C – Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

Types of C functions

There are two types of functions in C programming:

- Library function
- User defined function

Library function

Library functions are the in-built function in C programming system. For example:

main()

- The execution of every C program starts from this main() function.

printf()

- prinf() is used for displaying output in C.

scanf()

- scanf() is used for taking input in C.

C Standard Library Functions

C Standard library functions or simply C Library functions are inbuilt functions in C programming. Function prototype and data definitions of these functions are written in their respective header file. For example: If you want to use printf() function, the header file <stdio.h> should be included.

There is at least one function in any C program, i.e., the main() function (which is also a library function). This program is called at program starts.

There are many library functions available in C programming to help the programmer to write a good efficient program.

Suppose, you want to find the square root of a number. You can write your own piece of code to find square root but, this process is time consuming and the code you have written may not be the most efficient process to find square root. But, in C programming you can find the square root by just using sqrt() function which is defined under header file "math.h"

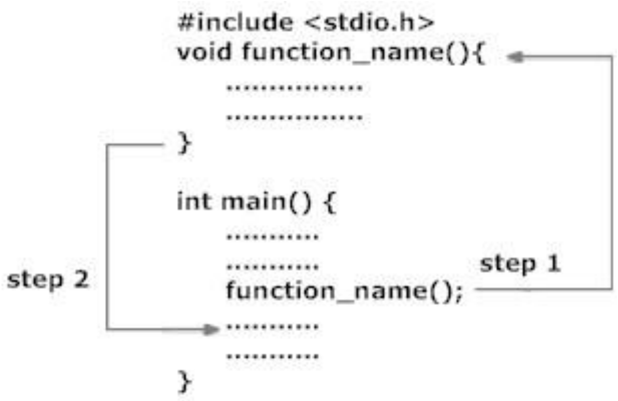
User defined function

C allows programmer to define their own function according to their requirement. These types of functions are known as user-defined functions. Suppose, a programmer wants to find factorial of a number and check whether it is prime or not in same program. Then, he/she can create two separate user-defined functions in that program: one for finding factorial and other for checking whether it is prime or not.

How user-defined function works in C Programming?

```
#include <stdio.h>

void function_name(){

................

................

}

int main(){

...........

...........

function_name();

...........

...........

}
```

As mentioned earlier, every C program begins from main() and program starts executing the codes inside main() function. When the control of program reaches to function_name() inside main() function. The control of program jumps to void function_name() and executes the codes inside it. When all the codes inside that user-defined function are executed, control of the program jumps to the statement just after function_name() from where it is called. Analyze the figure below for understanding the concept of function in C programming.



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

int main() {
    ...........
    ...........
    function_name();
    ...........
    ...........
}
```

Fig: Working of Functions

Remember, the function name is an identifier and should be unique.

Advantages of user defined functions

- User defined functions helps to decompose the large program into small segments which makes programmer easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- Programmer working on large project can divide the workload by making different functions.

Defining a Function:

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

A function definition in C programming language consists of a *function header* and a *function body*. Here are all the parts of a function:

Return Type: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.

Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body: The function body contains a collection of statements that define what the function does.

Example of user-defined function

A C program to add two integers. Make a function add to add integers and display sum in main() function.

```
/*Program to demonstrate the working of user defined function*/
#include <stdio.h>
int add(int a, int b);          //function prototype(declaration)
int main(){
    int num1,num2,sum;
    printf("Enters two number to add\n");
    scanf("%d %d",&num1,&num2);
    sum=add(num1,num2);         //function call
    printf("sum=%d",sum);
    return 0;
}
int add(int a,int b)            //function declarator
{
/* Start of function definition. */
    int add;
    add=a+b;
    return add;                 //return statement of function
/* End of function definition. */
}
```

Function prototype(declaration):

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax of function prototype

```
return_type function_name(type(1) argument(1),....,type(n)
argument(n));
```

In the above example,int add(int a, int b); is a function prototype which provides following information to the compiler:

- name of the function is add()
- return type of the function is int.
- two arguments of type int are passed to function.

Function prototype are not needed if user-definition function is written before main() function.

Function call

Control of the program cannot be transferred to user-defined function unless it is called invoked.

Syntax of function call

```
function_name(argument(1),....argument(n));
```

In the above example, function call is made using statement add(num1,num2); from main(). This make the control of program jump from that statement to function definition and executes the codes inside that function.

Function definition

Function definition contains programming codes to perform specific task.

Syntax of function definition

```
return_type function_name(type(1) argument(1),..,type(n)
argument(n))

{

                //body of function

}
```

Function definition has two major components:

1. Function declarator

Function declarator is the first line of function definition. When a function is called, control of the program is transferred to function declarator.

Syntax of function declaratory

```
return_type function_name(type(1) argument(1),....,type(n)
argument(n))
```

Syntax of function declaration and declarator are almost same except, there is no semicolon at the end of declarator and function declarator is followed by function body.

In above example, int add(int a,int b) in line 12 is a function declarator.

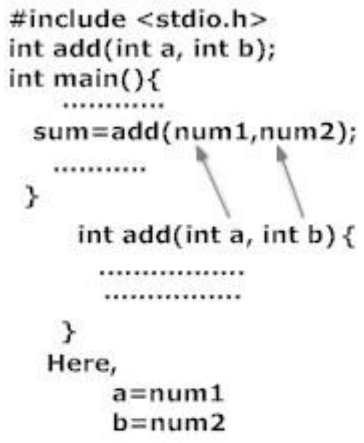
2. Function body

Function declarator is followed by body of function inside braces.

Passing arguments to functions

In programming, argument(parameter) refers to data this is passed to function(function definition) while calling function.

In above example two variable, num1 and num2 are passed to function during function call and these arguments are accepted by arguments a and b in function definition.



```
#include <stdio.h>
int add(int a, int b);
int main(){
     ...........
  sum=add(num1,num2);
    ..........
 }
      int add(int a, int b) {
        ................
        ...............

