FUNCTIONS

CHAPTER : 5

Prepared By: Er. Rudra Nepal
Nepal College of Information and Technology

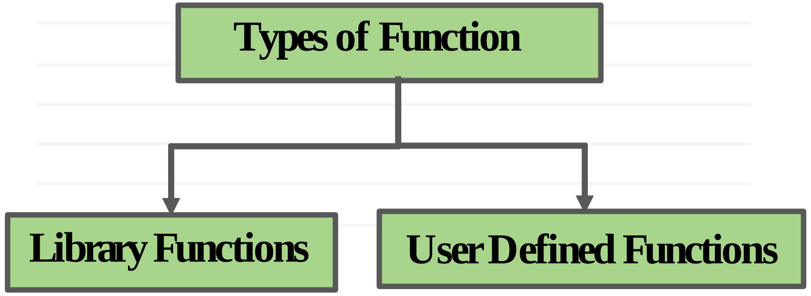
---OVERVIEW---

10Hrs

- Elements of Function
- Formal & Actual argument
- Category of Function
  - ✓ Function with no argument and no return type
  - ✓ Function with argument and no return type
  - ✓ Function with argument and return type
  - ✓ Function with no argument and return type
- Local variable & Global variable
- Types of Function Call
  - ✓ Call by value
  - ✓ Call by reference

FUNCTION:

- A Function is a block of statements that performs a specific task. By using functions, we can avoid rewriting same code again and again in a program. Reusability is the main achievement of C function.



1. **Library Function:**
   Library Functions are the functions which are declared in C header files such as scanf(), printf(), gets(), puts().

2. **User- Defined Function:**
   User Defined Functions are the functions which are created by the programmer according to his/ her reuirement.

<u>Syntax for creating a Function:</u>

```
return_type function_name(data_type perimeter, ..){

        / / code to be executed;

        }
```

Example:

```
int add(int a, int b);

void add();

int add();

void add(int a, int b);
```

ELEMENTS OF FUNCTION:

1. **<u>Function Declaration/ p rototype:</u>**

A Function protoype occurs at the top of a main function that describes what the function returns and what parameter it takes.
Function prototype should be follwed by semi-colon.

Syntax:

return_type function_name(arguments_list);

Syntax:

return_type function_name(arguments_lists);

Example:

```
int addition(int a, int b);
        or
int addition(int , int);
```

Name of function is addition.
Return Type is integer data.
Function takes two arguments of type int.

2. Function Defination:

A Function defination contains the block of code to perform a specific task.

It is written after the main function.

Syntax:

```
return_type function_name(arguments_lists){

        / / body of the function

}
```

Syntax:

```
return_type function_name(arguments_lists){

        / / body of the function

 }
```

Example:

```
int addition(int a, int b){
        int c;
        c = a + b;
        return c;
}
```

3. Calling a Function:

Control of the program is transferred to the user–defined function by calling it.

Function can be called anywhere in the program when nedded..

Syntax:

function_name(arguments_lists);

Example:

```
addition(2, 3); / / no return type
result = addition(2, 3); / / return type
```

CATEGORY OF FUNCTION:

1. <u>Function with no argumnet and no return type:</u>

- When a function has no argumnets , it doesn't receive any data from the calling function.
- Similarly, When it doesn't have return type , the calling function doesn't receive any data from the called function.
- Therefore, no data transfer between called and calling function.

Syntax:

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

```
#include<stdio.h>
# include< conio.h>
void add();  / / function prototype
int main(){
      printf("Transfer Control to add Function");
      add();  //function call
      printf("\n Control Back to main Function");
      getch();
     return 0;
  }
void add(){
      int a, b, c;
      printf("\ n Enter Two Numbers:");
      scanf("%d%d", &a, &b);
      c=a+b;
      printf("\ nSum of  %d and %d is:%d", a, b, c);
  }
```

CATEGORY OF FUNCTION:

2. **Function with argumnet and no return type:**

- When a function has argumnets , it receives data from the calling function.
- When it doesn’t have return type , the calling function doesn’t receive any data from the called function.

