Linear Search

Problem: Given an array arr[] of n elements, write a function to search a given element x in arr[].

1. Input : arr[] = {10, 20, 80, 30, 60, 50, 110, 100, 130, 170}

   x = 110;

   Output : 6

   Element x is present at index 6

2. Input : arr[] = {10, 20, 80, 30, 60, 50,110, 100, 130, 170}

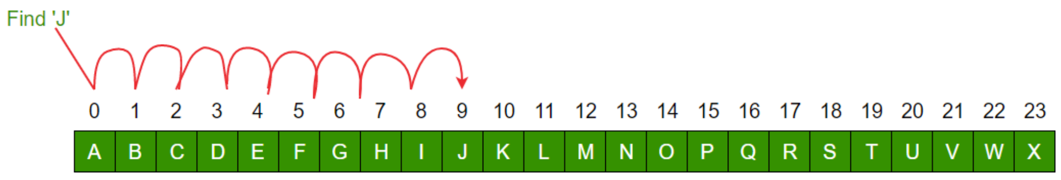
   x = 175;

   Output : -1

   Element x is not present in arr[].

A simple approach is to do linear search, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn't match with any of elements, return -1.



Iterative C Program for Linear Search

```
#include <stdio.h>

int main()

{

int array[100], search, c, n;

printf("Enter the number of elements in array\n");

scanf("%d", &n);

printf("Enter %d integer(s)\n", n);

for (c = 0; c < n; c++)
scanf("%d", &array[c]);
printf("Enter a number to search\n");
scanf("%d", &search);
for (c = 0; c < n; c++)
{
if (array[c] == search)
/* If required element is found */
{
printf("%d is present at location %d.\n", search, c+1);
break;
} }
if (c == n)
printf("%d isn't present in the array.\n", search);
return 0;
}
```

Binary Search

We basically ignore half of the elements just after one comparison.

- Compare x with the middle element.
- If x matches with middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
- Else (x is smaller) recur for the left half.

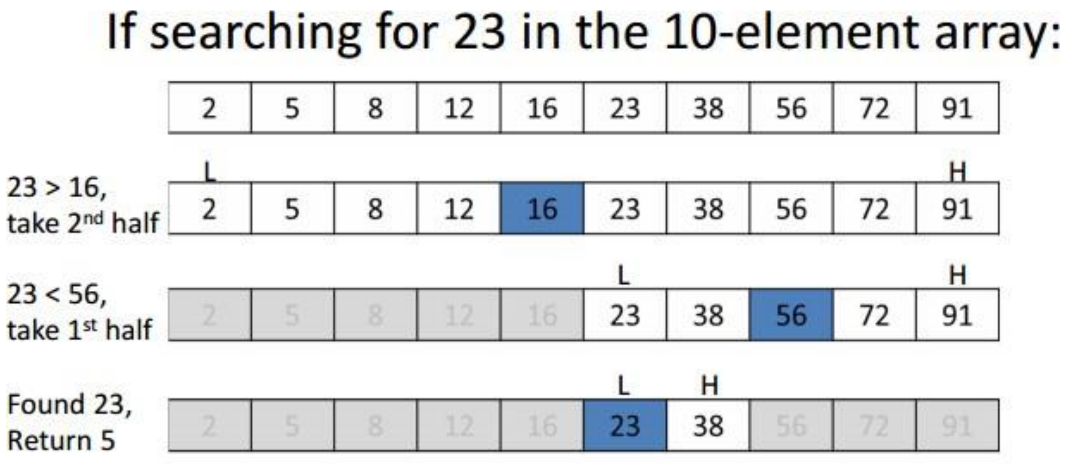
Binary Search- Recursive Method

```
#include <stdio.h>
 // A recursive binary search function. It returns location of x in
// given array arr[l..r] is present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
```

```
{
 if (r >= l)
{
int mid = l + (r - l)/2;
// If the element is present at the middle itself
if (arr[mid] == x)  return mid;
// If element is smaller than mid, then it can only be present
// in left subarray
if (arr[mid] > x) return binarySearch(arr, l, mid-1, x);
// Else the element can only be present in right subarray
return binarySearch(arr, mid+1, r, x);
}
// We reach here when element is not present in array
return -1;
}
int main(void)
{
int arr[] = {2, 3, 4, 10, 40};
int n = sizeof(arr)/ sizeof(arr[0]);
int x = 10;
int result = binarySearch(arr, 0, n-1, x);
if(result == -1)
printf("Element is not present in array")
printf("Element is present at index %d", result);
return 0;
}
```

