ACHARYA

ACHARYA

BENGALURU, INDIA

WHERE THE

WORLD COMES

TO LEARN

Module-3
ARRAYS

The Need for Arrays

To store and print the marks of a student in one subject, the following code can be used

```
int marks;
marks = 30;
printf("%d", marks);
```

Here, marks is a ordinary variable which is capable of holding only one value at a given point of time

If the situation is to store the marks of 100 students,
we may have to declare **100 different variables** say marks1, marks2, ..., marks100

To overcome this, C supports a powerful data structure called as **Arrays**

Definition

An array is a collection of items of similar data type under one name.

Example: int marks[100];

Here,
- marks is the **name** of the array,
- [ ] tells the compiler that we are **dealing with arrays**
- 100 is the **size** of the array

Classification

Arrays are classified as

- One dimensional arrays
- Two dimensional arrays
- Multidimensional arrays

Declaration of One dimensional arrays

Just like declaring variables before using in the program, arrays must be declared.

The general syntax for declaring arrays is

datatype arrayname [index value] ;
OR
datatype arrayname [expression] ;

where,

• **datatype** can be any of the basic datatypes such as int, float, char, double

• **arrayname** is the name of the array which should be a valid identifier

• **index value** is a integer constant, if expressions are used, they must be evaluated to a positive integer only

	.	.
	.	.
	.	.
marks[0]	25	1000
marks[1]	20	1004
marks[2]	22	1008
marks[3]	30	1012
marks[4]	27	1016
.	.	.
.	.	.
marks[98]	22	1392
marks[99]	30	1396
.	.	.

Properties

The declaration int marks[100]; allocates memory to store 100 integers and the memory map is as shown in the figure

- The array elements are stored in contiguous memory locations
- All the elements of the array share a common name and are differentiated by means index value.
- The **first element** of the array has a index **value = 0** and the **last element** of the array has a index **value = n-1**

Array Initialization

Array elements can be initialized at the time of declaration.

The syntax for initializing an array is as shown

datatype arrayname[expression] = {V_0, V_1, V_2, ..., V_{N-1}} ;

Here,

V_0, V_1, V_2, ..., V_{N-1} are **values** that should be written within the flower brackets, Separated by comma.

The **value V_0** will be assigned to the array element with **index 0**, **value V_1** will be assigned to the array element with **index 1** and so on.

Different ways of Initializing arrays

1) Initializing all the memory locations

- Arrays can be initialized at the time of declaration when their initial values are known in advance.
- The values are copied into the array in the order specified in the declaration.
- **Example:**

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

Index	0	1	2	3	4
a[ ]	**10**	**20**	**30**	**40**	**50**
Address	1000	1004	1008	1012	1016

Different ways of Initializing arrays

2) Partial array initialization

If the number of elements to be initialized is less than the size of the array, then the elements are initialized in the order from 0th location.

The remaining locations are initialized to zero automatically.

Example:

int a[5] ={10, 20}

Index	0	1	2	3	4
a[ ]	**10**	**20**	**0**	**0**	**0**
Address	1000	1004	1008	1012	1016

Different ways of Initializing arrays

3) Initialization without size

Consider the following declaration along with initialization

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

In this declaration, even though we have not specified the size of the array, the array size will be set to the total number of initial values specified.

The memory map for this will be set as shown below

Index	0	1	2
a[ ]	**10**	**20**	**30**
Address	1000	1004	1008

Program: Write a 'C' program to read and print an array of 'n' integers

```
#include<stdio.h>

void main( )

{
        int n, i, a[20] ;

        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("The entered elements are\n");
        for( i = 0; i< n ; i + + )
        {
                printf("%d\n", a[ i ] );
        }
}
```

Output:

Enter the number of elements

n=4

Enter the elements

1

2

3

4

Index	0	1	2	3
a[ ]	1	2	3	4
Address	1000	1004	1008	1012

The entered elements are

1

2

3

4

Program: Write a 'C' program to find sum of all elements in an array of 'n' integers

```
#include<stdio.h>

void main( )

{
        int n, i, a[20], sum ;

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

Output:

Enter the number of elements

n=4

Enter the elements

1

2

3

4

Index	0	1	2	3
a[ ]	1	2	3	4
Address	1000	1004	1008	1012

Sum=0

sum=sum + a [i];

Sum= 0 + 1 = **1**

Sum= 1 + 2 = **3**

Sum= 3 + 3 = **6**

Sum= 6 + 4 = **10**

Sum=10

Program: Write a 'C' program to find sum of positive and negative numbers in an array of 'n' integers. Also find the average

```
#include<stdio.h>
void main( )
{
        int  n, i, a[20], psum=0, nsum=0;
        float average;

        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] );
        }

        for( i = 0; i< n ; i + + )
        {
            if(a[i]>0)
                psum = psum + a[i];
                else
                nsum = nsum + a[i];
        }

        average = (float)(psum+nsum)/n;

        printf("sum of positive numbers
        = %d\n", psum);
        printf("sum of negative numbers
        = %d\n", nsum);

        printf("Average of numbers
        = %f\n",   average);
}
```

Output

n=6

2	4	-5	4	-10	-5
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**	**a[5]**

sum of positive numbers = **10**
sum of negative numbers = **-20**
Average of numbers = **-1.6666**

Program: Write a 'C' program to find biggest element in an array of 'n' integers

```
#include<stdio.h>

void main( )

{
        int  n, i, a[20], max;

        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] );
        }

        max = a[0];
        for( i = 1; i< n ; i + + )
        {
                if(a[i] > max)
                {
                        max = a[i];
                }
        }
printf("Biggest element = %d\n", max );

}
```

Output

n=5

2	4	10	20	5
a[0]	**a[1]**	**a[2]**	**a[3]**	**a[4]**

max = a[0];
max = **2**
if(a[i] > max)
4 > 2
max = a[i]
max = **4**
Biggest element = 20

Program: Write a C program to find the sum, mean, variance and standard deviation of 'n' real numbers stored in an array

$$\text{mean} = \frac{\sum_{i=0}^{n-1} \text{a[i]}}{n}$$

$$\text{variance} = \frac{\sum_{i=0}^{n-1} (a[i] - mean)^2}{n}$$

$$standard\ deviation = \sqrt{variance}$$

Program: Write a C program to find the sum, mean, variance and standard deviation of 'n' real numbers stored in an array

