UNIT 3:
Arrays and Functions

Array is the collection of similar data types or collection of similar entity stored in contiguous memory location. Array of character is a string. Each data item of an array is called an element. And each element is unique and located in separated memory location. Each of elements of an array share a variable but each element having different index no. known as subscript.

An array can be a single dimensional or multi-dimensional and number of subscripts determines its dimension. And number of subscript is always starts with zero. One dimensional array is known as vector and two dimensional arrays are known as matrix.

ADVANTAGES: array variable can store more than one value at a time where other variable can store one value at a time.

Example:

```
 int  arr[100];
int mark[100];
```

DECLARATION OF AN ARRAY :

```
        Its syntax is :
       Data type array name [size];
    int arr[100];
        int mark[100];
                      int a[5]={10,20,30,100,5}
```

The declaration of an array tells the compiler that, the data type, name of the array, size of the array and for each element it occupies memory space. Like for int data type, it occupies 2 bytes for each element and for float it occupies 4 byte for each element etc. The size of the array operates the number of elements that can be stored in an array and it may be a int constant or constant int expression.

We can represent individual array as :

```
int ar[5];
```

ar[0], ar[1], ar[2], ar[3], ar[4];

Symbolic constant can also be used to specify the size of the array as:

#define SIZE 10;

INITIALIZATION OF AN ARRAY:

After declaration element of local array has garbage value. If it is global or static array then it will be automatically initialize with zero. An explicitly it can be initialize that

Data type array name [size] = {value1, value2, value3…} Example:

in ar[5]={20,60,90, 100,120}

Array subscript always start from zero which is known as lower bound and upper value is known as upper bound and the last subscript value is one less than the size of array. Subscript can be an expression i.e. integer value. It can be any integer, integer constant, integer variable, integer expression or return value from functional call that yield integer value.

So if i & j are not variable then the valid subscript are ar [i*7],ar[i*i],ar[i++],ar[3];

The array elements are standing in continuous memory locations and the amount of storage required for hold the element depend in its size & type.

Total size in byte for 1D array is:

Total bytes=size of (data type) * size of array.

Example : if an array declared is:

int [20];

Total byte= 2 * 20 =40 byte.

ACCESSING OF ARRAY ELEMENT:

/*Write a program to input values into an array and display them*/

#include<stdio.h> int

main()

{

int arr[5],i; for(i=0;i<5;i++)

{

printf("enter a value for arr[%d] \n",i); scanf("%d",&arr[i]);

```
}
printf("the array elements are: \n"); for
(i=0;i<5;i++)
{
printf("%d\t",arr[i]);
}
return 0;
}
```

OUTPUT:

```
Enter a value for arr[0]  = 12
Enter a value for arr[1]  =45 Enter a
value for arr[2]  =59 Enter a value for
arr[3]  =98 Enter a value for arr[4]
=21
The array elements are 12  45  59  98  21
```

Example: From the above example value stored in an array are and occupy its memory addresses 2000, 2002, 2004, 2006, 2008 respectively.

a[0]=12, a[1]=45, a[2]=59, a[3]=98, a[4]=21

ar[0]	ar[1]	ar[2]	ar[3]	ar[4]
12	45	59	98	21
2000	2002	2004	2006	2008

Example 2:

```
  /* Write a program to add 10 array elements */
   #include<stdio.h>
void main()
{
 int i ; int arr [10]; int
sum=o; for (i=0;
i<=9; i++) {
printf (“enter the %d element \n”, i+1); scanf (“%d”,
&arr[i]);
}
for (i=0; i<=9; i++) {
sum = sum + a[i]; }
printf (“the sum of 10 array elements is %d”, sum);
}
```

OUTPUT:

```
Enter a value for arr[0]  =5
Enter a value for arr[1]  =10 Enter a
value for arr[2]  =15 Enter a value for
arr[3]  =20 Enter a value for arr[4]
=25 Enter a value for arr[5]  =30 Enter
a value for arr[6]  =35 Enter a value
for arr[7]  =40 Enter a value for arr[8]
=45 Enter a value for arr[9]  =50 Sum
= 275
```

while initializing a single dimensional array, it is optional to specify the size of array. If the size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers. For example:-

```
int  marks[]={99,78,50,45,67,89};
```

If during the initialization of the number the initializers is less then size of array, then all the remaining elements of array are assigned value zero . For example:-

```
int marks[5]={99,78};
```

Here the size of the array is 5 while there are only two initializers so After this initialization, the value of the rest elements are automatically occupied by zeros such as

Marks[0]=99 , Marks[1]=78 , Marks[2]=0, Marks[3]=0, Marks[4]=0 Again if we

initialize an array like int array[100]={0};

Then the all the element of the array will be initialized to zero. If the number of initializers is more than the size given in brackets then the compiler will show an error.