Binary Search- Iterative Method

```
    #include<stdio.h>
    // A iterative binary search function. It returns location of x ingiven array
    arr[l..r] if present, otherwise -1
    int binarySearch(int arr[],int l, int r, int x)
    {
        while (l <= r)
        {
         int m = l + (r-l)/2;
         // check if x is present at mid
         If(arr[m] == x)
                 return m;
       //if x is greater, ignore left half
         If(arr[m]<x)
                 l =m+1;
       //if x is smaller, ignore right half
       else
                 r =m -1;
       }
       // if we reach here, then elements was not present return -1;
}
       int main(void)
       {
       int arr[] = {2, 3, 4, 10, 40};
       int n = sizeof(arr)/ sizeof(arr[0]); int x = 10;
       int result = binarySearch(arr, 0, n-1, x);
       (result == -1)? printf("Element is not present in array")
       : printf("Element is present at index %d", result);
       return 0;
       }
```



Algorithm for Binary Search

```
int binarySearch(int arr[], int value, int left, int right)
{
        while (left <= right)
        {
                int middle = (left + right) / 2;
                if (arr[middle] == value)
                        return middle;
                else if (arr[middle] > value)
                        right = middle - 1;
                else
                        left = middle + 1;
        }
        return -1;
}
```

SORTING

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Example:

Given series is 5 1 4 2 8

First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) –> (1 4 2 5 8)

(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2

(1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8) Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

C Program for Bubble Sort

```
#include <stdio.h>
void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}
//A function to implement bubble sort
void bubbleSort(int arr[], int n)
{
        int i, j;
        for (i = 0; i < n-1; i++)
        // Last I elements are already in place
            for (j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                   swap(&arr[j], &arr[j+1]);
       }
       // Function to print an array
          Void printArray(int arr[],int size)
         {
            int i;
            for (i=0; i<size; i++)
            printf("%d", arr[i]);
            printf("n");
     }
     //Driver program to test above functions
     int main()
     {
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        int n =sizeof(arr)/sizeof(arr[0]);
        bubbleSort(arr, n);
        printf("Sorted array:");
        printArray(arr, n);
        return 0;
};
```

