# UNIT – 3 (UDF)

# Functions (Inbuilt and User Defined)

BCA SEM – 1

PROBLEM SOLVING METHODOLOGIS AND PROGRAMMING IN C

Code : CS-01

Presented by : Dhruvita Savaliya

## Topics :

## ❖ User Defined Functions :

- A function is a block of code that performs a specific task.
- C allows you to define functions according to your need. These functions are known as user-defined functions.
- It provides code reusability and modularity to our program.
- User-defined functions are different from built-in functions as their working is specified by the user and no header file is required for their usage.
- A user-defined function has three main components that are

1. **function declarations**
2. **function definition**
3. **function call**.

- Functions need to be written once and can be called as many times as required inside the program, which increases **reusability in code** and makes code more readable and easy to test, debug, and maintain the code.

- **How user-defined function works? (SYNTAX) :**

```
#include <stdio.h>
void functionName();        //function prototype
void main()
{
            … .. …
            … .. …
            functionName();   //function calling
            … .. …
            … .. …
}

void functionName()         //function definition
{
            … .. …
            //function body
            … .. …
}
```

## Example of User Defined Function :

```
#include <stdio.h>
void fybca();                    // function prototype
void main()
{
          printf("Main Function");
          fybca();    // function call
          printf("\nBack to the main function");
}
void fybca()                     // function definition
{
          printf("\n I am in the Function");
}
```

**Output :**

```
Main Function
I am in the Function
Back to the main function
```

## Components / Elements of User-defined Function in C :

- Functions in the C language have three parts.

1. **Function Declaration**
2. **Function Definition**
3. **Calling User-defined Functions**

## 1. Function Declaration (Function Prototype):

- Function declaration contains **function name, return type**, and **parameters** but does not contain the body of the function.
- **Syntax :**

returnType **functionName**(data-Type parameterName , data-Type parameterName2 , ...)**;**

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

- It is **not compulsory to mention parameter name in declaration** hence we can also use

returnType functionName(data-Type , data-Type , data-Type...);

```
                int  sum(int , int);
```

**NOTE : The function prototype is not needed if the user-defined function is defined before the main() function.**

- **Return type:** The type of data returned from the function is called return type.

  A function may not return any output, in that case, we use void as the return type.

  In function declaration return type is mentioned before the name of the function.

  Like void,int,float,char etc… .

- **Function name**: Function name is a **unique** name that can be used to identify our function in the program.

  Function names are used to create function calls, which is why they are unique identifiers for compilers.

  A valid function name in C can contain letters, underscore, and digits;

  the first letter must not be a digit.

- **Example :**

```
thisIsAfunction(); // valid
_getMaximum();     // valid
!getMinimum();     // invalid, symbols except underscore are not allowed
getPowerOf2();     // valid
2Root();           // invalid function name, must not start with a number
```

- **Parameter list**: Parameters required by the function are also defined inside the declaration to tell the compiler number of arguments required by the function along with their data types.
- **Semicolon**: Semicolon indicates the termination of a function declaration.
- **Note:** Function declaration is not required if the function is defined before it is called in the code.
- **Example :**

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

Here , sum is function name.

Before function name int is return type.

And after function name int a and int b are parameters or arguments which has int data type.

## 2. Function Definition :

- Function definition contains the actual block of code that is executed once the function is called. A function definition has four components:

1. Return type
2. Function name
3. Function parameters
4. Body of function

- **Function body** contains a collection of instructions that define what a function does. If the function returns any value, we use the keyword return to return the value from the function.

  For example, return (5 +10); returns value 15 of integer data type.

- **Syntax of function definition :**

```
return_Type  function_Name(functionParameters...)
{
        // function body
}
```

- We can also give default values to function parameters that are assigned to the parameter if no argument is passed. For example,

```
int  sum(int a = 10, int b = 5)
{
       return a + b;
}
```

## 3. Calling User-defined Functions :

- To transfer the control to a user-defined function, we need to call the function. A function can be called using a **function name followed by round brackets**.
- We can pass arguments to function inside brackets, if any.
- **Syntax :**
  - Without Arguments :