```
#include<stdio.h>
#include<math.h>
void main( )
{
int n, i;
float a[20], sum, mean, var, sd, temp;
printf("Enter the number of elements\n");
scanf("%d", &n);
printf("Enter the elements\n");
for( i = 0; i< n ; i + + )
{
scanf("%f",&a[i] );
}
// To find sum and mean
sum=0.0;
for( i = 0; i< n ; i + + )
{
sum = sum + a[i];
}
mean = sum/n;
// To find variance
temp=0.0
for( i = 0; i< n ; i + + )
{
temp = temp + (a[i] - mean) * (a[i] - mean);
}
var = temp/n;
// To find standard deviation
sd = sqrt(var);
printf("sum = %f\n", sum);
printf("mean = %f\n", mean);
printf("variance = %f\n", var);
printf("standard deviation = %f\n", sd);
}
```

Operations on Arrays

There are a number of operations that can be performed on arrays.

1. Traversing an array
2. Inserting an element from an array
3. Deleting an element from an array
4. Merging two arrays
5. Searching an element in an array
6. Sorting an array in ascending or descending order

1. Traversing an array means accessing each and every element of the array for a specific purpose.

```
#include<stdio.h>
void main( )
{
        int n, i, a[20] ;

        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("The entered elements are\n");
        for( i = 0; i< n ; i + + )
        {
                printf("%d\n", a[ i ] );
        }
}
```

2. Inserting an element in an array : means adding a new data element to an already existing array.

```
#include<stdio.h>

void main( )

{
        int n, i, a[20],num,pos ;

            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("\n Enter number to be inserted:");
        scanf("%d",&num);

        printf("\n Enter the position at which the
number has to be added :");
        scanf("%d",&pos);
```

```
        for(i=n-1;i>=pos;i++)
            a[i+1]= a[i];
            a[pos]=num;
            n++;


printf("The array after insertion
are\n",num);

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

3. Deleting an element from an array : means removing a data element from an already existing array.

```
#include<stdio.h>

void main( )

{
        int n, i, a[20],num,pos ;

            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("\n Enter number to be inserted:");
        scanf("%d",&num);

        printf("\n Enter the position from which the
number has to be deleted:");
        scanf("%d",&pos);
```

```
        for(i=pos;i<n-1;i++)
            a[i]= a[i+1];
            n--;


printf("The array after deletion
are\n",num);

for( i = 0; i< n ; i + + )
{
printf("\n Arr[%d] = %d",i, a[ i ] );
return 0;
}
```

4. Merging two arrays

```
#include<stdio.h>

void main( )

{
        int arr1[10],arr2[10],arr[30];
        int i, n1,n2,m,index=0;
 printf("Enter the number of  elements in array
1:");
scanf("%d",&n1);
printf("enter the elements of the first array");
for(i=0;i<n1;i++)
        scanf("%d",&arr1[i]);

printf("Enter the number of  elements in array
2:");
scanf("%d",&n2);
printf("enter the elements of the first array");
for(i=0;i<n2;i++)
        scanf("%d",&arr2[i]);
m=n1+n2;
```

```
for(i=0;i<n1;i++)
{

arr3[index]=arr1[i];
Index++;
}

for(i=0;i<n2;i++)
{
    arr3[index] = arr2[i];
    index++;
}

printf("\n\n The merged array is");

for(i=0;i<m;i++)
{
printf("\t Arr[%d] = %d",I ,arr3[i]);
return 0;

}
```

ARRAYS
5. Searching Techniques

Searching Techniques

The process of finding a particular element (key) in a large amount of data is called searching.

Two important searching techniques

1. Linear Search (Sequential Search)
2. Binary Search

Linear Search (Sequential Search)

In this technique, the key element is searched in a **sequential manner** starting from the first element till the last element.

In the process of searching, if the **key** element **matches** with any element then the **search is successful** and the **position** of the key element **(i+1)** is printed.

Otherwise, search is unsuccessful i.e. element is **not found**.

Program: Write a 'C' program to search for a given 'key' element in an array of 'n' elements using linear search technique

```
#include<stdio.h>

void main( )

{
int n, i, a[20], key, flag=0;
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);
for( i = 0; i< n ; i + + )
{       if(key = = a[i])
        {
                flag=1;
                break;
        }
}
```

```
 if(flag==1)
printf("Successful Search, Key element found at position %d\n", i+1);

else

printf("UNSUCCESSFUL SEARCH\n" );

}
```

Output: **flag=0**

n=6

2	4	5	40	10	1
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]

