UNIT - 2

CONTROL STATEMENTS

ARRAYS

STRINGS

Decision Making Statements / Control Statements

In C, until so far, in all the programs, the control is flowing from one instruction to next instruction. Such flow of control from one instruction to next instruction is known as sequential flow of control. But, in most of the C programs, while writing the logic, the programmer might want to skip some instructions or repeat a set of instructions again and again. This is can be called as non-sequential flow of control. The statements in C, which allows the programmers to make such decisions, are known as decision making statements or control statements.

In C, there are two types of decision making statements. One type is used to branch the control into different ways and the other type is used to repeat a set of instructions again and again. The two types of decision making statements are:

1. Selection statements or Branching statements
2. Looping statements

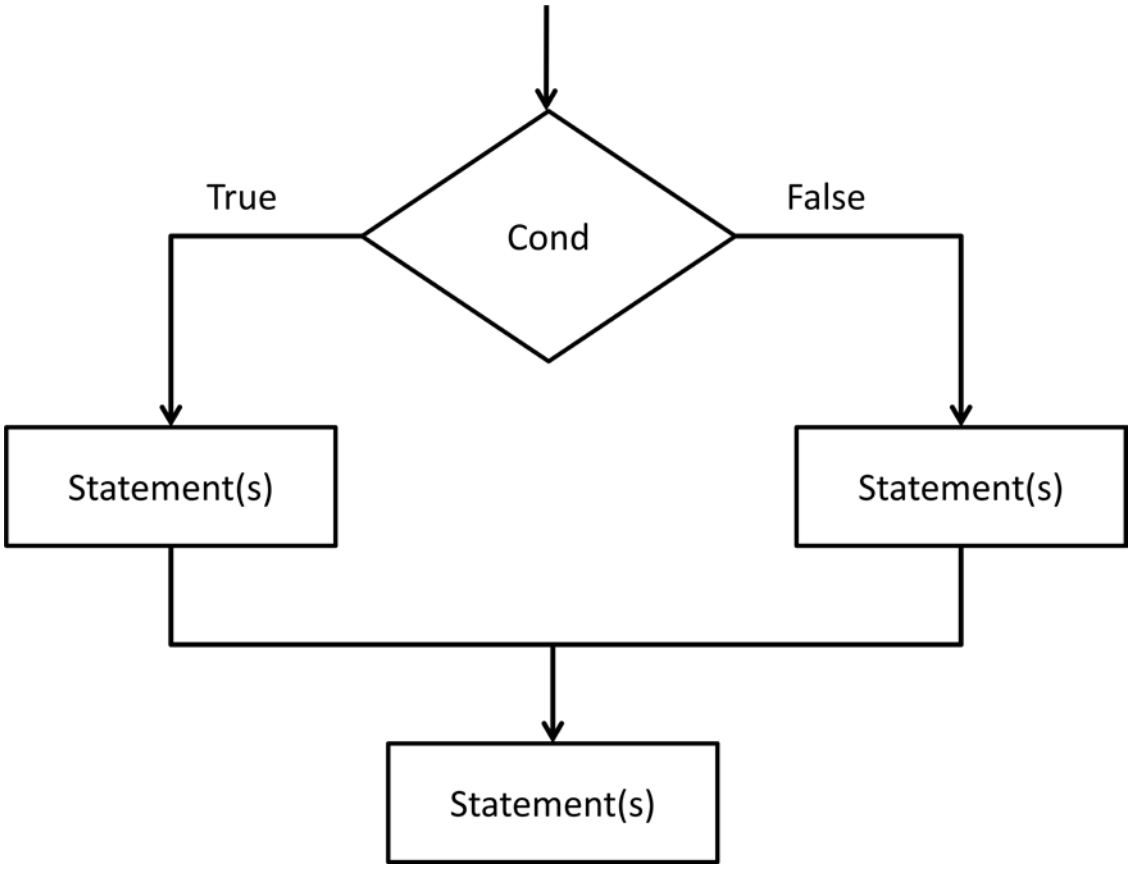
Selection Statements or Branching Statements

The selection statements in C enable the programmer to select a set of instructions to be executed by the CPU. This selection is based on a condition. C also supports a set of unconditional branching statements which transfers the control to some other place in the program. The selection statements in C are:

1. if statement
2. switch statement
3. Conditional operator statement
4. goto

if Statement

The **if** statement allows the programmer to select a set of instructions to be executed, based on a condition. If the condition is evaluated to **true,** then one set of instructions will be executed or if the condition is evaluated to **false,** another set of instructions will be executed. The general from of **if** statement is as shown below:

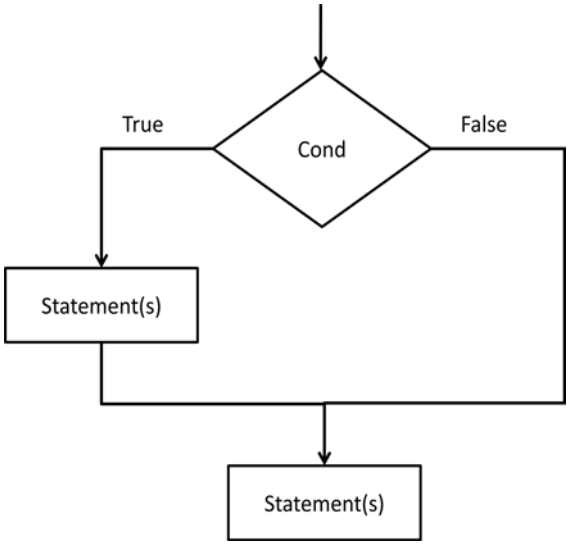


There are four variations of the **if** statement. They are:

1. Simple if or null else
2. if…else
3. Nested if
4. else if Ladder

Simple if or null else

The **simple if** allows the programmer to execute or skip a set of instructions based on the value of a condition. The **simple if** is a one way selection statement. If the condition is true, a set of statements will be executed. If the condition is false, the control will continue with the next statement after the **if** statement. The **simple if** can be represented diagrammatically as shown below:

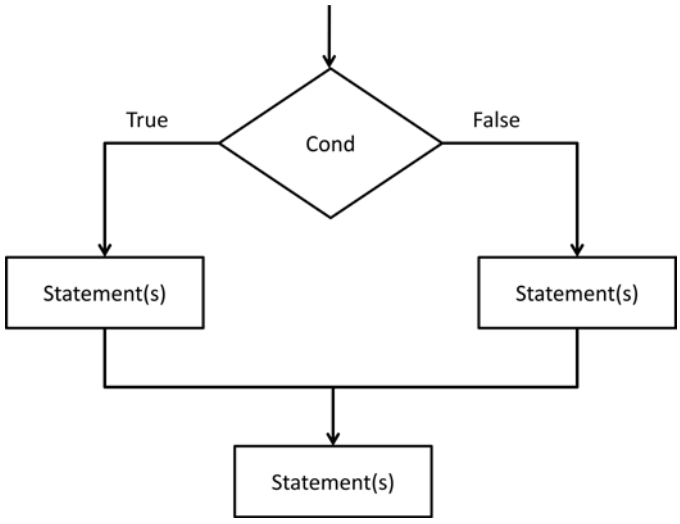


Syntax for **simple if** is as shown below:

```
if(cond/expr)
{
        Stmt(s);
}
Stmt(s);
```

if…else Statement

The **if…else** is a two way decision making selection statement. If the condition evaluates to true, one set of instructions will be executed. If the condition evaluates to false, another set of instructions will be executed. The **if…else** statement can be represented diagrammatically as shown below:

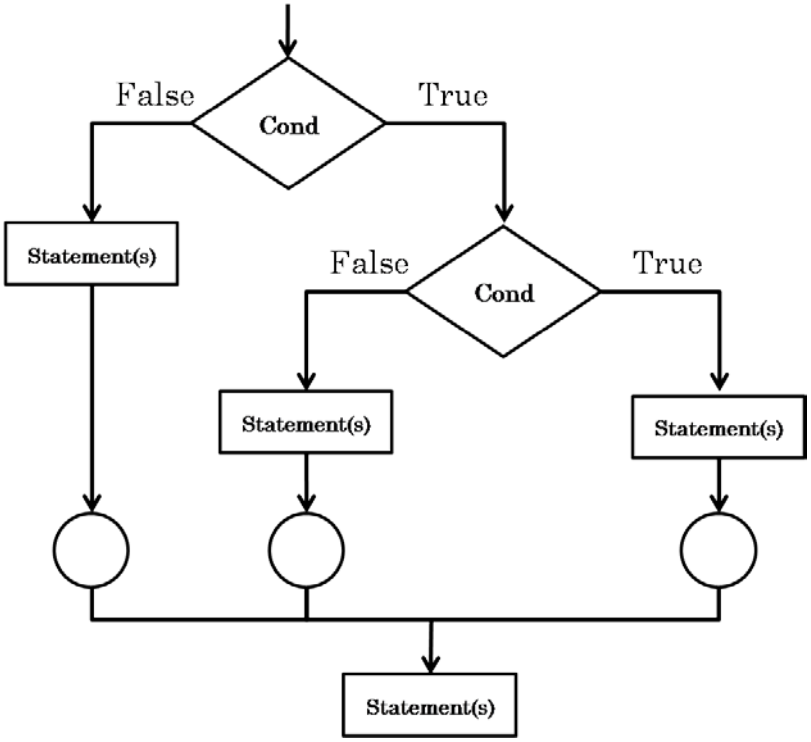


Syntax of **if…else** statement is as shown below:

```
if (cond/expr)
{
        Stmt(s);
}
else
{
        Stmt(s);
}
Stmt(s);
```