     }
   Here,
        a=num1
        b=num2
```

Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example:

If argument num1 was of int type and num2 was of float type then, argument variable a should be of type int and b should be of type float,i.e., type of argument during function call and function definition should be same.

A function can be called with or without an argument.

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the formal parameters of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Call by value in C

The call by value method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C programming language uses call by value method to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the following program and swap() function in it:

```
#include <stdio.h>

 /* function declaration */

void swap(int x, int y);

 int main ()

{

   /* local variable definition */

   int a = 100;

   int b = 200;

    printf("Before swap, value of a : %d\n", a );

   printf("Before swap, value of b : %d\n", b );

    /* calling a function to swap the values */

   swap(a, b);
```

```
    printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
    return 0;
}
/* function definition to swap the values */
void swap(int x, int y)
{
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
     return 0;
}
```

It will produce the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

It shows that there is no change in the values though they had been changed inside the function.

Call by reference in C

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

To pass the value by reference, argument pointers are passed to the functions just like any other value. So accordingly you need to declare the function parameters as pointer types as in the following function swap(), which exchanges the values of the two integer variables pointed to by its arguments.

Consider the following example:

```
#include <stdio.h>
 /* function declaration */
void swap(int *x, int *y);
 int main ()
{
   /* local variable definition */
   int a = 100;
   int b = 200;
    printf("Before swap, value of a : %d\n", a );
   printf("Before swap, value of b : %d\n", b );
    /* calling a function to swap the values */
   swap(&a, &b);
    printf("After swap, value of a : %d\n", a );
   printf("After swap, value of b : %d\n", b );
    return 0;
}
/* function definition to swap the values */
void swap(int *x, int *y)
{
   int temp;
   temp = *x; /* save the value of x */
  *x = *y;    /* put y into x */
   *y = temp; /* put temp into y */
     return 0;
}
```

It will produce the following result:

```
Before swap, value of a :100

Before swap, value of b :200

After swap, value of a :200

After swap, value of b :100
```

Which shows that the change has reflected outside of the function as well unlike call by value where changes does not reflect outside of the function.

Return Statement

Return statement is used for returning a value from function definition to calling function.

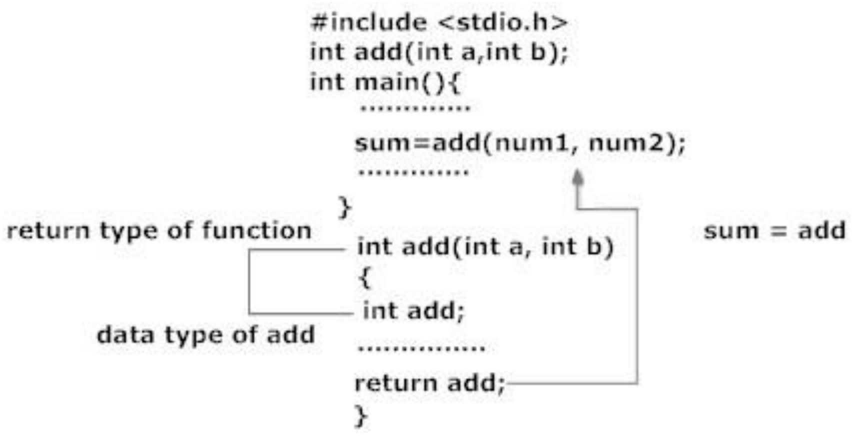
Syntax of return statement

```
return (expression);
```

For example:

```
return a;

return (a+b);
```

In above example, value of variable add in add() function is returned and that value is stored in variable sum in main() function. The data type of expression in return statement should also match the return type of function.



```
#include <stdio.h>
int add(int a,int b);
int main(){
    .............
    sum=add(num1, num2);
    .............
  }
    int add(int a, int b)
    {
    int add;
    ...............
    return add;
    }
```

Recursion

Recursion is the process of repeating items in a self-similar way. Same applies in programming languages as well where if a programming allows you to call a function inside the same function that is called recursive call of the function as follows.

```
void recursion()

{

   recursion(); /* function calls itself */

}


int main()

{

   recursion();

}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go in infinite loop.

Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

Advantages and Disadvantages of Recursion

Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.

In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

Number Factorial

Following is an example, which calculates factorial for a given number using a recursive function:

```
#include <stdio.h>


int factorial(unsigned int i)

{

   if(i <= 1)

   {
      return 1;

   }

   return i * factorial(i - 1);

}

int  main()

{

    int i = 15;

    printf("Factorial of %d is %d\n", i, factorial(i));

    return 0;

}
```

When the above code is compiled and executed, it produces the following result:

```
Factorial of 15 is 2004310016
```

Fibonacci Series

Following is another example, which generates Fibonacci series for a given number using a recursive function:

```
#include <stdio.h>


int fibonaci(int i)

{

   if(i == 0)

   {

      return 0;

   }

   if(i == 1)

   {

      return 1;

   }

   return fibonaci(i-1) + fibonaci(i-2);

}
```

```
int  main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
       printf("%d\t%n", fibonaci(i));
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
0        1        1        2        3        5        8        13       21       34
```

Matrix

A matrix (plural matrices) is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns that is interpreted and manipulated in certain prescribed ways. One such way is to state the order of the matrix. For example, the order of a matrix is represented as 2x3, because there are two rows and three columns. The individual items in a matrix are called its elements or entries.

NOTE

Two matrices can be **added** or **subtracted** element by element **only if they are the same size** (have the same number of rows and the same number of columns).

The rule for **matrix multiplication**, however, is that two matrices can be **multiplied only when the number of columns in the first equals the number of rows in the second**.

Transpose of a matrix is the matrix which is formed by turning all the rows of a given matrix into columns and vice-versa. The transpose of matrix A is written A^T.

Example: $\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}^T = \begin{bmatrix}1 & 4\\2 & 5\\3 & 6\end{bmatrix}$

A **symmetric matrix** is a square matrix that is equal to its transpose. Formally, matrix *A* is symmetric if

$$A = A^{\top}.$$

Because equal matrices have equal dimensions, only square matrices can be symmetric.

The entries of a symmetric matrix are symmetric with respect to the main diagonal. So if the entries are written as $A = (a_{ij})$, then $a_{ij} = a_{ji}$, for all indices *i* and *j*.

The following 3×3 matrix is symmetric:

$$\begin{bmatrix}1 & 7 & 3\\7 & 4 & -5\\3 & -5 & 6\end{bmatrix}.$$

The various matrix manipulation operations include matrix addition, subtraction, multiplication, finding trace of 2 matrices, finding transpose of a matrix etc.

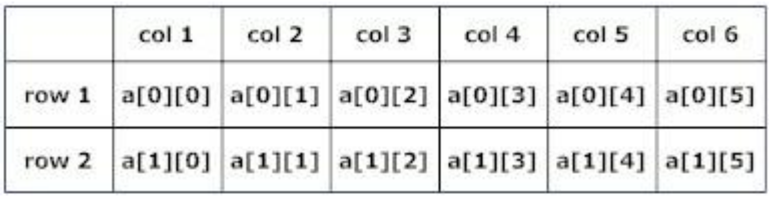
C Programming Multidimensional Arrays

C programming language allows programmer to create arrays of arrays known as multidimensional arrays. For example:

```
float a[2][6];
```

Here, *a* is an array of two dimension, which is an example of multidimensional array.

For better understanding of multidimensional arrays, array elements of above example can be considered as below:



	col 1	col 2	col 3	col 4	col 5	col 6
row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]
row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]

Figure: Multidimensional Arrays

Initialization of Multidimensional Arrays

In C, multidimensional arrays can be initialized in different number of ways.

```
int c[2][3]={{1,3,0}, {-1,5,9}};

               OR

int c[][3]={{1,3,0}, {-1,5,9}};

               OR

int c[2][3]={1,3,0,-1,5,9};
```

Initialization of three-dimensional Array

```
double cprogram[3][2][4]={

{{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},

 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},

 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}

};
```

Suppose there is a multidimensional array arr[i][j][k][m]. Then this array can hold i*j*k*m numbers of data.

