MODULE 3

Arrays

3.1 INTRODUCTION

Arrays: Array is a sequential collection of similar data items.

Pictorial representation of an array of 5 integers

10	20	30	40	50
A[0]	A[1]	A[2]	A[3]	A[4]

- An array is a collection of similar data items.
- All the elements of the array share a common name .
- Each element in the array can be accessed by the subscript(or index) and array name.
- The arrays are classified as:
  1. **Single dimensional array**
  2. **Multidimensional array.**

3.1.1 Single Dimensional Array.

- A single dimensional array is a linear list of related data items of same data type.
- In memory, all the data items are stored in contiguous memory locations.

Declaration of one-dimensional array(Single dimensional array)

Syntax:

```
datatype  array_name[size];
```

- **datatype** can be int,float,char,double.
- **array_name** is the name of the array and it should be an valid identifier.
- **Size** is the total number of elements in array.

For example:

```
int   a[5];
```

The above statement allocates 5*2=10 Bytes of memory for the array **a.**

a[0]	a[1]	a[2]	a[3]	a[4]

```
float b[5];
```

The above statement allocatests 5*4=20 Bytes of memory for the array **b.**

- Each element in the array is identified using integer number called as i**ndex.**
- If n is the size of array, the array index starts from **0** and ends at **n-1.**

Storing Values in Arrays

- Declaration of arrays only allocates memory space for array. But array elements are not initialized and hence values has to be stored.
- Therefore to store the values in array, there are 3 methods
    1. Initialization
    2. Assigning Values
    3. Input values from keyboard through **scanf()**

3.1.2 Initialization of one-dimensional array

- **Assigning the required values to an array elements before processing is called initialization.**

```
data type array_name[expression]={v1,v2,v3…,vn};
```

Where

- ✓ datatype can be char,int,float,double
- ✓ array name is the valid identifier
- ✓ size is the number of elements in array
- ✓ v1,v2,v3……..vn are values to be assigned.

- Arrays can be initialized at declaration time.

Example:

int a[5]={2,4,34,3,4};

2	4	34	3	4
a[0]	a[1]	a[2]	a[3]	a[4]

- The various ways of initializing arrays are as follows:
    1. **Initializing all elements of array(Complete array initialization)**
    2. **Partial array initialization**

3. **Initialization without size**
4. **String initialization**

1. Initializing all elements of array:

- Arrays can be initialized at the time of declaration when their initial values are known in advance.
- In this type of array initialization, initialize all the elements of specified memory size.
- Example:

int a[5]={10,20,30,40,50};

10	20	30	40	50

2. Partial array initialization

- If the number of values to be initialized is less than the size of array then it is called as partial array initialization**.**
- In such a case elements are initialized in the order from 0^{th} element.
- The remaining elements will be initialized to **zero automatically by the compiler.**
- Example:
  int a[5]={10,20};

10	20	0	0	0

3. Initialization without size

- In the declaration the array size will be set to the total number of initial values specified.
- The compiler will set the size based on the number of initial values.
- Example:

int a[ ]={10,20,30,40,50};

- **In the above example the size of an array is set to 5**

4. String Initialization

- Sequence of characters enclosed within double quotes is called as string.
- The string always ends with NULL character**(\0)**

char s[5]=”SVIT”;

We can observe that string length is 4,but size is 5 because to store NULL character we need one more location.

So pictorial representation of an array **s** is as follows:

S	V	I	T	\0
S[0]	S[1]	S[2]	S[3]	S[4]

3.1.2 Assigning values to arrays

Using assignment operators, we can assign values to individual elements of arrays.
For example:

```
int  a[3];
        a[0]=10;
        a[1]=20;
        a[2]=30;
```

10	20	30
a[0]	a[1]	a[2]

Reading and writing single dimensional arrays.

To read array elements from keyboard we can use **scanf()** function as follows:

To read **0th** element: scanf("%d",&a[0]);
To read **1st** element: scanf("%d",&a[1]);
To read **2nd** element: scanf("%d",&a[2]);
……
…….
To read **nth** element : scanf("%d",&a[n-1]);
In general
To read **ith** element:
scanf("%d",&a[i]); where i=0; i<n; i++

To print array elements we can use **printf()** function as follows:

To print **0th** element: printf("%d",a[0]);
To print **1st** element: printf("%d",a[1]);
To print **2nd** element :printf("%d",a[2]);
……..
……..

