n == 0)
          return 0; // 0 is not odd
        else
          return even(n - 1);
   }

   int main() {
        int num = 5;
        printf("%d is %s.\n", num, even(num) ? "even" : "odd");
        return 0;
   }
```

In this example, even and odd functions call each other to determine if a number is even or odd

Scope in C Programming

Extent : - The period of time during which memory is associated with the identifier is called extent of the variable.

Scope :- The region of the program over which the declaration of an identifier is visible is called the scope of the identifier.

The scope related to the

i) Accessibility
ii) The period of existence
iii) The boundary of usage of variably declared in a statement block (or) a function.

These features in turn define whether a variable is local (or) global in nature.

In C, the scope of a variable defines where the variable can be accessed and modified. The understanding of variable scope is crucial for writing clean and maintainable code.

There are three main types of variable scope in C: global scope, local scope, and block scope.

1. Global Scope:

- Variables declared outside of any function, at the beginning of the program.
- Accessible throughout the entire program.

Example:

```
#include <stdio.h>
  // Global variable
int globalVar = 10;
int main() {
  // Access global variable
  printf("Global Variable: %d\n", globalVar);
 return 0;
}
```

2. Local Scope:

- Variables declared within a function.
- Accessible only within the function where they are declared.

Example:

```
#include <stdio.h>
void localScopeExample() {
   // Local variable
   int localVar = 5;
   // Access local variable
   printf("Local Variable: %d\n", localVar);
}
int main() {
   // Call the function with local scope
   localScopeExample();
   return 0;
}
```

3. Block Scope:

- Variables declared within a block of code, typically within curly braces {} .

- Limited to the block where they are declared and any nested blocks.

Example:

```
#include <stdio.h>
int main() {
   // Variables in block-level scope
   int x = 5;
   if (x > 0) {
   // This block introduces a new scope
   int y = 10;
   printf("Inside if block: x=%d, y=%d\n", x, y);
   }
   // Variable 'y' is not accessible outside the if block
   // Uncommenting the line below would result in a compilation error
   // printf("Outside if block: x=%d, y=%d\n", x, y);
return 0;
}
```

Diffence Between Global & Local Variables :-

Global Variable	Local Variable

i) The variables which are declared above main() are called Global Variable.
 	
ii) These variables can be accessed by all the functions in the program.
 	
iii) The Default value for global variable is zero.
 	
Ex : int a= 5;
main()
{
--- 	
--- 	
}
	i) The variables which are declared within any function are called Local Variable.
	
	ii) These variables can be accessed only by the function in which they are declared. Other functions can't access these variables.
	
	iii) The Default value for local variable is garbage value.
	Ex : int add(int a,int b)
	{
	int c;
	--- 	
	--- 	
	}
	Where 'c' is Local Variable.

Storage Classes : -The storage class specifier the following information's about a variable.

a) Scope of variable
b) Lifetime of variable
c) Where the variable gets the storage place
d) What will be the default value of a variable

Scope represents within which it is accessible(variable is active).

The scope of a variable may be either Local or Global.
Local scope means the scope of the variable is restricted within the block in which that variable is declared.
Global scope means the scope of the variable is restricted outside that block ie., that variable is accessed in all modules.

Syntax | <storage-class-specifier> <data type> <variable-name>;

The Storage Classes can be characterized into 4 types

1. Automatic Variables
2. External Variables
3. Static Variables
4. Register Variables

1) **Automatic Variables :-** These variables are also called as Private or Local Variables. By default, all variables declared inside functions are automatic. The Keyword is auto. When function is calling these variables are created, but exit from function these variables are destroyed.

a) The default value of these variables is garbage.
b) Their Scope is Local
c) The storage location of these variables are memory.

The Syntax is

auto data-type $v_1,v_2, \ldots v_n$;

The Keyword auto is used optional.
Data-type may be int, float, char etc.,
$v_1,v_2, \ldots v_n$ are identifiers.

Ex : auto int a=4;

1. W.A.Prg., to illustrate the automatic variables.

```
main()
```

```
{
 int a = 5;
 printf("%d",a);
 fun1();
 printf("%d",a);
}
fun1()
{
 int a = 2;
 printf("%d",a);
 fun2();
 printf("%d",a);
}
fun2()
{
 int a = 12;
 printf("%d",a);
 fun3();
 printf("%d",a);
}
fun3()
{
  int a = 20;
  printf("%d",a);
}
```

OUTPUT : - 5, 2, 12, 20, 12, 2, 5

2. W.A.Prg., to illustrate the automatic variables.