Matrix Addition

Program:

```
#include <stdio.h>
int main(){
    int r,c,a[100][100],b[100][100],sum[100][100],i,j;
    printf("Enter number of rows (between 1 and 100): ");
    scanf("%d",&r);
    printf("Enter number of columns (between 1 and 100): ");
    scanf("%d",&c);
    printf("\nEnter elements of 1st matrix:\n");
/* Storing elements of first matrix entered by user. */
    for(i=0;i<r;i++)
       for(j=0;j<c;j++)
       {
           printf("Enter element a%d%d: ",i+1,j+1);
           scanf("%d",&a[i][j]);
       }
/* Storing elements of second matrix entered by user. */
    printf("Enter elements of 2nd matrix:\n");
    for(i=0;i<r;++i)
       for(j=0;j<c;++j)
       {
           printf("Enter element a%d%d: ",i+1,j+1);
           scanf("%d",&b[i][j]);
       }
/*Adding Two matrices */
  for(i=0;i<r;i++)
       for(j=0;j<c;j++)
           sum[i][j]=a[i][j]+b[i][j];
/* Displaying the resultant sum matrix. */
    printf("\nSum of two matrix is: \n\n");
    for(i=0;i<r;i++)
       for(j=0;j<c;j++)
       {
           printf("%d   ",sum[i][j]);
           if(j==c-1)
               printf("\n\n");
       }
    return 0;
}
```

Matrix Multiplication

Program:

```
#include <stdio.h>
int main()
{
   int a[10][10], b[10][10], mult[10][10], r1, c1, r2, c2, i, j, k;
   printf("Enter rows and column for first matrix: ");
   scanf("%d%d", &r1, &c1);
   printf("Enter rows and column for second matrix: ");
   scanf("%d%d",&r2, &c2);
```

/* If colum of first matrix in not equal to row of second matrix, asking user to enter the size of matrix again. */

```
    while (c1!=r2)

    {

        printf("Error! column of first matrix not equal to row of second.\n\n");

        printf("Enter rows and column for first matrix: ");

        scanf("%d%d", &r1, &c1);

        printf("Enter rows and column for second matrix: ");

        scanf("%d%d",&r2, &c2);

    }
```

/* Storing elements of first matrix. */

```
    printf("\nEnter elements of matrix 1:\n");

    for(i=0; i<r1; i++)

    for(j=0; j<c1; j++)

    {

        printf("Enter elements a%d%d: ",i+1,j+1);

        scanf("%d",&a[i][j]);

    }
```

/* Storing elements of second matrix. */

```
    printf("\nEnter elements of matrix 2:\n");

    for(i=0; i<r2; i++)

    for(j=0; j<c2; j++)

    {

        printf("Enter elements b%d%d: ",i+1,j+1);

        scanf("%d",&b[i][j]);

    }
```

```
/* Initializing elements of matrix mult to 0.*/
    for(i=0; i<r1; i++)
    for(j=0; j<c2; j++)
    {
       mult[i][j]=0;
    }
/* Multiplying matrix a and b and storing in array mult. */
    for(i=0; i<r1; i++)
    for(j=0; j<c2; j++)
    for(k=0; k<c1; k++)
    {
        mult[i][j]+=a[i][k]*b[k][j];
    }
/* Displaying the multiplication of two matrix. */
    printf("\nOutput Matrix:\n");
    for(i=0; i<r1; i++)
    for(j=0; j<c2; j++)
    {
        printf("%d  ",mult[i][j]);
        if(j==c2-1)
            printf("\n\n");
    }
    return 0;
}
```

Transpose of a Matrix

Program:

```
#include <stdio.h>
int main()
{
    int a[10][10], trans[10][10], r, c, i, j;
    printf("Enter rows and column of matrix: ");
    scanf("%d %d", &r, &c);
/* Storing element of matrix entered by user in array a[][]. */
    printf("\nEnter elements of matrix:\n");
    for(i=0; i<r; i++)
    for(j=0; j<c; j++)
    {
        printf("Enter elements a%d%d: ",i+1,j+1);
        scanf("%d",&a[i][j]);
    }
/* Displaying the matrix a[][] */
    printf("\nEntered Matrix: \n");
    for(i=0; i<r; i++)
    for(j=0; j<c; j++)
    {
        printf("%d  ",a[i][j]);
        if(j==c-1)
            printf("\n\n");
    }
/* Finding transpose of matrix a[][] and storing it in array trans[][]. */
    for(i=0; i<r; i++)
    for(j=0; j<c; j++)
    {
       trans[j][i]=a[i][j];
    }
/* Displaying the transpose,i.e, Displaying array trans[][]. */
    printf("\nTranspose of Matrix:\n");
    for(i=0; i<c; i++)
    for(j=0; j<r; j++)
    {
        printf("%d  ",trans[i][j]);
        if(j==r-1)
            printf("\n\n");
    }
    return 0;
}
```

Searching

Search as the name suggests, is an operation of finding an item from the given collection of items. In other words we check if a given element (called key) occurs in the array.

– Example: array of student records; rollno can be the key.

Linear Search

Linear search or sequential search is a method for finding a particular value in a list that checks each element in sequence until the desired element is found or the list is exhausted. The list need not be ordered. It is the simplest search algorithm; it is a special case of brute-force search.

<u>Basic idea:</u>

– *Start at the beginning of the array.*

– *Inspect elements one by one to see if it matches the key.*

<u>Time complexity:</u>

– It is a measure of how long an algorithm takes to run.

– If there are n elements in the array:

• Best case:

match found in first element (1 search operation)

• Worst case:

no match found, or match found in the last element (n search operations)

• Average case:

(n + 1) / 2 search operations

Program:

```
/*
 * C program to input N numbers and store them in an array.
 * Do a linear search for a given key and report success or failure.
 */
#include <stdio.h>
 void main()
{
    int array[10];
    int i, num, keynum, found = 0;
     printf("Enter the value of num \n");
    scanf("%d", &num);
    printf("Enter the elements one by one \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }
    printf("Input array is \n");
    for (i = 0; i < num; i++)
    {
        printf("%dn", array[i]);
    }
    printf("Enter the element to be searched \n");
    scanf("%d", &keynum);
    /*  Linear search begins */
    for (i = 0; i < num ; i++)
    {
        if (keynum == array[i] )
        {
            found = 1;
            break;
        }
    }
    if (found == 1)
        printf("Element is present in the array\n");
    else
        printf("Element is not present in the array\n");
}
```

<u>A small modification of the same program that returns the position of the element (key) if found.</u>

```
/*
 * C program to input N numbers and store them in an array.
 * Do a linear search for a given key and report success or failure also return the position of the element if found.
 */
#include <stdio.h>
 void main()
{
    int array[10];
    int i, num, keynum,pos, found = 0;
```

```
     printf("Enter the value of num \n");
    scanf("%d", &num);
    printf("Enter the elements one by one \n");
    for (i = 0; i < num; i++)
    {
        scanf("%d", &array[i]);
    }
    printf("Input array is \n");
    for (i = 0; i < num; i++)
    {
        printf("%d\n", array[i]);
    }
    printf("Enter the element to be searched \n");
    scanf("%d", &keynum);
    /*  Linear search begins */
    for (i = 0; i < num ; i++)
    {  if (keynum == array[i] )
        {
            found = 1;
            pos = i+1;
            break;
        } }
    if (found == 1)
        printf("Element is present in the array at position %d\n", pos);
    else
        printf("Element is not present in the array\n");
}
```

Binary Search

A binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomic divide-and-conquer search algorithm executes in logarithmic time.