Syntax:

```
void function_name(argument_lists){
            //function body
                    }
```

```
#include<stdio.h>
# include< conio.h>
void add(int , int); / / function prototype
int main(){
      int a, b;
      printf(“\ n Enter Two Numbers::”);
      scanf(“%d%d”, &a, &b);
     printf(“\ nTransfer Control to add Function”);
     add(a, b);  //function call
     printf(“\n Control Back to main Function”);
     getch();
    return 0;
  }
void add(int a, int b){
      int c;
      c=a+b;
      printf(“\ nSum of %d and %d is:%d”, a, b, c);
}
```

CATEGORY OF FUNCTION:

3. **Function with argumnet and return type:**

- When a function has argumnets , it receives data from the calling function.
- When it have return type , the calling function receives data from the called function.

Syntax:

```
return_type function_name(argument_lists){
            //function body
                    }
```

```
#include<stdio.h>
#include<conio.h>
int add(int , int); //function prototype
int main(){
     int a, b, c;
     printf("\ n Enter Two Numbers::");
     scanf("%d%d", &a, &b);
    printf("Transfer Control to add Function");
    c = add(a, b); / / function call
    printf("\ n Control Back to main Function");
    printf("\ nSum of %d and %d is:%d", a, b, c);
    getch();
   return 0;
 }
int add(int x, int y){
     int z;
     z=x+y;
     return z;
  }
```

CATEGORY OF FUNCTION:

4. Function with no argumnet and return type:

- When afunction has no argumnets , it doesn't receives data from the calling function.
- When it have return type , the calling function receives data from the called function.

Syntax:

```
return_type function_name(){
            //function body
                }
```

```
#include<stdio.h>
#include<conio.h>
int add(); / / function prototype
int main(){
      int c;
      printf("Transfer Control to add Function");
     c = add(); / / function call
     printf("\n Control Back to main Function");
     printf("\ nSum is:%d", c);
     getch();
    return 0;
  }
 int add(){
      int x, y, z;
      printf("\ n Enter Two Numbers::");
      scanf("%d%d", &x, &y);
      z=x+y;
      return z;
 }
```

ACTUAL AND FORMAL ARGUMENTS:

1. **Actual arguments:**

➢ The argument that are passed in a function is called Actual arguments.

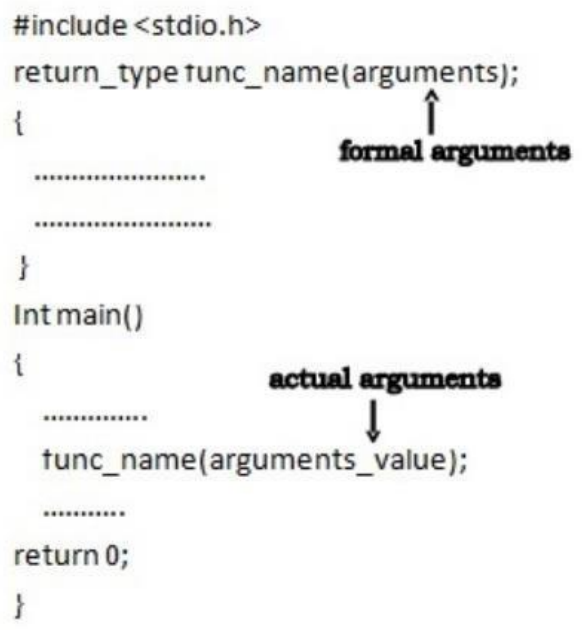
II. **Formal Arguments:**

➢ Argument mentoined in function defination is called Formal Arguments.

➢ These arguments are used to hold the values sent by calling function.

➢ Formal arguments are created when function starts and destroyed when funcrion ends.

(Formal and Actual arguments must match in number, order and type)



```
#include <stdio.h>
return_type func_name(arguments);
{
  ........................
  ........................
}
Int main()
{
  ...............
  func_name(arguments_value);
  ...........
return 0;
}
```

LOCAL AND GLOBAL VARIABLES:

Global Variables:

- Those variables which are defined outside of function block and are accessible to entire program are known as Global Variables.
- Scope is global i.e. they can be used anywhere in the program.
- Default value is Zero (0).
- When the value of the global variable is modified in one function changes are visible in the rest of the program
- It is stored on a fixed location decided by the compiler.

