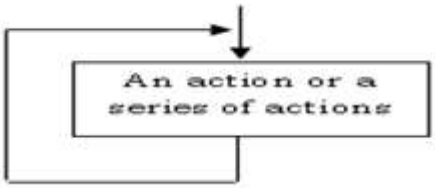
# UNIT – II

## ITERATIVE CONTROL STRUCTURES (DECISION MAKING AND LOOPING):

**Concept of a Loop:** The below diagram shows the concept of loop. In this diagram, the loop is repeated again and again it will never stop.



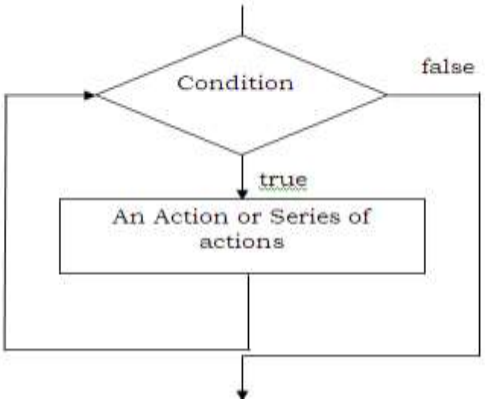

But we want the loop to be stopped after our work is over. To do this we need to put a condition to control the loop. This means that we have to design the loop so that after each iteration the condition must be checked. On checking the condition if it is true we repeat the loop again or if it is false we terminate the loop. This test condition is called **loop control expression** and the variable in the test expression that is used to control the loop is called **loop control variable**.

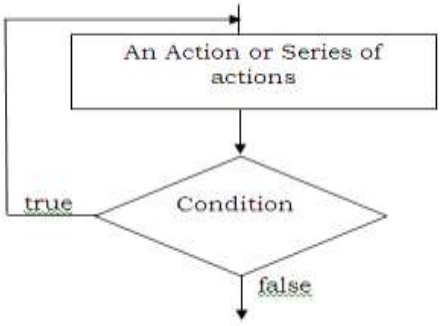
### Pretest and Post-test Loops (Entry controlled and Exit controlled loops)

Suppose if we need to test the end of the loop then where we have to test. The answer to this question is before the iteration or after the iteration of the loop. We can have both. They are

- ✓ Pretest loop or entry controlled loop
- ✓ Post-test loop or exit controlled loop

The diagrammatic representations for the pretest and post-test loops are as follows.



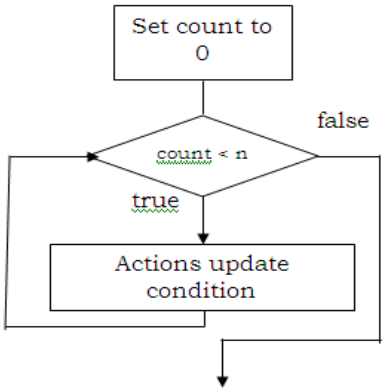

In Pretest loop, the condition is checked before the execution of the loop. If the test condition is true, then the body of the loop is executed again otherwise the loop gets terminated.

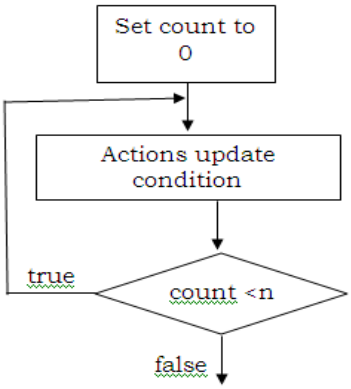
In Posttest loop, the loop is executed once and then the condition is checked. If the test condition is true, then the body of the loop is executed again otherwise the loop gets terminated.

### Counter Controlled Loops and Event Controlled Loops

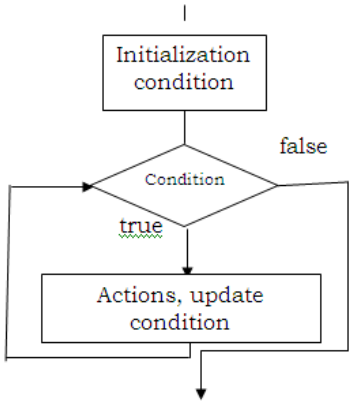
When we know exactly the number of times the loop is to be repeated then it is called as counter controlled loop. It is also known as fixed count loop. Here there will be initialization, updation and test counter. The initialization is to initialize a value to a variable. The updation can be an

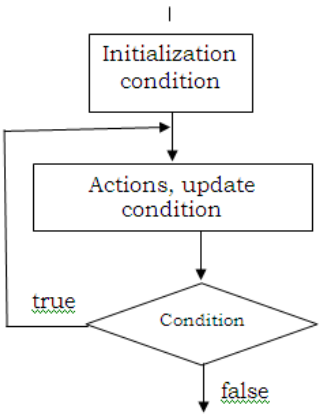
increment or decrement. The test can be a condition. The diagrammatic representations of entry and exit controlled loops for counter controlled loops are as follows.




In event controlled loop, an event changes the control expression from true to false. The test condition does not depend on the count but depends on the value that is either true or false. The diagrammatic representations of entry and exit controlled loops for event controlled loops are as follows.




**Loops in C:** In C, there are three loop statements:

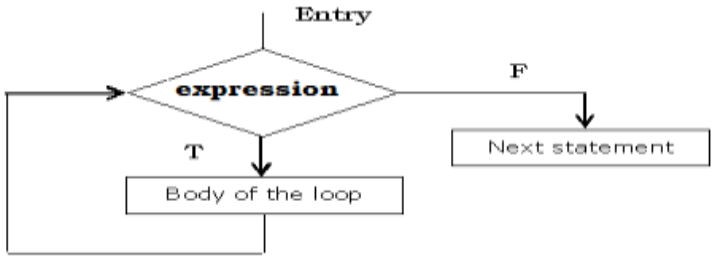
- ✓ while
- ✓ for
- ✓ do-while

The first and second are entry controlled or pretest loops and third is post test loop. We can use all of them for event and counter controlled loops. The while and do-while can be used for event controlled loops and for is used for counter controlled loop.