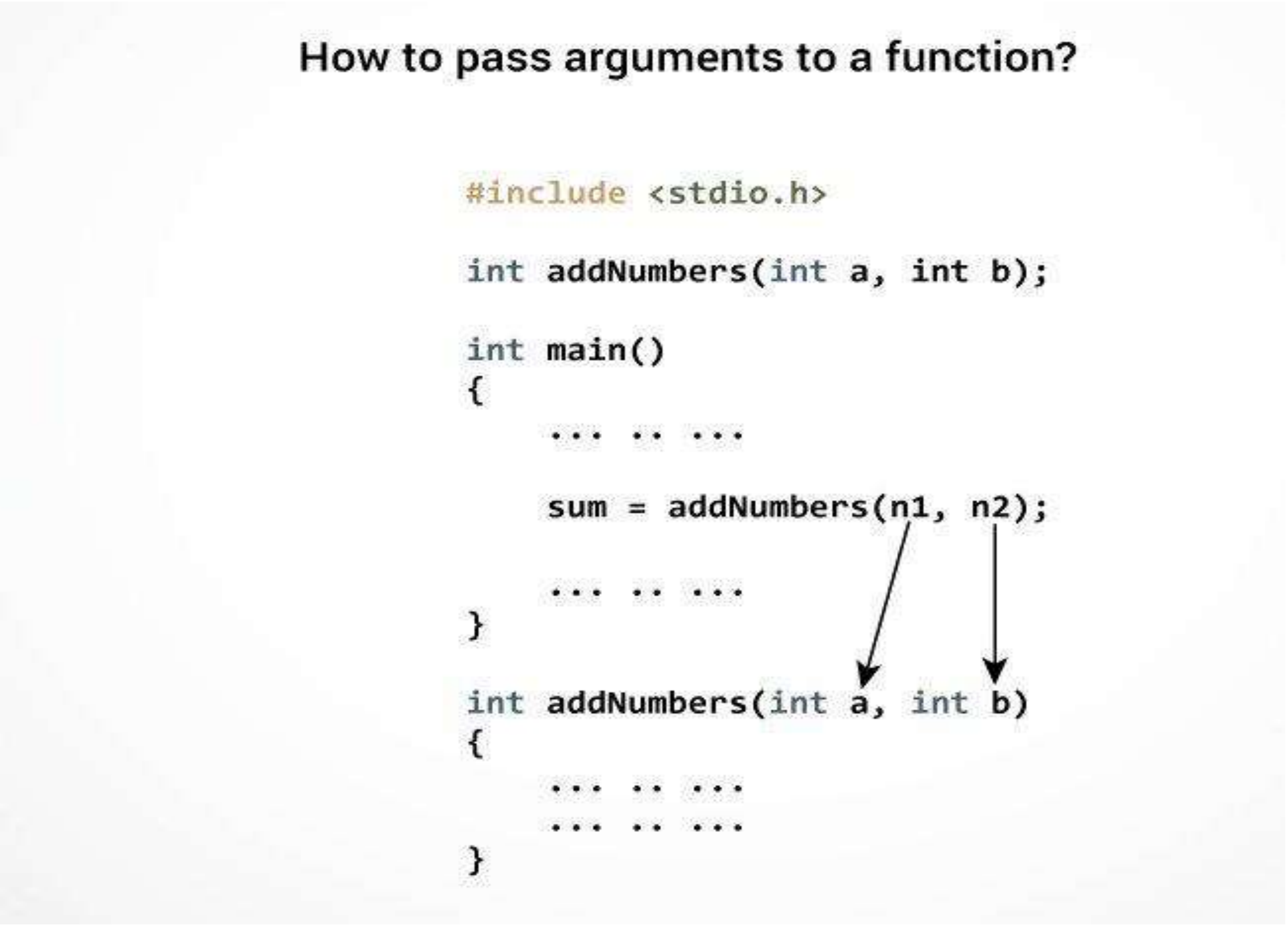
```
func_name();
```

  - With Arguments as Value :

```
func_name(10,5);
```

  - With Arguments as Variable :