Output

```
Sorted array:
11 12 22 25 34 64 90
```

Algorithm for bubble sort

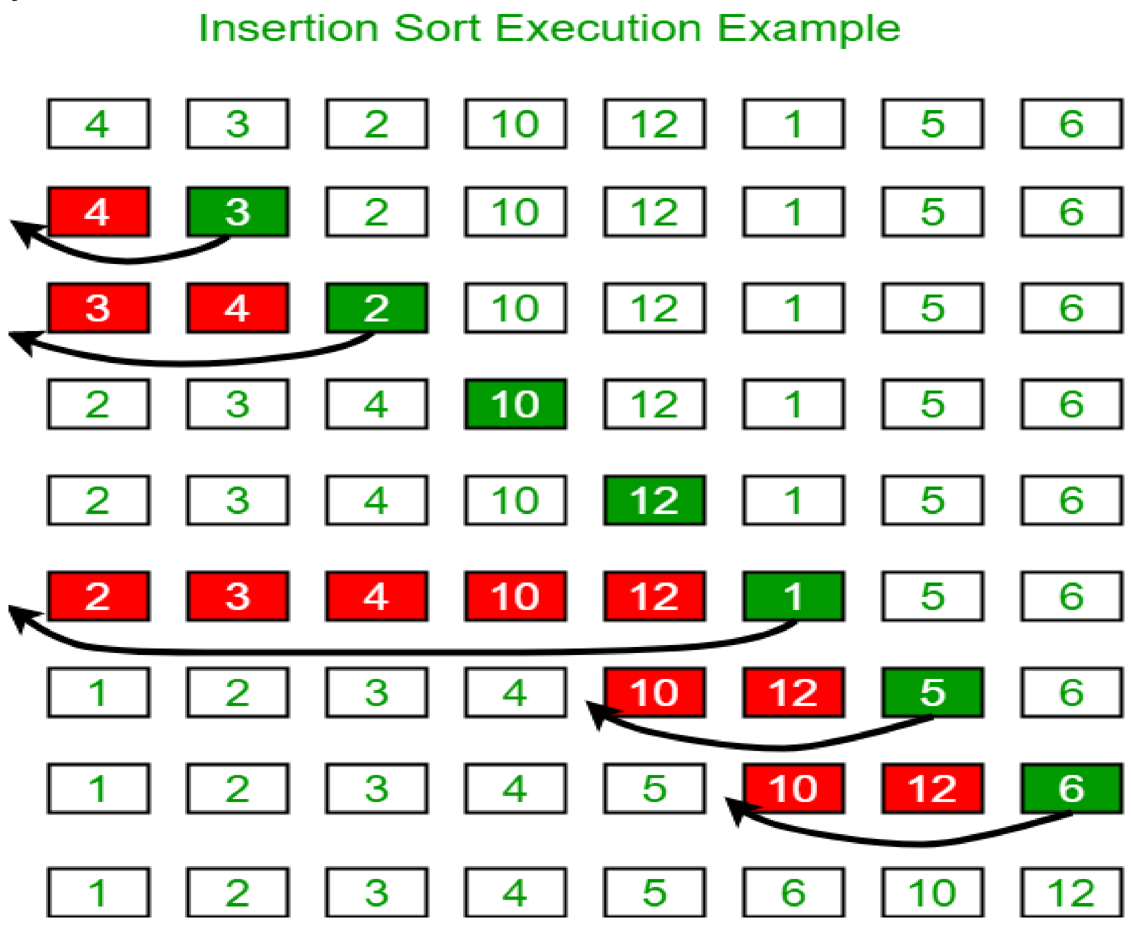
We assume list is an array of n elements. We further assume that swaps the values of given array elements.

```
 begin  BubbleSort(list)
     for all element of list
        if list[i]>list[i+1]
```

```
                swap(list[i], list[i+1])
            end if
        end for
            return list
end bubblesort
```

Insertion Sort

```
#include <stdio.h>
int main()
{
int n, array[1000], c, d, t;
printf("Enter number of elements\n");
scanf("%d", &n);
printf("Enter %d integers\n", n);
for (c = 0; c < n; c++)
{
scanf("%d", &array[c]);
 }
 for (c = 1 ; c <= n - 1; c++)
 {
 d = c;
 while ( d > 0 && array[d-1] > array[d])
 {
 t  = array[d];
 array[d]   = array[d-1];
 array[d-1] = t;
 d--;
 }
 }
 printf("Sorted list in ascending order:\n");
 for (c = 0; c <= n - 1; c++)
 {
 printf("%d\n", array[c]);
 }
 return 0;
 }
```


Insertion Sort Execution Example

MERGE SORT

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

ALGORITHM

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:
   middle m = (l+r)/2
2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