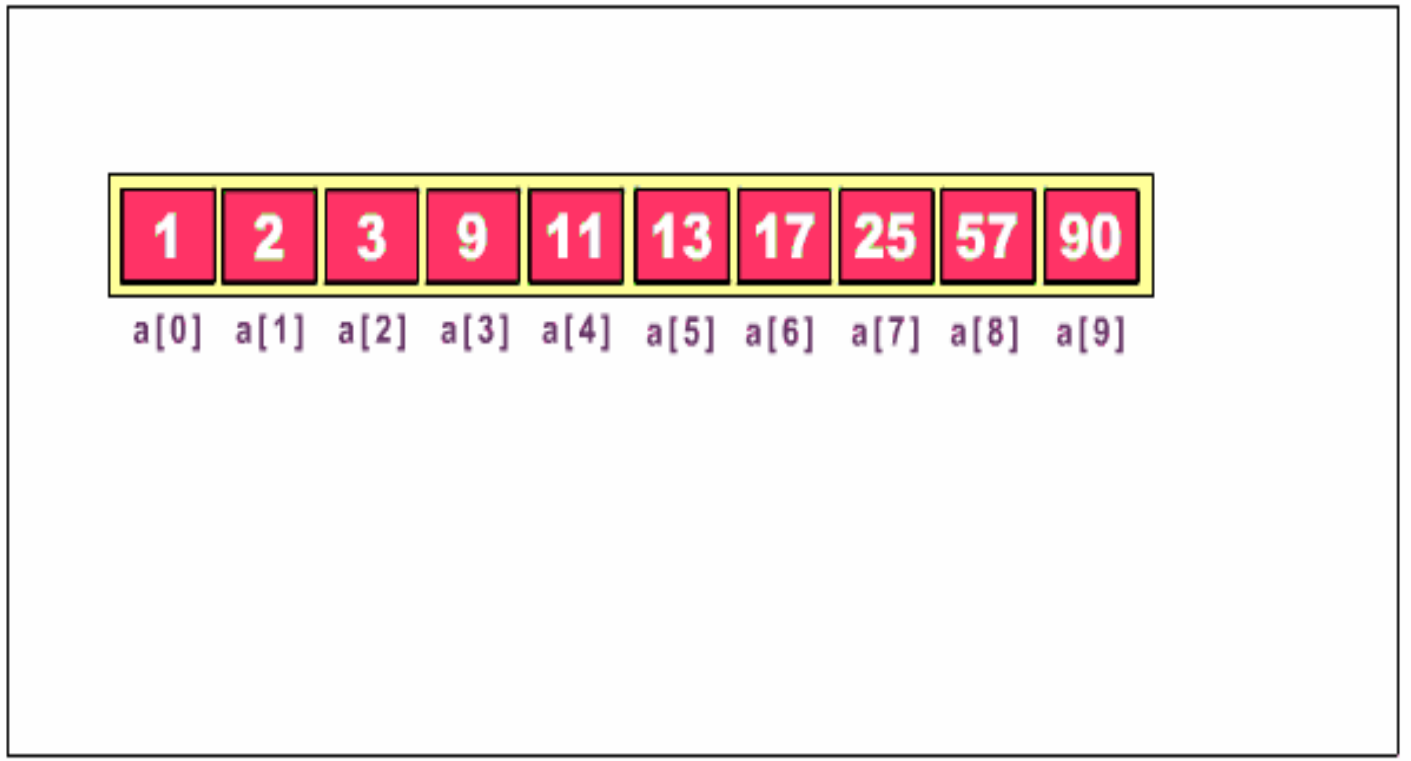
Key=10

flag=1

Successful Search, Key element found at position 5

Develop a program to Search for a given ‘key’ element in an array of ‘n’ elements using Binary Search

Binary search



Binary Search

1	2	3	9	13	17	25	57	90

Binary Search

This searching technique can be applied **only on sorted data** elements.

Three variables **low, high** and **mid** are used in this technique. **Low** is the index of **first element (low=0)**, **high** is the index of the **last element (high=n-1)** and **mid** is the index of the **middle element (mid = (low+high)/2)**.

The **key** element to be searched for is **compared with the middle element (a[mid])**, if they are equal, search is **successful** and the position of the key element is printed (mid+1).

If this condition fails, then the key may be in the **upper or lower part** of the array.

If **key is less than the middle** element, then it might be in the **upper part of the array** and hence searching needs to be continued only in the upper part. Now the search space is reduced to half i.e. from low to mid-1, hence high = mid-1.

If **key is greater than the middle** element, then it might be in the **lower part of the array** and hence searching needs to be continued only in the lower part. Again, the search space is reduced to half i.e. from mid+1 to high, hence low = mid+1.

Repeat this process of searching until either the key is found or the elements in the search space is exhausted. If the search space gets exhausted, then search is unsuccessful.


```
#include<stdio.h>

void main( )

{
int     n, i, a[20], key, flag=0,
         low, mid, high;

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])
{
flag=1;
break;
}
if(key < a[mid])
                high = mid – 1;
if(key > a[mid])
                low = mid +1;
}

if(flag==1)
{
printf("SUCCESSFUL SEARCH\n" );
printf("Element found at %d location\n", mid + 1 );

}
else
printf("UNSUCCESSFUL SEARCH\n" );
return 0;
}
```

ARRAYS
6. Sorting Techniques

Sorting Techniques

The process of arranging the given elements so that they are in ascending or descending order is called sorting

10	2	33	9	11	13	17	25
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]

Ascending order

2	9	10	11	13	17	25	33

Descending order

33	25	17	13	11	10	9	2

Two Sorting techniques

1. Bubble Sort (Sinking Sort)
2. Selection Sort

Develop a Program to sort the given set of N numbers using Bubble Sort

Program: Develop a program to sort the given set of N numbers using Bubble sort.

Input: `int a[5] = {50, 40, 30, 20, 10}`

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

Output: `int a[5] = {10, 20, 30, 40, 50}`

PASS-1				
A[0]=**50**	40	40	40	40
A[1]=**40**	**50**	30	30	30
A[2]=30	**30**	**50**	20	20
A[3]=20	20	**20**	**50**	10
A[4]=10	10	10	**10**	**50**
50 sinks to bottom after Pass-1				

PASS-2			
A[0]=**40**	30	30	30
A[1]=**30**	**40**	20	20
A[2]=20	**20**	**40**	10
A[3]=10	10	**10**	**40**
A[4]=50	50	50	**50**
40 sinks to bottom after Pass-2			

PASS-3		
A[0]=**30**	20	20
A[1]=**20**	**30**	10
A[2]=10	**10**	**30**
A[3]=40	40	**40**
A[4]=50	50	**50**
30 sinks to bottom after Pass-3		

PASS-4	
A[0]=**20**	10
A[1]=**10**	**20**
A[2]=30	**30**
A[3]=40	**40**
A[4]=50	**50**
20 sinks to bottom after Pass-4	

Program: Develop a program to sort the given set of N numbers using Bubble sort.

Input: `int a[5] = {50, 40, 30, 20, 10}`

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

Output: `int a[5] = {10, 20, 30, 40, 50}`

n=5	Pass-1	Pass-2	Pass-3	Pass-4	
j=	1	2	3	4	**for(j=1; j<n; j++)**
	0 – 1	0 - 1	0 - 1	0 - 1	
	1 – 2	1 - 2	1 - 2		
	2 – 3	2 - 3			
	3 – 4				
i =	0 to 3	0 to 2	0 to 1	0 to 0	
	0 to 5 - (1+1)	0 to 5 - (2+1)	0 to 5 - (3+1)	0 to 5 - (4+1)	
	0 to < n - j				**for(i=0; i< n-j ; i++)**

Input: **int a[5] = {50, 40, 30, 20, 10}**

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

Output: **int a[5] = {10, 20, 30, 40, 50}**


```
#include<stdio.h>

void main( )

{

 int n, i, j, temp, a[100];


 printf("Enter the value for n:\n");
scanf("%d",&n);

printf("Enter the array elements\n");
for(i=0;i<n;i++)
{
                scanf("%d",&a[i]);
 }
printf("The array elements before
sorting are\n");
for(i=0;i<n;i++)
{
                printf("%d\n",a[i]);

}
```

```
 for(j=1;j<n;j++)
{
     for(i=0 ; i< n-j ; i++)
        {
              if(a[i]>a[i+1])
              {
                   temp=a[i];
                   a[i]=a[i+1];
                   a[i+1]=temp;
              }
        }
}

printf("The elements after sorting are\n");

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

}
```

Selection Sort

Selection Sort

Input: **int a[5] = { 45, 20, 5, 15, 30}**

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

Output: **int a[5] = {5, 15, 20, 30, 45}**

For a given array of elements, two parts are considered sorted and unsorted. Initially, the sorted part is empty and unsorted part is the complete array.

