

# MODULE 4

## Module-4

### Table of contents

Topics	Page No
Introduction	1
Properties of User Defined Functions	1
Function types	2
Elements of User Defined Function	2
Function Parameter	5
Category of functions	6
Passing Arrays to functions	13
Passing Strings to functions	17
Recursion	20
Scope ,Visibility and Lifetime of Variables	22
Storage classes in C	23

## Chapter 4

### USER DEFINED FUNCTIONS

#### 4.1 Introduction

Function is a self contained block of statement that performs a specific task or job. C program does not execute the functions directly. It is required to invoke or call that functions. When a function is called in a program then program control goes to the function body. Then, it executes the statements which are involved in a function body. Therefore, it is possible to call function whenever we want to process that functions statements.

#### 4.2 Properties of functions:

- Every function has a unique name. This name is used to call function from main().
- A function is independent and it can perform its task without interfering with other parts of the program.
- A function can be called from within another function.
- A function performs a specific task.
- Function can return only one value to the calling function via return.

**Advantages of using functions:** There are many advantages in using functions in a program they are:

- It is easy to use.
- Debugging is more suitable for programs.
- It reduces the size of a program.
- It is easy to understand the actual logic of a program.
- Highly suited in case of large programs.
- By using functions in a program, it is possible to construct modular and structured programs.

#### 4.3 Types of functions:

1) Built-in functions or Pre-Defined functions

## 2) User-Defined functions

**Built-in functions or Pre-Defined functions:** A function defined by the C compiler. It is also called as Library functions or Pre-Defined function or Built-in functions. Example: printf(), scanf(), strcpy(), strlen(), strcmp(), strcat()

**User-Defined functions:** C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions.

### 4.4 Elements of User-Defined Functions

In order to make use of a user-defined function, we need to establish three elements that are related to functions.

Function Definition

Function call

Function declaration (Function prototype)

Function prototype are not needed if user-definition function is written before main() function.

**Arguments/Parameters:** The variables which are inside the function call are called arguments or parameters.

**Function Prototype(declaration)** Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

*Syntax of function prototype :*

return\_type function\_name(type(1) argument(1),...,type(n) argument(n));

In the below example,

`int addgk(int a, int b);` is a function prototype which provides following information to the compiler:

1. name of the function is addgk()

2. return type of the function is int.
3. two arguments of type int are passed to function.

#### **4.4.1 Function call**

Control of the program cannot be transferred to user-defined function unless it is called invoked.

*Syntax of function call :*

```
function_name(argument(1),...,argument(n));
```

In the above example, function call is made using statement `addaj(num1,num2);` from `main()`. This makes the control of program jump from that statement to function definition and executes the codes inside that function.

#### **4.4.2 Function definition**

Function definition contains programming codes to perform specific task.

*Syntax of function definition :*

```
return_type function_name(type(1) argument(1),...,type(n) argument(n)) { //body of function }
```

Function definition has two major components:

##### **4.4.2.1 Function declarator**

Function declarator is the first line of function definition. When a function is called, control of the program is transferred to function declarator.

*Syntax of function declarator :*

```
return_type function_name(type(1) argument(1),...,type(n) argument(n))
```

Syntax of function declaration and declarator are almost same except, there is no semicolon at the end of declarator and function declarator is followed by function body. In above example, `int addgk(int a,int b)` in line 12 is a function declarator.