Basic Concept:

• Binary search works if the array is sorted.

– Look for the target in the middle.

– If you don’t find it, you can ignore half of the array, and repeat the process with the other half.

• In every step, we reduce the number of elements to search in by half.

Algorithm:

First, the list has to be sorted in increasing order.

It starts with the middle element of the list.

1. If the middle element of the list is equal to the ‘input key’ then we have found the position the specified value.

2. Else if the ‘input key’ is greater than the middle element then the ‘input key’ has to be present in the last half of the list.

3. Or if the ‘input key’ is lesser than the middle element then the ‘input key’ has to be present in the first half of the list.

Hence, the search list gets reduced by half after each iteration.

[low, high] denotes the range in which element has to be present and [mid] denotes the middle element.

Initially low = 0, high = number_of_elements and mid = floor((low+high)/2).

In every iteration we reduce the range by doing the following until low is less than or equal to high(meaning elements are left in the array) or the position of the ‘input key’ has been found.

(i) If the middle element (mid) is less than key then key has to present in range

[mid+1 , high], so low=mid+1, high remains unchanged and mid is adjusted accordingly

(ii) If middle element is the key, then we are done.

(iii) If the middle element is greater than key then key has to be present in the range [low,mid-1], so high=mid-1, low remains unchanged and mid is adjusted accordingly.

Properties:

1. Best Case performance – The middle element is equal to the 'input key' O(1).

2. Worst Case performance - The 'input key' is not present in the list O(logn).

3. Average Case performance – The 'input key' is present, but it's not the middle element O(logn).

4. The List should be sorted for using Binary Search Algorithm.

5. It is faster than Linear Search algorithm, and its performance increases in comaparison to linear search as N grows.

To trace the algorithm with an example,

Refer to Attachment: Arrays_Binary_Search_Algorithm.pdf

Program:

```
#include<stdio.h>

/* Logic: [low, high] denotes the range in which element has to be present.

        Initially low = 0, high  = number_of_elements which covers entire range.

        In every step we reduce the range by doing the following

          (i) If the middle element (mid) is less than key then key has to present in range
[mid+1 , high]

         (ii) If middle element is the key, then we are done.

        (iii) If the middle element is greater than key then key has to be present in the
range[low,mid-1]

 */


int main()

{

        int number_of_elements;
```

```
scanf("%d",&number_of_elements);
int array[number_of_elements];
int iter;
/*Input has to be sorted in non decreasing order */
for(iter = 0;iter < number_of_elements;iter++)
{
        scanf("%d",&array[iter]);
}
/* Check if the input array is sorted */
for(iter = 1;iter < number_of_elements;iter++)
{
        if(array[iter] < array[iter - 1])
        {
                printf("Given input is not sorted\n");
                return 0;
        }
}
int key;
scanf("%d",&key);
int low = 0, high = number_of_elements-1, mid = (low + high)/2;
while(low <= high)
{
        if(array[mid] < key)
        {
                low = mid + 1;
        }
        else if(array[mid] == key)
        {
        printf("%d found at location %d.\n", key, mid+1);
        break;
        }
        else(array[mid] > key)
        {
            high = mid-1;
        }
        mid = (low + high)/2;
    }
    if(low>high)
    {
        printf("Element not found\n");
    }
    return 0;
}
```

Sorting

Sorting is any process of arranging items in a sequence ordered by some criterion(increasing or decreasing). Sorting can be done on names, numbers and records. Sorting reduces the For example, it is relatively easy to look up the phone number of a friend from a telephone dictionary because the names in the phone book have been sorted into alphabetical order.

Different algorithms are available to perform sorting operation. Some of the notable ones are:

- Bubble sort : Exchange two adjacent elements if they are out of order. Repeat until array is sorted.
- Insertion sort : Scan successive elements for an out-of-order item, then insert the item in the proper place.
- Selection sort : Find the smallest element in the array, and put it in the proper place. Swap it with the value in the first position. Repeat until array is sorted.
- Quick sort : Partition the array into two segments. In the first segment, all elements are less than or equal to the pivot value. In the second segment, all elements are greater than or equal to the pivot value. Finally, sort the two segments recursively.
- Merge sort : Divide the list of elements in two parts, sort the two parts individually and then merge it.

Bubble Sort

It's a sorting algorithm, in which each pair of adjacent items are compared and swapped if they are in wrong order. The comparison is repeated until no swaps are needed, indicating that the list is sorted. The smaller elements 'bubble' to the top of the list, hence, the name Bubble Sort. In this like selection sort, after every pass the largest element moves to the highest index position of the list.

<u>Algorithm:</u>

It starts with the first element of the list. The comparison of the adjacent pair of elements and swapping is done until the list is sorted as follows

1. Compare two adjacent elements and check if they are in correct order (that is second one has to be greater than the first).

2. Swap them if they are not in correct order.

Let iter denotes the number of iterations, then for iter ranging from 1 to n-1 (where n is total number of elements in the list) check

1. If value of the second item at position iter is lesser than the value of the first item i.e. at position iter-1, then swap them.

2. Else, move to the next element at position iter+1.

The effective size of the list is hence reduced by one after every pass and the largest element is moved to its final position in the sorted array. Repeat the two steps until the list is sorted and no swap is required i.e. the effective size is reduced to 1.

Properties:

1. Best Case performance – When the list is already sorted, we require only one pass to check, hence O(n).

2. Worst Case performance – When the list is in the reverse order, n number of comparisons and swap are required to be done n number of times, hence O(n2).