To **nth** element : printf("%d",&a[n-1]);
In general
To read **ith** element:
printf("%d",a[i]); where i=0; i<n; i++

1.	Write a C program to read N elements from keyboard and to print N elements on screen.
	/* program to read N elements from keyboard and to print N elements on screen */ **#include<stdio.h>** **void main()** **{** **int i,n,a[10];** **printf("enter number of array elements\n");** **scanf("%d",&n);** **printf("enter array elements\n");** **for(i=0; i<n;i++)** **{** **scanf("%d",&a[i]);** **}** **Printf("array elements are\n"):** **for(i=0; i<n;i++)** **{** **printf("%d",a[i]);** **}** **}**
2.	Write a C program to find sum of n array elements .
	/* program to find the sum of n array elements.*/ **#include<stdio.h>** **void main()** **{** **int i,n,a[10],sum=0;** **printf("enter number of array elements\n");** **scanf("%d",&n);** **printf("enter array elements\n");** **for(i=0; i<n; i++)** **{** **scanf("%d",&a[i]);** **}** **for(i=0; i<n;i++)** **{** **sum=sum+ a[i];**

	} **printf("sum is %d\n",sum):** **}**
3.	Write a c program to find largest of n elements stored in an array a.
	#include<stdio.h> **void main()** **{** **int i,n,a[10],big;** **printf("enter number of array elements\n");** **scanf("%d",&n);** **printf("enter array elements\n");** **for(i=0; i<n;i++)** **{** **scanf("%d",&a[i]);** **}** **big=a[0];** **for(i=0; i<n;i++)** **{** **if(a[i]>big)** **big=a[i];** **}** **printf("the biggest element in an array is %d\n",big);** **}**
4.	**Write a C program to generate Fibonacci numbers using arrays.**
	#include<stdio.h> **void main()** **{** **int i,n,a[10];** **a[0]=0;** **a[1]=1;** **printf("enter n\n");** **scanf("%d",&n);** **if(n==1)** **{** **printf("%d\t",a[0]);** **}** **if(n==2)** **{** **printf("%d\t %d\t",a[0],a[1]);** **}** **if(n>2)**

```
        {
                printf("%d \t %d\t",a[0],a[1]);
                for(i=2;i<n;i++)
                {
                        a[i]=a[i-1]+a[i-2];
                        printf("%d\t",a[i]);
                }
        }
}
```

3.2Searching

- The process of finding a particular item in the large amount of data is called searching.
- The element to be searched is called key element.

There are two methods of searching:

1] Linear search.

2] Binary search.

1] Linear Search:

- Linear search also called sequential search is a simple searching technique.
- In this technique we search for a given key item in linear order i.e,one after the other from first element to last element.
- The search may be successful or unsuccessful.
- If key item is present, the search is successful, otherwise unsuccessful search.

1. Program to implement linear search.

```
#include<stdio.h>
void main()
{
        int i,n,a[10],key;
        clrscr( );
        printf("enter array elements\n");
        scanf("%d",&n);
        printf("enter array elements\n");
        for(i=0; i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("enter the key element\n");
        scanf("%d",,&key);

        for(i=0; i<n;i++)
        {if(key==a[i])
```

```
                {
                        printf("successful search\n");
                        exit(0);
                }
        }
        printf("unsuccessful search\n");
}
```

Advantages of linear search

- Very simple Approach.
- Works well for small arrays.
- Used to search when elements are not sorted.

Disadvantages of linear search

- Less efficient if array size is large
- If the elements are already sorted, linear search is not efficient.

2] Binary Search:

- Binary search is a simple and very efficient searching technique which can be applied if the items are arranged in either ascending or descending order.
- In binary search first element is considered as low and last element is considered as high.
- Position of middle element is found by taking first and last element is as follows.

  mid=(low+high)/2

- Mid element is compared with key element, if they are same, the search is successful.
- Otherwise if key element is less than middle element then searching continues in left part of the array.
- If key element is greater than middle element then searching continues in right part of the array.
- The procedure is repeated till key item is found or key item is not found.

write a C program to perform binary search on the array of integers

```
/* C program to search a name in a list of names using Binary searching technique*/
        #include<stdio.h>
        void main()
        {
                int i, n, low, high, mid,a[50],key;
                printf("enter the number of elements\n"):
                scanf("%d",&n);
                printf("enter the elements\n");
                for(i=0;i<n;i++)
                {
                        Scanf("%d",&a[i]);
                }

                printf("enter the key element to be searched\n");
                scanf("%d",&key);
```

```
        low=0;
        high=n-1;
        while(low<=high)
        {
                mid=(low+high)/2;
                if(key==a[mid])
                {
                        printf("successful search\n");
                        exit(0);
                }
                if(key<a[mid])
                {
                        high=mid-1;
                }
                else
                {
                        low=mid+1;
                }
        }
        printf("unsuccesfull seasrch\n");
}
```

Advantages of binary search

1. Simple technique
2. Very efficient searching technique

Disadvantages

1. The elements should be sorted.
2. It is necessary to obtain the middle element, which are stored in array. If the elements are stored in linked list, this method cannot be used.