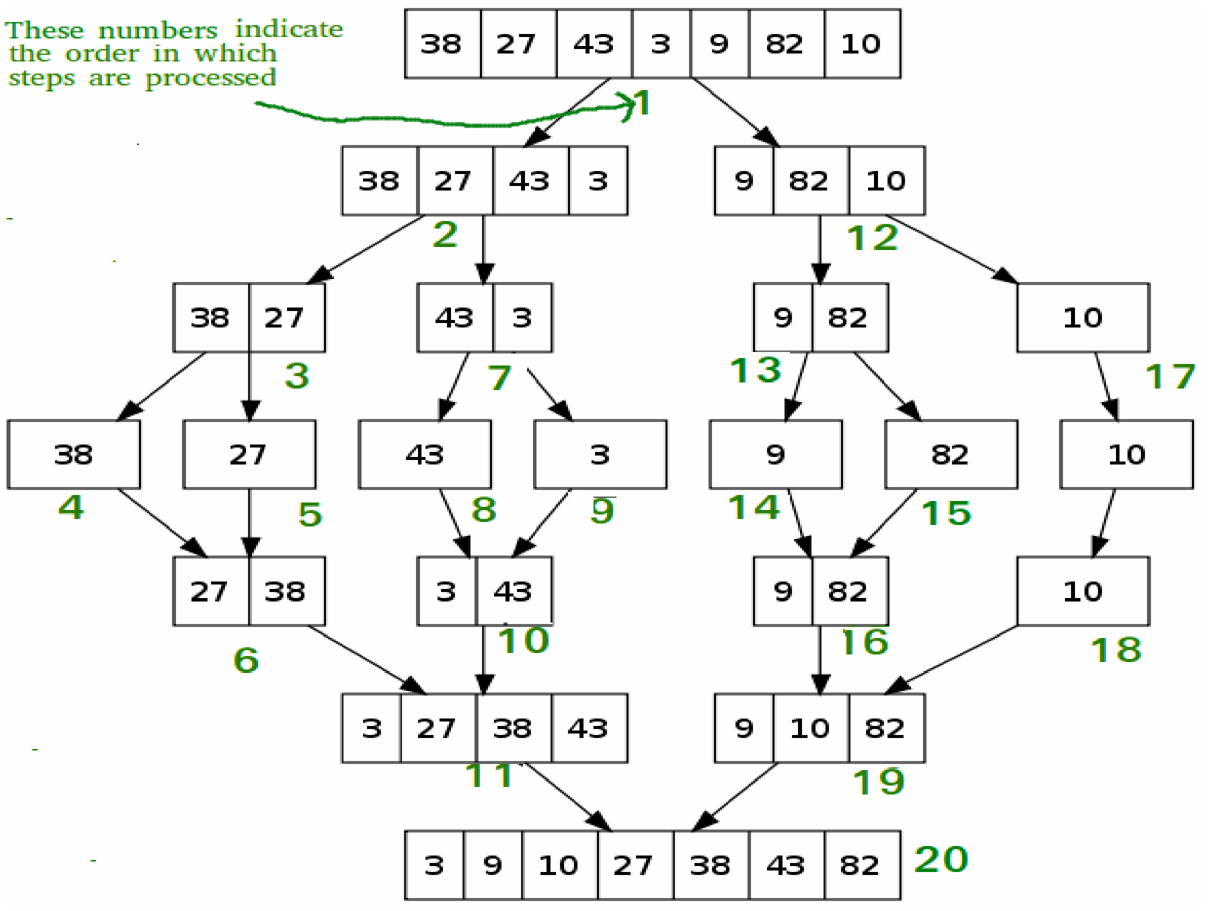
3. Call mergeSort for second half:

    Call mergeSort(arr,m+1,r)

4. Merge the two halves sorted in step 2&3

    Call merge(arr,l,m,r)