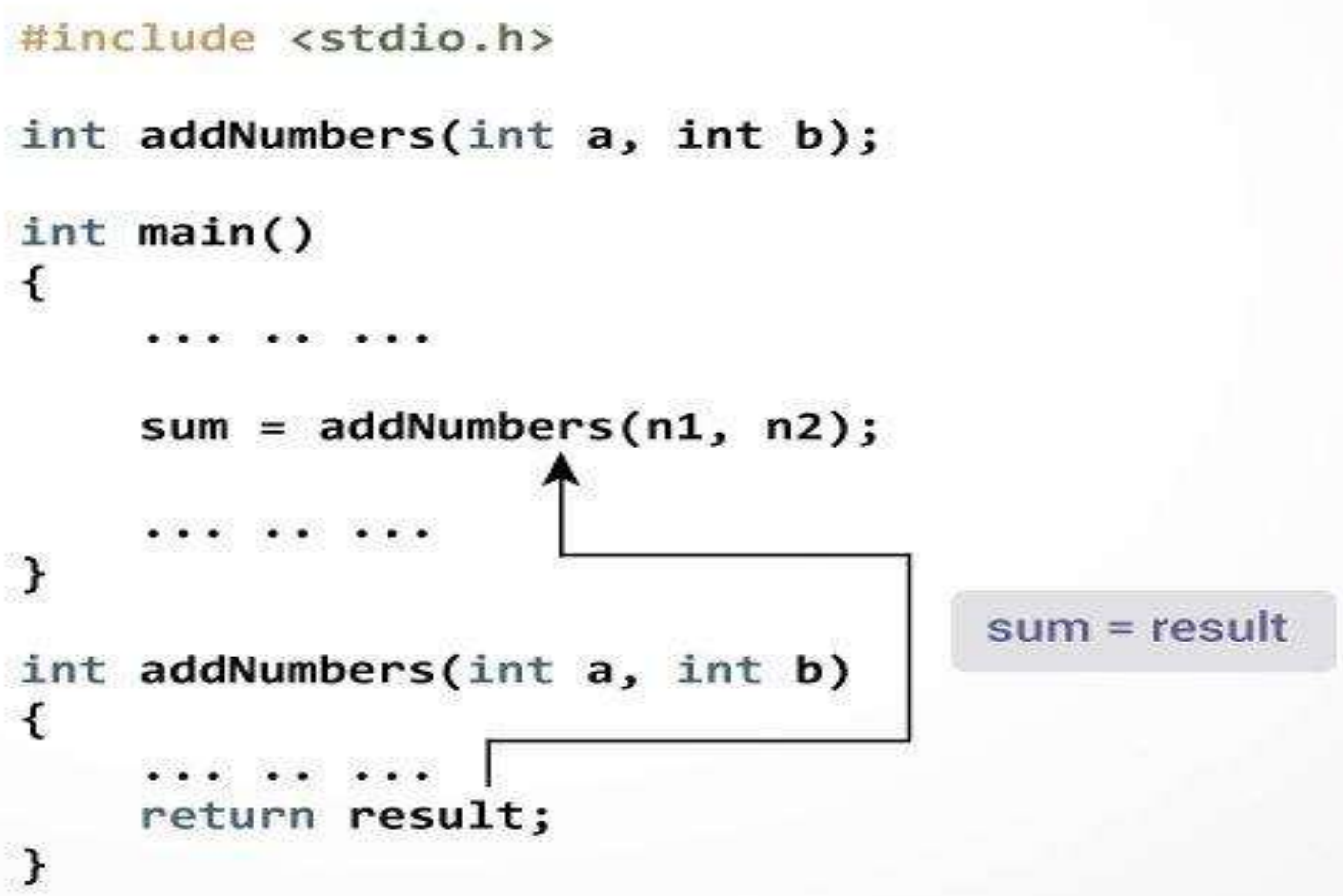
```
func_name(var1,var2);
```



## How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

## Return statement of a Function



```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
     ... .. ...

     sum = addNumbers(n1, n2);

     ... .. ...
}

int addNumbers(int a, int b)
{
     ... .. ...
     return result;
}
```

- **Advantages of user-defined function :**
- The program will be easier to understand, maintain and debug.
- Reusable codes that can be used in other programs.
- Functions are made for code reusability and for saving time and space.
- A large program can be divided into smaller modules. Hence, a large project can be divided among many programmers.