3.3Sorting

- The process of arranging elements in either ascending order or descending order is called Sorting.

Bubble Sort

- This is the simplest and easiest sorting technique.
- In this technique two successive elements of an array such as a[j] and a[j+1] are compared.
- If a[j]>=a[j+1] the they are exchanged, this process repeats till all elements of an array are arranged in ascending order.
- After each pass the largest element in the array is sinks at the bottom and the smallest element in the array is bubble towards top. So this sorting technique is also called as sinking sort and bubble sort.

PROGAM FOR BUBBLE SORT

```
#include<stdio.h>
void main()
{
        int a[20],n,.temp,i,j;
        printf("enter the number  of elements\n"):
        scanf("%d",&n);
        printf("enter the unsorted array elements\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        for(i=1;i<n;i++)
        {
                for(j=0;j<n-i;j++)
                {
                    If( a[ i ] > a[ i+1 ] )
                    {
                        temp=a[j];
                        a[j]=a[j+1];
                        a[j+1]=temp;
                     }
                }
        }
           printf("the sorted elements arer\n");
        for(i=0;i<n;i++)
        printf(" %d\t", a[i]);
    }
```

Write a C program to evaluate the polynomial using Horners method.

```
#include<stdio.h>
void main()
{
        int i,x,n,a[10],sum;
        printf("enter n:\n");
        scanf("%d",&n);
        printf("enter n+1 co efficient\n");
        for(i=0; i<=n;i++)
        {
                scanf("%d",&a[i]);
        }
        sum=a[n]* x;
        for(i=n-1; i>0; i--)
        {
                sum=(sum+a[i]) *x;
        }
        sum=sum+a[0];
        printf("sum of polynomial equation is %d",sum);
}
```

3.4 Two Dimensional arrays:

- In two dimensional arrays, elements will be arranged in rows and columns.
- To identify two dimensional arrays we will use two indices(say i and j) where I index indicates row number and j index indicates column number.

3.4.1Declaration of two dimensional array:

data_type array_name[exp1][exp2];

Or

data_type array_name[row_size][column_size];

- **data_type** can be int,float,char,double.
- **array_name** is the name of the array.
- **exp1 and exp2** indicates number of rows and columns

For example:

```
int   a[2][3];
```

✓ The above statements allocates memory for 3*4=12 elements i.e 12*2=24 bytes.

3.4.2 Initialization of two dimensional array

Assigning or providing the required values to a variable before processing is called initialization.

```
Data_type array_name[exp1][exp2]={
                                    {a1,a2,….an},
                                   {b1,b2,  .bn},
                                    …….
                                    ……..
                                   {z1,z2,…zn}
                                };
```

- Data type can be int,float etc.
- exp1 and exp2 are enclosed within square brackets .
- both exp1 and exp2 can be integer constants or constant integer expressions(number of rows and number of columns).
- a1 to an are the values assigned to 1st row ,
- b1 to bn are the values assigned to 2nd row and so on.

Example:

```
int a[3][3]={
                {10,20,30},
                {40,50,60},
                {70,80,90}
            };
```

10	**20**	**30**
40	**50**	**60**
70	**80**	**90**

Partial Array Initialization

- If the number of values to be initialized is less than the size of array, then the elements are initialized from left to right one after the other.
- The remaining locations initialized to zero automatically.
- Example:

```
int a[3][3]={
                {10,20},
                {40,50},
                {70,80}
            };
```

10	**20**	**0**
40	**50**	**0**

70	80	0

1. **Write a c program to read & print 2d array as a Array.**

```
#include<stdio.h>
void main()
{
    int m,n,i,j,a[3][3];
    printf("enter number of rows and columns\n");
    scanf("%d %d",&m,&n);
    printf("eneter array elements\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
    printf("array elements are\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d",a[i][j]);
        }
        printf("\n");
    }
}
```

2 **Write a c program to add two matrices.**

```
#include<stdio.h>
void main()
{
    int m,n,i,j,a[3][3],b[3][3] ,c[3][3];
    printf("enter number of rows and columns\n");
    scanf("%d %d",&m,&n);
    printf("enter array  a elements\n");
    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            scanf("%d",&a[i][j]);
        }
    }
```

```
        printf("enter array  b elements\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&b[i][j]);
                }
        }

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        c[i][j]=a[i][j]+b[i][j];
                }
        }
        printf("resultant matrix c is \n");

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t",c[i][j]);
                }
                printf("\n");
        }
}
```

3 **Write a c program to copy one 2d array in to another 2d array**

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int m,n,i,j,a[3][3],b[3][3];
        clrscr();
        printf("enter number of rows and columns\n");
        scanf("%d %d",&m,&n);
        printf("enter array  a elements\n");
        for(i=0; i<m; i++)
        {
                for(j=0; j<n; j++)
                {
                        scanf("%d", &a[i][j]);
                }
        }