For example:-

int arr[5]={1,2,3,4,5,6,7,8};//error

we cannot copy all the elements of an array to another array by simply assigning it to the other

array like, by initializing or declaring as int a[5] ={1,2,3,4,5}; int b[5]; b=a;//not valid

(**note**:-here we will have to copy all the elements of array one by one, using for loop.)

Deletion of an element from an array

To delete a specific element from an array, a user must define the position from which the array's element should be removed. The deletion of the element does not affect the size of an array. Furthermore, we should also check whether the deletion is possible or not in an array.

For example, suppose an array contains seven elements, arr[] = {10, 25, 14, 8, 12, 15, 5); and the user want to delete element 8. So, first, the user must define the position of the 8th element, which is the 4th, and then check whether the deletion is possible or not. The position of the particular element should not be more than the total elements of an array. Here, we have 7 elements in an array, and the user wants to delete the 8th position element, which is impossible.

Steps to remove an element from an array

Following is the steps to remove a particular element from an array in C programming.

Step 1: Input the size of the array arr[] using num, and then declare the pos variable to define the position, and i represent the counter value.

Step 2: Use a loop to insert the elements in an array until (i < num) is satisfied.

Step 3: Now, input the position of the particular element that the user or programmer wants to delete from an array.

Step 4: Compare the position of an element (pos) from the total no. of elements (num+1). If the pos is greater than the num+1, the deletion of the element is not possible and jump to step 7.

Step 5: Else removes the particular element and shift the rest elements' position to the left side in an array.

Step 6: Display the resultant array after deletion or removal of the element from an array.

Step 7: Terminate or exit from the program.

Example 1: Program to remove an element from an array using for loop

Let's create a program to delete an element from an array using for loop in the C programming language.

```
/* program to remove the specific elements from an array in C. */
#include <stdio.h>
#include <conio.h>

int main ()
{
    // declaration of the int type variable
    int arr[50];
    int pos, i, num; // declare int type variable
    printf (" \n Enter the number of elements in an array: \n ");
    scanf (" %d", &num);

    printf (" \n Enter %d elements in array: \n ", num);

    // use for loop to insert elements one by one in array
    for (i = 0; i < num; i++ )
    {   printf (" arr[%d] = ", i);
        scanf (" %d", &arr[i]);
    }

    // enter the position of the element to be deleted
    printf( " Define the position of the array element where you want to delete: \n ");
    scanf (" %d", &pos);

    // check whether the deletion is possible or not
    if (pos >= num+1)
    {
        printf (" \n Deletion is not possible in the array.");
    }
    else
    {
        // use for loop to delete the element and update the index
        for (i = pos - 1; i < num -1; i++)
        {
            arr[i] = arr[i+1]; // assign arr[i+1] to arr[i]
        }

        printf (" \n The resultant array is: \n");

        // display the final array
        for (i = 0; i< num - 1; i++)
        {
            printf (" arr[%d] = ", i);
            printf (" %d \n", arr[i]);
        }
    }
    return 0;
}
```

Output:

```
Enter the number of elements in an array:
8

Enter 8 elements in array:
arr[0] = 3
arr[1] = 6
```

```
arr[2] = 2
arr[3] = 15
arr[4] = 10
arr[5] = 5
arr[6] = 8
arr[7] = 12
Define the position of the array element where you want to delete:
5

The resultant array is:
arr[0] = 3
arr[1] = 6
arr[2] = 2
arr[3] = 15
arr[4] = 5
arr[5] = 8
arr[6] = 12
```

In the above program, we input 8 elements for an array arr[] from the user. After that, the user enters the position of the particular element is 5. And then, we checked the defined position by the If statement (if (pos > num + 1)); Here, 5 is less than the (num+1). So the condition is false, and it executes the else block to remove the 5th position of the element is 10 using for loop and then prints the resultant array.