**While Statement**

This type of loop is also called an entry controlled loop construct or pretest loop. As it is pretest loop, it will test the expression before every iteration of the loop. The expression is executed and if is true then the body of the loop is executed this process is repeated until the expression becomes false. Once it becomes false the control is a transferred out of the loop. The general form or syntax of the while statement is:

```
while(expression)
{
        body of while loop;
}
next statement;
```




**Flow Chart**

**Example:**

```
i = 1;
while(i<=5)
{
        printf("%d",i);
        i++;
 }
```

In the above example the loop with be executed until the condition is false.

**Program to illustrate about while statement**

```
#include<stdio.h>
int main()
{
        int n, i=1,c=0;
        printf("Enter the value of n: ");
        scanf("%d", &n);
        while(i<=n)
        {
                if(n%i==0)
                        c++;
                i++;
        }
        if(c==2)
                printf(" the given number is prime %d", n);
        else
                printf(" the given number is not prime %d", n);
        return 0;
}
```

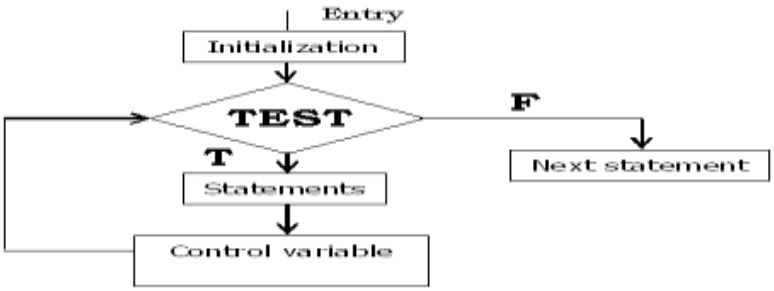
**for Statement**

for loop is another entry controlled loop that provides a more concise loop control structure. The general form or syntax of for loop is:

```
for ( initialization; test condition; increment)
{
        body of the loop;
}
next statement;
```

Where initialization is used to initialize some parameter that controls the looping action, 'test condition' represents if that condition is true the body of the loop is executed, otherwise the loop is terminated. After executing the body of the loop the new value of the control variable is again tested with the loop condition. If the condition is satisfied the body of the loop is again executed. This process continues until the value of the control variable becomes false to satisfy the condition.

**Flow Chart**




**Example:**

```
for (i=1; i<=5; i++)
{
        printf("%d",i);
}
```

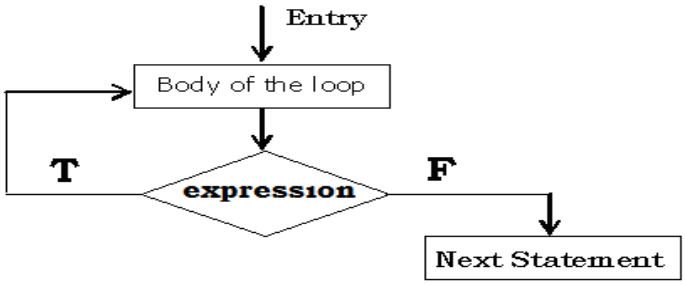
**Program to illustrate about for statement**

```
#include<stdio.h>
int main()
{
        int n, i, sum=0;
        printf("Enter the value of n: ");
        scanf("%d", &n);
        for(i=1; i<=n; i++)
        {
                if(n % i == 0)
                {
                        sum=sum+i;
                }
        }
        if(sum == n)
                printf(" Perfect Number");
        else
                printf(" Not Perfect Number");
        return 0;
}
```

**do-while Statement:**

This type of loop is also called an exit controlled loop i.e., the expression is evaluated at the bottom of the loop and if it is true then body of the loop is executed again and again until the expression becomes false. Once it becomes false the control is transferred out of the loop executing the next statement. The general form of do-while statement is:

```
do
{
        body of the loop ;
}while( expression);
next statement;
```




**Flow Chart**

**Example:**

```
i = 1;
do
{
        printf("%d",i);
        i++;
}while(i<=5);
```

**Program to illustrate about do while statement**

```
#include<stdio.h>
main()
{
        int n,r,rev=0;
        printf("Enter the value of n: ");
        scanf("%d", &n);
        do
        {
                r=n%10;
                rev=rev*10+r;
                n=n/10;
        }while(n>0);
        printf(" the reverse of the given number is %d", rev);
        return 0;
}
```

**Nested Loops:**

There are several situations where we might have one loop to be contained in another loop. Such loops are called nested loops. If one for loop contains another for loop then it is called nested for loop. The syntax is as follows

```
for(     ;         ;         )
{
        ----------------
        ----------------
        for(     ;         ;         )
        {
                ---------------
                ---------------
        }
        --------------------
        --------------------
}
```

**Example:**

```
for(i=1;i<=5;i++)
{
        printf("%d",i);
        for(j=1;j<=4;j++)
                printf("%d",j);
}
```

The first loop is controlled by "i" is called outer loop control variable. The second loop is controlled by "j" is called inner loop control variable. We have used different variables to control each loop. For each iteration of the outer loop, the inner loop is executed for its entire sequence. Thus each time "i" increases the inner loop executes completely. It is shown as follows:

When i=1 as the first iteration
j=1, j=2, j=3, j=4
When i=2 as the second iteration
j=1, j=2, j=3, j=4
When i=3 as the third iteration
j=1, j=2, j=3, j=4
When i=4 as the fourth iteration
j=1, j=2, j=3, j=4
When i=5 as the fifth iteration
j=1, j=2, j=3, j=4

**Program to illustrate about nested loop statement**

```
#include<stdio.h>
int main()
{
        int i, j;
        for(i=1;i<=4;i++)
        {
                for(j=1;j<=i;j++)
                        printf("* ");
```