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
```

```
                {
                        b[i][j]=a[i][j];
                }
        }
        printf("matrix b  is \n");

        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t",b[i][j]);
                }
                printf("\n");
        }
}
```

4 **Write a c program to find biggest element in a matrix or 2D array.**

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

        int m,n,i,j,a[3][3];
        clrscr();
        printf("enter number of rows and columns\n");
        scanf("%d %d",&m,&n);
        printf("enter array elements\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }
        big=a[0][0];
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        if(big>a[i][j])
                        big=a[i][j];
                }
        }
        printf("big is %",big);
}
```

5	**Write a C program to implement Matrix Multiplication**

```
#include<stdio.h>
void main()
{
        int m,n,i,j,sum,p,q,k,a[3][3],b[3][3],c[3][3];
        printf("enter number of rows and columns of matrix a \n");
        scanf("%d %d",&m,&n);
        printf("enter number of rows and columns of matrix b \n");
        scanf("%d %d",&p,&q);
        if(n!=p)
        {
                printf("multiplication not possible\n"):
                exit(0);
        }

        printf("enter matrix   a elements\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }

        printf("enter array  b elements\n");
        for(i=0;i<p;i++)
        {
                for(j=0;j<q;j++)
                {
                        scanf("%d",&b[i][j]);
                }
        }

        for(i=0;i<m;i++)
        {
                for(j=0;j<q;j++)
                {
                        {
                                c[i][j]=0;
                                for(k=0;k<n;k++)
                                {
                                        c[i][j]= c[i][j]+a[i][k]*b[k][j];
                                }

                }
        }
        printf("resultant matrix a is \n");

        for(i=0;i<m;i++)
```

```
        {
                for(j=0;j<n;j++)
                {
                        printf("%d\t",a[i][j]);
                }
                printf("\n");
        }
        printf("resultant matrix a is \n");

        for(i=0;i<p;i++)
        {
                for(j=0;j<q;j++)
                {
                        printf("%d\t",b[i][j]);
                }
                printf("\n");
        }
printf("resultant matrix a is \n");

        for(i=0;i<m;i++)
        {
                for(j=0;j<q;j++)
                {
                        printf("%d\t",c[i][j]);
                }
                printf("\n");
        }

}
```

6 **Write a program to find sum of each row and sum of each column**

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

        int m,n,i,j,rsum,csum,a[3][3];
        printf("enter number of rows and columns\n");
        scanf("%d %d",&m,&n);
        printf("enter array elements\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }
        for(i=0;i<m;i++)
        {
                rsum=0;
```

```
                for(j=0;j<n;j++)
                {
                        rsum=rsum+a[i][j];
                }
                printf("sum is %d",rsum);
        }

        for(i=0;i<m;i++)
        {
                csum=0;
                for(j=0;j<n;j++)
                {
                        csum=csum+a[i][j];
                }
                printf("sum is %d",csum);
        }

}
```

7 **Write a C program to add all 2D array elements**

```
#include<stdio.h>
void main()
{
        int m,n,i,j,sum=0,a[3][3];
        printf("enter number of rows and columns\n");
        scanf("%d %d",&m,&n);
        printf("enter array elements\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                {
                        scanf("%d",&a[i][j]);
                }
        }
        for(i=0;i<m;i++)
        {

                sum=sum+a[i][j];