Example 2: Program to remove the specific character from an array using for loop

Let's consider an example to delete the particular position of the defined character from an array using for loop in the C programming language.

```c
/* program to remove the specific elements from an array in C. */
#include <stdio.h>
#include <conio.h>

int main ()
{
    // declaration of the int type variable
    int arr[50];
    int pos, i, num; // declare int type variable
    printf (" \n Enter the number of elements in an array: \n ");
    scanf (" %d", &num);

    printf (" \n Enter %d elements in array: \n ", num);
```

```c

// use for loop to insert elements one by one in array
for (i = 0; i < num; i++ )
{   printf (" arr[%d] = ", i);
    scanf (" %c", &arr[i]);
}

// enter the position of the element to be deleted
printf( " Define the position of the array element where you want to delete: \n ");
scanf (" %d", &pos);

// check whether the deletion is possible or not
if (pos >= num+1)
{
    printf (" \n Deletion is not possible in the array.");
}
else
{
    // use for loop to delete the element and update the index
    for (i = pos - 1; i < num -1; i++)
    {
        arr[i] = arr[i+1]; // assign arr[i+1] to arr[i]
    }

    printf (" \n The resultant array is: \n");

    // display the final array
    for (i = 0; i< num - 1; i++)
    {
        printf (" arr[%d] = ", i);
        printf (" %c \n", arr[i]);
    }
}
return 0;
```

48. }

Output:

```
Enter the number of elements in an array:
8
Enter 8 elements in array:
arr[0] = a
arr[1] = b
arr[2] = f
arr[3] = h
arr[4] = e
arr[5] = k
arr[6] = w
arr[7] = p
Define the position of the array element where you want to delete:
4

The resultant array is:
arr[0] = a
arr[1] = b
arr[2] = f
arr[3] = e
arr[4] = k
arr[5] = w
arr[6] = p
```

Largest Element in an array

```
#include <stdio.h>
int main() {
 int n;
 double arr[100];
 printf("Enter the number of elements (1 to 100): ");
 scanf("%d", &n);

 for (int i = 0; i < n; ++i) {
  printf("Enter number%d: ", i + 1);
  scanf("%lf", &arr[i]);
 }

 // storing the largest number to arr[0]
 for (int i = 1; i < n; ++i) {
  if (arr[0] < arr[i]) {
   arr[0] = arr[i];
  }
 }

 printf("Largest element = %.2lf", arr[0]);

 return 0;
}
```
Run Code

Output

```
Enter the number of elements (1 to 100): 5
Enter number1: 34.5
Enter number2: 2.4
Enter number3: -35.5
Enter number4: 38.7
Enter number5: 24.5
Largest element = 38.70
```

This program takes `n` number of elements from the user and stores it in the `arr` array.

To find the largest element,

- the first two elements of array are checked and the largest of these two elements are placed in `arr[0]`
- the first and third elements are checked and largest of these two elements is placed in `arr[0]`.
- this process continues until the first and last elements are checked
- the largest number will be stored in the `arr[0]` position

```
// storing the largest number at arr[0]
for (int i = 1; i < n; ++i) {
  if (arr[0] < arr[i]) {
    arr[0] = arr[i];
  }
}
```

Finding the smallest element in an array

In this program, we need to find out the smallest element present in the array. This can be achieved by maintaining a variable min which initially will hold the value of the first element. Loop through the array by comparing the value of min with elements of the array. If any of the element's value is less than min, store the value of the element in min.

Consider above array. Initially, min will hold the value 25. In the 1st iteration, min will be compared with 11. Since 11 is less than 25. Min will hold the value 11. In a 2nd iteration, 11 will be compared with 7. Now, 7 is less than 11. So, min will take the value 7. Continue this process until the end of the array is reached. At last, min will hold the smallest value element in the array.