− The smallest element is selected from the unsorted part and exchanged with the first element. With this, the first element is sorted.

− The next smallest element is selected from the unsorted part and exchanged with the second element. With this, the second element is sorted.

− Continue this process, until all the elements are sorted.

Selection Sort

Input: **int a[5] = { 45, 20, 5, 15, 30}**

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

Output: **int a[5] = {5, 15, 20, 30, 45}**

Given array n=5					
a[0]	45	5	5	5	5
a[1]	20	20	15	15	15
a[2]	5	45	45	20	20
a[3]	15	15	20	45	30
a[4]	30	30	30	30	45
Sorted upto		a[0]	a[1]	a[2]	a[3]
j=		0	1	2	3
		for(j=0; j<=n-2; j++)			
min=		a[0]	a[1]	a[2]	a[3]
		min=a[j] pos = j			
i=		1 to 4	2 to 4	3 to 4	4 to 4
		for(i=j+1; i<n; i++)			

Input: **int a[5] = { 45, 20, 5, 15, 30}**

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

```
#include<stdio.h>
void main()
{
	int i, a[20], n, min,j,temp;
	printf("Enter the number of element\n");
	scanf("%d",&n);
	printf("Enter the elements\n");
	for(i=0;i<n;i++)
	{
		scanf("%d",&a[i]);
	}
for(i=0;i<=n-1;i++)
{
	min=i;
	for(j=i+1;j<n;j++)
	{
		if(a[i]<a[min])
		{
			min=j;
		}
	}
temp=a[i];
a[i]=a[min];
a[min]=temp;
}

printf("The sorted numbers are\n");
for(j=0;j<n;j++)
{
	printf("%d\t",a[j]);
}
}
```

Passing Arrays to a function

Write a program to read and print an array of n numbers


```
#include<stdio.h>
void read_array(int arr[] , int);
void display_array(int arr[], int);

Int main()
{
int num[10], n;
printf("Enter the size of the array :");
scanf("%d",&n);
read_array(num,n);
 display_array(num,n);
return 0;
}
```

```
void read_array(int arr[10],int n)
{
    int i;
   printf("Enter the elements of the array");
   for(i=0;i<n;i++)
            {
            scanf("%d",&arr[i]);
            }
}
```

```
void display_array(int arr[10],int n)
{
   printf("Enter the elements of the array");
   for(i=0;i<n;i++)
   printf("\t %d",arr[i]);
}
```

TWO-DIMENSIONAL ARRAYS

A two-dimensional array is declared as

data_type array_name[row_size] [column_size];

A two-dimensional m x n array is an array that contains m x n data elements and each element is accessed using two 2 subscripts, i and j where i<=m and j<=n.

Operations on Two-Dimensional ARRAYS

1. **Transpose** of a m x n matrix A is given as a n x m matrix B
2. **Sum** Two matrices that are compatible with each other can be added together thereby storing the result in the third matrix.
3. **Difference** Two matrices that are compatible with each other can be subtracted together thereby storing the result in the third matrix.
4. **Product** Two matrices that are multiplied with each other if the number of columns in the 1st matrix is equal to the number of rows in the second matrix.

Implement Matrix multiplication and validate the rules of multiplication

```
#include<stdio.h>
void main( )
{
int m, n, p, q, i, j, a[20] [20], b[20][20],
c[20][20];
printf("Enter the number of rows and
columns of matrix A:\n");
scanf("%d %d", &m,&n);
printf("Enter the number of rows and
columns of matrix B:\n");
scanf("%d%d", &p,&q);
if(n!=p)
{
printf("Matrix multiplication is not
possible\n"); exit(0);
}
printf("Enter the elements of matrix A:\n");
for( i = 0; i < m ; i + + )
{
for( j = 0; j < n ; j + + )
{
scanf("%d", &a[i][j] );
}
}
printf("Enter the elements of matrix B:\n");
for( i = 0; i < p ; i + + )
{
for( j = 0; j < q ; j + + )
{
scanf("%d", &b[i][j] );
}
}
```


```
printf("The elements of matrix A are:\n");
for( i = 0; i < m ; i + + )
{
for( j = 0; j < b ; j + + )
{
printf("%d \t",a[i][j] );
}
printf("\n");
}
printf("The elements of matrix B are:\n");
for( i = 0; i < p ; i + + )
{
for( j = 0; j < q ; j + + )
{
printf("%d \t",b[i][j] );
}
printf("\n");
}
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("The resultant matrix is\n");
for( i = 0; i < m ; i + + )
{
for( j = 0; j < q ; j + + )
{
printf("%d \t",c[i][j] );
}
printf("\n");
}
}
```

Passing Two-Dimensional Arrays to Functions

```
#include <stdio.h>

void printArray(int rows, int cols, int arr[rows][cols])
{
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
    printArray(2, 3, arr);
    return 0;
}
```

Output

```
1 2 3
4 5 6
```

Applications of Arrays

1. Storing Data Collections
2. Matrix Operations
3. Image Processing
4. Database Implementation
5. Sorting and Searching
6. Implementing Data Structures

Functions

Why functions?

If the program is very lengthy, there are too many disadvantages.

- It is very difficult for the programmer to write a large program.
- Very difficult to identify the logical errors and correct.
- Very difficult to read and understand.
- Large programs are more prone to errors and so on.

These disadvantages can be overcome using functions

What is a functions?

- In c language, a large program can be divided into a series of individual related programs called **modules**.
- These modules are called functions(also called subprogram)
- Thus a function is a program segment that carries out some specific and well defined tasks.