        }
        printf("sum is %d",rsum);
}
```

FUNCTIONS

- A function as series of instructions or group of statements with one specific purpose.
- A function is a program segment that carries out some specific, well defined task.
- A function is a self contained block of code that performs a particular task.

4.1 Types of functions

- C functions can be classified into **two** types,
  1. Library functions /pre defined functions /standard functions /built in functions
  2. User defined functions

1. Library functions /pre defined functions /standard functions/Built in Functions

- These functions are defined in the **library of C compiler** which are used frequently in the C program.
- These functions are written by designers of c compiler.
- C supports many built in functions like
  - Mathematical functions
  - String manipulation functions
  - Input and output functions
  - Memory management functions
  - Error handling functions
- EXAMPLE:
  - pow(x,y)-computes $\mathbf{x^y}$
  - sqrt(x)-computes square root of x
  - printf()- used to print the data on the screen
  - scanf()-used to read the data from keyboard.

2. User Defined Functions

- The functions written by the programmer /user to do the specific tasks are called user defined function(UDF's).
- The user can construct their own functions to perform some specific task. This type of functions created by the user is termed as User defined functions.

4.2 Elements of User Defined Function

The Three Elements of User Defined function structure consists of :

1. **Function Definition**
2. **Function Declaration**
3. **Function call**

1. Function Definition:

- A program Module written to achieve a specific task is called as function definition.
- Each function definition consists of two parts:
  - **i. Function header**
  - **ii. Function body**

General syntax of function definition

Function Definition Syntax	**Function Definition Example**

datatypefunctionname(parameters)
{
declaration part;
executable part;
return statement;
 	
}
	void add()
	{
	int sum,a,b;
	printf("enter a and b\n");
	scanf("%d%d",&a,&b);
	sum=a+b;
	printf("sum is %d",sum);
	}

i. Function header

Syntax

datatype functionname(parameters)

- It consists of **three** parts
  - **a) Datatype:**
    - ✓ **The data type can be int,float,char,double,void.**
    - ✓ This is the data type of the value that the function is expected to return to calling function.
  - **b) <u>functionname</u>**:
    - ✓ The name of the function.

- ✓ It should be a valid identifier.

c) <u>parameters</u>

- ✓ The parameters are list of variables enclosed within parenthesis.
- ✓ The list of variables should be separated by comma.

Ex: **void add(int a, int b)**

- In the above example the return type of the function is **void**
- the name of the function is **add** and
- The parameters are '**a**' **and** '**b**' of type integer.

ii. Function body

- The function body consists of the set of instructions enclosed between { **and** } **.**
- The function body consists of following three elements:

a) **declaration part:** variables used in function body.

b) **executable part:** set of Statements or instructions to do specific activity.

c) **return :** It is a keyword,it is used to **return control back to calling function.**

If a function is not returning value then statement is:

return;

If a function is returning value then statement is:

return value;

2. Function Declaration

- The process of declaring the function before they are used is called as function declaration or function prototype.
  - function declaration Consists of the data type of function, name of the function and parameter list ending with semicolon.

Function Declaration Syntax
datatypefunctionname(type p1,type p2,.....type pn);
Example
int add(int a, int b);
void add(int a, int b);

Note: The function declaration should end with a <u>semicolon ;</u>

3. Function Call:

- The method of calling a function to achieve a specific task is called as function call.
- A function call is defined as **function name followed by semicolon.**
- A function call is nothing but invoking a function at the required place in the program to achieve a specific task.

Ex:

```
void main()
{
        add( ); // function call without parameter
}
```

4.3 Formal Parameters and Actual Parameters

- **Formal Parameters:**

- The variables defined in the **function header of function definition** are called **formal** parameters.
- All the variables should be separately declared and each declaration must be separated by **commas.**
- The formal parameters **receive the data from actual parameters.**

- **Actual Parameters:**

- The variables that are used when a function is invoked)in function call) are called **actual** parameters.
- Using actual parameters, the data can be transferred from calling function. to the called function.
- The corresponding **formal** parameters in the **function definition** receive them.
- The **actual** parameters and **formal** parameters must match in number and type of data.

- **Differences between Actual and Formal Parameters**

Actual Parameters	**Formal Parameters**
Actual parameters are also called as **argument list.** Ex: add(m,n)	Formal parameters are also called as **dummy parameters.** **Ex:**int add(int a, int b)
The variables used in function call are called as actual parameters	The variables defined in function header are called formal parameters
Actual parameters are used in calling function when a function is called or invoked Ex: add(m,n) Here, m and n are called actual parameters	Formal parameters are used in the function header of a called function. Example: int add(int a, int b) { ……. } Here, a and b are called formal parameters.
Actual parameters sends data to the formal parameters Example:	Formal parameters receive data from the actual parameters.

4.4 Categories of the functions

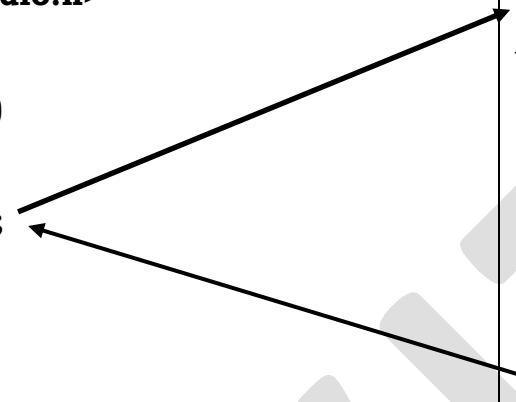
1. **Function with no parameters and no return values**
2. **Function with no parameters and return values.**
3. **Function with parameters and no return values**
4. **Function with parameters and return values**

1. Function with no parameters and no return values

1. Function with no parameters and no return values
(void function without parameter)

Calling function	**Called function**

/*program to find sum of two numbers using function*/
#include<stdio.h>
void add( );
void main( )
{
add( );
}
	void add ( )
	{
	int sum;
	printf("enter a and b values\n");
	scanf("%d%d",&a,&b);
	sum=a+b;
	printf("\n The sum is %d", sum);
	return;
	}



- ✓ In this category **no data** is transferred from **calling function to called function**, hence called function cannot receive any values.
- ✓ In the above example,no arguments are passed to user defined function **add( ).**
- ✓ Hence no parameter are defined in function header.
- ✓ When the control is transferred from calling function to called function a ,and b values are read,they are added,the result is printed on monitor.
- ✓ When return statement is executed ,control is transferred from called function/add to calling function/main.

2. **Function with parameters and no return values**
   (void function with parameter)

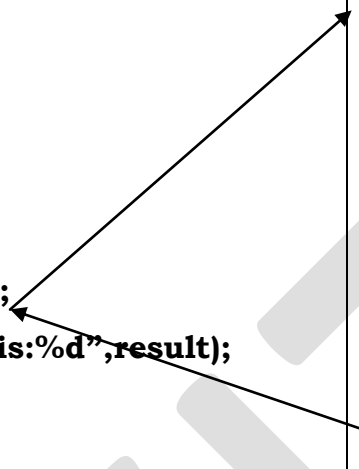
Calling function	**Called function**

/*program to find sum of two numbers using function*/
#include<stdio.h>
void add(int m, int n);
void main()
{
int m,n;
printf("enter values for m and n:");
scanf("%d %d",&m,&n);
add(m,n);
}
	void add(int a, int b)
	{
	int sum;
	sum = a+b;
	printf("sum is:%d",sum);
	return;
	}

- ✓ In this category, **there is data transfer** from the calling function to the called function using parameters.
- ✓ **But there is no data transfer from** called function to the calling function.
- ✓ The values of actual parameters m and n are copied into formal parameters a and b.
- ✓ The value of a and b are added and result stored in sum is displayed on the screen in called function itself.

3. Function with no parameters and with return values

Calling function	**Called function**
/*program to find sum of two numbers using function*/ **#include<stdio.h>** **int add();** **void main()** **{** **int result;** **result=add( );** **printf("sum is:%d",result);** **}**	**int add( ) /* function header */** **{** **int a,b,sum;** **printf("enter values for a and b:");** **scanf("%d %d",&a,&b);** **sum= a+b;** **return sum;** **}**



- ✓ 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.
- ✓ No arguments are passed to the function add( ). So, no parameters are defined in the function header
- ✓ When the function returns a value, the calling function receives one value from the called function and assigns to variable result.
- ✓ The result value is printed in calling function.

4. Function with parameters and with return values

Calling function	**Called function**

/*program to find sum of two numbers using function*/
#include<stdio.h>
int add();
void main()
{
int result,m,n;
printf("enter values for m and n:");
scanf("%d %d",&m,&n);
result=add(m,n);
printf("sum is:%d",result);
}
	int add(int a, int b) /* function header */
	{
	int sum;
	sum= a+b;
	return sum;
	}

- ✓ In this category, there is data transfer between the calling function and called function.
- ✓ When Actual parameters values are passed, the formal parameters in called function can receive the values from the calling function.
- ✓ When the add function returns a value, the calling function receives a value from the called function.
- ✓ The values of actual parameters m and n are copied into formal parameters a and b.
- ✓ Sum is computed and returned back to calling function which is assigned to variable result.

4.5 Passing parameters to functions or Types of argument passing

The different ways of passing parameters to the function are:

- ✓ **Pass by value or Call by value**
- ✓ **Pass by address or Call by address**

1. Call by value:

- In call by value, the values of actual parameters are copied into formal parameters.
- The formal parameters contain only a copy of the actual parameters.
- So, even if the values of the formal parameters changes in the called function, the values of the actual parameters are not changed.
- The concept of call by value can be explained by considering the following program.

Example:

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

void swap(int a, int b)
 {
        int temp;
        temp=a;
        a=b;
        b=temp;
  }
```

- Execution starts from function main( ) and we will read the values for variables m and n, assume we are reading 10 and 20 respectively.
- We will print the values before swapping it will print 10 and 20.
- The function swap( ) is called with actual parameters m=10 and n=20.
- In the function header of function swap( ), the formal parameters a and b receive the values 10 and 20.
- In the function swap( ), the values of a and b are exchanged.

- But, the values of actual parameters m and n in function main( ) have not been exchanged.
- The change is not reflected back to calling function.

2. Call by Address

- In Call by **Address,** when a function is called, the addresses of actual parameters are sent.
- In the called function, the formal parameters should be declared as pointers with the same type as the actual parameters.
- The addresses of actual parameters are copied into formal parameters.
- Using these addresses the values of the actual parameters can be changed.
- This way of changing the actual parameters indirectly using the addresses of actual parameters is known as pass by address.