**Output:**

```
*
* *
* * *
* * * *
```

```
            printf("\n");
        }
        return 0;
}
```

**Examples on Loops:**

**1. Write a C Program to find the Fibonacci series for given number.**

```
#include<stdio.h>
main()
{
        int n, f1=0,f2=1,f3;
        printf("Enter the value of n: ");
        scanf("%d", &n);
        printf("%d\t%d\t", f1,f2);
        f3=f1+f2;
        while(f3<=n)
        {
                printf("%d\t", f3);
                f1=f2;
                f2=f3
                f3=f1+f2;
        }
}
```

**Output:**

Enter the value of n: 6

0 1 1 2 3 5

**2. Write a C Program to check whether the given number is palindrome or not.**

```
#include<stdio.h>
main()
{
        int n, r,m,rev=0;
        printf("Enter the value of n: ");
        scanf("%d",&n);
        m=n;
        while(n!=0)
        {
                r=n%10;
                rev=rev*10+r;
                n=n/10;
        }
        printf(" The reverse of the given number: %d\n", sum);
        if(rev==m)
                printf("Palindrome Number");
        else
                printf("Not Palindrome Number");
}
```

**Output:**

Enter the value of n: 6446

The reverse of the given number: 6446

Palindrome number

**3. Write a C Program to check whether the given number is Strong or not.**

```
#include<stdio.h>
main()
{
        int n, r,m,sum=0;
        printf("Enter the value of n: ");
        scanf("%d", &n);
        m=n;
        while(n!=0)
        {
                f=1;
                r=n%10;
                for(i=1;i<=r;i++)
                        f=f*i;
                sum=sum+f;
                n=n/10;
        }
        if(sum==m)
                printf("Strong Number");
        else
                printf("Not Strong Number");
}
```

**Output:**

Enter the value of n: 145

Strong number

**4. Write a C Program to generate prime numbers between 1 to n.**

```
#include<stdio.h>
main()
{
        int n, i,j,c;
        printf("Enter the value of n: ");
        scanf("%d",&n);
        printf("The prime numbers between 1 to n are: ");
        for(i=2;i<=n;i++)
        {
                c=0;
                for(j=2;j<=i/2;j++)
                {
                        if(i%j==0)
                        {
                                c++;
                                break;
                        }
                }
        if((c==0)
```

**Output:**

Enter the value of n: 50

The prime numbers between 1 to n are: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47

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

**5. Write a C Program to print sum of digits**

```
#include<stdio.h>
int main(()
{
        int n, sum=0, i, r;
        printf("Enter the value of n: "));
        scanf("%d", &n);
        while(n!=0)
        {
            r = n % 10;
            sum = sum + r;
            n = n / 10;
        }
        printf("The sum of individual digits is %d", sum);
        return 0;
}
```

**Output:**

Enter the value of n: 537
The sum of individual digits is 15

**Other Statements Related to Looping**

**Break Statement:** The break statement can be used to exit the loop. When break is encountered inside a loop, the loop is immediately exited and the program continues with the statement which is followed by the loop. If used in nested loops then the break statement inside one loop transfers the control to the next outer loop.

**Example:**

```
for (i=1; i<5; i++)
{
        if( i == 4)
                break;
        printf("%d",i);
}
```

**Continue Statement:** The continue statement is used to skip the present iteration and continues with the next iteration of the loop.

Continue statement is used for continuing next iteration of loop after skipping some statement/s of loop. When continue is encountered control automatically passes to the beginning of the loop. It is usually associated with if statement. It is useful when we want to continue the next iteration of the loop without executing some part of the loop.

**Example:**

```
for (i=1; i<5; i++)
{
```

```
        if( i == 3)
                continue;
        printf("%d",i);
}
```

In the above example when i=3 then the continue statement will execute and skip the rest of the statements in the loop and continues with the next iteration i.e., i=4.

The difference between break and continue is, when the break is encountered loop is terminated and the control is transferred to the statement after the loop and when continue is encountered control is transferred to the beginning of the loop.

**ARRAYS**

**"An array is defined as fixed size sequenced collection of elements of same data type"**

**"An array is defined as collection of elements that share a common name"**

**"An array is defined as collection of homogeneous elements"**

**"An array is defined as collection of data items stored in contiguous memory locations"**

Arrays are used to represent list of values in dimensions. Hence arrays are classified into three categories

- ✓ One dimensional array
- ✓ Two-dimensional
- ✓ Multidimensional arrays

**One - Dimensional Array:**

A list of items can be specified under one variable name using only one subscript and such a variable is called a **single-subscripted variable** or a **one-dimensional array** or **1D array** or **Single dimensional array**. It is simply called as **array**. The subscript begins with the number 0.

If we want to represent a set of five numbers, say (45, 65, 10, 93, 50) in an array variable marks, we may declare the variable marks as follows:

**int marks[5];**

Now the computer stores the values of the array marks after reserving five storage locations and assigning the values to array elements.

| | |
|---|---|
| marks[0] | |
| marks[1] | |
| marks[2] | |
| marks[3] | |
| marks[4] | |

```
marks [0]=45;
marks [1]=65;
marks [2]=10;
marks [3]=93;
marks [4]=40;
```

| | |
|---|---|
| marks[0] | 45 |
| marks[1] | 65 |
| marks[2] | 10 |
| marks[3] | 93 |
| marks[4] | 40 |

**Declaration of One Dimensional Array:** An array variable is declared by specifying first the data type of the array, then the name of the array variable, and then the number of elements of the array that is specified between a pair square brackets ([]). The general format or syntax of array declaration is:

**type variable_name[size];**

**Example:** int marks [100];

Here **int** specifies the data type and **marks** specifies the name of the array and **100** specifies the maximum size of the array.

**Accessing Elements of the Array:** Once the array is declared it is possible to refer the individual elements of the array. This is done with the index or subscript which contains a number in the square brackets following the array name. The number specifies the element position in the array. All the array elements are numbered starting from 0. Thus marks[2] specifies the third element of the array.

**Input and Output of Array Values:**

**Entering data into an array/ Storing into an array:**

```
for(i=0;i<30;i++)
{
        printf(" enter the marks");
        scanf("%d",&marks[i]);
}
```

Here the variable "i" is used as the subscript for referring various elements of the array. The for loop causes the process to be repeated for 30 times in asking to enter the students marks from user. For the first time the value of "i" is 0 read through scanf() statement and get the value of marks[0] which is the first element of the array. The process is repeated until "i" value terminates the condition.