Example: sqrt(),scanf(),printf(),

Different types of functions

- Library function
- User defined function

Library function(built in functions):

- The standard C library is a collection of various types of functions which perform some standard and pre-defined tasks.
- These functions which re part of the C compiler that have been written for general purpose are called library functions.

Example: sqrt(), sin(), cos(),scanf(),printf()

Advantages :

- The programmer job is made easier because the functions are already available.
- The library functions can be used whenever required

Disadvantages:

- Since the standard library functions are limited, Programmer cannot completely rely on these library functions.
- The programmer has to write his/her own programs

User–Defined Functions:

The functions which are written by the programmer / user to do some specific tasks are called User Defined functions(UDFs)

The various advantages of functions are

- Reusability and Reduction of code size
- Readability of the program can be increased
- Modular programming approach
- Easier debugging
- Build library
- Function sharing

Example : Consider the program

```
void main()
{
   int a, b, sum;
   printf("Enter the values for a and b\n");
   scanf("%d%d",&a,&b);
   sum=a+b;
   printf("%d",sum);
}
```

Calling function	Called function
#include<stdio.h> void main() { add(); }	void add() { int a, b, sum; printf("Enter a and b values\n"); scanf("%d%d",&a,&b); sum=a+b; printf("%d",sum); }

Called function:

Invoking the function add() is done by simply writing the name of the function as shown in function main().We say that the function add() is called .So, the function add() is often called "**Called function**".

Calling function:

Who is calling the function add() ? It is clear from the above program that the function main() is calling . So, the function main() is called "**Calling function**".

Elements of user – defined functions

- Function definition
- Function call
- Function declaration

```
#include<stdio.h>
void add();

void main()
{
        add();
}

void add()
{
Int a, b, sum;
printf("Enter the values for a and b\n");
Scanf("%d%d",&a,&b);
sum=a+b;
printf("%d",sum);
}
```

Function declaration

Function call

Function definition

Function definition

The program module that is written to achieve a specific task is called **function definition.**

The various elements of the function definition are

- Function header
  - → type
  - → name
  - → parameters
- Function body
  - → Declaration statements
  - → Executable statements
  - → return statements

General format	Example
function header type name(parameters) { Declaration part; Executable part; return exp; }	**function header** int add (int m, int n) { int sum; sum=m+n; return sum; }

Return values and their type

- Using this statement, the function returns the evaluated result.
- If a function do not return any value, the return statement can be omitted.

return; function is not sending any value to the calling function.

return exp; function is sending a value to the Calling function.

Function to return the larger of two numbers

```
int maximum(int m, int n)
{
        if(m>n)
                return m;
        else
                return n;
}
```

Function calls

- After writing the functions, appropriate functions have to be invoked to get the job done from those functions.
- This invocation of a function is called **function call**.
- The variables that are used while calling the function are called **actual parameters**.

```
void main()
{
  int m,n,res;
 printf("Enter m and n\n");
 scanf("%d%d",&m,&n);

 res=maximum(m,n)

 printf("%d",res);
}
```

Actual parameters

Calling function

Formal parameters

```
int maximum(int m,int n)
{

 if(m>n)
   return m;
 else
   return n;
}
```

Called function

Formal parameters are those which contain only a copy of the actual variables to do a specific operation in the called function and return/output a result without modifying the parameters.

Difference between Actual parameters and Formal parameters

Actual parameters	Formal parameters
Actual parameters are used in calling function when a function is invoked **Example : c=sum(a,b)**	Formal parameters are used in the function header of a called function **Example : int sum(int m, int n)**
Actual parameters can be constants, variables or expression **Example : c= sum (a+4,b)**	Formal parameters should be only variables. Expressions and constants are not allowed. **Example : int sum(int m, int n)**
Actual parameters sends values to the formal parameters **Example : c= sum (4, 5)**	Formal parameters receive values from the actual parameter **Example : int sum(int m, int n)**
Addresses of actual parameters can be send to formal parameters	If formal parameters contains addresses, they should be declared as pointers.

Function declaration

- The declaration of functions are similar to the declaration of variables.
- We normally declares the functions before they are defined.
- The declaration of each function should end with semicolon.
- This is called **function declaration** or function prototype.

Syntax :

type function_name(type a1, type a2,....type an);

Example : the function declaration for maximum()

int maximum(int m, int n);
or
int maximum(int, int);
or
maximum(int, int);

Categories of functions

Based on the parameters and return value, the functions are categorized as shown below.

1. Function with No parameters and No return values

Calling function	Called function
#include<stdio.h> void sum(); void main() { sum(); }	void sum() **/* No parameters */** { int a, b, c; printf("Enter the values of a and b\n"); Scanf("%d%d",&a,&b); c=a+b; printf("%d",c); **/* No return values */** }

2. Function with No parameters and return values

Calling function	Called function
#include<stdio.h> void sum(); void main() { int c; c=sum(); printf("Sum=%d",c); }	int sum() **/* No parameters */** { int a, b, c; printf("Enter the values of a and b\n"); scanf("%d%d",&a,&b); c=a+b; return c; **/* return values */** }

3. Function with parameters and No return values

Calling function	Called function
#include<stdio.h> void sum(int a, int b); void main() { int m,n; printf("Enter m and n\n"); scanf("%d%d",&m,&n); sum(m,n); }	void sum(int a, int b) { **/* with parameters */** int c; c=a+b; printf("Sum=%d",c); **/* No return values */** }

4. Function with parameters and return values

Calling function	Called function
#include<stdio.h> void sum(int a, int b); void main() { int m,n,c; printf("Enter m and n\n"); scanf("%d%d",&m,&n); c=sum(m,n); printf("Sum=%d",c); }	void sum(int a, int b) { **/* with parameters */** int c; c=a+b; return c; **/* with return values */** }

Passing parameters to functions

The various ways of passing parameters to the functions are

- Pass by value (call by value)
- Pass by reference (call by reference)

Pass by value (call by value)

- When a function is called with **actual parameters**, the values of actual parameters are copied into formal parameters.
- If the values of the **formal parameters** changes in the functions, the values of the actual parameters are not changed.
- This way of passing parameters is called **pass by value or call by value**

.