Nested if

The **nested if** can also be called as **cascaded if**. In **nested if** statement, one **if** statement is nested within another **if** statement. A **nested if** statement can be used as an alternative to logical AND operator. If the condition is evaluated to true in the first **if** statement, then the condition in the second **if** statement is evaluated and so on. The **nested if** statement can be represented diagrammatically as shown below:

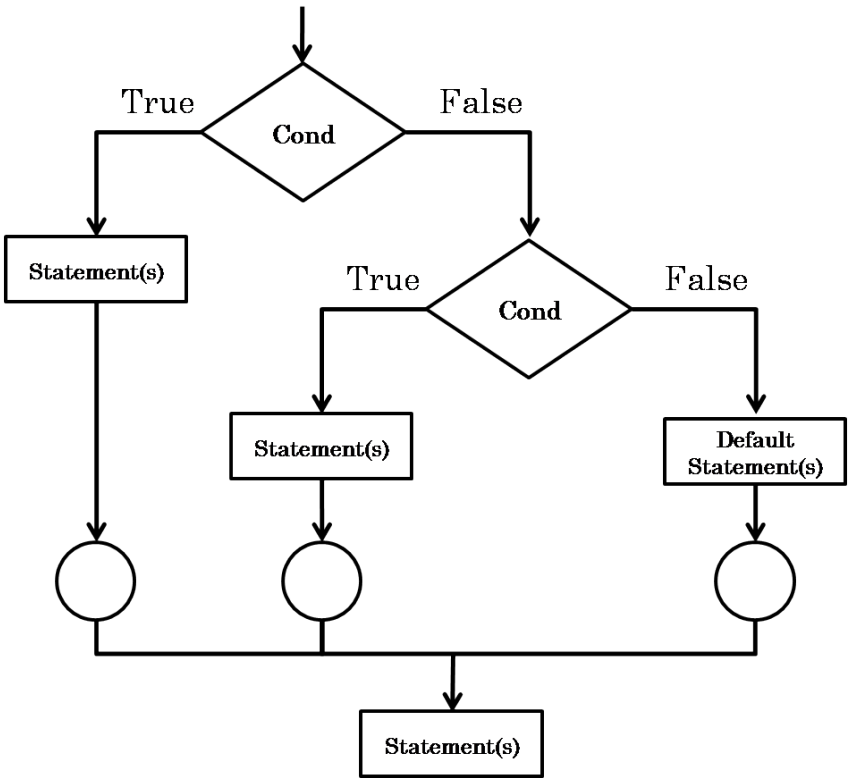


Syntax of **nested if** statement is as shown below:

```
if(cond/expr)
{
        if(cond/expr)
        {
                Stmt(s);
        }
        else
        {
                Stmt(s);
        }
}
else
{
        Stmt(s);
}
```

else if Ladder

The **else if** ladder in C is a multi way selection operator. It allows the programmer to select one set of instructions among various other sets of instructions. The **else if** ladder is similar to the logical OR operator. If first condition is **true**, then corresponding set of instructions will be executed. If the condition is **false**, then the next condition is checked and so on. If all the conditions fail, the statements in the default block will be executed. The **else if** ladder can be represented diagrammatically as shown below:



Syntax of **else if** statement is as shown below:

```
if(cond/expr)
{
        Stmt(s);
}
else if(cond/expr)
{
        Stmt(s);
}
else if(cond/expr)
{
        Stmt(s);
}
....
else
{
        Stmt(s);
}
Stmt(s);
```

/* Program to demostrate if statement */

```
#include<stdio.h>
#include<conio.h>
main()
{
   int a,b;
   printf("Enter the value of a:");
   scanf("%d",&a);
   printf("Enter the value of b:");
   scanf("%d",&b);
   if(a>b)
   {
      printf("a is greater than b!");
   }
}
```

/* Program to demostrate if...else statement */

```
#include<stdio.h>
#include<conio.h>
main()
{
   int a,b;
   printf("Enter the value of a:");
   scanf("%d",&a);
   printf("Enter the value of b:");
   scanf("%d",&b);
   if(a>b)
   {
      printf("a is greater than b!");
   }
   else
   {
      printf("b is greater than a!");
   }
}
```

/* Program to demostrate nested if statement */

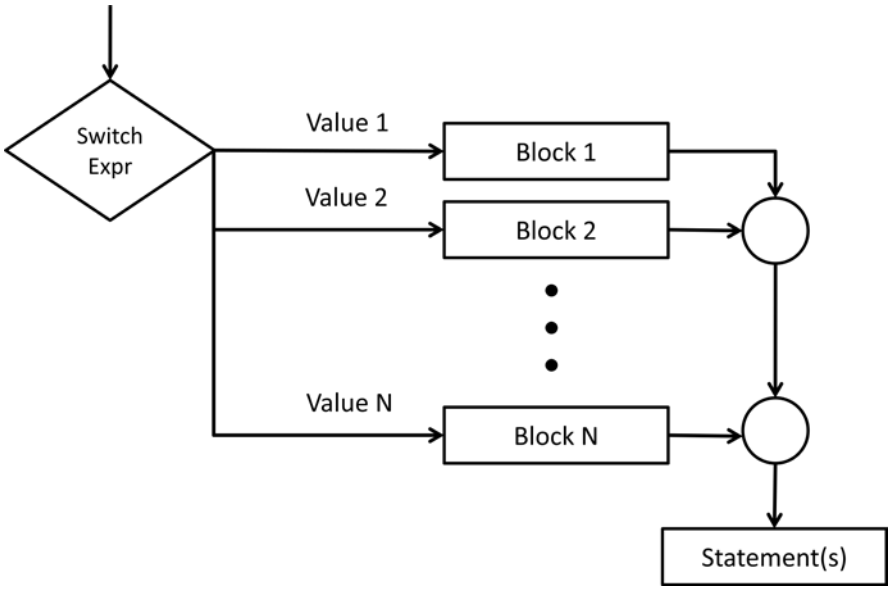
```
#include<stdio.h>
#include<conio.h>
main()
{
   int a,b,c;
   printf("Enter the value of a:");
   scanf("%d",&a);
   printf("Enter the value of b:");
   scanf("%d",&b);
   printf("Enter the value of c:");
   scanf("%d",&c);
   if(a>b)
   {
      if(a>c)
      {
         printf("a is greater than b and c");
      }
      else
      {
         printf("a is not greater than c");
      }
   }
   else
   {
      printf("a is not greater than b");
   }
}
```

/*Program to demonstrate else if ladder*/

```
#include<stdio.h>
main()
{
   int marks;
   printf("Enter the marks: ");
   scanf("%d",&marks);
   if(marks>90)
   {
      printf("\nGarde - A");
   }
   else if(marks>75&&marks<=90)
   {
      printf("\nGrade - B");
   }
   else if(marks>60&&marks<=75)
   {
      printf("\nGrade - C");
   }
   else if(marks>45&&marks<=60)
   {
      printf("\nGrade - D");
   }
   else
   {
      printf("\nFAIL!");
   }
}
```

switch Statement

Although, C provides a multi way selection statement like **else if**, when the number of conditions increases, the program will become less readable. To solve this problem, C provides an easy to understand multi way selection statement called **switch** statement. The **switch** statement to easy to understand when there are more than 3 alternatives. The **switch** statement can be represented diagrammatically as shown below:



As seen in the above diagram, the **switch** statement switches between the blocks based on the value of the expression. Each block will have a value associated with it. The expression in the **switch** statement must always reduce to an integer value. So the expression in the **switch** statement can be either an integer value or a character constant or an expression which reduces to an integer value. The label for each block can be either an integer value or a character constant.

Syntax for the **switch** statement is as shown below:

```
switch(expression)
{
        case label1:
                Stmt(s);
                break;
        case label2:
                Stmt(s);
                break;
        case label3:
                Stmt(s);
                break;
        …
        case labelN:
                Stmt(s);
                break;
        default:
                Stmt(s);
                break;
}
```

As seen in the above syntax, each block is represented using the **case** keyword and the **case** keyword follows with the label of the block. In a **switch** statement, both the **default** block and the **break** statement are optional. If none of the blocks are matched, then the statements in the **default** block are executed. Every block is ended with a **break** statement. If we remove the **break** statement from a particular block, all the subsequent blocks are also executed until the next **break** statement is encountered.