**Printing data from an array**

```
for(i=0;i<30;i++)
{
        printf("%d", marks[i]);
}
```

Here the variable "i" is used in the subscript for referring various elements of the array. The for loop causes the process to be repeated for 30 times in printing the students marks. For the first time the value of "i" is 0 printed through printf() statement and print the value of marks[0] which is the first element of the array. The process is repeated until "i" value terminates the condition.

**Initialization of Arrays:** After declaring an array the individual elements of an array are initialized otherwise they will contain garbage values. The initialization takes place in two ways.

- ✓ Compile time initialization
- ✓ Run time initialization

**Compile time initialization:** Elements of an array can be assigned initial values by following the array definition with a list of values enclosed in braces and separated by commas. The general format of initialization is: **type array_name[size]={list of values};**

**Example:** **int marks[5] = {65, 98, 62, 48, 57};**

Defines the array marks to contain five integer elements and initializes marks[0] to 65, marks[1] to 98, marks[2] to 62, marks[3] to 48 and marks[4] to 57. If the number of initializers are less than the number of element in the array, the remaining elements are set to zero.

**Example:**

int m[5] = {3, 4, 8}; is equivalent to int m [5]= {3, 4, 8, 0, 0};

If initializers are provided for an array, it is not necessary to explicitly specify the array length, in which case the length is derived from the initializers. A character array may be initialized by a string constant, resulting in the first element of the array being set to the first character in the string, the second element to the second character, and so on. The array also receives the terminating '\0' in the string constant.

**Example:**

```
char name[10] ="computers";
char name[10] = {'c', 'o', 'm', 'p', 'u','t', 'e', 'r', 's', '\0'};
```

**Run time initialization:** Arrays are initialized at run time also. This approach is used when large arrays are considered.

```
for(i=0;i<100;i++)
{
        if(i<50)
                sum[i]=0;
        else
                sum[i]=1;
}
```

From the above example the first 50 elements of the array are initialized to 0 and second 50 elements of the array are initialized to 1 at run time.

We can also use scanf() to initialize an array.

```
int x[3];
scanf("%d%d%d",&x[0],&x[1],&x[2]);
```

**Storing an Array / Representing One Dimensional Array in Memory**

Let us consider the array declaration

**int a[10];**

Now 40 bytes get immediately reserved in memory because integer occupies 4 bytes of memory. So for 10 elements 40 bytes of memory is required. Array elements occupy contiguous memory locations. The diagrammatic representation of how the array elements are stored in the array after initialization is shown below.

**int a[10]={1,2,3,4,5,6,7,8,9,10};**

| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9] |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2000 | 2004 | 2008 | 2012 | 2016 | 2020 | 2024 | 2028 | 2032 | 2036 |

**TWO - DIMENSIONAL ARRAYS:**

If an array contains two dimensions then it is called **two dimensional arrays**. This two dimensional array is also called as **matrix**. It is called as **array of arrays**. Since it contains two subscripts it is also known as **double subscripted variable**. It is used to represent table of values containing row values and column values.

**Two Dimensional Array Declaration:**

The two-dimensional array can be declared by specifying first the base type of the array, then the name of the array variable, and then the number of rows and column elements the array should be specified between a pair square brackets ([] []). Note that this value cannot be a variable and has to be an integer constant.

**type variable_name[row][column];**

Total no. of elements in 2-D array is calculated as **row*column**

**Example:**

```
int student[4][3]; //Total number of elements in the array will be 4*3=12
```

Here **int** specifies the data type and **student** specifies the name of the array and **size** specifies the row size and column size of the array. From the above example there are 4 rows and 3 columns for the array student.

**Two Dimensional Array Initialization:**

Elements of an array can be assigned initial values by following the array definition with a list of initializers enclosed in braces and separated by commas.

**Example:**

```
int marks[2][3] = {
                    {65, 98, 62},
                    {48, 57, 40}
                  };
```

Defines the array marks to contain six integer elements and initializes marks[0][0] to 65, marks[0][1] to 98, marks[0][2] to 62, marks[1][0] to 48, marks[1][1] to 57 and marks[1][2] to 40.

**Input and Output of 2D - Array Values**

**Entering data into 2D array**

```
for(i=0;i<30;i++)
{
        for(j=0;j<30;j++)
                scanf("%d",&marks[i][j]);
}
```

**Printing data from a 2D array**

```
for(i=0;i<30;i++)
{
        for(j=0;j<30;j++)
                printf("%d", marks[i][j]);
}
```

**Storing two dimensional Array / Representing two dimensional Array in Memory**

Consider the following example to explain about storing two dimensional arrays.

| | m1 | m2 | m3 |
|---|---|---|---|
| student 1 | 75 | 85 | 90 |
| student 2 | 85 | 75 | 95 |
| student 3 | 90 | 70 | 75 |
| student 4 | 95 | 65 | 85 |

| | [0][0] | [0][1] | [0][2] |
|---|---|---|---|
| row 0 | 75 | 85 | 90 |

| | [1][0] | [1][1] | [1][2] |
|---|---|---|---|
| row 1 | 85 | 75 | 95 |

| | [2][0] | [2][1] | [2][2] |
|---|---|---|---|
| row 2 | 90 | 70 | 75 |

| | [3][0] | [3][1] | [3][2] |
|---|---|---|---|
| row 3 | 95 | 65 | 85 |

**LARGER DIMENSIONAL ARRAYS / THREE DIMENSIONAL ARRAYS / MULTIDIMENSIONAL ARRAYS:** The programming language C allows three or more dimensions. But they are used rarely. Three dimensional arrays are called as array of array of arrays. The general format of three or multidimensional arrays is

**type variable_name[s1][s2][s3]……[sn]**

type specifies the data type and variable_name specifies the name of the array and [s1][s2]……[sn] specifies the sizes of the multidimensional array.