Local Variables:

- Those variables which are defined within some function and are accessible to that function only are called Local Variables.
- Scope is Local to that block or function where they are defined.
- Default value is unpredictable(garbage).
- When the value of the Local variable is modified in one function changes are not visible in the other function
- It is stored in stack unless specified.

```
#include<stdio.h>
# include<conio.h>
int add();
void modify();
int a , b;/ / Global Variable
int main(){
       int result, num=2 ; / / Local Variables
       result =  add();
       printf("\ nSum  of %d and %d is:%d",a,b,result);
       modify();
       num=num*2;
       printf("\ n a= %d , b= %d" , a, b, num);
       getch();
      return 0;
  }
 int add(){
       printf("\ n Enter Two Numbers::");
       scanf("%d%d", &a, &b);
       return (a +  b);
 }
```

```
void  modify(){
        a=a*2;
        b= b*2;
```

num = num++; //ERROR(num is local variable visible only within main function)

```
}
```

```
Enter Two Numbers: 5, 6
Sum of 5 and 6 is: 11
a=12, b=12, num=4
```

TYPES OF FUNCTION CALL:

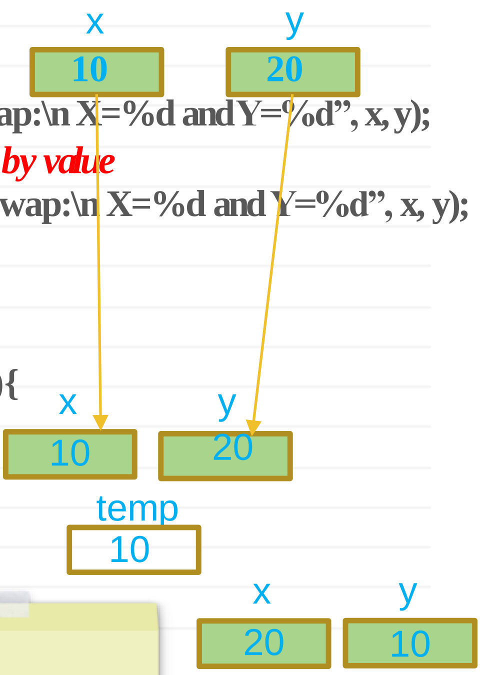
In programming language, function can be invoked in two ways:

1. <u>Call by Value:</u>

If Function is called by passing the value of variables as actual arguments , the function call is called called by value.

In this method, values of actual arguments are copied to function’s formal arguments. So, any changes made inside function are not reflected in actual parameters of caller.

```
#include<stdio.h>
# include<conio.h>
void swap(int ,  int);  / / function prototype
int main(){
       int x= 10, y= 20;
       printf(“Before Swap:\n X=%d and Y=%d”, x, y);
       swap(x, y); / / call by value
       printf(“\n After Swap:\n X=%d and Y=%d”, x, y);
       getch();
     return 0;
  }
 void  swap(int x, int y){
       int temp;
       temp=x;
       x=y;
       y=temp;
}
```



```
Before Swap:
X=10 and Y=20
After Swap:
X=10 and Y=20
```

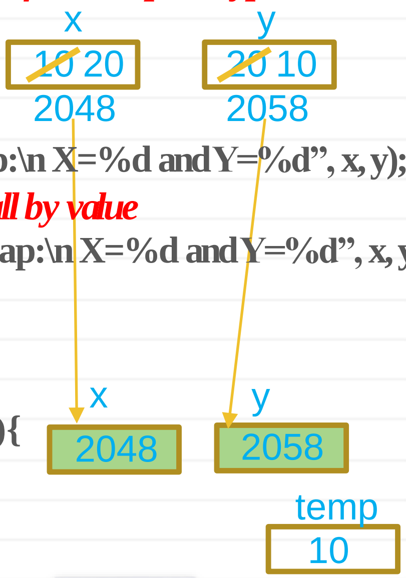
TYPES OF FUNCTION CALL:

2. Call by Refrence:

If Function is called by passing the address of variables as actual arguments , the function call is called called by reference.