##### **4.2.2.2 Function body**


Function declarator is followed by body of function inside braces

### Passing arguments to functions

In programming, argument (parameter) refers to data this is passed to function(function definition) while calling function. In above example two variable, num1 and num2 are passed to function during function call and these arguments are accepted by arguments a and b in function definition.

```
#include<stdio.h>
int addgk(int, int);
void main( )
{
    .....
    Sum=addgk(x, y);
    .....
}

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



Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example: If argument *num1* was of int type and *num2* was of float type then, argument variable *a* should be of type int and *b* should be of type float, i.e., type of argument during function call and function definition should be same. A function can be called with or without an argument.

#### 4.2.2.3 Return Statement

Return statement is used for returning a value from function definition to calling function.

*Syntax of return statement :*

return (expression);

For example: return a; return (a+b);

In above example, value of variable add in addgk() function is returned and that value is stored in variable *sum* in main() function. The data type of expression in return statement should also match the return type of function.

#### 4.5 Function Parameter

Function parameters are the means of communication between the calling and the called functions. They are classified into Actual parameter and Formal parameters.

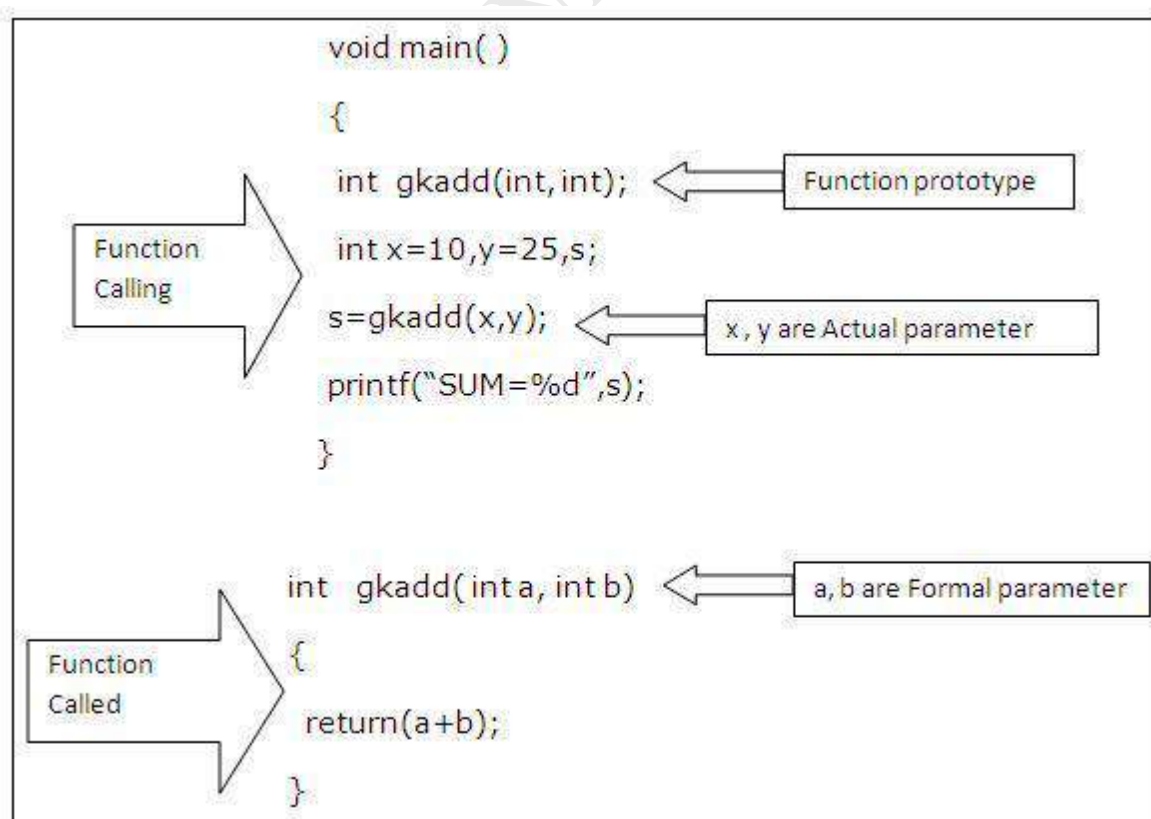
**4.5.1 Actual parameters** are specified in the function call often known as arguments.

**4.5.2 Formal parameters** are the parameters given in the function declaration and function definitions.

Number of actual parameters should be equivalent to the number of formal parameters.

Data types of the actual and formal parameters should be same.

The variable name of the formal parameters may be same as the actual parameters



## 4.6 Category of functions

A function depending on whether arguments are present or not and whether a value is returned or not, may belong to the following categories

- 1) Function with no arguments and no return value
- 2) Function with argument and no return value.
- 3) Function with no argument and return value.
- 4) Function with argument and return value
- 5) Function returns multiple values.

**4.6.1 Function with no arguments and no return value:** When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function. So, there is no data transfer between the calling function and the called function.

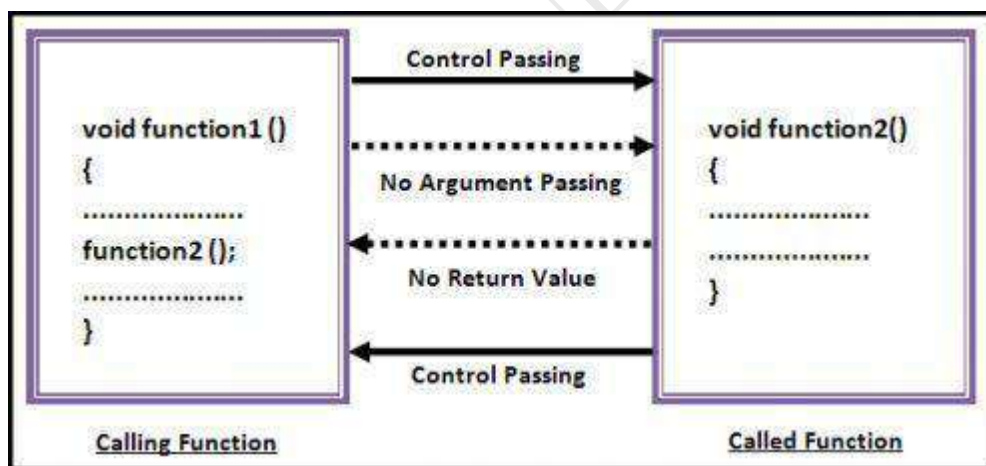


Fig:4.6.1: Function with no argument and no return value

Example to illustrate Function with no arguments and no return value:

```
void main()
{
printf("Hello\n");
```



```

ajmv( );

printf("I am fine\n");

}

void ajmv( )

{

printf("How are you?\n");

}

```

**Output:** Hello How are you? I am Fine

**4.6.2 Function with argument and no return value:** In this category, there is data transfer from the calling function to the called function using the arguments. But, there is no data transfer from the called function to the calling function. When arguments are passed, the function can receive values from the calling function. When the function does not return a value, the calling function cannot receive any value from the called function.

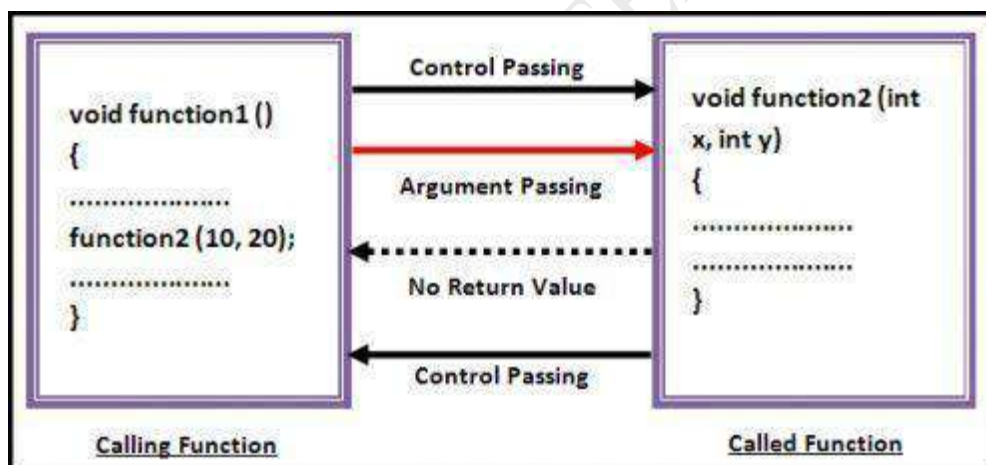


Fig:4.6.2: Function with argument and no return value

Example to illustrate Function with arguments and no return value:

```

void add(int, int);

void main()

{

int a=10,b=5;

```

```

add(a, b);

}

void add(int a, int b)

{

printf("Sum=%d",a+b);