**Example: int a[3][4][2];**

**Initialization of three dimensional/multi-dimensional arrays:**

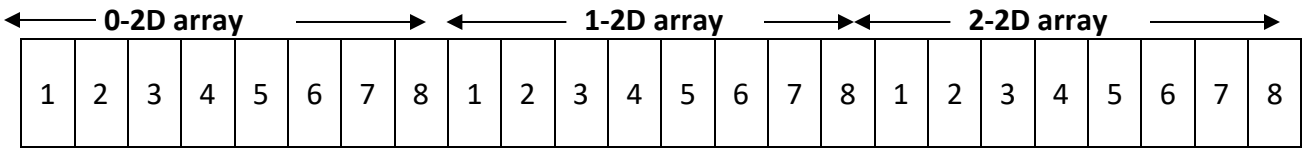
The initialization of three or multi-dimensional arrays is shown below.

```
int a[3][4][2]={  {       {1,2},
                          {3,4},
                          {4,5},
                          {6,8}
                   },
                  {       {1,2},
                          {3,4},
                          {4,5},
                          {6,8}
                  },
                  {       {1,2},
                          {3,4},
                          {4,5},
                          {6,8}
                   },
                };
```

The outer array contains three elements each of which is a two dimensional array of four one dimensional arrays each of which contains two integers.

**Storing larger dimensional arrays / three dimensional Array / Representing three dimensional Array in Memory**

The three dimensional or multidimensional array elements stored in the memory is represented as shown below for the above example



**Program to illustrate one-dimensional array (sorting array elements)**

```
main()
{
	int a[10], n, i, j, t;
	printf("Enter the array size: ");
	scanf("%d", &n);
	printf("Enter the array elements: ");
	for(i=0;i<n;i++)
	{
		scanf("%d",&a[i]);
	}
	printf(" The elements after sorting are: ");
	for(i=0;i<n;i++)
	{
		for(j=0;j<n-1-i;j++)
		{
			if(a[j]>a[j+1])
			{
				t=a[j];
				a[j]=a[j+1];
				a[j+1]=t;
			}
		}
	}
	for(i=0;i<n;i++)
	{
		printf("%d",a[i]);
	}
}
```

**Program to illustrate about largest and smallest element in the array**

```
main()
{
	int n, a[10], i, large, small;
	printf("Enter the array size: ");
	scanf("%d", &n);
	printf("Enter the array elements: ");
	for(i=0;i<n;i++)
	{
		scanf("%d", &a[i]);
	}
	large=small=a[0];
	for(i=0;i<n;i++)
	{
```

```
            if(a[i]>large)
                large=a[i];
            else
                if(a[i]<small)
                small=a[i];
    }
    printf("%d\t%d", large, small);
}
```

**Matrix operations on two dimensional arrays:**

**Programs to illustrate about matrix addition, subtraction, multiplication and transpose**

**/* matrix addition*/**

```
main()
{
    int a[2][2],b[2][2],c[2][2],n,m,i,j;
    printf("Enter the size of the matrix: ");
    scanf("%d%d", &n, &m);
    printf("Enter the a matrix elements: ");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        scanf("%d",&a[i][j]);
    }
    printf("Enter the b matrix elements: ");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        scanf("%d",&b[i][j]);
    }
    printf("The matrix addition is: ");
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
        c[i][j]=a[i][j]+b[i][j];
    }
    for(i=0;i<n;i++)
    {
        for(j=0;j<m;j++)
            printf("%d ",c[i][j]);
        printf("\n");
    }
}
```

**/* matrix subtraction*/**

```
main()
{
	int a[2][2],b[2][2],c[2][2],n,m,i,j;
	printf("Enter the size of the matrix: ");
	scanf("%d%d", &n, &m);
	printf("Enter the a matrix elements: ");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		scanf("%d",&a[i][j]);
	}
	printf("Enter the b matrix elements: ");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		scanf("%d",&b[i][j]);
	}
	printf("The matrix subtraction is: ");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		c[i][j]=a[i][j]-b[i][j];
	}
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
			printf("%d ",c[i][j]);
		printf(“\n”);
	}
}
```

**/* matrix multiplication*/**

```
#include<stdio.h>
main()
{
	int a[2][2],b[2][2],c[2][2],n,m,i,j,k;
	printf("Enter the size of the matrix");
	scanf("%d%d",&n,&m);
	printf("Enter the a matrix elements");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		scanf("%d",&a[i][j]);
	}
```

```
	printf("Enter the b matrix elements");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		{
			scanf("%d",&b[i][j]);
		}
	}
	printf("The matrix multiplication is ");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		for(k=0;k<n;k++)
		{
			c[i][j]=0;
			c[i][j]=c[i][j]+a[i][k]*b[k][j];
		}
	}
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		printf("%d\t",c[i][j]);
	}
}
```

**/* Matrix Transpose*/**

```
#include<stdio.h>
int main()
{
	int a[10][10], t[10][10], i, j, n, m;
	printf("Enter the size of the matrix: ");
	scanf("%d%d", &n, &m);
	printf("Enter the matrix elements\n");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
		{
			scanf("%d", &a[i][j]);
		}
	}
	printf("The transpose of matrix is\n");
	for(i=0;i<n;i++)
	{
		for(j=0;j<m;j++)
```