3. Average Case performance – O(n2)

To trace the algorithm with an example,

Refer to Attachment: Arrays_Bubble_Sort.pdf

Program:

```
#include<stdio.h>

/* Logic : Do the following thing until the list is sorted

            (i) Compare two adjacent elements and check if they are in correct order(that is
second one has

                to be greater than the first).

            (ii) Swap them if they are not in correct order.

 */

int main()

{

        int number_of_elements;

        printf("Enter the no of elements\n");

        scanf("%d",&number_of_elements);

        int array[number_of_elements];

        int iter;
```

```
printf("Enter the elements\n");
for(iter = 0;iter < number_of_elements;iter++)
{
    scanf("%d",&array[iter]);
}
/* Calling this functions sorts the array */
int temp, swapped;
do
{
    swapped = 0; /* If no element is swapped array is sorted */
    /* In the following loop compare every pair of adjacent elements and check
       if they are in correct order */
    for(iter = 1; iter < number_of_elements; iter++)
    {
        if(array[iter-1] > array[iter])
        {
            temp = array[iter-1];
            array[iter-1] = array[iter];
            array[iter] = temp;
            swapped = 1;
        }
    }

}while(swapped);
printf("Elements after sorting is as follows: \n");
for(iter = 0;iter < number_of_elements;iter++)
{
        printf("%d ",array[iter]);

    }

    printf("\n");

    return 0;

}
```

Selection Sort

In this technique, the first element is selected and compared with all other elements. If any other element is less than the first element swapping should take place. By the end of this comparison, the least element most top position in the array. This is known as pass1. In pass II, the second element is selected and compared with all other elements. Swapping takes place if any other element is less than selected element. This process continuous until array is sorted.

Property:

1. Best case performance – When the list is already sorted O(n2).

2. Worst case performance - When the list is sorted in reverse order O(n2).

3. Average case performance – O(n2).

4. O(n2) time complexity makes it difficult for long lists.

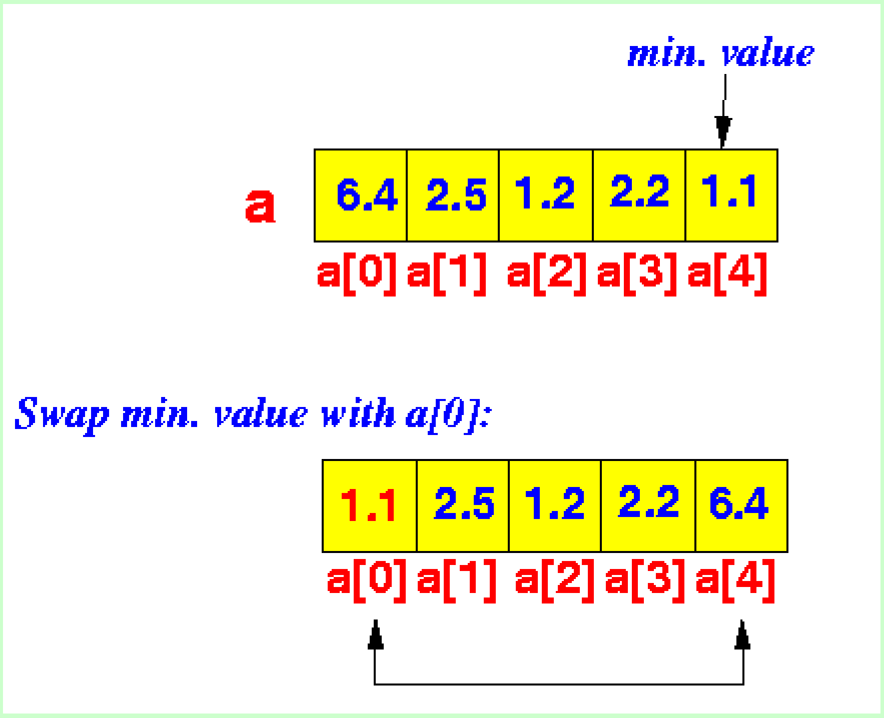
Algorithm:

1. Find the minimum element in the list.
2. Swap it with the element in the first position of the list.
3. Repeat the steps above for all remaining elements of the list starting from the second position.
4. The iteration stops when no more swap is further required.

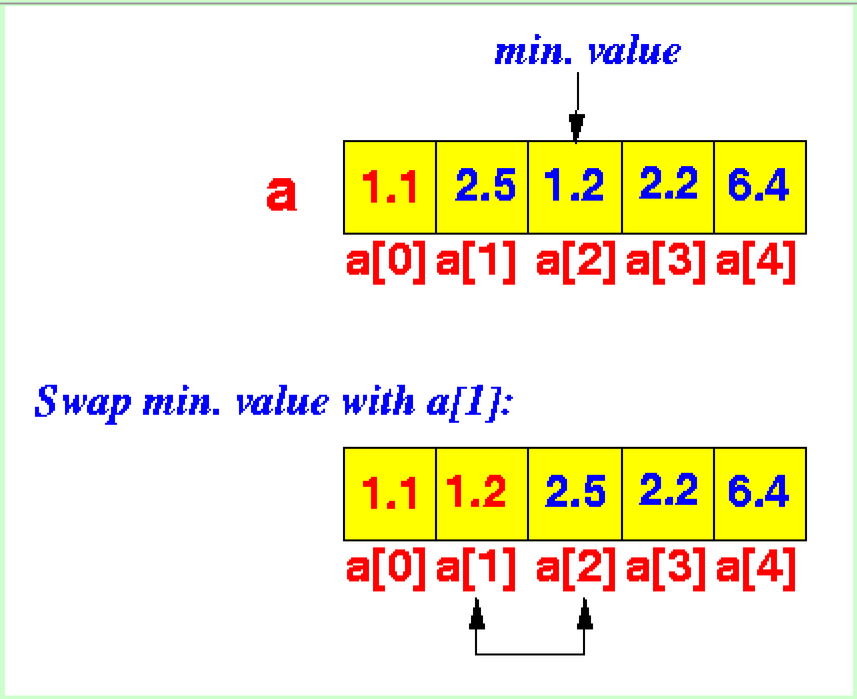
The *Selection Sort* algorithm - development step

- **Pseudo code of the *Selection Sort* algorithm:**

```
   Let: a = array containing the values
   Let: n = # of elements

   1. Find the array element with the min. value among a[0],
a[1], ..., a[n-1];
   2. Swap this element with a[0]

       Illustration:
```



```
  Repeat:

   1. Find the array element with the min. value among a[1],
..., a[n-1];
   2. Swap this element with a[1];

       Illustration:
```



```
Repeat:

  1. Find the array element with the min. value among a[2],
..., a[n-1];
  2. Swap this element with a[2];

  And so on (until we reach the last element in the array)
```

Program:

```
#include<stdio.h>

int main(){

  int s,i,j,temp,a[20];

  printf("Enter the limit: ");

  scanf("%d",&s);

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

  for(i=0;i<s;i++)

      scanf("%d",&a[i]);

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

 printf("After sorting is: \n");
 for(i=0;i<s;i++)
    printf(" %d\n",a[i]);
 return 0;
}
```

Structure

Structure is a user-defined data type in C which allows you to combine different data types to store a particular type of record. Structure helps to construct a complex data type in more meaningful way. It is somewhat similar to an Array. The only difference is that array is used to store collection of similar datatypes while structure can store collection of any type of data.