## ❖ Types of User Defined Functions :

- There can be 4 different types of user-defined functions, they are :

1. Function **without** arguments and **without** return value
2. Function **without** arguments and **with** return value
3. Function **with** arguments and **without** return value
4. Function **with** arguments and **with** return value

## 1. Function without Arguments and without Return Value :

- In this method, we don't pass any arguments and mention void as a function return type, so a function will not return any value.
- We declare a few variables in the function definition and perform certain operations on them.
- **Syntax :**

```
 void functionName();
             OR
 void functionName(void);
```

**Example :**

```
#include <stdio.h>
void sum();          // function prototype
void main()
{
        printf("Main Function");
        sum();   // function call
        printf("\nBack to the main function");
}
void sum()           // function definition
{
        int a=10,b=5;
        printf("\nI am in the Function");
        printf("\nHelo BCA");
        printf("\n Sum is %d",a+b);
}
```

**Output :**

```
Main Function
I am in the Function
Hello BCA
Sum is 15
Back to the main function
```

## 2. Function without arguments and with return value :

- In this function call method, we don’t pass any arguments in the function but mention a return type.
- A return type can be any data type; by default, the return type is int, and the value returned by a function is an integer value.
- **Syntax :**

```
int functionName();
           OR
int functionName(void);
```

**Example :**

```
#include <stdio.h>
int sum();  // function prototype
void main()
{
        int ans;
        printf("Main Function");
        ans=sum(); // function call
        printf("\nBack to the main");
        printf("\nSum of 2 number is : %d",ans);
}
int sum() // function definition
{
        int a=10,b=5;
        printf("\nI am in the Fybca");
        printf("\nHelo BCA");
        return a+b;
}
```

**Output :**

```
Main Function
I am in the Fybca
Hello BCA
Back to the main
Sum of 2 number is : 15
```

## 3. Function with arguments and without return value :

- In this function call method, with an argument and no returning value, we pass arguments to the function, but the function doesn't return a value.
- It is also necessary to give the arguments in the function call.
- **Syntax :**

```
void  sum(int a, int b);
                OR
void  sum(int , int);
```

## Example :

```
#include <stdio.h>
void sum(int a,int b);  // function prototype
void main()
{
        printf("Main Function");
        sum(10,5); // function call
        printf("\nBack to the main");
}
void sum(int x,int y) // function definition
{
        printf("\nI am in the Function");
        printf("\nHelo BCA");
        printf("\nX : %d  Y : %d",x,y);
        printf("\nSum is :",x+y);
}
```

**Output :**

```
Main Function
I am in the Function
Hello BCA
X : 10 Y : 5
Sum is : 15
Back to the main
```

## 4. Function with arguments and with return value :

- This method passes arguments to the function, which will also return a value.
- **Syntax :**

```
int  sum(int a, int b);
                 OR
int  sum(int , int);
```

**Example :**

```
#include <stdio.h>
int fybca(int a,int b);  // function prototype
void main()
{
        int sum;
        printf("Main Function");
        sum=fybca(10,5); // function call
        printf("\nBack to the main");
        printf("\nSum of 2 number  is : %d",sum);
}
int fybca(int a,int b) // function definition
{
        printf("\nI am in the Fybca");
        printf("\nHelo BCA");
        printf("\nA : %d  B : %d",a,b);
        return a+b;
}
```

**Output :**

```
Main Function
I am in the Fybca
Hello BCA
A : 10  B : 5
Back to the main
Sum of 2 number is : 15
```

- **Call by value and Call by reference in C :**

- There are two methods to pass the data into the function in C language, i.e., *call by value and call by reference.*

## ❖ Call by value in C :

- In call by value method, the value of the actual parameters is copied into the formal parameters.
- In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

## Example :

```
#include<stdio.h>
void change(int num) {
        printf("Before adding value inside function num=%d \n",num);
        num=num+100;
        printf("After adding value inside function num=%d \n", num);
}
void main() {
        int x=100;
        printf("Before function call x=%d \n", x);
        change(x);//passing value in function
        printf("After function call x=%d \n", x);
}
```

***Output***

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=100
```

## Pointer :

- The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2 byte.

**Example :**

int n = 50; //normal

int*p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.

int a=50;

int *b=&a;

**Declaring a pointer**

- The pointer in c language can be declared using * (asterisk symbol). It is also known as indirection pointer used to dereference a pointer.
- 1. int *a;//pointer to int
- 2. char *c;//pointer to char