/*Program to demonstrate switch statement*/

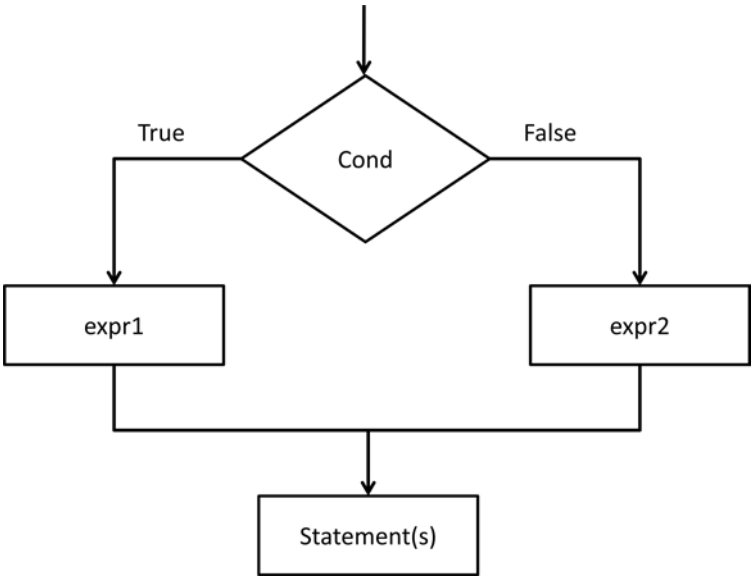
```
#include<stdio.h>
#include<conio.h>
main()
{
   char ch;
   printf("\nEnter a character: ");
   scanf("%c",&ch);
   switch(ch)
   {
      case 'a':
         printf("Entered character is a vowel!");
         break;
      case 'e':
         printf("Entered character is a vowel!");
         break;
      case 'i':
         printf("Entered character is a vowel!");
         break;
      case 'o':
         printf("Entered character is a vowel!");
         break;
      case 'u':
         printf("Entered character is a vowel!");
         break;
      default:
         printf("Entered character is not a vowel!");
         break;
   }
}
```

Conditional Operator Statement

C language provides an unusual operator called conditional operator which is represented as **? :** . The conditional operator is a two way selection operator. The syntax for conditional operator statement is as shown below:

```
(cond/expr) ? expr1 : expr2
```

As shown in the above syntax, when the condition evaluates to true, **expr1** is executed. Otherwise, if the condition is false, **expr2** is executed. The conditional operator statement can be represented diagrammatically as shown below:



Every conditional operator statement can be converted into an **if…else** statement. But the vice versa is not true.

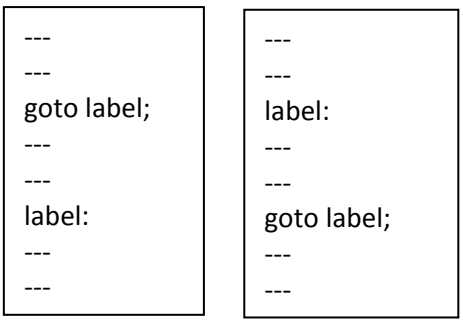
```
/*Program to calculate the value of y by using the following equations
  y = -x, if x <= 0
  y = x+1, if x > 0
*/
#include<stdio.h>
#include<conio.h>
main()
{
   int x,y;
   printf("Enter the value of x: ");
   scanf("%d",&x);
   y = (x<=0) ? (-x) : (x+1);
   printf("y = %d",y);
}
```

goto Statement

Unlike other selection or branching statements that we have seen so far which branches based on a condition, the **goto** statement branches unconditionally. That is why the **goto** statement is also referred to as unconditional jump statement. There are two more unconditional branch statements in C. They are: **break** and **continue.** We have already seen the **break** statement in **switch** statement. But both **break** and **continue** are extensively used inside loops. So, we will discuss about these two unconditional branch statements later. By using the **goto** branch statement, we can either skip some instructions and jump forward in the program or jump back and again repeat a set of instructions. So there are two ways in which we can use the **goto** statement. They are:

1. Forward jump
2. Backward jump.

Syntax of forward jump and backward jump is as shown below:



Forward jump Backward jump

As shown in the above syntax, if the label is after the **goto** statement, then it is known as forward jump and if the label is before the **goto** statement, it is known as backward jump.

/*Program to demonstrate goto Statement*/

```
#include<stdio.h>
#include<conio.h>
main()
{
   int number;
   read:
   printf("\nEnter a number: ");
   scanf("%d",&number);
   if(number>0)
   {
      printf("Number is +ve\n");
   }
   else if(number<0)
   {
      printf("Number is -ve\n");
   }
   else
   {
      goto end;
```

```
    }
    goto read;
    end:
    printf("You entered zero!");
}
```

Decision Making Looping Statements

While writing C programs, the programmer might want a set of instructions to be repeated again and again until some condition is satisfied. For this purpose, C provides decision making looping statements. The looping statements provided by C are:

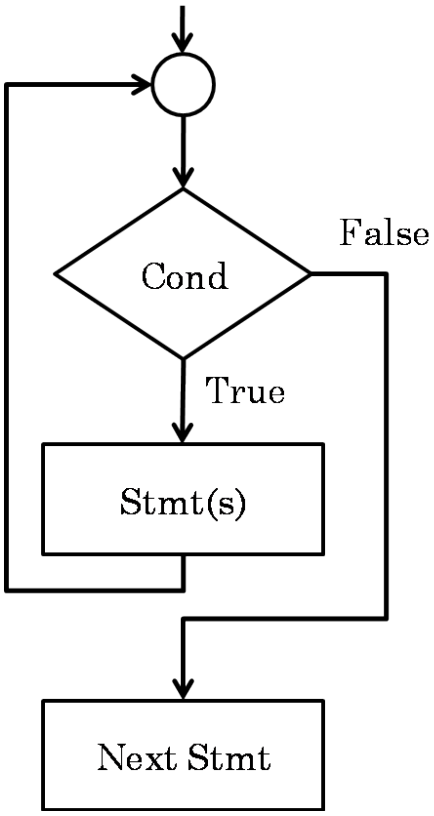
1. while
2. do…while
3. for

All the looping statements in C essentially consist of two parts namely: **control statement** and **body of the loop.** The control statement decides when the loop will be stopped and the body of the loop contains the instructions that are to be repeated. Based on where the control statement is placed in the loop, the looping statements are categorized to two categories. They are:

1. Entry controlled loops
2. Exit controlled loops

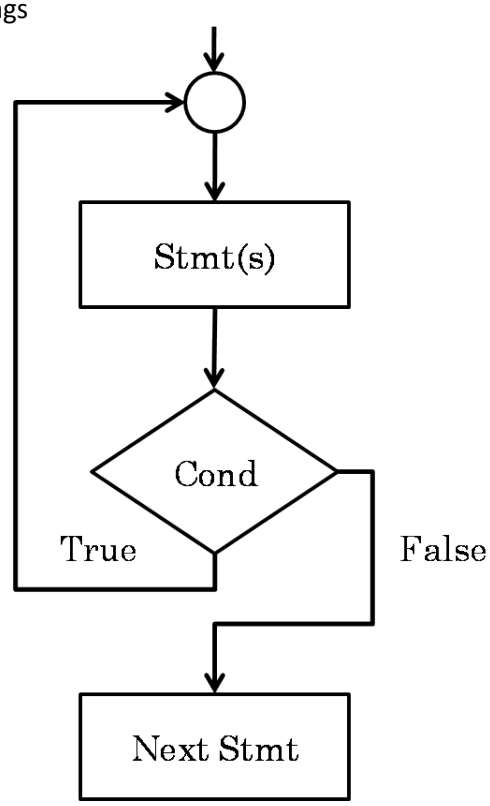
Entry controlled loops

The looping statements in which the control statement is placed before the body of the loop are known as entry controlled loops. Ex: while and for loops. In entry controlled loops, the condition is checked first and if the value is true, then the body of the loop is executed. Otherwise the body is never executed. An entry controlled loop can be represented diagrammatically as shown below:



Exit controlled loops

The looping statements in which the control statement is placed after the body of the loop are known as exit controlled loops. Ex: do…while loop. In exit controlled loops, the body of the loop is executed once and then the condition is checked. If the value is true, the body of the loop is executed again. Otherwise, the execution of the loop stops. An exit controlled loop is represented diagrammatically as shown below:



Based on the nature of the loops, the loops can be categorized into two types namely:

1. Definite loops (Ex: for)
2. Indefinite loops (Ex: while and do…while)

Definite loops: If the programmer exactly knows how many times he/she is going to repeat the set of instructions (loop), such loops are known as definite loops.

Indefinite loops: If the programmer does not know exactly how many times he/she is going to repeat the set of instructions (loop), such loops are known as indefinite loops.