}

```

**Output**

Sum=15

**4.6.3 Function with no argument and return value:** In this category, there is no data transfer from the calling function to the called function. But, there is data transfer from called function to the calling function. When no parameters are there, the function cannot receive any values from the calling function. When the function returns a value, the calling function receives one value from the called function.

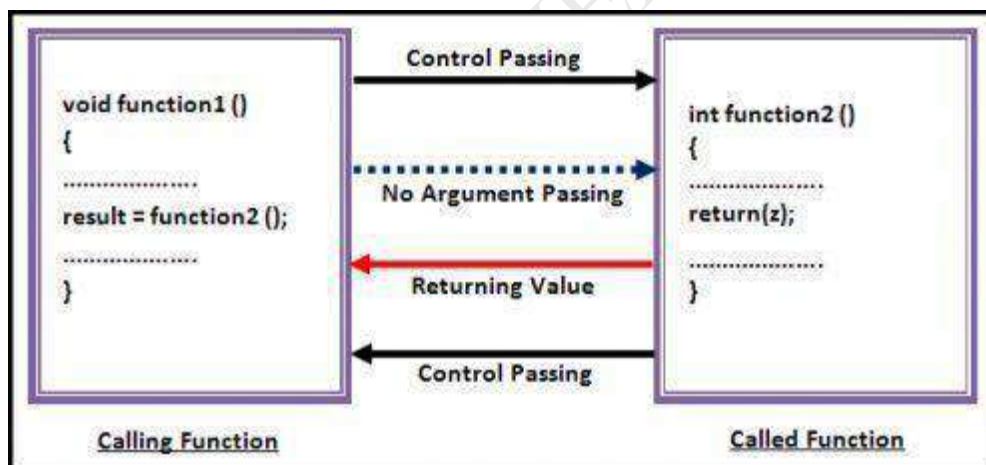


Fig:4.6.3: Function with no argument and return value

Example to illustrate Function with no arguments and return value:

```

int add();

void main()

{

printf("Sum=%d",add());

```

```

}

int add()
{
    int a=10,b=5;

    return(a+b);
}

```

### Output

Sum=15

**4.6.4 Function with argument and return value:** In this category, there is data transfer between the calling function and called function. When parameters are passed, the called function can receive values from the calling function. When the function returns a value, the calling function can receive a value from the called function.

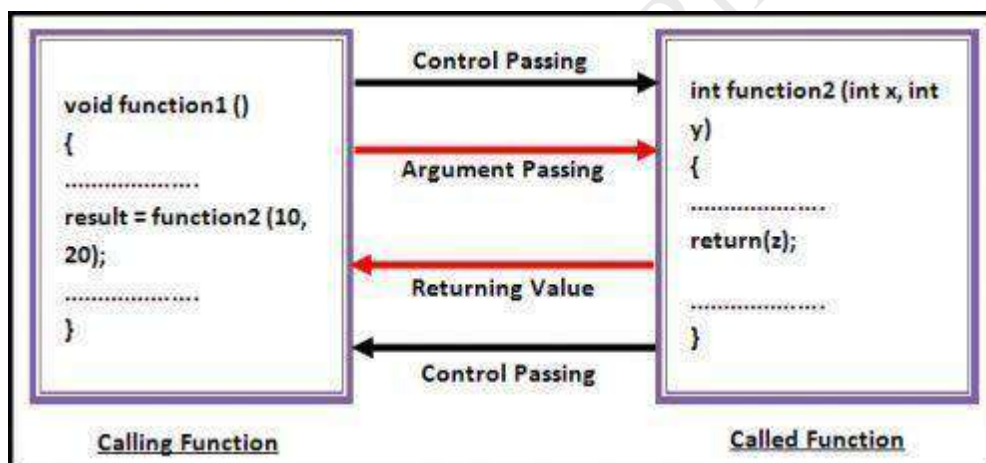


Fig:4.6.4: Function with argument and return

Example to illustrate Function with arguments and return value:

```

int add(int, int);

void main()
{
    int a=10,b=5;

```

```
printf("Sum=%d",add(a,b));  
  
}  
  
int add(int a, int b)  
{  
  
return (a+b);  
  