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



```
#include<stdlib.h>
#include<stdio.h>
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
```

```
 void merge(int arr[], int l, int m, int r)
{      int i, j, k;
 int n1 = m - l + 1;
int n2 =  r - m;
 /* create temp arrays */
int L[n1], R[n2];
 /* Copy data to temp arrays L[] and R[] */
for (i = 0; i < n1; i++)
L[i] = arr[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1+ j];
/* Merge the temp arrays back into arr[l..r]*/
i = 0; //
Initial index of first subarray
 j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2)
{
  if (L[i] <= R[j])
 {
 arr[k] = L[i];
i++;
}
else
{
arr[k] = R[j];
j++;
}
k++;
}
/* Copy the remaining elements of L[], if there are any */
while (i < n1)
{
arr[k] = L[i];
i++;
k++;
}
/* Copy the remaining elements of R[], if there are any */
while (j < n2)
{
arr[k] = R[j];
j++;
k++;
}  }
 /* l is for left index and r is right index of the     sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
 if (l < r)
{
 // Same as (l+r)/2, but avoids overflow for
 // large l and h
int m = l+(r-l)/2;
```

```
 // Sort first and second halves
mergeSort(arr, l, m);
mergeSort(arr, m+1, r);
merge(arr, l, m, r);
}  }
/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
int i;
for (i=0; i < size; i++)
printf("%d ", A[i]);
printf("\n");
 }
/* Driver program to test above functions */
int main()
{
int arr[] = {12, 11, 13, 5, 6, 7};
int arr_size = sizeof(arr)/sizeof(arr[0]);
printf("Given array is \n");
printArray(arr, arr_size);
mergeSort(arr, 0, arr_size - 1);
printf("\nSorted array is \n");
printArray(arr, arr_size);
return 0;
}
```

Output:

Given array is 12 11 13 5 6 7

Sorted array is 5 6 7 11 12 13

STRING

In C language a string is group of character (or) array of characters, which is terminated by delimiter \0 (null). Thus, C uses variable-length delimited strings in programs.

Declaration Strings:

C does not support string as a data type. It allows us to represent strings as character arrays. In C, a string variable is any valid C variable name and is always declared as an array of characters.

Syntax:- char string_name[size];

The size determines the number of characters in the string name.

Ex:- char city[10];

char name[30];

Initializing strings:-

There are several methods to initialize values for string variables.

Eg:

char str1[6]=”HELLO”;

char month[]=”JANUARY”;

char city[8]={„N‟,‟E‟,‟W‟,‟Y‟,‟O‟,‟R‟,‟K‟,‟\0‟};

The string city size is 8 but it contains 7 characters and one character space is for NULL terminator.

Storing strings in memory:-

In C a string is stored in an array of characters and terminated by \0 (null).

Ex:

H	E	L	L	O	\0

A string is stored in array, the name of the string is a pointer to the beginning of the string.

The character requires only one memory location.

If we use one-character string it requires two locations. The difference is shown below,

H

Character

H	\0

one-character string

\0

empty string

The difference between array and string is shown below,

H	E	L	L	O	\0

String

H	E	L	L	O

array

Because strings are variable-length structure, we must provide enough room for maximum length string to store and one byte for delimiter.

EG: char str[15]

W	E	L	C	O	M	E	\0							

NULL OPERATOR:

A string is not a datatype but a data structure. String implementation is logical not physical. The physical structure is array in which the string is stored. The string is variable-length, so we need to identify logical end of data in that physical structure.

Reading and Writing strings

Reading strings from terminal:

formatted input function:-

scanf can be used with %s format specification to read a string.

Ex:- char name[10];

scanf("%s",name);

Here don"t use „&" because name of string is a pointer to array. The problem with scanf is that it terminates its input on the first white space it finds.

Ex:- NEW YORK

Reads only NEW (from above example).

Unformatted input functions:-

(1) **getchar():-** It is used to read a single character from keyboard. Using this function repeatedly we may read entire line of text
Ex:- char ch=‟z‟; ch=getchar();

(2) **gets():-** It is more convenient method of reading a string of text including blank spaces.
Ex:- char line[100];

gets(line);

Writing strings on to the screen:-

(1) Using formatted output functions:-

printf with %s format specifier we can print strings in different formats on to screen.