Defining a structure

struct keyword is used to define a structure. **struct** define a new data type which is a collection of different type of data.

Syntax :

```
struct structure_name
{
 //Statements
};
```

Example of Structure

```
struct Book
{
 char name[15];
 int price;
 int pages;
};
```

Here the **struct Book** declares a structure to hold the details of book which consists of three data fields, namely*name*, *price* and *pages*. These fields are called **structure elements or members**. Each member can have different data type,like in this case, **name** is of char type and **price** is of int type etc. **Book** is the name of the structure and is called structure tag.

Declaring Structure Variables

It is possible to declare variables of a **structure**, after the structure is defined. **Structure** variable declaration is similar to the declaration of variables of any other data types. Structure variables can be declared in following two ways.

1) Declaring Structure variables separately

```
struct Student

{

 char[20] name;

 int age;

 int rollno;

} ;


struct Student S1 , S2;   //declaring variables of Student
```

2) Declaring Structure Variables with Structure definition

```
struct Student

{

 char[20] name;

 int age;

 int rollno;

} S1, S2 ;
```

Here **S1** and **S2** are variables of structure **Student**. However this approach is not much recommended.

Accessing Structure Members

Structure members can be accessed and assigned values in number of ways. Structure member has no meaning independently. In order to assign a value to a structure member, the member name must be linked with the **structure** variable using dot `.` operator also called **period** or **member access** operator.

```
struct Book

{

 char name[15];

 int price;
 int pages;
} b1 , b2 ;

b1.price=200;      //b1 is variable of Book type and price is member of Book
```

We can also use `scanf()` to give values to structure members through terminal.

```
scanf(" %s ", b1.name);
scanf(" %d ", &b1.price);
```

Structure Initialization

Like any other data type, structure variable can also be initialized at compile time.

```
struct Patient
{
 float height;
 int weight;
 int age;
};

struct Patient p1 = { 180.75 , 73, 23 };    //initialization
or,
struct patient p1;
p1.height = 180.75;     //initialization of each member separately
p1.weight = 73;
p1.age = 23;
```

Array of Structure

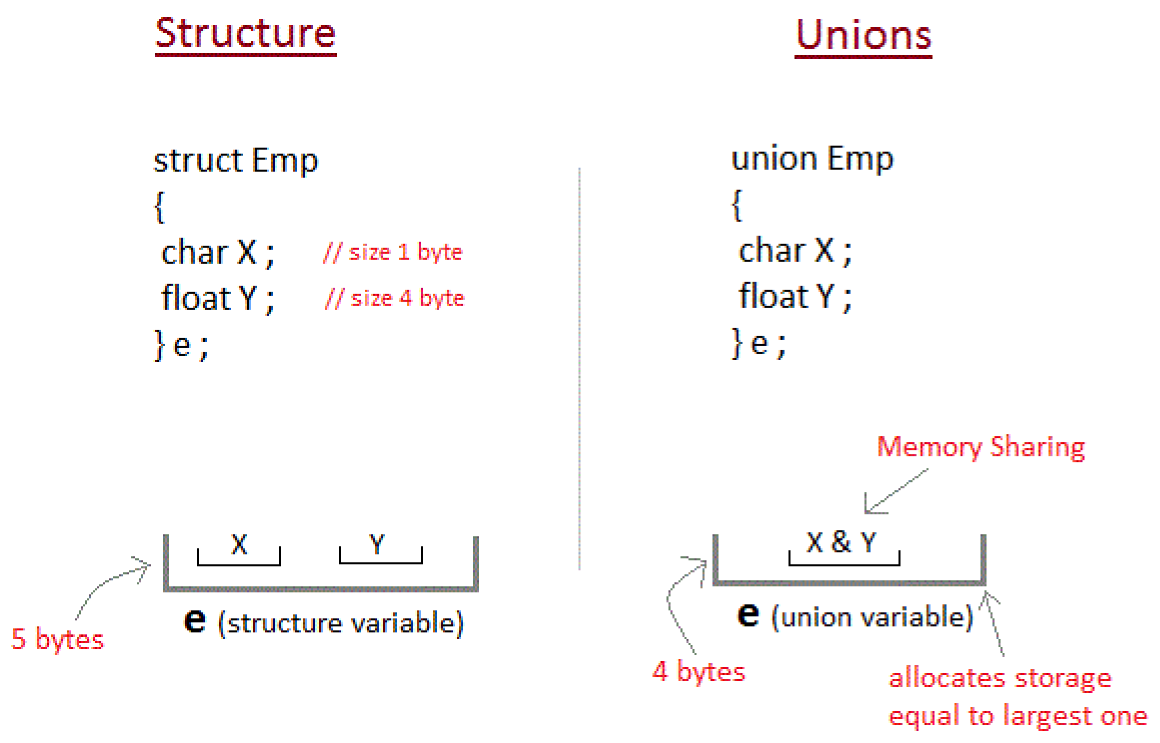
We can also declare an array of **structure**. Each element of the array representing a **structure** variable.**Example :** `struct employee emp[5];`

The above code define an array **emp** of size 5 elements. Each element of array **emp** is of type **employee**

```
#include<stdio.h>

#include<conio.h>

struct employee

{

 char ename[10];

 int sal;

};


struct employee emp[5];

int i,j;

void ask()

{

 for(i=0;i<3;i++)

 {

  printf("\nEnter %dst employee record\n",i+1);

  printf("\nEmployee name\t");

  scanf("%s",emp[i].ename);

  printf("\nEnter employee salary\t");

  scanf("%d",&emp[i].sal);

 }

 printf("\nDisplaying Employee record\n");

 for(i=0;i<3;i++)

 {

  printf("\nEmployee name is %s",emp[i].ename);

  printf("\nSlary is %d",emp[i].sal);

 }

}

void main()

{

 clrscr();
 ask();

 getch();

}
```

Unions in C Language

Unions are conceptually similar to **structures**. The syntax of **union** is also similar to that of structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of **union** uses a single shared memory location which is equal to the size of its largest data member.



This implies that although a **union** may contain many members of different types, **it cannot handle all the members at same time**. A **union** is declared using **union** keyword.

```
union item

{

 int m;

 float x;

 char c;

}It1;
```

This declares a variable **It1** of type union **item**. This **union** contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for a **union** variable, irrespective of its size. The compiler allocates the storage that is large enough to hold largest variable type in the **union**. In the **union** declared above the member **x** requires 4 bytes which is largest among the members in 16-bit machine. Other members of **union** will share the same address.