}
```

**Output**

Sum=15

**Write a program to compute the sum of even numbers and the sum of odd numbers using a function.**

```
int n,i,osum,esum;  
  
int ajodd(int);  
  
int ajeven(int);  
  
void main()  
{  
  
printf("Enter the value for n \n");  
  
scanf("%d",&n);  
  
printf("\n Sum of ODD number= %d ",ajodd(n)); printf("\n Sum of EVEN number= %d  
",ajeven(n));  
  
}  
  
int ajodd(int n)  
{  
  
int osum=0;  
for(i=1;i<=n;i+=2)  
osum=osum+i;  
return(osum);  
  
}
```

```
int ajeven(int n)
{
    int esum=0;
    for(i=2;i<=n;i+=2)
        esum=esum+i;
    return(esum);
}
```

#### 4.6.5 Function that return multiple values

The return statement can return only one value. Suppose we want more information from function, we can achieve this using arguments not only to receive the information but also to send back information to the calling function.

The arguments that are used to send out information are called output parameters. This mechanism of sending back information through arguments is achieved using address(&)operator and indirection operator(\*).

Example:

```
void calc(int x, int y, int *s, int *d);
main()
{
    int x=200,y=100,s,d;
    calc(x,y,&s,&d);
    printf("s=%d\nd=%d\n",s,d);
}

void calc(int a,int b,int *sum,int *dif)
{
    *sum=a+b;
    *dif=a-b;
}
```

The actual arguments x and y are input arguments, s and d are output arguments. In the function call, while we pass the actual values of x and y to the function, we pass the addresses of locations where the values of s and d are stored in memory.

The function works as follows

Value of x to a

Value of y to b

Address of s to sum

Address of d to dif

The operator \* is known as indirection operator because it gives an indirect reference to a variable through its address.

The variables sum and dif point to the memory locations of s and d respectively.

\*sum=a+b; adds the values a and b and the result is stored in the memory location pointed to by sum. This memory location is same as the memory location of s. The value stored in the location pointed to by sum is the value of s.

\*dif=a-b; the value of a-b is stored in the location pointed to by dif, which is same as memory location d.

Output will be s=300,d=100.

The variables \*sum and dif are called as pointers and sum and dif as pointer variables. Since they are declared as int, they can point to locations of int type data.

## 4.7 Passing arrays to functions

The arrays can be passed to functions using two methods:

- 1) Passing Individual Elements of Array
- 2) Passing the Whole Array

### 4.7.1 Passing Individual Elements of Array

- All array-elements can be passed as individual elements to a function like any other variable.
- The array-element is passed as a value parameter i.e. any change in the formal-parameter will not affect the actual-parameter i.e. array-element.
- Example: Program to illustrate passing individual elements of an array to a function.

```
#include<stdio.h>
void display(int a)
{
    printf("%d", a);
}

void main()
{
    int c[3]={2,3,4};
    display(c[2]);    //Passing array-element c[2] only.
}
```

**Output:**

4

#### 4.7.2. Passing entire one-dimensional array to a function

While passing arrays to the argument, the name of the array is passed as an argument(i.e, starting address of memory area is passed as argument).

- Array is passed to function Completely.
- Parameter Passing Method : Pass by Reference
- It is Also Called “Pass by Address“
- Original Copy is Passed to Function
- Function Body Can Modify Original Value.

**Write a C program to pass an array to a function. This function should find sum of the array elements and display the sum in main function.**

```
#include <stdio.h>

int sumofarray(int a[]);

void main()
{
    int sum, a[]={23, 55, 42, 15, 64, 18};

    sum=sumofarray(a); /* Only name of array is passed as argument.*/

    printf("Sum=%d",sum);
}
```

```
}  
  
int sumofarray(int A[])  
{  
    int i,sum=0;  
    for(i=0;i<6;i++)  
    {  
        sum=sum+A[i];  
    }  
    return(sum);  
}
```

Note: Here “a” is same as “A” because Base Address of Array “a” is stored in Array “A”.

#### **Output:**

Sum=217

### **4.8 Two Dimensional Array**

Like simple arrays, we can also pass multi-dimensional arrays to functions. The approach is similar to one-dimensional arrays. The rules are simple

1. The functions must be called by passing only the array name.
2. In the function definition, we must indicate that the array has two dimensions by including two sets of brackets
3. The size of the second dimension must be specified
4. The prototype declaration should be similar to the function header.

The function given below shows reading and printing of two dimensional array