In this method, address of actual arguments are copied to function's formal arguments. So, any changes made inside function are reflected in actual parameters of caller through pointer operation.

```
#include<stdio.h>
# include<conio.h>
void swap(int* , int*);  / / function prototype
int main(){
       int x= 10, y= 20;

       printf("Before Swap:\n X=%d and Y=%d", x, y);
       swap(&x, &y); / / call by value
       printf("\n After Swap:\n X=%d and Y=%d", x, y);
       getch();
     return 0;
  }
 void  swap(int *x, int *y){
        int temp;
        temp=*x;
        *x=*y;
         *y= temp;
}
```



Before Swap:
X=10 and Y=20
After Swap:
X=20 and Y=10

EXERCISE:

- WAP to program to calculate Sum, Difference, Product and Division of two number using functions(use function with argument and return type).

- Write a menu driven program using function to calculate:
  1. palindrome (with argument & no return type)
  2. odd/even (no argument, no return type)
  3. factorial(no argument, return type)
  4. print sum n naturan numbers(with argument and return type)
  5. exit

- Write a menu driven program using function to calculate:
  1. count number of digits in numbers(with argument and return type)
  2. Find reverse of number(without argument and return type)
  3. check for prime or not(with argument and no return type)
  4. check for armstrong or not.(with no argument and no return type)

- **WAP to evaluate following series using factorial function:**

  a. $1! + 2! + 3! + 4! + \ldots\ldots$

  b. $\frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \ldots\ldots$

  c. $x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} \ldots \ldots.$

  d. $1 - \frac{x}{2!} + \frac{x}{4!} - \frac{x}{6!} + \frac{x}{8!} \ldots \ldots.$

---OVERVIEW---

- Recursive function
- Recursion Vs Iteration
- Macros
- Storage class

MACROS:

- The Macros in C are basically fragment of code that has been given a name.
- The name is replaced by the contents of the macro, whenever the name is used.
- We can define a Macro in C using #define preprocessor directive
- There are generally two types of Macros:
  1. Object like Macro
  2. Function like Macro

1. Object Like Macros:

The object like macro is an identifier that is replaced by value. It is widely used to represent numeric constant

```
#include<stdio.h>
#include<conio.h>
#define height 10
#define PIE 3.15
#define letter ‘A’
int main(){
    printf(“Value of HEIGHT: %d”, height);
    printf(“Value of PIE: %f”, PIE);
    printf(“Value of LETTER: %c”, letter);
    getch();
    return 0;
            }
```

2. Function Like Macros:

It is an expression , used to perform particular operation. It is an alternative way to define function.

```
#include<stdio.h>
#include<conio.h>
#define area(l, b) l*b
int main(){
    int a, b, c
    printf("Value length and breadth:);
    scanf("%d%d", &a, &b);
    c= area(a, b);
    printf("Area of Rectangle:%d", c);
    getch();
    return 0;
              }
```

Multi-line macros are similar to single line macros. The only difference is that you need to add one \ at the end of each line except last line.

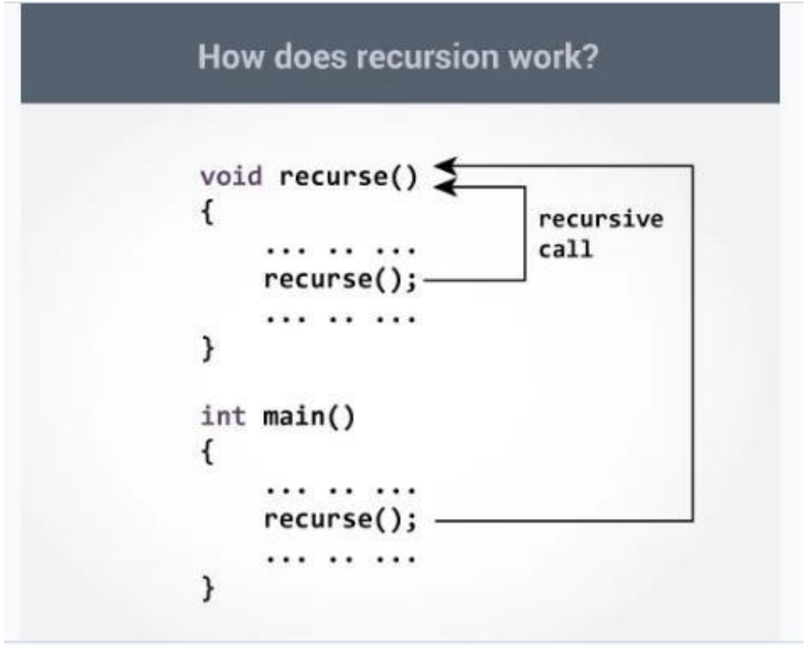
```
# include < stdio.h>
# define MAX(num1, num2) \
            if (num1> num2) \
            printf("%d is Max", num1); \
            else \
            printf("%d is MAX", num2);

int main() {
int num1, num2;
printf("Enter any two number: ");
 scanf("%d%d", &num1, &num2);
 MAX(num1, num2);
return 0;
}
```