Example:

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

void swap(int*a, int*b)
 {
        int temp;
        temp=*a;
        *a=*b;
        *b=temp;
  }
```

NOTE:

Pointer: A pointer is a variable that is used to store the address of another variable.
Syntax: datatype *variablename;
Example: int *p;

```
#include<stdio.h>
void main()
{
        inta ,*p;
        p=&a;
}
```

In the above program **p** is a **pointer** variable, which is storing the address of variable **a**.

Differences between Call by Value and Call by reference

Call by Value	Call by Address
When a function is called the **values of variables are passed**	When a function is called the **addresses of variables are passed**
The type of formal parameters should be same as type of actual parameters	The type of formal parameters should be same as type of actual parameters, but they have to be declared as **pointers.**
Formal parameters contains the values of actual parameters	Formal parameters contain the addresses of actual parameters.

4.6 Scope and Life time of a variable

Scope of a variable is defined as the region or boundary of the program in which the variable is visible. There are two types

(i) Global Scope

(ii) Local Scope

i. Global Scope:

- The variables that are defined outside a block have global scope.
- That is any variable defined in global area of a program is visible from its definition until the end of the program.
- For Example, the variables declared before all the functions are visible everywhere in the program and they have global scope.

ii. Local Scope

a. The variables that are defined inside a block have local scope.
b. They exist only from thepoint of their declaration until the end of the block.
c. They are not visible outside the block.

❖ **Life Span of a variable**

➢ The life span of a variable is defined as the period during which a variable is active during execution of a program.

For Example

❖ The life span of a global variable is the life span of the program.
❖ The life span of local variables is the life span of the function, they are created.

❖ **Storage Classes**

➢ *There are following storage classes which can be used in a C Program:*

i. Global variables
ii. Local variables
iii. Static variables
iv. Register variables

i. Global variables:

➢ These are the variables which are defined before all functions in global area of the program.
➢ Memory is allocated only once to these variables and initialized to zero.
➢ These variables can be accessed by any function and are alive and active throughout the program.
➢ Memory is deallocated when program execution is over.

ii. Local variables(automatic variables)

➢ These are the variables which are defined within a functions.
➢ These variables are also called as automatic variables.