Ex:- char name[10];

printf("%s",name);

Ex:- char name[10];

printf("%0.4",name);

(2**) Using unformatted output functions:-**

(a) **putchar():-** It is used to print a character on the screen.

Ex:- putchar(ch);

(b) **puts():-** It is used to print strings including blank spaces.

Ex:- char line[15]="Welcome to lab";

puts(line);

STRING HANDLING FUNCTIONS:

(i) Srtlen
(ii) Strcpy
(iii) Strcmp
(iv) Strcat

(i). strlen( )

This function is used to find the length of the string excluding the NULL character. In other words, this function is used to count the number of characters in a string. Its syntax is as follows:

Int strlen(string);

Example:

```
char str1[ ] = “WELCOME”;
int n;
n = strlen(str1);
```

/* A program to calculate length of string by using strlen() function*/

```
#include<stdio.h>
#include<string.h>
main()
{
char string1[50];
int length;
printf(“\n Enter any string:”);
gets(string1);
length=strlen(string1);
printf(“\n The length of string=%d”,length);
}
```

(ii). strcpy( )

This function is used to copy one string to the other. Its syntax is as follows:

Strcpy(string1,string2)

where string1 and string2 are one-dimensional character arrays.

This function copies the content of string2 to string1.

E.g., string1 contains master and string2 contains madam, then string1 holds madam after execution of the strcpy (string1,string2) function.

```
Example:  char str1[ ] = “WELCOME”;
                char str2[ ] =”HELLO”;
                strcpy(str1,str2);
```

/* A program to copy one string to another using strcpy() function */

```
#include<stdio.h>
#include<string.h>
main( )
{
char string1[30],string2[30];
printf(“\n Enter first string:”);
gets(string1);
printf(“\n Enter second string:”);
gets(string2);
strcpy(string1,string2);
printf(“\n First string=%s”,string1);
printf(“\n Second string=%s”,string2);
}
```

(iii). strcmp ( )

This function compares two strings character by character (ASCII comparison) and returns one of three values {-1,0,1}. The numeric difference is „0‟ if strings are equal .If it is negative string1 is alphabetically above string2 .If it is positive string2 is alphabetically above string1.

Its syntax is as follows:

Int strcmp(string1,string2);

```
Example:  char str1[ ] = “ROM”;
                char str2[ ] =”RAM”;
                strcmp(str1,str2);
          (or)
```

```
strcmp("ROM","RAM");
```

/* A program to compare two strings using strcmp() function */

```
#include<stdio.h>
#include<string.h>
main()
{
char string1[30],string2[15];
int x;
printf("\n Enter first string:");
gets(string1);
printf("\n Enter second string:");
gets(string2);
x=strcmp(string1,string2);
if(x==0)
printf("\n Both strings are equal");
else if(x>0)
printf("\n First string is bigger");
else
printf("\n Second string is bigger");
}
```

(iv). strcat ( )

This function is used to concatenate two strings. i.e., it appends one string at the end of the specified string. Its syntax as follows:

Strcat(string1,string2);

where string1 and string2 are one-dimensional character arrays.

This function joins two strings together. In other words, it adds the string2 to string1 and the string1 contains the final concatenated string. E.g., string1 contains prog and string2 contains ram, then string1 holds program after execution of the strcat() function.

Example: char str1[10] = “VERY”;

char str2[5] =”GOOD”;

strcat(str1,str2);

/* A program to concatenate one string with another using strcat() function*/

```
#include<stdio.h>
#include<string.h>
main()
{
char string1[30],string2[15];
printf(“\n Enter first string:”);
gets(string1);
printf(“\n Enter second string:”);
gets(string2);
strcat(string1,string2);
printf(“\n Concatenated string=%s”,string1);
}
```