C program swapping of two numbers

Calling function	Called function
#include<stdio.h> void exchange(int m, int n); void main() { int a,b; a=10, b=20; printf("Before exchange"); printf("a=%d and b=%d\n",a,b); exchange(a,b); printf("After exchange"); printf("a=%d and b=%d\n",a,b); } **OUTPUT** **Before exchange** a=10 and b=20 **After exchange** a=10 and b=20	void exchange(int m,int n) { int temp; printf("Before exchange\n"); printf("m=%d and n=%d\n",m,n); temp=m; m=n; n=temp; printf("After exchange\n"); printf("m=%d and n=%d\n",m,n); } **OUTPUT** **Before exchange** m=10 and n=20 **After exchange**

Pass by reference (call by reference)

- In pass by reference, a function is called with addresses of actual parameters.
- In the function header, the format parameters receive the addresses of actual parameters.
- Now, the formal parameters do not contain values, instead they contain addresses.
- Any variable if it contains an address, it is called a **pointer variable.**
- Using pointer variables, the values of the actual parameters can be changed.
- This way of passing parameters is called **Pass by reference or call by reference.**

.

C program swapping of two numbers

Calling function	Called function
#include<stdio.h> Void exchange(int *m, int *n); void main() { int a,b; a=10, b=20; printf("Before exchange"); printf("a=%d and b=%d\n",a,b); exchange(&a,&b); printf("After exchange"); printf("a=%d and b=%d\n",a,b); } **OUTPUT** **Before exchange** a=10 and b=20 **After exchange** a=20 and b=10	Void exchange(int *m,int *n) { int temp; printf("Before exchange\n"); printf("m=%d and n=%d\n",m,n); temp=m; m=n; n=temp; printf("After exchange\n"); printf("m=%d and n=%d\n",m,n); } **Before exchange** m=10 and n=20 **After exchange** m=20 and n=10

Develop a program to find the square root of a given number N and execute for all possible inputs with appropriate messages. Note: Don't use library function sqrt(n).

```
#include<stdio.h>
void main( )
{
  float n, x1, x2;

 printf("Enter the number\n");
 scanf("%f", &n);

 x2 = 0;
x1 = (n + (n/n)) / 2;

while(x1 != x2)
 {
x2 = (x1 + (n/x1)) / 2;
x1 = (x2 + (n/x2)) / 2;
  }
  printf("Square root of %f is %f\n", n, x1);
  printf(“Using Built in function Square root of a given
number
  %f is %f\”, n, sqrt(n));
 }
```

OUTPUT :

Enter the number
4
Square root of 4 is 2

Using Built in function Square root of a given number **4 is 2**.

Trace1	Trace2
N=4 X2=0 X1=(4+(4/4))/2=2 While(2!=0) { X2=(2+(4/2))/2=2 X1=(2+(4/2))/2=2 }	N=16 X2=0 X1=16+(16/16))/2=17/2= 8 while(8!=0) { X2=(8+(16/8))/2=10/2=5 X1=(5+(16/5))/2=8/2=4 } while(4!=5) { X2=(4+(16/4))/2=8/2=4 X1=(4+(16/4))/2=8/2=4 } While(4!=4)

Implement using functions to check whether the given number is prime and display appropriate messages. (No built-in math function).

```
 #include<stdio.h>
void checkprime( int n )
{
int i;

 if( n== 0 || n==1)
{
 printf("Number is neither prime nor composite\n");
 exit(0);
}
for(i=2; i<=n/2; i++)
{
 if( n % i = = 0 )
{
printf("Number is not prime\n");
 exit(0);
}
 }
printf("Number is prime\n");
}
```

```
int main( )
 {
 int n;

 printf("Enter the value of n\n");
 scanf("%d", &n);

 checkprime( n );
 return 0;
}
```

OUTPUT

Enter the value of n
1
Number is neither prime nor composite

Enter the value of n
9
Number is not prime

Enter the value of n
13
Number is prime

N=13
Computations
i=2 3 4 5 6

13	13	13	13	13
2	3	4	5	6

(2,3,4,5,6 can't divide 13)
13 is prime

N=9
Computations
i=2 3 4

9	9
2	3

9 is not prime

Function that returns multiple values

- The arguments that are used to “send out” information are called output parameters.

- Mechanism of sending back information through arguments is achieved using what are known as the address operator (&) and indirection operator(*)

Example :

```
void mathoperation (int x, int y, int *s, int *d);
 main()
{
  int x = 20, y = 10, s, d;
  mathoperation(x,y, &s, &d);

  printf(“s=%d\n d=%d\n”,s,d);
  }

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

Nesting of functions

- C permits nesting of functions freely. main can call function1, which calls function2, which calls function3,….and so on.
- There is in principle no limit as to how deeply functions can be nested

Example

```
float ratio(int x, int y, int z);
main()
{
        int a,b,c;
        scanf(“%d %d %d”, &a, &b, &c);
        printf(“%f \n “,ratio(a,b,c));
}
float ratio(int x, int y, int z)
{
        if(difference(y, z))
        return(x/(y-z));
        else
        return(0.0);
}
int difference(int p, int q)
{
         if (p!=q)
         return (1)
         else
         return(0);
}
```

Recursion

- Recursion is a substitute for iteration in making a function repetitive in terms of itself.
- Recursion is a powerful programming concept.
- It helps the programmer to build a function which is clearer and easily readable.
- Recursion is the name given to the ability of a function to call itself, until the desired condition is satisfied.
- Here, the function is calling and recalling itself repeatedly, until the stopping condition.

Types of recursion

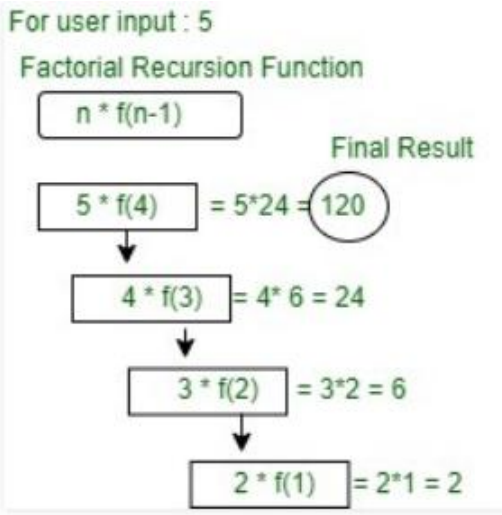
Direct recursion	Indirect recursion
Function_1() { --- Function_1() --- }	Function_1() { --- Function_2() }— Function_2() { Function_1() -- }

/*Write a c program to find factorial of a number using recursion */