```
#include <stdio.h>
void main()
{
   int c,*pc;
   clrscr();
   c = 22;
       printf("Address of c: %p\n",&c);
       printf("Value of c: %d\n\n",c);  // 22
   pc = &c;
       printf("Address of pointer pc: %p\n", pc);
       printf("Content of pointer pc: %d\n\n", *pc); // 22
   c = 11;
       printf("Address of pointer pc: %p\n", pc);
       printf("Content of pointer pc: %d\n\n", *pc); // 11
   *pc = 2;
       printf("Address of c: %p\n", &c);
       printf("Value of c: %d\n\n", c); // 2

       printf("Address of pointer pc: %p\n", pc);
       printf("Content of pointer pc: %d\n", *pc);//2
   getch();
}
```

## ❖ Call by reference in C :

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

## Example :

```
#include<stdio.h>
void change(int *num) {
        printf("Before adding value inside function num=%d \n",*num);
        (*num) += 100;
        printf("After adding value inside function num=%d \n", *num);
}
void main() {
        int x=100;
        printf("Before function call x=%d \n", x);
        change(&x);//passing reference in function
        printf("After function call x=%d \n", x);
}
```

***Output***

```
Before function call x=100
Before adding value inside function num=100
After adding value inside function num=200
After function call x=200
```

## ❖ Recursion :

- It is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
        recursion();     /* function calls itself */
}
int main() {
        recursion();
}
```

- The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.
- Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

**Example :**

```
#include<stdio.h>
#include<conio.h>
void count_to_ten(int);
void main()
{
            clrscr();
            count_to_ten(0);
            getch();
}
void count_to_ten(int cnt)
{
            Printf(" %d ",cnt);
            If(cnt < 10)
            {
                        count_to_ten(cnt+1);
            }
}
```

**Output :**

```
0
1
2
3
4
5
6
7
8
9
10
```

## ❖ Storage Classes :

- C Storage Classes are used to describe the features of a variable/function.
- These features basically include the scope, visibility, and lifetime which help us to trace the existence of a particular variable during the runtime of a program.
- **C language uses 4 storage classes :**

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

## Summary of Storage Classes in C :

| Class | Name of Class | Place of Storage | Scope | Default Value | Lifetime |
|---|---|---|---|---|---|
| auto | Automatic | RAM | Local | Garbage Value | Within a function |
| extern | External | RAM | Global | Zero | Till the main program ends. One can declare it anywhere in a program. |
| static | Static | RAM | Local | Zero | Till the main program ends. It retains the available value between various function calls. |
| register | Register | Register | Local | Garbage Value | Within the function |

## 1. Automatic :

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.
- Every local variable is automatic in C by default.

## Example :

```
#include<stdio.h>
void main()
{
        int a; //auto
        //auto int a; //both are same
        char b;
        float c;
        clrscr();
        printf("%d %c %f",a,b,c);
        // printing initial default value(garbage value) of automatic
        variables a, b, and c.
        getch();
}
```

## 2. Static :

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

## EXAMPLE :

```
#include<stdio.h>
void sum()
{
    static int a = 10;//create static variable
    int b = 24;       //create normal variable
    printf("Static Variable a : %d",a);
    printf("\t\tNormal Variable b : %d \n",b);
    a++;
    b++;
}
void main()
{
    int i;
    clrscr();
for(i = 1; i<=3; i++)
{
        printf("\ncall %d\n",i);
        sum();
        // The static variables holds their
        value between multiple function
        calls.
}
getch();
}
```

## OUTPUT :

```
call 1
Static Variable a : 10     Static Variable b : 24
call 2
Static Variable a : 11     Static Variable b : 24
call 3
Static Variable a : 12     Static Variable b : 24
```

## 3. Register :

- The **register** storage class is used to define local variables that should be stored in a register instead of RAM.
- This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

- **Example :**

```
#include <stdio.h>
void main()
{
    register int a;
    clrscr();
    printf("%d",a);   //garbage value
    printf("%d",&a);
    // This will give a compile time
    error since we can not access
    the address of a register
    variable.
    getch();
}
```

## 4. extern :

- Extern stands for external storage class.
- Extern storage class is used when we have global functions or variables which are shared between two or more files.
- Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.
- The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program.
- Notice that the extern variable cannot be initialized it has already been defined in the original file.
- Example: **extern** void display();

  **extern** int a;

## Example :

- **First File: main.c**

```
#include <stdio.h>
extern i;
void main()
{
		printf("value of the external integer is = %d\n", i);
}
```

- **Second File: original.c**

```
#include <stdio.h>
i=48;
//assign value of variable i that is declared in main.c
```

**Output :**

```
value of the external integer is = 48
```