➢ The scope of these variables are limited only to the function in which they are declared and cannot be accessed outside the function.

iii. Static variables

➢ The variables that are declared using the keyword static are called static variables.

- The static variables can be declared outside the function and inside the function. They have the characteristics of both local and global variables.
- Static can also be defined within a function.

  Ex:

  static int a,b;

iv. Register variables

- Any variables declared with the qualifier register is called a register variable.
- This declaration instructs the compiler that the variable under use is to be stored in one of the registers but not in main memory.
- Register access is much faster compared to memory access.

  Ex:

  register int a;

Recursion

- Recursion is a method of solving the problem where the solution to a problem depends on solutions to smaller instances of the same problem.
- Recursive function is a function that calls itself during the execution.
- The two types of recursion are

  1.Direct Recursion

  2. Indirect Recursion

1.Direct recursion

- A recursive function that invokes itself is said to have direct recursion.
- For example factorial function calls itself hence it is called direct recursion.

2. Indirect recursion

- A function which contains call to another function which in turn contains calls another function, and so on.

Design of recursive function

- ***Any recursive function has two elements:***

i. Base case

ii. General case

i. **Base case**

- The statement that solves the problem is called base case.
- Every recursive function must have at least one base case.
- It is a special case whose solution can be obtained without using recursion.

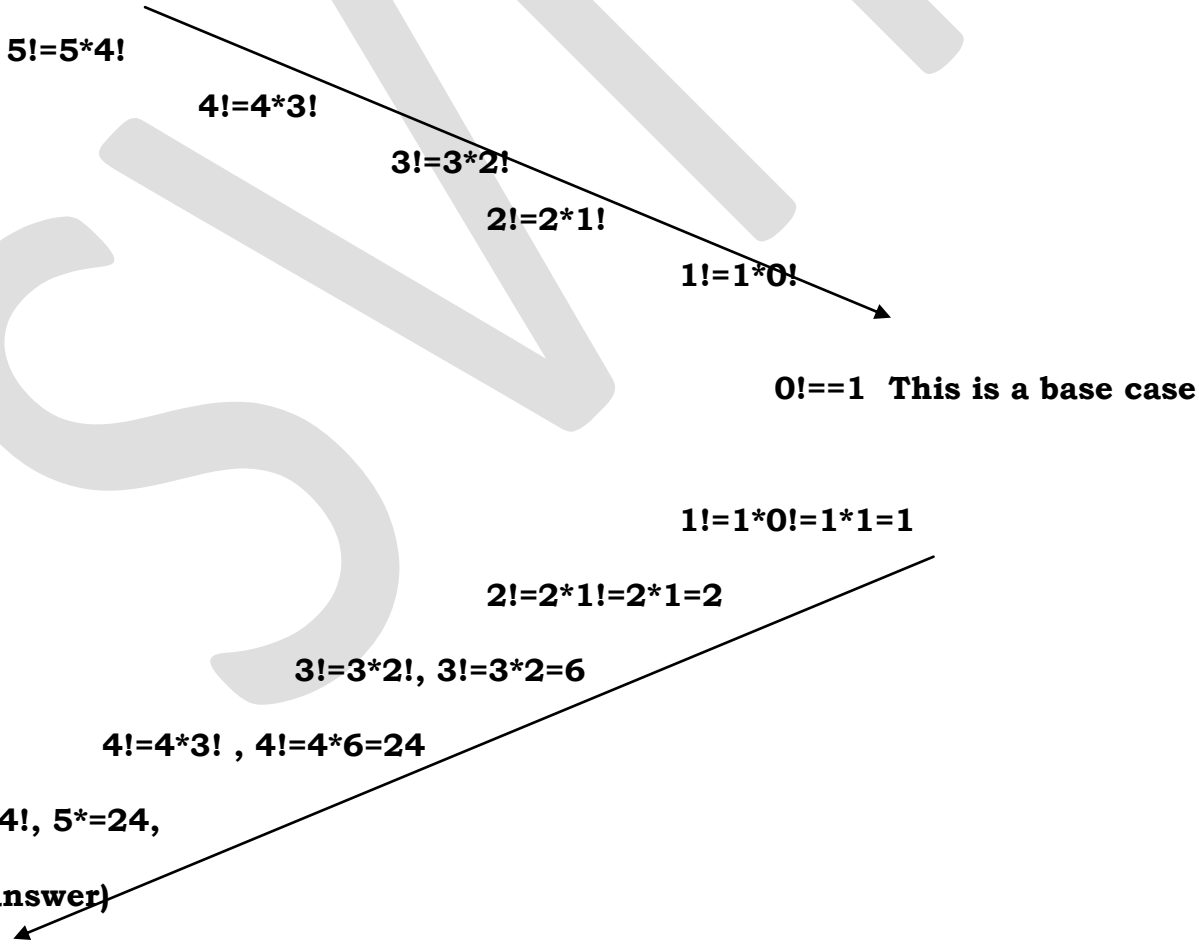
A base case serves two purposes:

i). It **act as a terminating condition**.

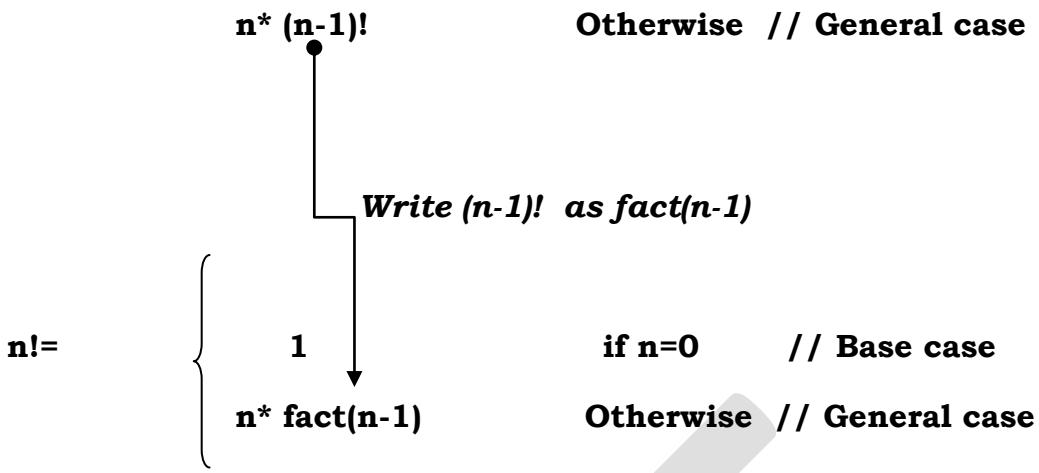
ii). the recursive function obtains **the solution from the base case it reaches**.

ii. **General case:**

- The statement that reduces the size of the problem is called general case.
- This is done by calling the same function with reduced size.
- In general recursive function of factorial problem can be written as



$$n! = \begin{cases} 1 & \text{if } n=0 \quad \text{// Base case} \end{cases}$$



Example 1.

```
/******* Factorial of a given number using Recursion ******/
#include<stdio.h>
int fact(int n);
void main( )
{
        int num,result;
        printf("enter number:");
        scanf("%d",&num);
        result=fact(num);
        printf("The factorial of a number is: %d",result);
}
int fact(int n)
{
        if(n==0)
                return 1;
        else
                return (n*fact(n-1));
}
```