MACRO	FUNCTION
It is not compiled, it is pre processed	It is not pre processed, it is compiled
Faster in execution than function	Bit slower in execution than macros
Macros are useful when small piece of code is used multiple times in a program	Function are useful when large piece of code is used multiple times in a program
No type checking required	Type checking required
It cannot be defined recursively	It can be defined recursively

Example:
#define PIE 3.14
#define area(l, b) l*b
	Example:
	Int add(int a, int b){
	//code
	}

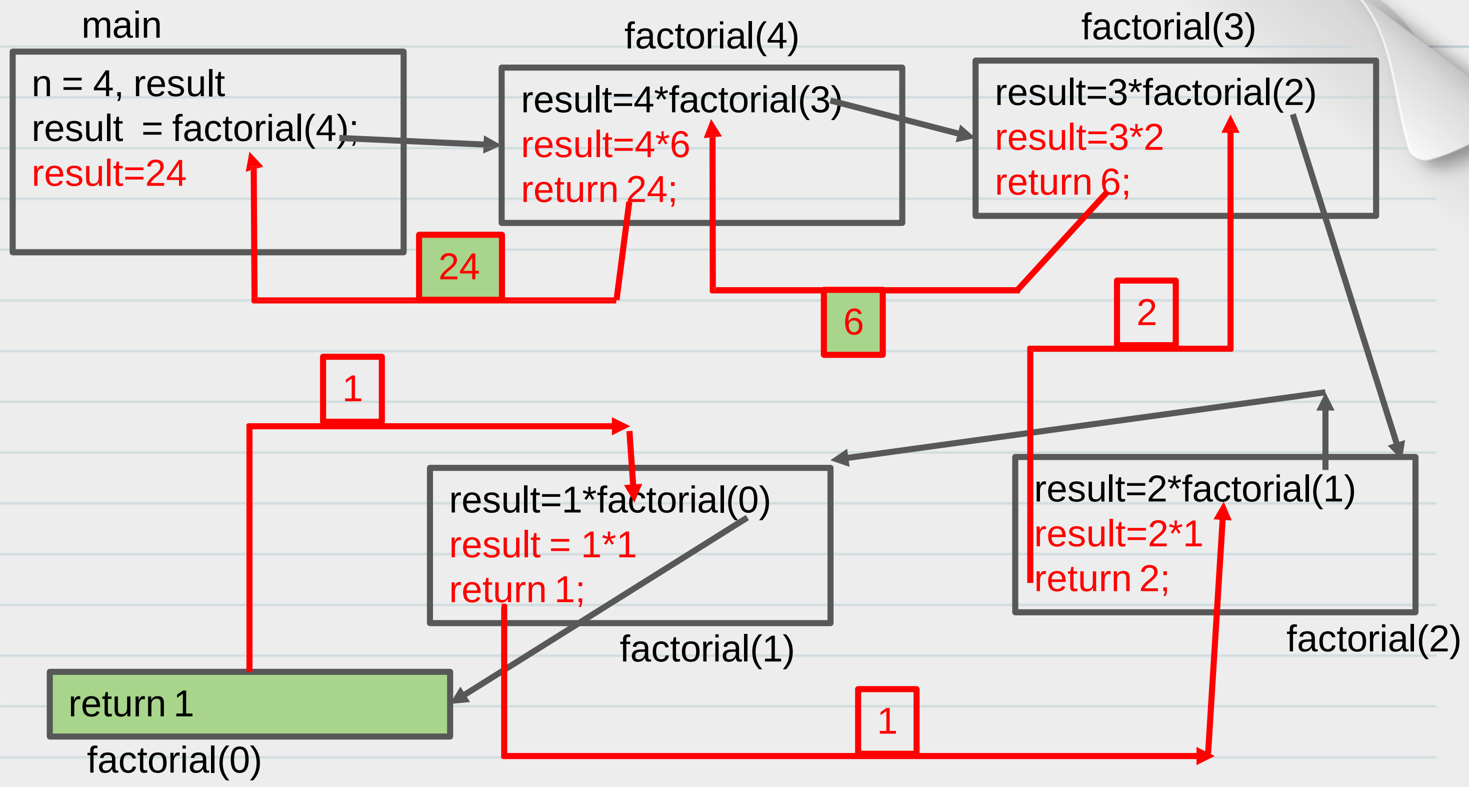
RECURSIVE FUNCTION:

- A function that calls itself is known as Recursive Function and this technique is called recursion.
- Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself .


How does recursion work?

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

```
//factorial of number using recursion
#include<stdio.h>
#include<conio.h>
int  factorial(int);
int main(){
                    int n , result;
                    printf("Enter a Number:");
                    scanf("%d", &n);
                    result= factorial(n);
                    printf("Factorial is:%d", result);
                    return 0;
}
int factorial(int n){
                    int result;
    if(n==0){
           return 1;
   }
   else{
                    result=n*factorial(n-1);
                    return result;
    }
}
```


main
n = 4, result
result = factorial(4);
result=24
factorial(4)
result=4*factorial(3)
result=4*6
return 24;
factorial(3)
result=3*factorial(2)
result=3*2
return 6;
24
6
2
1
result=1*factorial(0)
result = 1*1
return 1;
factorial(1)
result=2*factorial(1)
result=2*1
return 2;
factorial(2)
1
return 1
factorial(0)

```
//sum of n natural of number using recursion
#include<stdio.h>
#include<conio.h>
int sum(int);
int main(){
                int n , result;
                printf("Enter a Number:");
                scanf("%d", &n);
                result= sum(n);
                printf("Sum of natural number to %d term is:%d", n,result);
                return 0;
}
int sum(int  n){
                int result;
   if(n==1){
         return 1;
   }
   else{
                result=n+sum(n-1);
                return result;
    }
}
```

//Program to find power of any number using recursion

```
#include<stdio.h>
#include<conio.h>
int Calculate_power(int , int);
int main(){
            int base, power;
            long int result;
            printf("Enter a Base and Power:");
            scanf("%d%d", &base,  &power);
            result= Calculate_power (base, power);
            printf("Value of %d to power %d is:%d", base, power, result);
            return 0;
}
int Calculate_power (int  b, int p){
            long int result;
  if(p==0){
     return 1;
  }
  else{
            result=b*Calculate_power (b, p-1);
            return result;
  }
}
```

```
//Program to print Fibonacci series upto n terms using recursion
#include<stdio.h>
int  Fibonacci(int);
int main()
{
   int i, n;
   scanf("%d",&n);
   printf("Fibonacci series\n");
   for ( i = 0 ; i < n ; i++ )
   {
      printf("%d\t", Fibonacci(i));
   }
   return 0;
}
int Fibonacci(int n)
{
    if ( n == 0 )
       return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

RECURSION	ITERATION
The statement in a body of a function calls the function itself.	Allows set of instruction to be repeatedly executed.
Slower in execution.	Fast in execution
A conditional statement is included in the body of function for termination.	Statement is repeatedly executed until certain condition is reached.
Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Not all problem can be solved by recursion.	All problem that can be solved by recursion can be solved by iteration.

STORAGE CLASS:

- Every varialbes in C programming has two properties: Type and Storage Class;

  storage_class data_type variable_name;

- A Storage class determines:

  a. scope and visibility of variable
  b. Lifetime of variable
  c. Location of variable
  d. Default value of variable

- There are Four types of storage class in C:

  i. auto
  ii. Extern
  iii. Static
  iv. register

I. Auto Storage Class (Local Variables):

- **Syntax:** *auto int a;*

  int a;

- **Scope**= *the block where variable has been declared.*

- **Lifetime**= *Variable exist till the block is executing and destroyed when block finishes execution*

- **Default Value**= *Garbage Value*

- **Location**= *RAM*

2. Extern Storage Class:

- ➢ **Syntax:** *extern int a;*
- ➢ **Scope**= *Throughout the program.*
- ➢ **Lifetime**= *It remains alive till the program finishes execution.*
- ➢ **Default Value**= *Zero*
- ➢ **Location**= *RAM*

- Extern storage class is used to give refernce of global variable that is visible to ALL the program files.
- When we use extern , the variable cannot be initialized.