```
int factorial(int n);
int main()
{
int n;
 printf("Enter a +ve integer:");
scanf("%d",&n);
Printf("Factorial of %d =%d",n,factorial(n));
return 0;
}
```

/*Recursive function*/

```
int factorial(int n)
{
    int fact;
    if(n==1)
    return(1);
    else
    fact= n*(factorial(n-1);
    return(fact);
}
```



OUTPUT

N=4

Factorial of 4 = 24

Implement Recursive functions for Binary to Decimal Conversion.

The general procedure:

$(111)_2 = (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) = (7)_{10}$

```
#include<stdio.h>
int binarytodecimal(int n)
{
 if(n = = 0)
  return 0;
 else
  return( n%10 +  binarytodecimal (n/10) * 2 );
}
void main( )
{
 int decnum, binnum;
 printf("Enter the binary number : only 0s and 1s");
 scanf("%d",&binnum);

 decnum= binarytodecimal (binnum);

 printf("The decimal equivalent =%d", decnum);
 }
```

OUTPUT

Enter the binary number : only 0s and 1s
111
The decimal equivalent =7

Trace:

111%10+ 2* binarytodecimal(111/10))
1 + 2 * binarytodecimal(11)

1 + 2* 11%10 +2 * binarytodecimal**(11/10)**
3 * 1 + 2 * binarytodecimal**(1)**

3 + 2 * 1%10 + 2 * binarytodecimal(1/10)

5 * 1 +2
5 + 2
7

Passing arrays to functions

One-Dimensional Arrays

- Like the values of simple variables, it is also possible to pass the values of an array to a function.
- To pass a one-dimensional an array to a called function, it is sufficient to list the name of the array, without any subscripts, and the size of the array as arguments.

Example, the call

largest(a,n)

The largest function header

float largest(float array[], int size)

The declaration : float array[];

```
main()
{
float largest(float a[],int n);
float value[4]={2,5,-4.75,1.2};

printf("%f\n",largest(value,4));
}
```

```
float largest(float a[], int
n)
{
int i;
float max;
 max=a[0];
for(i=1;i<n; i++)
If(max<a[i])
max = a[i] ;
return(max);
}
```

Two Dimensional Arrays

- Like simple arrays, we can also pass multi-dimensional arrays to functions.
- The function must be called by passing only the array name.
- In the function definition, we must indicate that the array has two-dimensions by including two sets of brackets.
- The size of the second dimension must be specified.
- The prototype declaration should be similar to the function header.

Example :

```
main()
{
int M=3, N=2;
double average(int [M][N],int,int);
double mean;
int matrix[M][N]=
{
  {1,2},
  {3,4},
  {5,6}
};
mean = average(matrix,M,N);
printf("%lf\n",mean);
}
```

```
double average(
  int [M][N],int,int)
{
int i,j;
double sum=0.0;
for(i=0;i<M; i++)
for(j=1;i<N; j++)

sum+=X[i][j]
return(sum/(M*N));
}
```

Passing strings to functions

The strings are treated as character arrays in c and therefore the rules for passing strings to function are very similar to those for passing arrays to functions.

Basic rules are:

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

   Example:

```
void display(char item_name[])
{
 ----
 ----
}
```

2. The function prototype must shown that the arguments is a string.

```
void display(char str[])
```

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

Example: display(names);

where name is a properly declared string array is the calling function.

/*Write functions to implement string operations such as compare, concatenate, string length. Convince the parameter passing techniques. */

```
#include<stdio.h>
int my_strcmp(char str1[ ], char str2[ ])
{
int i = 0;

 while( str1[i] != '\0' && str2[i] != '\0')
 {
 if(str1[i] > str2[i])
 return 1;
 else if(str1[i] < str2[i])
  return -1;
 else
 i + +;
 }
 if(str1[i] != '\0' && str2[i] = = '\0')
return 1;

 if(str1[i] = = '\0' && str2[i] != '\0')
  return -1;
   return 0;
 }
```

```
void my_strcat(char str1[ ], char str2[ ])
{
int i, j;

 i = 0;
while( str1[i] != '\0')
 {
  i + +;
 }

 j=0;
  while( str2[j] != '\0')
 {
str1[i] = str2[j];
  i + +;
  j + +;
  }
str1[i] = '\0';
}
```

```
int my_strlen(char str[ ])
 {
 int i = 0;

 while( str[i] != '\0')
  {
 i + +;
  }
 return i;
}
```

```
main( )
{
int choice, difference, len;
char str1[20], str2[20], str[20];

 for( ; ;)
 {
  printf("1.Compare  2.Concatenate
              3.Length      4.Exit\n");
 printf("Enter your choice\n");
 scanf("%d", &choice);
```

```
switch(choice)
 {
case 1:
printf("Enter the string1\n");
scanf("%[^\n]", str1);  // gets(str1);
printf("Enter the string2\n");
scanf("%[^\n]", str2);  // gets(str2);
difference = my_strcmp(str1, str2);

   if(difference = = 0)
  printf("%s = %s\n", str1, str2);
 else if (difference = = 1)
 printf("%s > %s\n", str1, str2);
 else if (difference = = -1)
printf("%s < %s\n", str1, str2);
  break;
```

OUTPUT:

```
case 2:
printf("Enter the string1\n");
scanf("%[^\n]", str1);  // gets(str1);
printf("Enter the string2\n");
scanf("%[^\n]", str2);  // gets(str2);
my_strcat(str1, str2);
printf("Concatenated string = %s\n", str1);
break;
```

```
case 3:
printf("Enter the string\n");
scanf("%[^\n]", str);  // gets(str);
 len = my_strlen(str);
   printf("Length of the string =
%d\n",len);
break;
  case 4: exit(0);
 }

 }
 return 0;
 }
```

Storage Classes

- The variables may be declared within the function or they may be declared outside the functions.
- Based on where the variables are declared, the scope and lifetime of variables change.
- Scope of a variable is defined as the region or a boundary of the program over which a variable can be accessed during execution of the program.
- The lifespan of a variable is defined as the period during which a variable retains a given value during execution of a program.

The various storage classes in C language are classified as shown below

- Local variables
- Global variables
- Static variables
- Register variables

Local variables (Automatic variable)

- Local variables are the variables which are defined within a function.
- As and when the control enters into the function, memory is allocated for these variables and when control goes out of the function, memory will be de-allocated.
- The scope of these variables is limited only to the function so, variables can not be accessed by any function.
- No other function can use a local variables and change its value.