```
main()
{
    int a = 5;
    printf("%d",a);
    fun1();
    printf("%d",a);
}
fun1()
{
    int a = 2;
    printf("%d",a);
    fun2();
    printf("%d",a);
}
fun2()
{
    int a = 12;
    printf("%d",a);
```

```
        fun3();
        printf("%d",a);
    }
     fun3()
    {
        printf("%d",a);
    }
```

Note :- We can execute above prg., the third function can't be executed.

2) <u>External Variables</u> :- These variables are also called as Public or Global Variables. These variables are declared before main function. The Keyword is extern.

A program in 'C', particularly when it is large, can be broken up into smaller programs. After compiling these, each program file can be joined together to form the large program. These small program modules that combine together may need some variables that are used by all of them. In 'C', such a provision can be made by specifying these variables, accessible to all the small program modules that are formed as separate files.

When function is calling these variables are created, but exit from function these variables are not destroyed.

a) The default value of these variables is zero.
b) Their scope is Global
c) The storage location of these variables are memory.
   The Syntax is

extern data-type $v_1, v_2, \ldots v_n$;

Data-type may be int, float, char etc., $v_1, v_2, \ldots v_n$ are identifiers.
Ex : extern int a = 4;

1. W.A.Prg., to illustrate the External Variables**.**

```
int a = 5;
main()
{
printf("%d",a);
fun1();
printf("%d",a);
}
fun1()
{
int a = 2;
printf("%d",a);
fun2();
printf("%d",a);
}
fun2()
```

```
{
printf("%d",a);
fun3();
printf("%d",a);
}
fun3()
{
printf("%d",a);
}
```

OUTPUT : - 5, 2, 5, 5, 5, 2, 5

c) <u>Static Variables</u> :- These variables are declared as either automatic or external variables. The Keyword is static. When we can't define keyword 'static' that variable is act as static automatic otherwise static extern. When function is calling these variables are created, but exit from function these variables are not destroyed.

a) The default value of these variables is 0.
b) Their Scope may be Local or Global.
c) The storage location of these variables(Local or Global static) are memory.

The Syntax is

static data-type $v_1,v_2, \ldots v_n$;

Data-type may be int, float, char etc.,

$v_1,v_2, \ldots v_n$ are identifiers.

Ex : static int a;

Example :

```
#include <stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

OUTPUT : 1 2

The above program prints 1 2 because static variables are only initialized once and live till the end of the program. That is why they can retain their value between multiple function calls.

Following are some interesting facts about static variables in C:

1) A static int variable remains in memory while the program is running. A normal or auto variable is destroyed when a function call where the variable was declared is over.
2) Static variables are allocated memory in the data segment, not the stack segment.
3) Static variables (like global variables) are initialized as 0 if not initialized explicitly.

d) <u>Register Variables</u> :- These variables are very faster than computer's memory. The keyword is 'register'. This keyword precedes the normal declaration statement of a variable. The variables are preceded by register then whose variables stored in some register of the CPU. In most 'C' Compilers, the register specifier can only be applied to int and char type of data.
The default value of these variables is garbage.

a) Their Scope is Local
<u>b)</u> The storage location of these variables are CPU.

The Syntax is

register data-type $v_1,v_2, \ldots v_n$;

Data-type may be int, float, char etc., & $v_1,v_2, \ldots v_n$ are identifiers.

Ex : register int a;

***** W.A.Prg., to display the number and its square from 1 to 5 using register variable.

```
main()
{
register  int i,c;
for(i=1;i<=5;i++)
{
c = i * i;
printf("%2d%2d",c,i);
}
}
```

OUTPUT :- 1 2 3 4 5
1 4 9 16 25

***** W.A.Prg., to display the number and its square from 1 to 5 using register variable.

```
#include <stdio.h>
int   power(int,int);
main()
{
int   n = 4, k = 3, x;
x = power(n,k);
printf("%d %d %d",n,k,x);
}
int  power(int  s, register int  t)
{
register int  u;
u = 1;
for ( ; t; t - - )
u * = s;
return(u);
}
```

Output : 4 3 64