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

**/* Program for finding frequency of the elements in the array */**

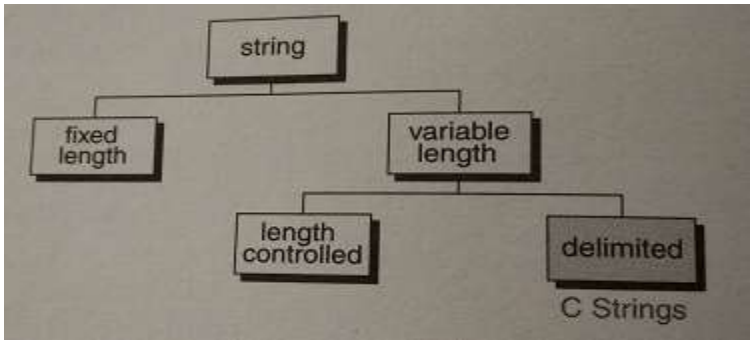
```
#include <stdio.h>
int main()
{
        int arr1[100], fr1[100];
        int n, i, j, ctr;
        printf("Input the number of elements to be stored in the array: ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Element - %d : ",i);
                scanf("%d",&arr1[i]);
                fr1[i] = -1;
         }
         for(i=0; i<n; i++)
         {
                ctr = 1;
                for(j=i+1; j<n; j++)
                {
                        if(arr1[i]==arr1[j])
                        {
                                ctr++;
                                fr1[j] = 0;
                        }
                }
                if(fr1[i]!=0)
                {
                        fr1[i] = ctr;
                }
```

```
        }
        printf("\nThe frequency of all elements of array: \n");
        for(i=0; i<n; i++)
        {
                if(fr1[i]!=0)
                {
                printf("%d occurs %d times\n", arr1[i], fr1[i]);
                }
        }
        return 0;
}
```

**STRINGS**

A string is a series of characters treated as a unit. The string is classified into two categories.

- ✓ Fixed length strings
- ✓ Variable length strings




**Fixed Length Strings**

- ✓ In fixed length string format, the first decision is size of the variable.
- ✓ If it is too small, we cannot store all data. If it is too big, we waste memory.
- ✓ The problem is how to tell the data from nondata. The solution is to add nondata at the end of the data.

**Variable Length Strings**

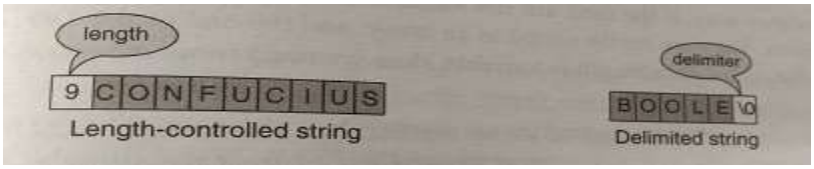
- ✓ A variable length strings can expand or contract to contain the data.
- ✓ To store a person name consisting of five characters provide five character space as well a person name consisting of 20 character provide 20 character space to contain it.
- ✓ There are two techniques for variable length strings
    - ✓ Length controlled strings
    - ✓ Delimited strings

**Length Controlled Strings**

It has a count that specifies the number of characters in the string. This count is then used by the string function to determine the length of the string.

**Delimited Strings**

To identify the end of the string is a delimiter at the end of the delimited strings. The delimiter is specified by null character(\0).




**C STRINGS:**

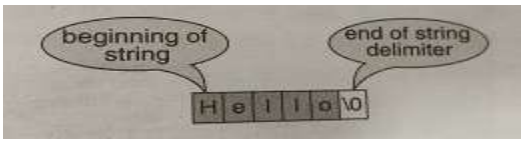
**"A string is defined as sequence of characters that is treated as single data item".**

**"A string is variable length array of characters that is delimited by the null character".**

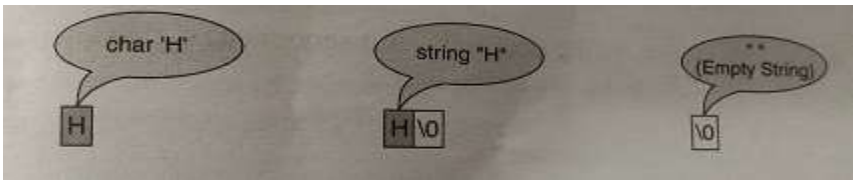
**Examples**: abcd
Ab123cde
Ramu etc

**Storing Strings**

- ✓ String is stored as array of characters. It is terminated by null character ('\0').
- ✓ As string is stored as an array, the name of the string is a pointer to the beginning of the string.
- ✓ The below diagram shows how a string is stored in memory:

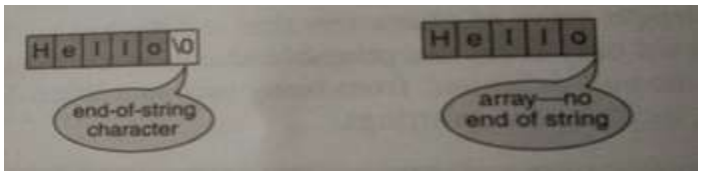



- ✓ We need to observe the difference about storing a character and a string in memory
- ✓ The character requires one memory location and single character string requires two memory locations since one memory location for data and one for null character.




**String Delimiter**

- ✓ Why do we need null character at the end of the string? The answer is string is not a type but it is data structure.
- ✓ The physical structure is the array and the logical structure is its definition.
- ✓ We need to identify the logical end of the string within the physical structure.
- ✓ If data is variable length, then we need to determine the end of the data. Therefore we use null character to determine the end of the string.
- ✓ The difference between array and string is shown below:



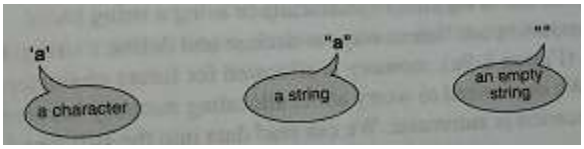

**String Literals**

**"A group of characters defined between double quotation marks is called as string constant"**

Examples "abcd"
"Ab123cde"
"Ramu" etc

**Strings and Characters**

- ✓ When we need to store a single character, we have two options.
- ✓ The first as a single character constant then we need to use single quotes and second as a string constant then we need to use double quotes.
- ✓ The difference is the way we manipulate data.
- ✓ Moving a character from one location to another needs assignment but moving a string needs a function call.
- ✓ The important another difference is the absence of data. The null character is used to represent the absence of data.