The variable used in the condition inside of a definite loop is known as a counter and such loops are also known as **counter controlled loops.** The variable used in the condition inside of a indefinite loop is known as sentinel and such loops are also known as **sentinel loops.**

while Loop

In C, the **while** loop is an entry controlled loop. The body of the **while** loops is only executed when the condition evaluates to true. If the condition evaluates to false, the body of the loop is not executed. The **while** loops are generally used when there is a need for repeating a set of instructions for indefinite amount of times. The syntax of a **while** loop is as shown below:

```
while(cond/expr)
{
        -----
        -----
        -----
}
```

do…while Loop

In C, the **do…while** loop is an exit controlled loop. The body of the **do…while** loop is executed first and then the condition is evaluated. If the value is true, the body of the loop is executed again and if the value is false, the execution of the body of the loop stops. The difference between **while** and **do…while** is, unlike the **while** loop, the body of the **do…while** loop is guaranteed to be executed atleast once (>=1). The syntax of the **do…while** loop is as shown below:

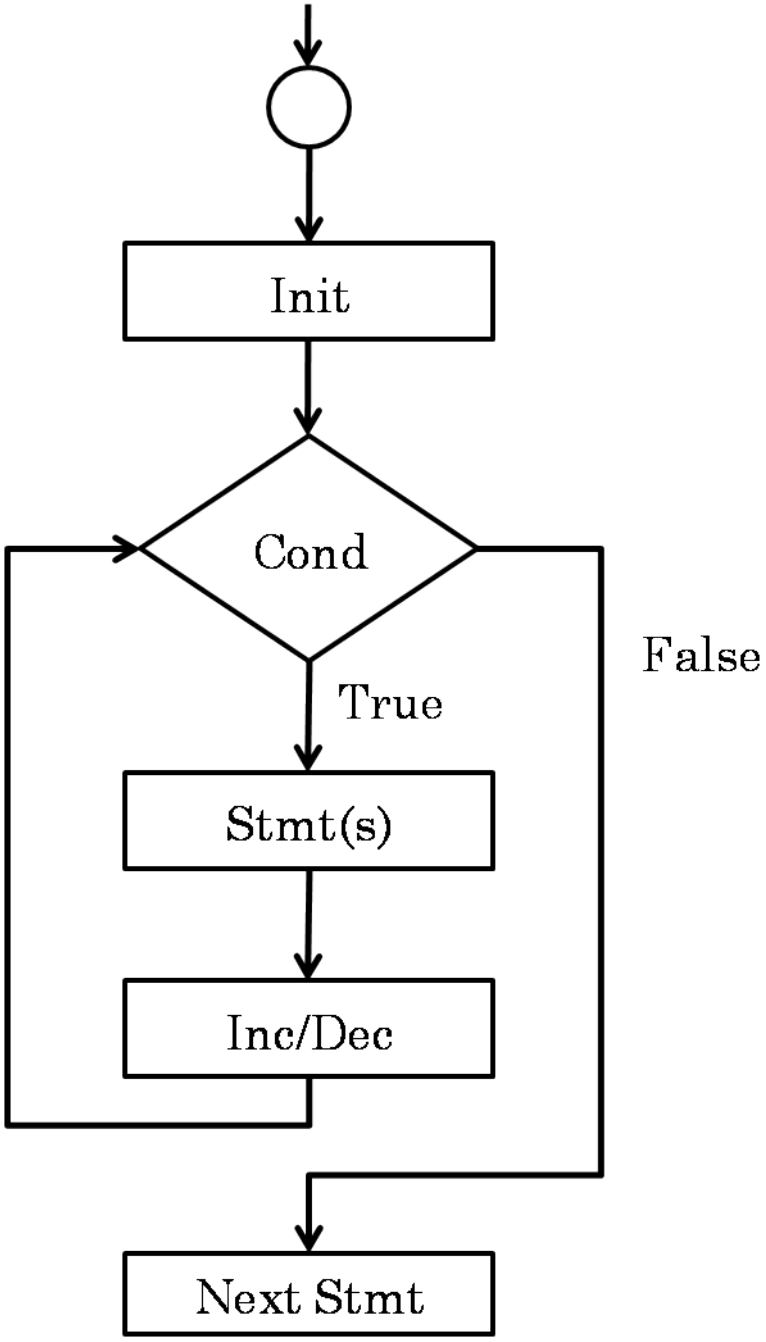
```
do
{
        ----
        ----
        ----
}while(cond/expr);
```

for Loop

In C, the **for** loop is an entry controlled loop. The **for** is generally used while implementing definite loops in C programs. The **for** loop's syntax is a little bit different from the other loops. In the syntax of the **for** loop, first the counter is initialized, and then the condition is evaluated. If the value of the condition is true, the body of the **for** loop is executed. Otherwise, the body of the loop is not executed. After the execution of the **for** loop's body, the counter is either incremented or decremented. Then the condition is evaluated again and so on. The syntax of the **for** loop is as shown below:

```
for(init; cond; incr/decr)
{
        ----
        ----
        ----
}
```

The flowchart of the **for** loop is as shown below:


Init
Cond
False
True
Stmt(s)
Inc/Dec
Next Stmt

```
/*Program to demonstrate while loop*/
/*Program which prompts the user to enter a number and performs
the
 sum of those numbers. The program terminates when the user enters
 -1 and prints the sum*/
#include<stdio.h>
#include<conio.h>
main()
{
   int number = 0, sum = 0;
   clrscr();
   while(number!=-1)
   {
      printf("Enter a number: ");
      scanf("%d",&number);
      if(number!=-1)
      {
         sum = sum + number;
      }
   }
   printf("Sum is: %d",+sum);
   getch();
}
```

```
/*Program to demonstrate do...while loop*/
/*Program which prompts the user to enter a number and performs
the
 sum of those numbers. The program terminates when the user enters
 -1 and prints the sum*/
#include<stdio.h>
#include<conio.h>
main()
{
   int number = 0, sum = 0;
   clrscr();
   do
   {
      printf("Enter a number: ");
      scanf("%d",&number);
      if(number!=-1)
      {
         sum = sum + number;
      }
   }while(number!=-1);
   printf("Sum is: %d",+sum);
   getch();
}
```

```
/*Program to demonstrate for loop*/
/*Program to print the sum of the numbers from 1 to 5*/
#include<stdio.h>
#include<conio.h>
main()
{
   int i, sum = 0;
   clrscr();
   for(i = 1; i <= 5; i++)
   {
      sum += i;
   }
   printf("Sum is: %d",+sum);
   getch();
}
```

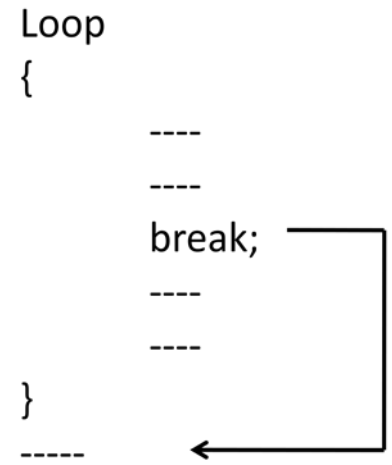
break and continue

C provides two unconditional branching statements which are extensively used inside the looping statements. They are:

1. break
2. continue

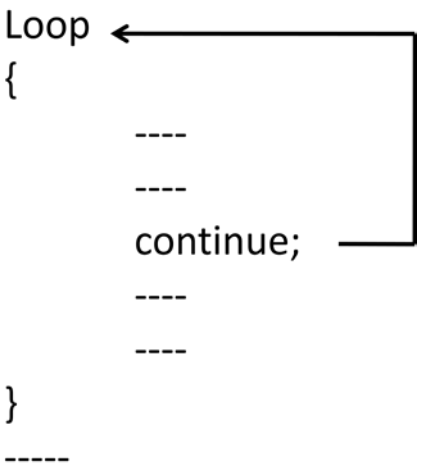
break Statement

The **break** statement is used inside the looping statements to break the execution of the loop. When the **break** statement is encountered inside the loop, the execution of the body of the loop stops and the control is given to the next instruction after the body of the loop. The syntax of the **break** statement is as shown below:



continue Statement

The **continue** statement is used inside the looping statements to skip the execution of a set of instructions and return the control back to the loop. When a **continue** statement is encountered within the body of the loop, the statements after the **continue** statement are skipped and the control is passed back to the loop. The syntax of the **continue** statement is as shown below:



```
/*Program to demonstrate break inside a loop*/
/*Program which asks the number to enter a number repeatedly
  and prints out the entered number. Program must terminate
  when the user enters 0 or negative number*/
#include<stdio.h>
#include<conio.h>
main()
{
   int number;
   clrscr();
   while(1)
   {
      printf("Enter a number: ");
      scanf("%d",&number);
      if(number<=0)
      {
         printf("Program Terminated!");
         break;
      }
      else
      {
         printf("You enetered: %d\n",number);
      }
   }
   getch();
}
```

```
/*Program to demonstrate continue inside a loop*/
/*Program which prints the odd numbers between 1 and 100*/
#include<stdio.h>
#include<conio.h>
main()
{
   int i;
   clrscr();
   for(i = 1; i <= 100; i++)
   {
      if(i%2==0)
      {
         continue;
      }
      else
      {
         printf("%d ",i);
      }
   }
   getch();
}
```

```
/*Program to calculate the sum of digits in the given integer*/
#include<stdio.h>
#include<conio.h>
main()
{
   int number, sum=0;
   clrscr();
   printf("Enter a number: ");
   scanf("%d",&number);
   while(number>0)
   {
      sum = sum + (number % 10);
      number = number / 10;
   }
   printf("Sum of the digits in the given number is: %d",sum);
   getch();
}
```

```
/*Program to reverse the given integer*/
#include<stdio.h>
#include<conio.h>
main()
{
   int number, sum = 0;
   clrscr();
   printf("Enter a number: ");
   scanf("%d",&number);
   printf("\nReverse of the given number is: ");
   while(number>0)
   {
      sum = sum*10 + (number % 10);
      number = number / 10;
   }
   printf("%d",sum);
   getch();
}
```

```
/*Program to count the number of digits in the given integer*/
#include<stdio.h>
#include<conio.h>
main()
{
   int number, count=0;
   clrscr();
   printf("Enter a number: ");
   scanf("%d",&number);
   printf("\nNumber of digits is: ");
   while(number>0)
   {
      count++;
      number = number / 10;
   }
   printf("%d",count);
   getch();
}
```

```
/*Program to accept an arithmetic operator and perform the
corresponding operation.
  Program should prompt the user to proceed with the execution of the
program
  again or not. Using do while*/

#include<stdio.h>
#include<conio.h>
main()
{
   int n1,n2;
   char op,option;
   clrscr();
   n1 = 3, n2 = 2;
   do
   {
      printf("Enter the operator: ");
      scanf("%c",&op);
      fflush(stdin);
      switch(op)
      {
         case '+':
            printf("Sum of n1 and n2 is: %d",(n1+n2));
            break;
         case '-':
            printf("Subtracting n2 from n1 gives: %d",(n1-n2));
            break;
         case '*':
            printf("Multiplication of n1 and n2 gives: %d",(n1*n2));
            break;
         case '/':
            printf("Dividing n1 by n2 gives: %d",(n1/n2));
```

```
            break;
         case '%':
            printf("n1 mod n2 gives: %d",(n1%n2));
            break;
          default:
             printf("Invalid Operator!");
             break;
      }
      printf("\n\nDo you want to proceed (y/n)?: ");
      scanf("%c",&option);
      fflush(stdin);
   }while(option=='y');
   printf("Program Terminated!");
   getch();
}
```

```
/* C program to print the smallest number in a set of numbers */
#include<stdio.h>
#include<conio.h>
main()
{
   int numbers[10], small, i;
   clrscr();
   for(i = 0; i < 10; i++)
   {
      printf("Enter the value %d: ",(i+1));
      scanf("%d",&numbers[i]);
   }
   small = numbers[0];
   for(i = 1; i < 10; i++)
   {
      if(small > numbers[i])
      {
         small = numbers[i];
      }
   }
   printf("\n\nSmallest number is: %d",small);
   getch();
}
```

```
/* C program to print the largest number in a set of numbers */
#include<stdio.h>
#include<conio.h>
main()
{
   int numbers[10], large, i;
   clrscr();
   for(i = 0; i < 10; i++)
   {
      printf("Enter the value %d: ",(i+1));
      scanf("%d",&numbers[i]);
   }
   large = numbers[0];
   for(i = 1; i < 10; i++)
   {
      if(large < numbers[i])
      {
         large = numbers[i];
      }
   }
   printf("\n\nLargest number is: %d",large);
   getch();
}
```

```
/* C program to print the alphabets along with ascii values */
#include<stdio.h>
#include<conio.h>
main()
{
   int i;
   clrscr();
   for(i = 65; i <= 122; i++)
   {
      if(i > 90 && i < 97)
      {
         continue;
      }
      printf("%c - %d\t",i,i);
   }
   getch();
}
```

Arrays

In all the programs we have done until now, to store and operate on values we have used variables. But at a point in time, a variable can hold only a single value. For example, in the following syntax: **int a = 10;** we are able to store only **10** in the variable **a,** which is a single value. It is normal in programming to work with a list of values or a group of values at once. For such purposes, variables cannot be used. So, C language provides the construct **array** for holding multiple values at once.

Definition: "*An array is a sequential collection/group of homogeneous (same type) elements which are referred by the same name".* The **type** refers to the data types like int, char, float etc. All the elements/values in the array are of the same type (data type). We cannot store values of different data types in the same array.

Uses:

1. To maintain a list of values.
2. To maintain a table of values.

One Dimensional Array

For maintaining a list of items in C, we can declare an array with a single subscript like a_i . Such arrays with only a single subscript are known as one dimensional arrays. Common uses of one dimensional arrays are: to maintain a list of numbers, to maintain the list of marks, to maintain a list of student names etc.

Declaration of one dimensional array

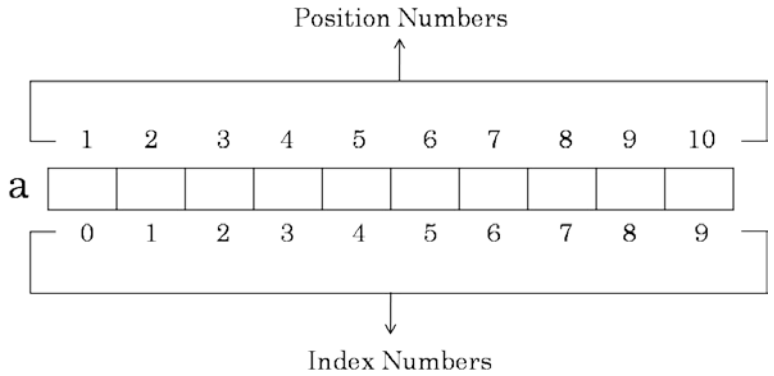
For using arrays in C programs, just like we are declaring variables before using them, we should declare arrays before using them. The syntax for declaring a one dimensional array is as shown below:

```
type  arrayname[size];
```

Example:

```
int a[10];
```

In the above example, **int** is the data type, **a** is the array name and **size** is the number of elements that we can store in the array. So, in the above example **a** is an integer array, which can hold 10 integer values. All the 10 elements in the array will be stored sequentially one after another inside the main memory (RAM). The one dimensional array declared above will be maintained in the memory as shown below:



Initialization of one dimensional array

Initialization means assigning values. To assign values to the elements in the array, we use the following syntax:

```
type arrayname[index] = value;
```

Example:

```
int a[5];

a[0] = 10;
```

In the above syntax, **index** refers to the element in the array. The **index** of an array always starts with zero. So, the index values for the array **a** in the above example are: 0, 1, 2, 3, 4. If we want to access **nth** element in the array, the **index** value will be **n-1**. For example, if we want to refer **first** element in the array, the **index** value will be 1-1=0. In, the above example, we are assigning value **10** to the first element of the array.

Note: If the array elements are not initialized, the values stored in the elements of the array will be garbage values.

Generally, there are two types of initializing an array. They are: 1) Static Initialization (at compile time) and 2) Dynamic Initialization (at run time). In static initialization, the elements of an array are assigned values when the program is compiled. In dynamic initialization, the elements of an array are assigned values when the program is executed (runtime). The above way of initialization is static initialization. We can also perform static initialization in other ways as shown below:

```
type arrayname[size] = {value1, value2, …. , valueN};

Or

type arrayname[ ] = {value1, value2, …. , valueN};
```

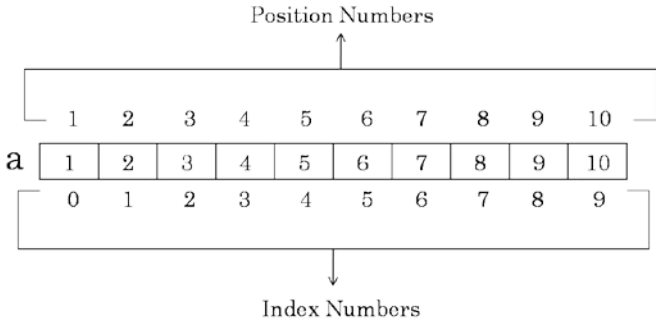
Example:

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

or

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

In the above example, in the first line, the size is specified as **10**. But, in the second line, the size was not specified. In such cases, the size is automatically calculated. In this example, the size is automatically computed as **10** by the compiler. The array will be represented in the memory as shown below:



In dynamic initialization, the values are assigned to the elements of the array during the execution of the program. Let's see the following piece of code:

```
int a[10], i;
for(i = 0; i < 10; i++)
{
        scanf("%d",&a[i]);
}
```

In the above example, the user will be storing the values into the array while executing the program. Until then, garbage values will be stored in the array.

Note: If we use braces i.e., { and }, for initializing the array, the default values for all the elements in the array will be zero.