```
#include<stdio.h>
#include<conio.h>
void func_1();
int main(){
              func_1();
              return 0;
              }
   void func_1(){
              int a=1;
              extern int b;
              a++;
              b++;
              printf("a=%d and b=%d", a, b);
 }

 int b= 100;
```

3. Register Storage Class:

- **Syntax:** *register int a;*
- **Scope**= within *the block where variable has been declared.*
- **Lifetime**= *Variable exist till the block is executing and destroyed when block finishes execution.*
- **Default Value**= *Garbage Value*
- **Location**= *register(CPU)*

- We use register storage class when we want quick access to the variables because register has faster access than that of the main memory.

4. Static Storage Class:

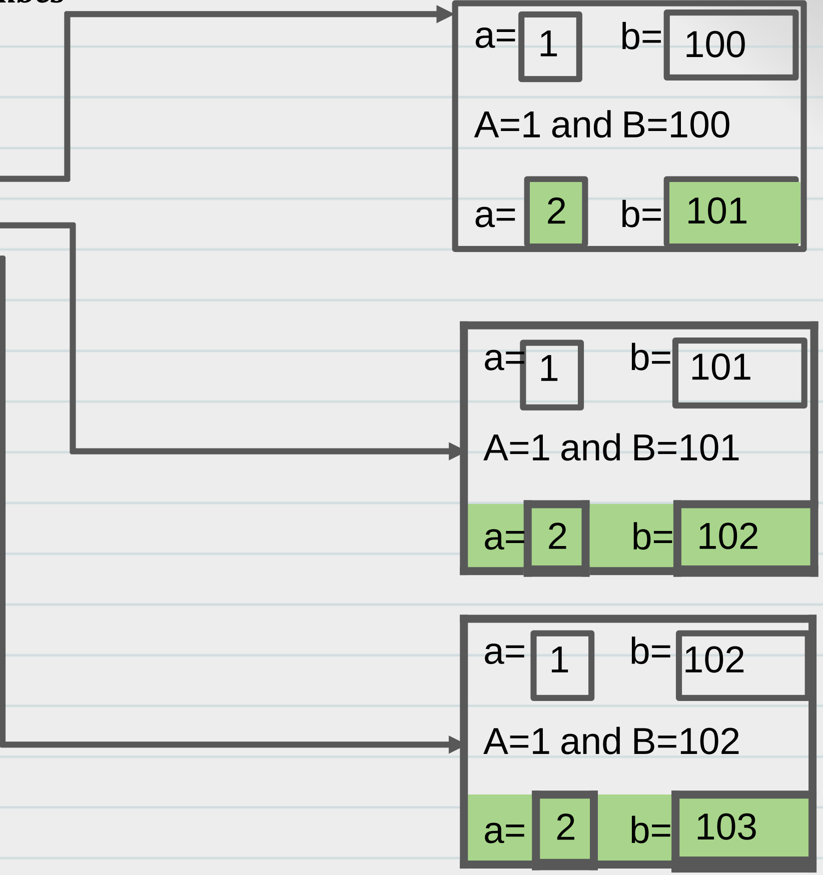
- **Syntax:** *static int a;*
- **Scope**= *the block where variable has been declared.*
- **Lifetime**= *It remains alive till the program finishes execution.*
- **Default Value**= *Zero*
- **Location**= *RAM*

- Static vaiables preserves their previous value in their previous scope and are not initialized again in new scope.
- Default storage class for global variable.

//Program Demonstrate static varialbes

```
#include<stdio.h>
void func1();
int main()
{
          func1();
          func1();
          func1();
          return 0;
}
void func1()
{
   int a=1;
   static int b=100;
    printf("A=%d and B=%d");
    a++;
    b++;
}
```

```
A=1 and B=100
A=1 and B=101
A=1 and B=102
```


a= 1 b= 100
A=1 and B=100
a= 2 b= 101
a= 1 b= 101
A=1 and B=101
a= 2 b= 102
a= 1 b= 102
A=1 and B=102
a= 2 b= 103