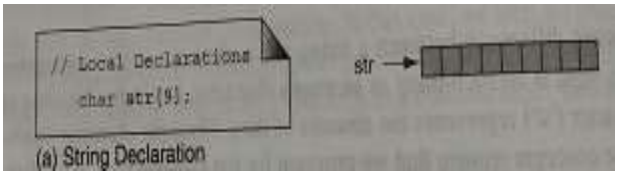

**Declaration of strings / character array declaration:**

The C language does not support string data type. As a string is declared as array of characters hence the general form of string declaration is as follows:

**char string_name[size];**

In the above syntax the size specifies the number of characters in the string_name. For example

**char str[9];**




**Initialization of strings**

There are two types of initializations. They are as shown below:

**char str[9]= "Good Day";**
**char str[9]={'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y', '\0'};**

When we specify the list of character array elements we need to supply the null character to indicate the end of the strings. The C language also permits to initialize the strings without specifying the size. It is shown as follows:

**char month[ ]= "January";**


**Strings and Assignment Operator**

As string is an array, the name of the string is a pointer constant. As pointer constant, it is an rvalue and therefore it cannot be used as left operand of the assignment operator. For example,

```
char str1[6] = "hello";
char str2;
str2 = str1;    //Compile error
```

**Reading and Writing Strings**

A string can be read and written. There are several string functions for input and output.

**STRING INPUT AND OUTPUT FUNCTIONS**

We have two set of functions to read and write strings

1. formatted input / output functions (scanf & printf)
2. string input // output functions (gets & puts)

**Formatted Functions**

**Using scanf()function**

The **scanf()** function is used to read string of characters using the format specification **%s.**

```
char address[10];
scanf("%s", address);
```

The problem with scanf() function is that it terminates its input on the first white space it finds. A white space may be a blank, tab, carriage return, form feed and new line.

**Example:** **NEW YORK**

Then only the **NEW** of the string **NEW YORK** will be read into the address because after **NEW** there is blank space so the scanf() function terminates reading the string.

We can also specify the field width using the form **%ws** in the scanf() statement. The width field specifies the number of characters to be read from the input string.

**scanf("%ws",name);**

**Using printf() function**

The printf() function is used to print the strings to screen. We use the format specification for printf() function as **%s.** For example

**printf("%s",name);**

The above function displays the entire contents of **name.** The specification of precision such as **%10.4** indicates that the first **four** characters are printed in the field width of 10 columns.

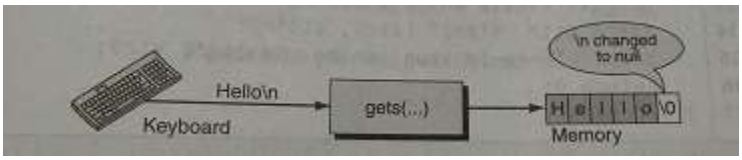
**String Input / Output Functions**

C has two sets of string functions to read and write.

1. Line to string
2. String to line

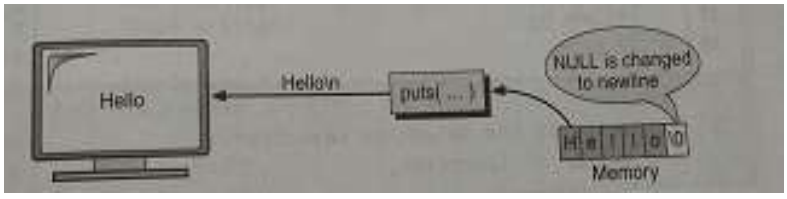
**Line to string**

- ✓ The gets function take a line terminated by a newline from the input stream and make a null terminated string.
- ✓ It is called line-to-string input functions
- ✓ The syntax is as follows: **gets(str);**
- ✓ The diagrammatic representation about how it works is shown below:




**String to line**

- ✓ The puts function take null terminated string from memory and write it to a file.
- ✓ It is called as string –to-line output function.
- ✓ The syntax is as follows: **puts(str);**
- ✓ The diagrammatic representation about it works is shown below:




**STRING MANIPULATION FUNCTIONS (Defined in string.h header file)**

The following are the string handling functions. They are used to carry out operations on strings known as string manipulations.

- ✓ String concatenation – **strcat()** – used to concatenate two strings
- ✓ String comparison – **strcmp()** – used to compare two strings
- ✓ String copy – **strcpy()** – used to copy one string to another string
- ✓ String length – **strlen()** – used to find the length of the string

**strcat() function**

The **strcat** function is used to join two strings together. It takes the following form:

**strcat(string1, string2);**

**or**

**char *strcat(char *str1, const char *str2);**

Here the string1 and string2 are character arrays. When the function **strcat** is executed, the **string2** is appended to **string1** and string1 size should be large enough to hold the entire string after concatenation. It does by removing the null character at the end of the first string and placing the string2 from there. The **string2** remains unchanged.

**Example:**

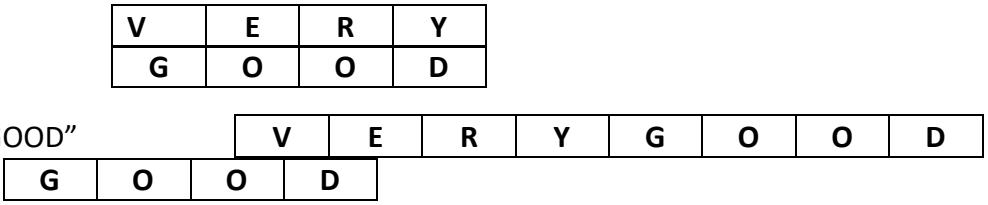
String1=”VERY”

String2=”GOOD”

strcat(string1,string2)

Result is string1=”VERY GOOD”

String2=”GOOD”




**Program to illustrate about string concatenation using string library function**

```
#include<string.h>
#include<stdio.h>
main()
{
        char str1[10], str2[10];
        printf("Enter first string: ");
        gets(str1);
        printf("Enter second string: ");
        gets(str2);
        strcat(str1,str2);
        printf(“The string concatenation is %s\n”,str1);
        puts(str2);
}
```

**Program to illustrate about string concatenation without using string library function**

```
#include <stdio.h>
int main()
{
        char s1[50], s2[50];
        int i, j;
        printf("Enter first string: ");
        gets(s1);
        printf("Enter second string: ");
        gets(s2);
        for(i=0; s1[i]!='\0'; i++)  /* i contains length of string s1. */
        for(j=0; s2[j]!='\0'; ++j, ++i)
        {
                s1[i]=s2[j];
        }
        s1[i]='\0';
        printf("After concatenation: %s",s1);
}
```