```
#include <stdio.h>
```



```
void displayNumbers(int num[2][2]);

int main()
{
    int num[2][2];

    printf("Enter 4 numbers:\n");

    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);

    // passing multi-dimensional array to a function
    displayNumbers(num);

    return 0;
}

void displayNumbers(int num[2][2])
{
    printf("Displaying:\n");

    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            printf("%d\n", num[i][j]);
        }
    }
}
```

Output:

Enter 4 numbers:

2

3

4

5

Displaying:

2

3

4

5

## 4.9 Passing Strings to functions

The strings are treated as character arrays and so the rules for strings to functions are similar to passing arrays to functions.

Rules are:

1. The strings to be passed must be declared as a formal argument of the function when it is defined.

Example:

```
void show(char cust_name[])
```

```
{
```

```
.....
```

```
}
```

2. The function prototype may show that the argument is a string for the above example the prototype is given as

```
void show(char str[]);
```

3. A call to the function must have a string name without subscripts as its actual argument.

Example

show(name); where name is a properly declared string array in the calling function.

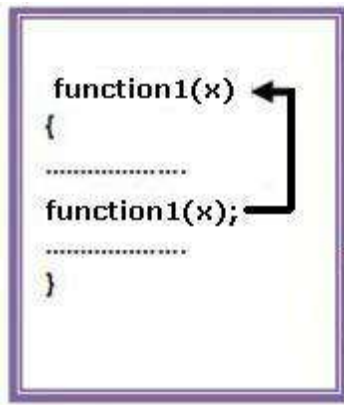
**Difference between Pass by value and Pass by Reference:** The following table gives the d

Table : Difference between Pass by value and Pass by Reference

PASS BY VALUE	PASS BY REFERENCE
Mechanism of copying the function parameter value to another variable	Mechanism of passing the actual parameters to the function
Changes made inside the function are not reflected in the original value	Changes made inside the function are reflected in the original value
Makes a copy of the actual parameter	Address of the actual parameter passes to the function
Function gets a copy of the actual content	Function accesses the original variable's content
Requires more memory	Requires less memory
Requires more time as it involves copying values	Requires a less amount of time as there is no copying

## 4.10 Recursions

A function that calls itself is known as recursive function and this technique is known as recursion in C programming. Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of previous result.



In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in recursive form, and second, the problem statement must include a stopping condition.

### Advantages and Disadvantages of Recursion

Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type. In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

**Write a recursive program to find the sum of first N natural numbers.**

```
void main()
{
    int ajsun(int);

    int n;

    printf("Enter the number\n");
```

```
scanf("%d",&n);

printf("Sum of the first %d natural numbers is %d\n",n,ajsum(n));

}

int ajsum(int n)

{

    if(n==0) return 0;

    else return (n+ajsum(n-1));

}
```

**Write a recursive program to find the factorial of a given number:**

```
void main()

{

    long int ajfact(int);

    int n;

    printf("Enter the number\n");

    scanf("%d",&n);

    printf("\n the Factorial of %d is %ld",n,ajfact(n));

}

long int ajfact(int n)

{

    if(n<=1)

        return 1;

    else return(n*ajfact(n-1));

}
```

**Write a C program to display Fibonacci series using recursive function.**

```
#include<stdio.h>

#include<conio.h>

int fib(int);
```

```
void main()
{
    int n,i;
    printf("Enter the value of n \n");
    scanf("%d",&n);
    printf("Fibonacci series\n");
    for ( i = 1 ; i <= n ; i++ )
    {
        printf("%d\n", fib(i));
    }
}

int fib(int n)
{
    if ( n == 0 || n == 1) return 0;
    else if ( n == 2 ) return 1; else
    return ( fib(n-1) + fib(n-2) );
}
```

#### 4.11 Scope ,Visibility and Lifetime of Variables

Scope of a variable is defined as the region or block of the program in which the variable is visible.

Global scope: Variables defined outside the block have global scope.

Local scope: Variables defined in the block have local scope.

Global variable: Defined before all functions in the global area of the program. Memory is allocated for these variables only once and all the allocated memory is initialized to 0. These variables can be accessed by any function and are active throughout the program.

Local variable: These are the variables defined within the function.

Example:

```
#include<stdio.h>
int x,y;
void main()
{
    int a=10,b=20;    //Local variable
    add(a,b);
    printf("a=%d,b=%d,x=%d",a,b,x);
}

void add(int a,int b)
{

    x=a+b;
}
```

Output:a=10,b=20,x=30

## 4.12 Storage Classes in C

Every variable in C programming has two properties: type and storage class. Type refers to the data type of variable whether it is character or integer or floating-point value etc. And storage class determines how long it stays in existence. There are 4 types of storage class: **auto register static external**

### a) auto

**auto** is the default storage class for all local variables. Variables declared inside the function body are automatic by default. These variable are also known as local variables as they are local to the function and doesn't have meaning outside that function. Since, variable inside a function is automatic by default, keyword **auto** are rarely used. Example: auto int x; Default value of *auto* variables is garbage value. *auto* variables are stored in the memory.

### b) register

**register** 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). Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

Example: register int p;

### c) static

A variable for which memory remains allocated throughout the execution of the entire program is called *static* variable. ie, the value of the *static* variables continue until the end of the program.

Example: static int b;

```
#include <stdio.h>
```

```
void Check();
```

```
void main()
```

```
{
```

```
    checkgk();
```

```
    checkgk();
```

```
    checkgk();
```

```
}
```

```
void checkgk()
```

```
{
```

```
    static int a=0;
```

```
    printf("%d\t",a);
```

```
    a=a+5;
```

```
}
```

The output is 0 5 10



**d) external**

External variable can be accessed by any function. They are also known as global variables. Variables declared outside every function are external variables.

By default, all the global variables are external variables. We may also use the keyword **extern** to declare external variables explicitly. Example: `extern int a;`

Default value of external storage class is zero. External variables are stored in the memory.

```
#include <stdio.h>
void extergk();
int a=5; /* a is global variable because it is outside every function */
void main( )
{
    a+=4;
    extergk ();
}
void extergk ( )
{
    ++a;
    printf("a=%d",a);
}
```

The Output is a=10

## Web Resources

1. User defined functions: <https://www.w3resource.com/c-programming-exercises/function/index.php>

2. Recursion: <https://www.w3resource.com/c-programming-exercises/recursion/index.php>

## Video Resources

### 1. User defined functions:

[https://www.youtube.com/watch?v=JenkXelHmM&feature=emb\\_logo](https://www.youtube.com/watch?v=JenkXelHmM&feature=emb_logo)

[https://www.youtube.com/watch?v=zUzEbLdt8zU&feature=emb\\_logo](https://www.youtube.com/watch?v=zUzEbLdt8zU&feature=emb_logo)

[https://www.youtube.com/watch?v=AJvCmpt1UU8&feature=emb\\_logo](https://www.youtube.com/watch?v=AJvCmpt1UU8&feature=emb_logo)

### 2. Recursion:

[https://www.youtube.com/watch?time\\_continue=1&v=LoIe\\_9cTtPE&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=1&v=LoIe_9cTtPE&feature=emb_logo)

[https://www.youtube.com/watch?time\\_continue=32&v=6ZAitmMODIE&feature=emb\\_logo](https://www.youtube.com/watch?time_continue=32&v=6ZAitmMODIE&feature=emb_logo)

## Question Bank

1. What is function parameter? Explain different types of parameters in c functions?
2. Illustrate the elements of user defined functions with examples.
3. Write a program to find the factorial of a given integer using functions.
4. Explain how call by value differs from call by reference while invoking a function.
5. Explain the categories of user defined functions.
6. Define the following i) Global variable ii) Local variable
7. Define Recursion. Write a program to compute the fibonacci series upto n using recursion.
8. List the storage class specifiers and explain.