```
#include<stdio.h>
void add(int a,int b)
{
  int c;
  c=a+b;
   printf("Result =
%d\n",c);
```

```
void main()
{
int a=10, b=20;
Add(a,b);
}
```

Global variables

- Global variables are the variables which are defined before all functions.
- These variables declared before the functions.
- Memory is **allocated** for these variables only once and are alive and active throughout the program.
- Any function can use a global variable and change its value.
- Memory is **deallocated** once the execution of the program is over.

Example:

```
#include<stdio.h>
int a=10,b=20;
Int c;
void add(int a,int b)
{
  c=a+b;
}
void subtract()
{
  c=a-b;
}
```

```
void main()
{

  add();
  printf("a=
%d\tb=%d\n",a,b);
  printf("a+b=%d\n",c);

  subtract()
  printf("a-b=%d\n",c);
}
```

Static variables

- Static variables can be declared outside the function or within the function.
- They have the characteristics of both local and global variables.
- The declaration of a static variable should begin with the keyword static.
- Memory is allocated for those variables only once and memory contents are not initialized to zero.
- If these variables are defined inside a function, No other function can use a static variable and change its value.
- Memory is de-allocated for static variables when the execution of the program stops.

Example: static int a,b; static float c;

```
#include<stdio.h>
void display()
{
  static int i=0;
  i++;
   printf(" %d\n",i);
}
```

```
void main()
{
Int j;
for(j=0;j<5;j++)
display();
}
```

Register variable

- Any variable 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 register but, not in main memory.
- Register accessing is much faster compared to the memory access and hence, frequently accessed variables such as loop variables are usually stored in register variable.
- This leads to the faster execution of programs.

The register declaration is

register int x;

This declaration is allocated only for the local(automatic) variables and to the formal parameters.

Storage Classes

Storage Classes

C supports **4 types** of storage class specifiers **extern, static, register** and **auto**

The storage class specifiers tells the compiler **how to store** the subsequent variables, like

where the variable should be stored

what will be the default initial value

the scope : determines over what region of the program a variable is actually available for use.

life time : refers to the period during which a variable retains a given value during the execution of a program.

visibility : refers to the accessibility of a variable from the memory.

Storage Classes

Depending on the place of their declaration, variables can be categorized as

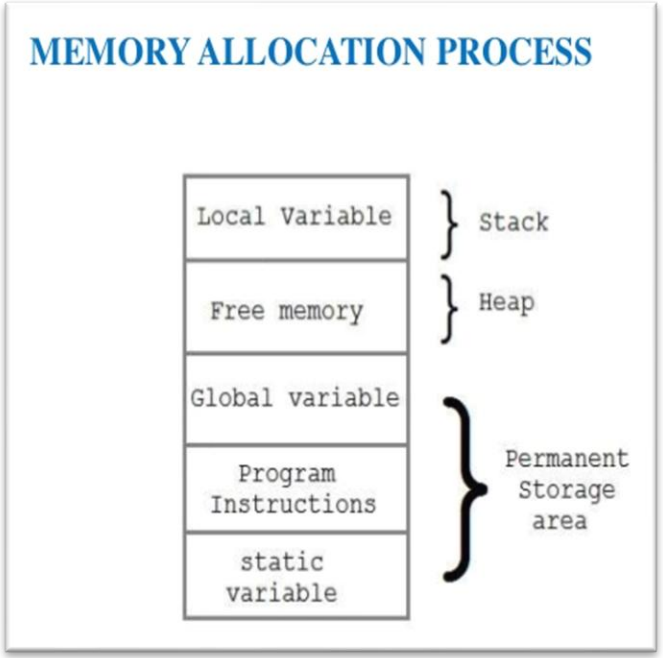
internal : are those which are declared within a particular function

external : are those which are declared outside of any function

Apart from their data type, all variable also have a storage class.

There are 4 storage classes

1) automatic or local or internal
2) global or external
3) static
4) register variables



1) Automatic Variables

They are created **(memory is allocated)** each time the **function is called and destroyed automatically** when the function is exited.

storage : RAM

default initial value : garbage value

scope : local to the function in which it is defined

life time : till the control remains within the function

Since they are private or local to the function in which they are declared, they are also called as **internal or local** variables.

Example:

```
void main( )
{       int num;          // the keyword auto is optional
        auto float x;
}
```

2) Global Variables

They are created **(memory is allocated)** when the program execution starts i.e. **only once and destroyed when the program terminates**.

storage : RAM

default initial value : Zero

scope : global

life time : till the end of the program

Since they are global, these variables can be accessed by any of the functions in the program and are generally declared outside the functions (usually in the beginning)

Example:

```
int num;
float x = 3.5;
void main( )
{          ……..
}
function1( )
{          ……..
}
function2( )
{          ……..
}
```

3) Static Variables

They are **permanent variables** and maintain their values **between function calls** i.e. the value of these variables persists until the end of the program

storage : RAM

default initial value : Zero

scope : local to the function in which it is defined

life time : till the end of the program

It has the features of **both local and global** variables. Like a local variable the static variables **cannot be accessed outside the function** in which it is defined and like a global variable memory for a static variable is **allocated only once** when the function is called and the memory is deallocated only when the program terminates.

Example:

```
function1( )
{
Static int a=10;
.....
}
void main( )
{
function1( );
function1( );
.....
}
```

4) Register Variables

This specifier tells the compiler to keep the value of the **variable in a register of the CPU** rather than in memory (where normal variables are stored)

storage : CPU register

default initial value : garbage value

scope : local to the function in which it is defined

life time : till the control remains within that function

Note:

1) Registers are **faster** than memory to access, so variables which are most frequently used in a C program can be put in registers using the register keyword.

2) If **registers are not free**, the compiler ignores this and **treats** the variables as to be of **auto class**.

Example:

```
void main( )
{
register int i;
for(i = 1; i<=10; i++)
printf("%d\n", i);
}
```

ACHARYA

Thank you