Accessing a Union Member

Syntax for accessing **union** member is similar to accessing structure member,

```
union test
{
 int a;
 float b;
 char c;
}t;

t.a ;     //to access members of union t
t.b ;
t.c ;
```

Complete Example for Union

```
#include <stdio.h>
#include <conio.h>

union item
{
 int a;
 float b;
 char ch;
};

int main( )
{
 union item it;

 it.a = 12;

 it.b = 20.2;

 it.ch='z';

 clrscr();

 printf("%d\n",it.a);

 printf("%f\n",it.b);

 printf("%c\n",it.ch);

 getch();

 return 0;

}
```

Output :

```
-26426

20.1999

z
```

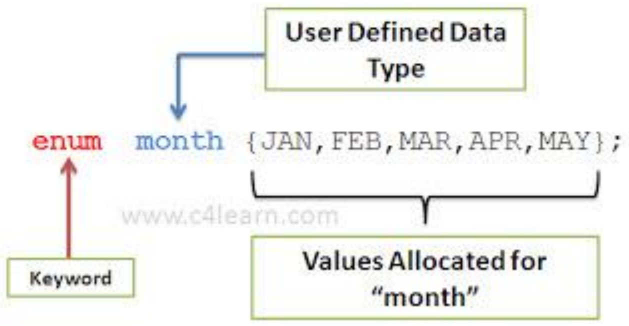
As you can see here, the values of **a** and **b** get corrupted and only variable **c** prints the expected result. Because in **union**, the only member whose value is currently stored will have the memory.

Enum [Enumerated] : User defined Data Type in C

Syntax:

```
enum identifier {value1, value2,.... Value n};
```

- enum is **" Enumerated Data Type "**.
- enum is **user defined** data type
- In the above example **"identifier"** is nothing but the **user defined data type** .
- Value1,Value2,Value3….. etc creates **one set of enum values**.
- Using **"identifier"** we are **creating our variables**.


User Defined Data Type
enum month {JAN,FEB,MAR,APR,MAY};
www.c4learn.com
Keyword
Values Allocated for "month"

Let's Take Look at Following Example … Example :

```
enum month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,DEC};
enum month rmonth;
```

1. First Line Creates **"User Defined Data Type"** called **month**.
2. It has 12 values as given in the **pair of braces**.
3. In the second line "rmonth" variable is declared of type "month" which can be initialized with any "data value amongst 12 values".

```
rmonth = FEB;
```

1. **Default Numeric value** assigned to first enum value is "0".
2. Numerically JAN is given value "0".

3. FEB is given value “1”.
4. MAR is given value “2”.
5. APR is given value “3”.
6. MAY is given value “4”.
7. JUN is given value “5”.
8. and so on…..

```
printf("%d",rmonth);
```

- It will Print “1” on the screen because “Numerical Equivalent” of “FEB” is 1 and “rmonth” is initialized by “FEB”.
- Generally Printing Value of enum variable is as good as printing “**Integer**“.

Sample Program :

```
#include< stdio.h>
void main()
{
int i;
enum month {JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,DEC};
clrscr();
  for(i=JAN;i<=DEC;i++)
      printf("\n%d",i);
}
```

Output :

```
01234567891011
```

Same notes presented in different manner for REFERENCE

Enumeration in C

An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword `enum` is used to defined enumerated data type.

```
enum type_name{ value1, value2,...,valueN };
```

Here, *type_name* is the name of enumerated data type or tag.
And *value1*, *value2*,....,*valueN* are values of type *type_name*.

By default, *value1* will be equal to 0, *value2* will be 1 and so on but, the programmer can change the default value.

```
// Changing the default value of enum elements

enum suit{

    club=0;

    diamonds=10;

    hearts=20;

    spades=3;

};
```

Declaration of enumerated variable

Above code defines the type of the data but, no any variable is created. Variable of type `enum` can be created as:

```
enum boolean{

    false;

    true;

};

enum boolean check;
```

Here, a variable check is declared which is of type **`enum boolean`**.

Example of enumerated type

```
#include <stdio.h>
enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};
int main(){
   enum week today;
   today=wednesday;
   printf("%d day",today+1);
   return 0;
  }
```

Output

```
4 day
```

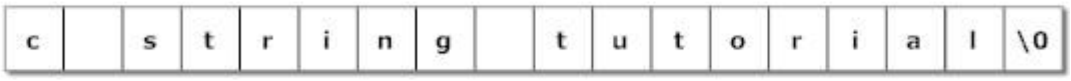
You can write any program in C language without the help of enumerations but, enumerations helps in writing clear codes and simplify programming.

C Programming String

In C programming, array of character are called strings. A string is terminated by null character /0. For example:

```
"c string tutorial"
```

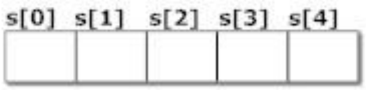
Here, "c string tutorial" is a string. When, compiler encounters strings, it appends null character at the end of string.



Declaration of strings

Strings are declared in C in similar manner as arrays. Only difference is that, strings are of char type.

```
char s[5];
```



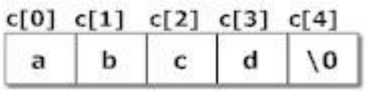
Strings can also be declared using pointer.

```
char *p
```

Initialization of strings

In C, string can be initialized in different number of ways.

```
char c[]="abcd";

    OR,

char c[5]="abcd";

    OR,

char c[]={'a','b','c','d','\0'};

    OR;

char c[5]={'a','b','c','d','\0'};
```



String can also be initialized using pointers

```
char *c="abcd";
```

Reading Strings from user.

Reading words from user.

```
char c[20];

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

String variable *c* can only take a word. It is beacause when white space is encountered, the scanf()function terminates.

Write a C program to illustrate how to read string from terminal.

```
#include <stdio.h>
int main(){
    char name[20];
    printf("Enter name: ");
    scanf("%s",name);
    printf("Your name is %s.",name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie

Your name is Dennis.
```

Here, program will ignore Ritchie because, scanf() function takes only string before the white space.

Reading a line of text

C program to read line of text manually.

```
#include <stdio.h>
int main(){
    char name[30],ch;
    int i=0;
    printf("Enter name: ");
    while(ch!='\n')    // terminates if user hit enter
    {
        ch=getchar();
        name[i]=ch;
        i++;
    }
    name[i]='\0';       // inserting null character at end
    printf("Name: %s",name);
    return 0;
}
```

This process to take string is tedious. There are predefined functions gets() and puts in C language to read and display string respectively.

```
int main(){
   char name[30];
   printf("Enter name: ");
   gets(name);     //Function to read string from user.
   printf("Name: ");
   puts(name);    //Function to display string.
   return 0;
}
```

Both, the above program has same output below:

Output

```
Enter name: Tom Hanks

Name: Tom Hanks
```

Passing Strings to Functions

String can be passed to function in similar manner as arrays as, string is also an array. Learn more about passing array to a function.