Programs:

```
/* C program to declare an integer array and initialize the elements in the array
*/
#include<stdio.h>
#include<conio.h>
main()
{
   int a[5];
   clrscr();
   a[0] = 1;
   a[1] = 2;
   a[2] = 3;
   a[3] = 4;
   a[4] = 5;
   printf("a[0]: %d\n",a[0]);
   printf("a[1]: %d\n",a[1]);
   printf("a[2]: %d\n",a[2]);
   printf("a[3]: %d\n",a[3]);
   printf("a[4]: %d",a[4]);
   getch();
}
/* C program to declare an integer array and initialize the elements in the array
*/
#include<stdio.h>
#include<conio.h>
main()
{
   int a[5] = {1,2,3,4,5};
```

```
   clrscr();
   printf("a[0]: %d\n",a[0]);
   printf("a[1]: %d\n",a[1]);
   printf("a[2]: %d\n",a[2]);
   printf("a[3]: %d\n",a[3]);
   printf("a[4]: %d",a[4]);
   getch();
}
```

```
/* C program to declare an integer array and initialize the elements in the array
*/
#include<stdio.h>
#include<conio.h>
main()
{
   int a[ ] = {5,2,1,9,10};
   clrscr();
   printf("a[0]: %d\n",a[0]);
   printf("a[1]: %d\n",a[1]);
   printf("a[2]: %d\n",a[2]);
   printf("a[3]: %d\n",a[3]);
   printf("a[4]: %d",a[4]);
   getch();
}
```

```
/* C program to print the capacity of an array */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[] = {5,2,1,9,10};
   int size;
   clrscr();
   size = sizeof(a);
   printf("Size of the array is: %d bytes",size);
   getch();
}
```

```
/* C program to print no of elements in an array */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[] = {5,2,1,9,10};
   int count;
   clrscr();
   count = sizeof(a)/sizeof(int);
   printf("Number of elements in the array is: %d",count);
   getch();
}
```

```
/* C program to initialize array elements at runtime */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[10], i;
   clrscr();
   for(i = 0; i < 10; i++)
   {
      printf("Enter value %d: ",(i+1));
      scanf("%d",&a[i]);
   }
   printf("The elements in the array are: \n");
   for(i = 0; i < 10; i++)
   {
      printf("a[%d]: %d\n",i,a[i]);
   }
   getch();
}
```

```
/* C program to search for a number n in a group of 10 numbers */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[10], i, n;
   /*clrscr();*/
   for(i = 0; i < 10; i++)
   {
      printf("Enter value %d: ",(i+1));
      scanf("%d",&a[i]);
   }
   printf("Enter the value you want to search: ");
   scanf("%d",&n);
   for(i = 0; i < 10; i++)
   {
      if(n==a[i])
      {
         printf("Number found at index %d",i);
      }
   }
   getch();
}
```

```
/* C program to calculate the average of 10 numbers */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[10], i;
   float sum, avg;
   /*clrscr();*/
   for(i = 0; i < 10; i++)
   {
      printf("Enter value %d: ",(i+1));
      scanf("%d",&a[i]);
   }
   for(i = 0; i < 10; i++)
   {
      sum += a[i];
   }
   avg = sum / 10;
   printf("Average of 10 numbers is: %f", sum);
   getch();
}
```

```
/* C program to print the multiplication table of number n */
#include<stdio.h>
#include<conio.h>
main()
{
   int i, n;
   /*clrscr();*/
   printf("Enter the value of n: ");
   scanf("%d",&n);
   for(i = 1; i <= 10; i++)
   {
      printf("%d\t*\t%d\t=\t%d\n”, n, i, (n*i));
   }
   getch();
}
```

```
/* C program to print the elements in a array in reverse order */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[10], i;
   clrscr();
   for(i = 0; i < 10; i++)
   {
      printf("Enter the value %d: ",(i+1));
      scanf("%d",&a[i]);
   }
   printf("Elements in reverse order: ");
   for(i = 9; i >= 0; i--)
   {
      printf("%d ",a[i]);
   }
   getch();
}
```

Two Dimensional Array

An array that has two subscripts, for example: a_{ij} is known as a two dimensional. A two dimensional array is used to store a table of values which has rows and columns. Some of the uses/applications of the two dimensional arrays are: to maintain the marks of students, to maintain prices of items etc…

Declaration of a two dimensional array

Like a one dimensional array, the two dimensional array must also be declared before using it in the program. The syntax for declaring a two dimensional array is shown below:

```
type arrayname[rows][columns];
```

Example:

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

In the above example, **int** refers to the data type, **a** refers to the array name, first set of square brackets represents the number of rows and the second set of square brackets represents the number of columns. So, our two dimensional array contains 9 elements in total.

Initialization of a two dimensional array

A two dimensional array can be initialized in different ways either statically at compile time or dynamically at run time. The syntax for initializing a two dimensional array statically is as shown below:

```
arrayname[rowindex][columnindex] = value;

(or)

type arrayname[rows][columns] = {val1,val2,….,valn};
```

Example:

```
a[0][0] = 10;

(or)

int a[3][3] = {1,1,1,2,2,2,3,3,3};
```

The memory representation of the two dimensional array will be as shown below:

	0	1	2
0	1 [0,0]	1 [0,1]	1 [0,2]
1	2 [1,0]	2 [1,1]	2 [1,2]
2	3 [2,0]	3 [2,1]	3 [2,2]

The values will be assigned starting with the first element in the first row and so on. If insufficient values are provided, then the remaining elements will be initialized to zero. There is another way of initializing a two dimensional array in which we can specify the values for each row separately. We can see the example below:

```
int a[3][3] = {{1,1,1},{2,2,2},{3,3,3}};
```

Programs:

```
/* C program to declare a two dimensional array and initialize it */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[2][3];
   int i,j;
   clrscr();
   a[0][0] = 1;
   a[0][1] = 2;
   a[0][2] = 3;
   a[1][0] = 4;
   a[1][1] = 5;
   a[1][2] = 6;
   for(i = 0; i < 2; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("%d ",a[i][j]);
      }
      printf("\n");
   }
   getch();
}
```

```
/* C program to declare a two dimensional array and initialize it */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[2][3] = {1,2,3,4,5,6};
   /* int a[2][3] = {{1,2,3},{4,5,6}}; */
   int i,j;
   clrscr();
   for(i = 0; i < 2; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("%d ",a[i][j]);
      }
      printf("\n");
   }
   getch();
}
```

```
/* C program to declare a two dimensional array and initialize it dynamically */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[3][3];
   int i,j;
   /*clrscr();*/
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("Enter a[%d][%d]: ",i,j);
         scanf("%d",&a[i][j]);
      }
   }
   printf("The elements in the array are: \n");
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("%d ",a[i][j]);
      }
      printf("\n");
   }
   getch();
}
```

```
/* C program to perform matrix addition */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[3][3], b[3][3], c[3][3];
   int i,j;
   /*clrscr();*/
   printf("Enter the elements of matrix a: \n");
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("Enter a[%d][%d]: ",i,j);
         scanf("%d",&a[i][j]);
      }
   }
   printf("Enter the elements of matrix b: \n");
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("Enter b[%d][%d]: ",i,j);
         scanf("%d",&b[i][j]);
      }
   }
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         c[i][j] = a[i][j] + b[i][j];
      }
   }
   printf("The elements in the array are: \n");
   for(i = 0; i < 3; i++)
   {
```

```
        for(j = 0; j < 3; j++)
        {
            printf("%d ",c[i][j]);
        }
        printf("\n");
    }
    getch();
}
```

```
/* C program to multiply two matrices */
#include<stdio.h>
#include<conio.h>
main()
{
   int a[3][3], b[3][3], c[3][3] = {};
   int i, j, k;
   /*clrscr();*/
   printf("Enter the elements of matrix a: \n");
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("Enter a[%d][%d]: ",i,j);
         scanf("%d",&a[i][j]);
      }
   }
   printf("Enter the elements of matrix b: \n");
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         printf("Enter b[%d][%d]: ",i,j);
         scanf("%d",&b[i][j]);
      }
   }
   for(i = 0; i < 3; i++)
   {
      for(j = 0; j < 3; j++)
      {
         for(k = 0; k < 3; k++)
         {
            c[i][j] += a[i][k] * b[k][j];
         }
      }
   }
```

```
    printf("The elements in the array c are: \n");
    for(i = 0; i < 3; i++)
    {
        for(j = 0; j < 3; j++)
        {
            printf("%d\t",c[i][j]);
        }
        printf("\n");
    }
}
```

Strings

A string is a collection of characters which is treated as a single data item. A group of characters enclosed in double quotes is known as a string constant. Some of the examples of string constants are:

1. "hai"
2. "hello world"
3. "My name is suresh"

In C programming, there is no predefined data type to declare and use strings. So, we use character arrays to declare strings in C programs. The common operations that can be performed on a string are:

- Reading and writing strings
- Combining strings together
- Copying one string to another
- Comparing strings for equality
- Extracting a portion of the string (substring)

Declaring and Initializing Strings

Like variables, we should declare strings before using them in the program. Since we already known that strings are implemented as character arrays, the syntax for declaring a string is as shown below:

```
char stringname[size];
```

The **size** refers to the number of characters in the string. When the compiler assigns a character string to a character array, it appends a **'\0'** to the end of the array. So, the **size** of the character array should always be number of characters plus 1.

Like numeric arrays and variables, character arrays can also be initialized when they are declared. Some of the examples for initializing the string are as shown below:

```
char str[10] = {'N','E','W',' ','D','E','L','H','I','\0'};

or

char str[10] = "NEW DELHI";
```

If less number of characters are provided than the size of the string, the rest of the characters are initialized to **'\0'.** If we try to assign more characters then the size of the string, compiler gives an error.