**strcmp() function**

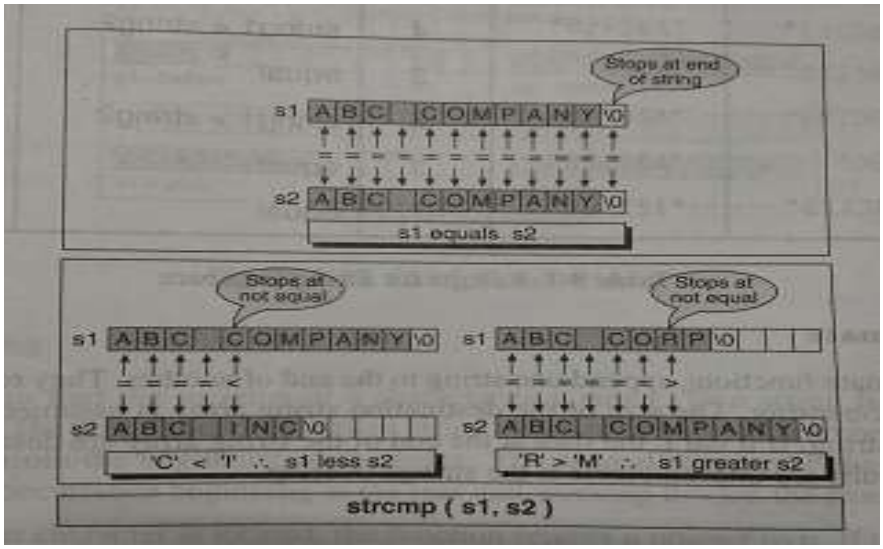
The **strcmp** function is used to compare two strings. It is done character by character. If the value is 0 then they are equal. It they are not equal then we should consider the numeric difference between them. It takes the following form:

**strcmp(string1, string2);**
**or**
**int strcmp(const char *str1, const char *str2);**

Here the string1 and string2 are string variables or string constants. For example

**strcmp(name1,name2);**
**strcmp(name1,"john");**
**strcmp("rom","ram");**

If you consider the third statement such as **strcmp("rom","ram").** The second characters in both the strings are different then we have to consider their ASCII values to get the result. The result may be zero, positive or negative.




**Program to illustrate about string comparison using string library function**

```
#include<stdio.h>
#include<string.h>
main()
{
        int x;
        char str1[10], str2[10];
        gets(str1);
        gets(str2);
        x=strcmp(str1,str2);
        if(x==0)
                printf("The strings are equal");
        else
```

```
                printf("The strings are not equal");
}
```

**Program to illustrate about string comparison without using string library function**

```
#include<stdio.h>
main()
{
        char str1[5],str2[5];
        int i;
        printf("Enter the string1: ");
        gets(str1);
        printf(" Enter the String2: ");
        gets(str2);
        i=0;
        while(str1[i]==str2 && str1[i]!='\0' && str2[i]!='\0')
            i++;
        if(str1[i]=='\0' && str2[i]=='\0')
            printf("The strings are equal");
        else
            printf("The strings are not equal");
}
```

**strcpy() function**

The **strcpy** function is used to copy one string over another string. It takes the following form:

**strcpy(string1, string2);**

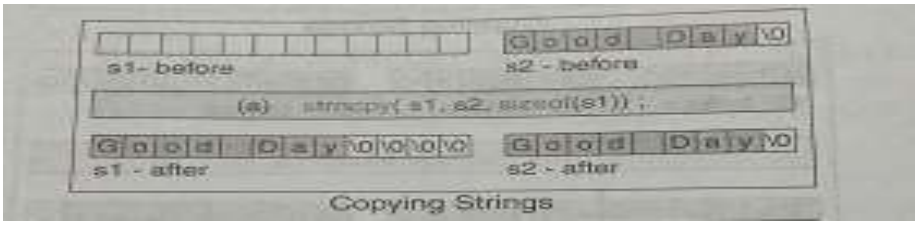
**or**

**char *strcpy(char *tostr, const char *fromstr);**

Here the contents of string2 are assigned to string1 where string2 may be a string constant. For example

**strcpy(city,"delhi");**

will assign the string **"delhi"** to the string variable **city**.




**Program to illustrate about string copy using string library function**

```
#include<string.h>
#include<stdio.h>
main()
{
        char str1[10], str2[10];
```

```
        printf("Enter the string1: ");
        gets(str1);
        printf(" Enter the String2: ");
        gets(str2);
        strcpy(str1,str2);
        puts(str1);
        puts(str2);
}
```

**Program to illustrate about string copy without using string library function**

```
#include <stdio.h>
main()
{
        char s1[100], s2[100];
        int i;
        printf("Enter string s1: ");
        gets(s1);
        for(i=0; s1[i]!='\0'; ++i)
        {
                s2[i]=s1[i];
        }
        s2[i]='\0';
        pintf("String s2: ");
        puts(s2);
}
```

**strlen() function**

The **strlen** function counts and returns the number of characters in a string. It takes the following form:

**n=strlen(string);**

**or**

**int strlen(const char *string);**

Here **"n"** is integer variable, which receives the integer value of length of the string. The parameter can be a string constant. For Example

**n=strlen("good"); //returns 4 to n as length**

**Program to illustrate about string length using string length function**

```
#include<string.h>
#include<stdio.h>
main()
{
        int x;
        char str[10];
        printf("Enter a string: ");
```

```
        gets(str);
        x=strlen(str);
        printf(" the length of the string is %d",x);
}
```

**Program to illustrate about string length without using string length function.**

```
#include<string.h>
main()
{
        int i;
        char str[100];
        printf("Enter a string: ");
        gets(str);
        i=0;
        while(str[i] !='\0')
        {
                i++;
        }
        printf(" The length of the string is %d",i);
}
```