```
#include <stdio.h>
void Display(char ch[]);
int main(){
   char c[50];
   printf("Enter string: ");
   gets(c);
   Display(c);     // Passing string c to function.
   return 0;
}
void Display(char ch[]){
   printf("String Output: ");
   puts(ch);
}
```

Here, string c is passed from main() function to user-defined function Display(). In function declaration, ch[] is the formal argument.

String handling functions

You can perform different type of string operations manually like: finding length of string, concatenating(joining) two strings etc. But, for programmers ease, many library function are defined under header file <string.h> to handle these commonly used talk in C programming.

String Manipulations In C Programming Using Library Functions

Strings are often needed to be manipulated by programmer according to the need of a problem. All string manipulation can be done manually by the programmer but, this makes programming complex and large. To solve this, the C supports a large number of string handling functions.

There are numerous functions defined in "string.h" header file. Few commonly used string handling functions are discussed below:

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

Strings handling functions are defined under "string.h" header file, i.e, you have to include the code below to run string handling functions.

```
#include <string.h>
```

gets() and puts()

Functions gets() and puts() are two string functions to take string input from user and display string respectively as mentioned in previous chapter.

```
#include<stdio.h>
int main(){
    char name[30];
    printf("Enter name: ");
    gets(name);     //Function to read string from user.
    printf("Name: ");
    puts(name);    //Function to display string.
    return 0;
}
```

Though, gets() and puts() function handle string, both these functions are defined in "stdio.h"header file.

Program to find the Frequency of Characters

```
#include <stdio.h>
int main(){
   char c[1000],ch;
   int i,count=0;
   printf("Enter a string: ");
   gets(c);
   printf("Enter a character to find frequency: ");
   scanf("%c",&ch);
   for(i=0;c[i]!='\0';i++)
   {
       if(ch==c[i])
           count++;
   }
   printf("Frequency of %c = %d", ch, count);
   return 0;
}
```

Find Number of Vowels, Consonants, Digits and White Space Character

```
#include<stdio.h>
int main(){
    char line[150];
    int i,v,c,ch,d,s,o;
    o=v=c=ch=d=s=0;
    printf("Enter a line of string:\n");
    gets(line);
    for(i=0;line[i]!='\0';i++)
    {
        if(line[i]=='a' || line[i]=='e' || line[i]=='i' || line[i]=='o' || line[i]=='u' || line[i]=='A' ||
line[i]=='E' || line[i]=='I' || line[i]=='O' || line[i]=='U')

            v++;

        else if((line[i]>='a'&& line[i]<='z') || (line[i]>='A'&& line[i]<='Z'))

            c++;

        else if(line[i]>='0'&&c<='9')

            d++;

        else if (line[i]==' ')

            s++;

    }

    printf("Vowels: %d",v);

    printf("\nConsonants: %d",c);

    printf("\nDigits: %d",d);

    printf("\nWhite spaces: %d",s);

    return 0;

}
```

Calculated Length without Using strlen() Function

```
#include <stdio.h>

int main()

{

    char s[1000],i;

    printf("Enter a string: ");

    scanf("%s",s);

    for(i=0; s[i]!='\0'; i++);

    printf("Length of string: %d",i);

    return 0;}
```

String Concatenation

You can concatenate two strings easily using standard library function strcat() but, this program concatenates two strings manually without using strcat() function.

```
#include <stdio.h>

int main()

{

   char s1[100], s2[100], i, j;

   printf("Enter first string: ");

   scanf("%s",s1);

   printf("Enter second string: ");

   scanf("%s",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);

   return 0;

}
```

Program to copy contents of one string to another string

You can use the strcpy() function to copy the content of one string to another but, this program copies the content of one string to another manually without using strcpy() function.

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

Program to check for String Palindrome

Palindrome is a number or string that remains the same even if reversed for example , MALAYALAM if reversed again it is MALAYALAM

```
/*
 * C program to read a string and check if it's a palindrome, without
 * using library functions. Display the result.
 */
```

```
#include <stdio.h>

#include <string.h>


void main()

{

    char string[25], reverse_string[25] = {'\0'};

    int  i, length = 0, flag = 0;

    printf("Enter a string \n");

    gets(string);

    /*  keep going through each character of the string till its end */

    for (i = 0; string[i] != '\0'; i++)

    {

        length++;

    }

    for (i = length - 1; i >= 0; i--)

    {

       reverse_string[length - i - 1] = string[i];

    }

    /*

     * Compare the input string and its reverse. If both are equal

     * then the input string is palindrome.

     */

    for (i = 0; i < length; i++)

    {

        if (reverse_string[i] == string[i])

            flag = 1;

        else
```

```
            flag = 0;
    }
    if (flag == 1)
        printf("%s is a palindrome \n", string);
    else
        printf("%s is not a palindrome \n", string);
}
```

Program to accept 2 strings and compare them

```
/*
 * C Program to accepts two strings and compare them. Display
 * the result whether both are equal, or first string is greater
 * than the second or the first string is less than the second string
 */
#include <stdio.h>
 void main()
{
    int count1 = 0, count2 = 0, flag = 0, i;
    char string1[10], string2[10];
     printf("Enter a string:");
    gets(string1);
    printf("Enter another string:");
    gets(string2);
    /*  Count the number of characters in string1 */
    while (string1[count1] != '\0')
        count1++;
    /*  Count the number of characters in string2 */
    while (string2[count2] != '\0')
        count2++;
    i = 0;
    while ((i < count1) && (i < count2))
    {
        if (string1[i] == string2[i])
        {
            i++;
            continue;
        }
        if (string1[i] < string2[i])
        {
            flag = -1;
            break;
        }
        if (string1[i] > string2[i])
        {
            flag = 1;
            break;
        }   }
    if (flag == 0)
        printf("Both strings are equal \n");
    if (flag == 1)
        printf("String1 is greater than string2 \n", string1, string2);
    if (flag == -1)
        printf("String1 is less than string2 \n", string1, string2);
}
```

Program to sort n names alphabetically

```
/*
 * C program to read N names, store them in the form of an array
 * and sort them in alphabetical order. Output the given names and
 * the sorted names in two columns side by side.
 */
#include <stdio.h>
#include <string.h>

void main()
{
    char name[10][8], tname[10][8], temp[8];
    int i, j, n;

    printf("Enter the value of n \n");
    scanf("%d", &n);
    printf("Enter %d names\n", n);
    for (i = 0; i < n; i++)
    {
        scanf("%s", name[i]);
        strcpy(tname[i], name[i]);
    }
    for (i = 0; i < n - 1 ; i++)
    {
        for (j = i + 1; j < n; j++)
        {
            if (strcmp(name[i], name[j]) > 0)
            {
                strcpy(temp, name[i]);
                strcpy(name[i], name[j]);
                strcpy(name[j], temp);
            }
        }
    }
    printf("\n----------------------------------------\n");
    printf("Input Names     Sorted names\n");
    printf("------------------------------------------\n");
    for (i = 0; i < n; i++)
    {
        printf("%s\t\t%s\n", tname[i], name[i]);
    }
    printf("------------------------------------------\n");
}
```