Note: *The string termination character **'\0'**, is used to terminate a string. In C, there is no data type provided available. We maintain strings using character arrays. A string is a variable length structure stored inside a fixed size array. The size of the array is often larger than the number of characters in the string. So, the compiler must have some means to detect the end of the string. For this purpose we use **'\0'**.*

Note: *If '\0' is not provided then the compiler will treat the array as a normal character array. Only when we provide the '\0' character, compiler treats it as a string.*

Programs:

```
/*C program to declare a string */
#include<stdio.h>
#include<conio.h>
main()
{
   char str[6];
   clrscr();
   str[0] = 'h';
   str[1] = 'e';
   str[2] = 'l';
   str[3] = 'l';
   str[4] = 'o';
   str[5] = '\0';
   printf("%s",str);
   getch();
}
```

```
/*C program to declare a string */
#include<stdio.h>
#include<conio.h>
main()
{
   char str1[6] = {'h','e','l','l','o','\0'};
   char str2[] = {'h','e','l','l','o','\0'};
   char str3[] = "hello world";
   clrscr();
   printf("%s",str1);
   printf("\n%s",str2);
   printf("\n%s",str3);
   getch();
}
```

```
/*C program to print a string */
#include<stdio.h>
#include<conio.h>
main()
{
   char str1[6] = {'h','e','l','l','o','\0'};
   int i;
   clrscr();
   printf("%s\n",str1);
   printf("\nPrinting the string using for loop...\n");
   for(i = 0; i < 5; i++)
   {
      printf("%c",str1[i]);
   }
   getch();
}
```

```
/*C program to calculate the length of a string */
#include<stdio.h>
#include<conio.h>
main()
{
   char str[6] = {'h','e','l','l','o','\0'};
   int length;
   clrscr();
   length = sizeof(str) - 1;
   printf("Length of the string is: %d",length);
   getch();
}
```

```
/*C program to calculate the length of a string */
#include<stdio.h>
#include<conio.h>
main()
{
   char str[6] = {'h','e','l','l','o','\0'};
   int length, i;
   clrscr();
   i = 0;
   while(str[i] != '\0')
   {
      i++;
   }
   length = i;
   printf("Length of the string is: %d",length);
   getch();
}
```

```
/*C program to search for a character in a string */
#include<stdio.h>
#include<conio.h>
main()
{
   char str[6] = {'h','e','l','l','o','\0'}, ch;
   int i;
   clrscr();
   printf("Enter a character to search: ");
   scanf("%c",&ch);
   for(i = 0; i < 5; i++)
   {
      if(str[i] == ch)
      {
         printf("Character found at position %d",(i+1));
         break;
      }
   }
   getch();
}
```

Reading Strings

Using scanf function

Strings can be read from the terminal by using the familiar **scanf** function. The format specifier to read strings is **%s**. An example of reading a string using **scanf** function is shown below:

```
char str[10];

scanf("%s", str);
```

There is a downside of using **scanf** function for reading strings. The downside is, **scanf** cannot read strings with white spaces. For example, if we provide "New Delhi" as the string, **scanf** will read only "New" and the rest of the characters are neglected or not processed. So, to read "New Delhi", we have to declare two character arrays and read "New" and "Delhi" separately.

Reading and printing a line of text

Reading Text (getchar and gets)

As we have learned that **scanf** function cannot be used for reading a line of text, we have to search for other alternatives for reading a line of text which has white spaces embedded in it. One alternative is to read the line of text character by character until the enter key (\n) is pressed. Here we will use a predefined function called **getchar( )** which reads a single character from the terminal.

```
/* C program to read a line of text using getchar function */
#include<stdio.h>
#include<conio.h>
main()
{
   char line[20], ch;
   int i;
   clrscr();
   printf("Enter the line of text: ");
   i = 0;
   do
   {
      ch = getchar();
      line[i] = ch;
      i++;

   }while(ch != '\n');
   line[i] = '\0';
   printf("%s",line);
   getch();
}
```

In the above program we are reading characters until a new line character (\n) is encountered. So, by using the above approach we can read a line of text. There is a more efficient way for reading a line of text with white spaces. We can use the **gets** function which is available in the **stdio.h** header file. The purpose of **gets** function is to read a line of text from the terminal/keyboard. The usage of **gets** function is as shown below:

```
gets(arrayname);
```

The arrayname in the above piece of code is the array in which we are going to store the string. By using the **gets** function the above program can be rewritten as shown below:

```
/* C program to read a line of text using gets function */
#include<stdio.h>
#include<conio.h>
main()
{
   char line[20], ch;
   int i;
   clrscr();
   printf("Enter the line of text: ");
   gets(line);
   printf("%s",line);
   getch();
}
```

Printing Text (putchar and puts)

We can print strings in multiple ways. First is by using **printf** function. We can use the format specifier **%s** to print a string. Let us consider the following example:

```
char str[4] = {'h','a','i','\0'};

printf("%s", str);
```

The output of the above piece of code is "hai". The second way is to print the string character by character. For this purpose we can use the format specifier **%c** for printing each character or we can use the predefined function **putchar**. The usage of this function is as shown below:

```
putchar(ch);
```

In the above code, **ch** is the character that we want to print on the screen. By using the **putchar** function we can print a line of text or string as shown below:

```
/* C program to print a string using putchar function */
#include<stdio.h>
#include<conio.h>
main()
{
   char line[20];
   int i;
   clrscr();
   printf("Enter the line of text: ");
   gets(line);
   i = 0;
   while(line[i] != '\0')
   {
      putchar(line[i]);
      i++;
   }
```

```
   getch();
}
```

The last alternative for printing a string or a line of text is by using the predefined function pu**ts**. This function is available in the **stdio.h** header file. By using the **puts** function we can rewrite the above program as shown below:

```
/* C program to print a string using puts function */
#include<stdio.h>
#include<conio.h>
main()
{
   char line[20];
   clrscr();
   printf("Enter the line of text: ");
   gets(line);
   puts(line);
   getch();
}
```

String Manipulations

As we have already seen in the previous unit, a string is a collection of characters and strings are maintained as character arrays in C programs. The common operations or manipulations that can be performed on strings are:

1. Reading and writing strings
2. Concatenating/combining/joining strings
3. Comparing strings
4. Copying one string into another string
5. Extracting a portion of the string (substring)

C provides predefined functions for performing all the above operations or manipulations on strings. Most of these predefined functions are available in **string.h** header file. The list of predefined functions is given below:

Functions	Purpose
scanf, gets, getchar	To read a string
printf, puts, putchar	To print a string
strcat	To concatenate two strings
strcmp	To compare two strings
strcpy	To copy one string into another string
strstr	To locate a substring in the string
strlen	To calculate the length of the string
strrev	To reverse the given string

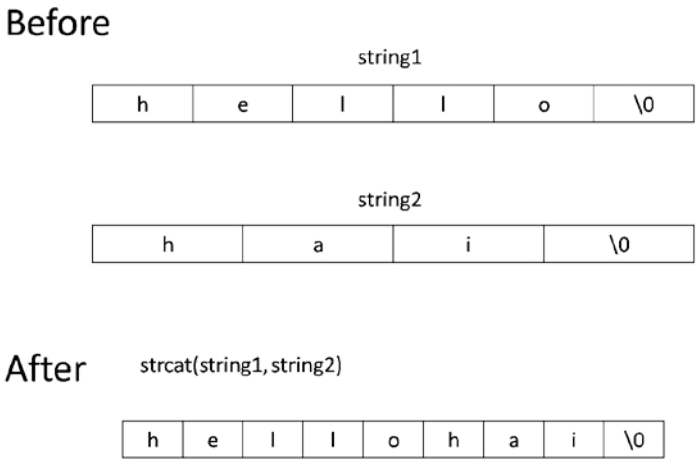
Note: *The difference between **scanf** and **gets** is, **scanf** can read strings which does not contain any white spaces. Whereas **gets** can read strings both without spaces and with spaces.*

strcat( )

The **strcat** predefined function is used to concatenate/join two strings together. The syntax of the **strcat** function is shown below:

```
strcat(string1, string2)
```

The **strcat** function accepts two parameters which are strings. The **string2** parameter can be either a character array or a string constant. The **strcat** function takes the content of **string2** and merges it with the content in **string1** and the final result will be stored in **string1.** Let us see an example:



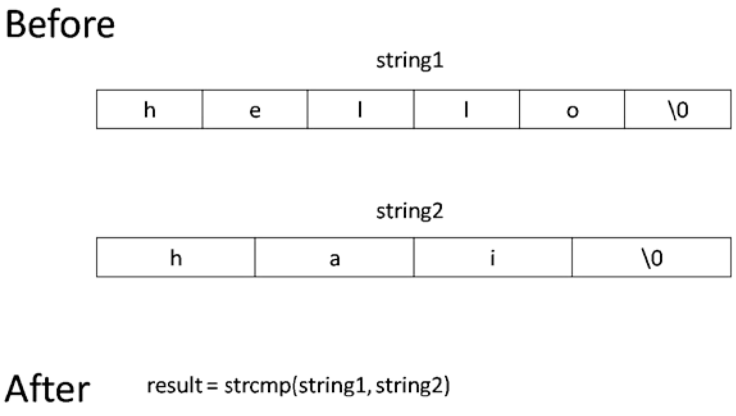
strcmp( )

The **strcmp** predefined function is used to compare two strings. After comparison, if the two strings are equal, then the function returns a **0**. Otherwise if the first string comes before the second string in alphabetical

order, the function returns a **-1.** If the first string comes after the second string in alphabetical order, the function returns a **1** as the return value. The syntax of **strcmp** function is as shown below:

```
strcmp(string1, string2)
```

Let us consider an example:



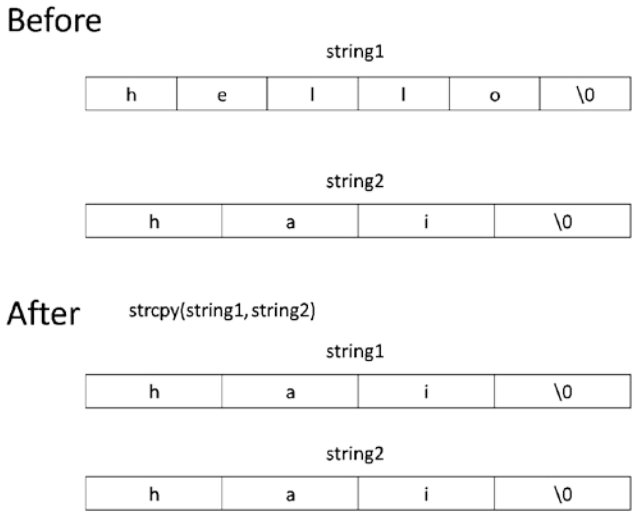
As seen from the above example, the **strcmp** function the first letter in both the strings and since they are equal, now it compares the second letter in both the strings, which are **e** and a. Since, **e** comes after **a** according to dictionary order, the result will be **1** which means, the string **hello** comes after (greater than) the string **hai**.

strcpy( )

The **strcpy** function is used to copy one string into another string. This function can be used for creating a copy of an existing string. The syntax of **strcpy** function is as shown below:

```
strcpy(string1, string2)
```

In the above syntax, the **string2** can be either a string or string constant. The string in **string2** is copied into **string1** and the result will be stored in **string1.** Let us consider an example:

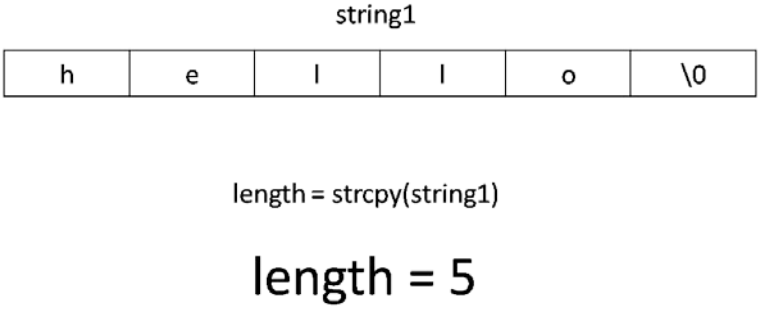


strlen( )

The **strlen** function is used to retrieve the length of a given string. The return type of this function will be an integer. The syntax of **strlen** function is as shown below:

```
strlen(string)
```

The parameter **string** can be either a character array or a string constant. The function returns the length of the **string** which will be the number of characters in the string excluding the '\0' character. Let us consider an example:

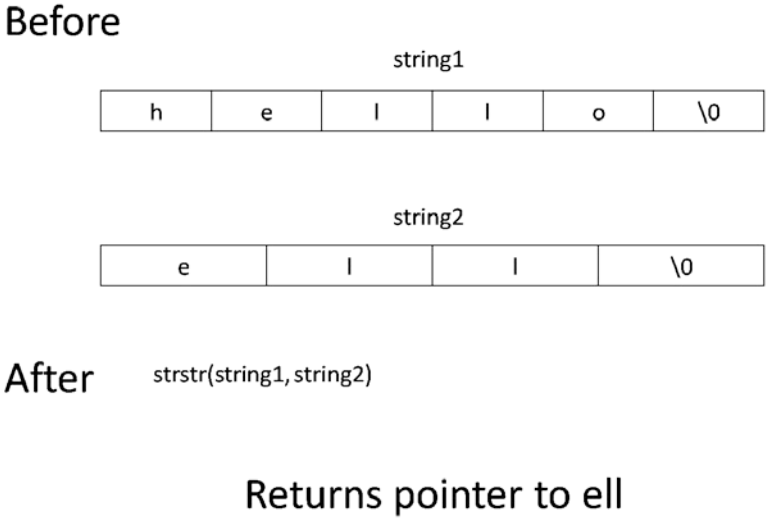


strtstr( )

The **strstr** function returns a character pointer of the first occurrence of the given substring in the string. If the substring is not found in the string, the function **strstr** returns NULL. The syntax for **strstr** function is shown below:

```
strstr(string1, string2)
```

In the above syntax, **strstr** searches for string2 in the string1. If found, it returns a pointer to the first occurrence of the string2 in string1. If not found, it returns NULL. Let us consider an example:

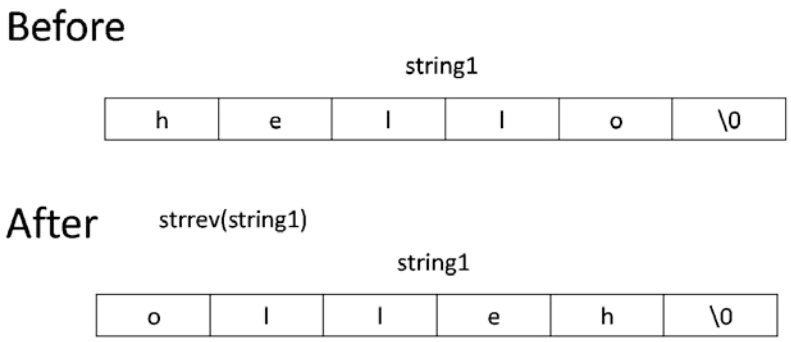


strrev( )

The **strrev** function is used to reverse a given string. We can use this predefined function to check whether a given string is a palindrome or not. The syntax for using the **strrev** function is as shown below:

```
strrev(string)
```

In the above syntax, the **strrev** function reverses the given **string** and returns it back. The content of the string also changes. Let us see an example:



Other predefined functions

Some of the other predefined functions available in the **string.h** header file are shown in the below table:

Function	Purpose	Syntax
strncat	To concatenate n number of left most characters	strncat(str1, str2, n)
strncmp	To compare n number of left most characters	strncmp(str1, str2, n)
strncpy	To copy n number of left most characters	strncpy(str1, str2, n)

Programs:

```
/* C program to concatenate two strings without using any functions*/
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
   char str1[20] = {'h','e','l','l','o','\0'};
   char str2[4] = {'h','a','i','\0'};
   char result[10];
   int i, j;
   clrscr();
   for(i = 0; i < strlen(str1); i++)
   {
      result[i] = str1[i];
   }
   result[i] = ' ';
   for(i = strlen(str1)+1, j = 0; j < strlen(str2); i++,j++)
   {
      result[i] = str2[j];
   }
   result[i] = '\0';
   printf("Concatenated String is: %s",result);
   getch();
}
```

```
/* C program to concatenate two strings using strcat function */
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
   char str1[20] = {'h','e','l','l','o',' ','\0'};
   char str2[6] = {'w','o','r','l','d','\0'};
   clrscr();
   strcat(str1, str2);
   printf("Concatenated string is: %s",str1);
   getch();
}
```

```
/* C program to compare two strings using strcmp function */
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
   char str1[20] = {'h','e','l','l','o',' ','\0'};
   char str2[6] = {'w','o','r','l','d','\0'};
   int result;
   clrscr();
   result = strcmp(str1, str2);
   printf("Result is: %d",result);
   getch();
}
```

```
/* C program to copy one string into another string using strcpy function */
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
   char str1[20] = {'h','e','l','l','o',' ','\0'};
   char str2[6] = {'w','o','r','l','d','\0'};
   clrscr();
   strcpy(str1, str2);
   printf("Copied string is: %s",str1);
   getch();
}
```

```
/* C program to print the length of strings using strlen function */
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
   char str1[] = {'h','e','l','l','o','\0'};
   char str2[] = {'h','a','i','\0'};
   clrscr();
   printf("Length of str1 is: %d\n",strlen(str1));
   printf("Length of str2 is: %d",strlen(str2));
   getch();
}
```

```
/* C program to locate a substring in the given string using strstr function */
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
   char str1[] = {'h','e','l','l','o','\0'};
   char str2[] = "llo";
   int index;
   clrscr();
   if(strstr(str1, str2) != NULL)
   {
      printf("Substring str2 is present in str1");
   }
   else
   {
      printf("Substring str2 is not present in str1");
   }
   getch();
}
```

```
/* C program to reverse a string using strrev function */
#include<stdio.h>
#include<conio.h>
#include<string.h>
main()
{
   char str[6] = "hello";
   clrscr();
   printf("Reverse of the string is: %s",strrev(str));
   getch();
}
```

```
/* C program to check for a string palindrome */
#include<stdio.h>
```

```
#include<conio.h>
#include<string.h>
main()
{
   char org[10], dup[10];
   clrscr();
   printf("Enter a string: ");
   gets(org);
   strcpy(dup, org);
   strrev(org);
   if(strcmp(org, dup))
   {
      printf("Entered string is not a palindrome!");
   }
   else
   {
      printf("Entered string is a palindrome");
   }
   getch();
}
```