ALGORITHM:

- **STEP 1:** START
- **STEP 2:** INITIALIZE arr[] = {25, 11, 7, 75, 56}
- **STEP 3:** length= sizeof(arr)/sizeof(arr[0])
- **STEP 4:** min = arr[0]
- **STEP 5:** SET i=0. REPEAT STEP 6 and STEP 7 UNTIL i<length
- **STEP 6:** if(arr[i]<min)   min=arr[i]
- **STEP 7:** i=i+1.
- **STEP 8:** PRINT "Smallest element present in given array:" by assigning min
- **STEP 9:** RETURN 0.
- **STEP 10:** END.

PROGRAM:

```
#include <stdio.h>

int main()
{
   //Initialize array
   int arr[] = {25, 11, 7, 75, 56};

   //Calculate length of array arr
   int length = sizeof(arr)/sizeof(arr[0]);
```

```

    //Initialize min with first element of array.
    int min = arr[0];

    //Loop through the array
    for (int i = 0; i < length; i++) {
        //Compare elements of array with min
        if(arr[i] < min)
            min = arr[i];
    }
    printf("Smallest element present in given array: %d\n", min);
    return 0;
}
```

Output:

```
Smallest element present in given array: 7
```

Two dimensional arrays

Two dimensional array is known as matrix. The array declaration in both the array i.e.in single dimensional array single subscript is used and in two dimensional array two subscripts are is used.

Its syntax is

Data-type array name[row][column];

Or we can say 2-d array is a collection of 1-D array placed one below the other.

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

int a[2][3];

Total no of elements=row*column is 2*3 =6

It means the matrix consist of 2 rows and 3 columns

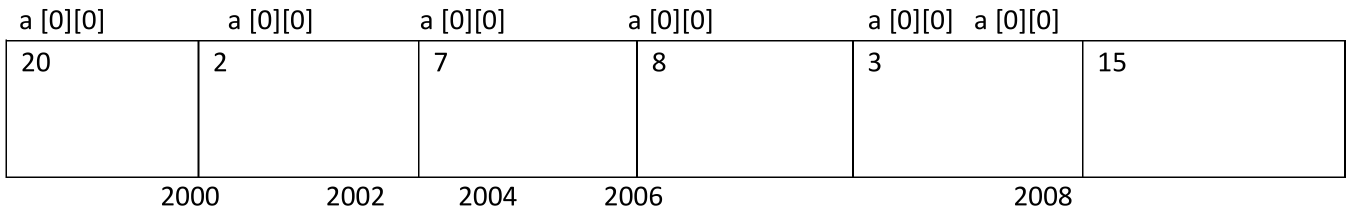
For example:-

```
 20     2      7
8       3       15
```

Positions of 2-D array elements in an array are as below

00   01   02 10

       11   12



Accessing 2-d array /processing 2-d arrays

For processing 2-d array, we use two nested for loops. The outer for loop corresponds

to the row and the inner for loop corresponds to the column. For example int a[4][5];

for reading value:-

```
        for(i=0;i<4;i++)

        { for(j=0;j<5;j++)

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

                }

        }
```

For displaying value:for(i=0;i<4;i++)

```
        { for(j=0;j<5;j++)

                {

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

                }

        }
```

Initialization of 2-d array:

2-D array can be initialized in a way similar to that of 1-D array. for example:int

mat[4][3]={11,12,13,14,15,16,17,18,19,20,21,22};

These values are assigned to the elements row wise, so the values of elements after this initialization are

Mat[0][0]=11, Mat[1][0]=14, Mat[2][0]=17 Mat[3][0]=20

Mat[0][1]=12, Mat[1][1]=15, Mat[2][1]=18 Mat[3][1]=21

Mat[0][2]=13, Mat[1][2]=16, Mat[2][2]=19 Mat[3][2]=22

While initializing we can group the elements row wise using inner braces. for example:-

```
   int mat[4][3]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};
```

And while initializing , it is necessary to mention the 2nd dimension where 1st dimension is optional.

```
int mat[][3];

int mat[2][3];
```

```
int mat[][];
                  } invalid
int mat[2][];
```

If we initialize an array as

```
   int mat[4][3]={{11},{12,13},{14,15,16},{17}};
```

Then the compiler will assume its all rest value as 0,which are not defined.

Mat[0][0]=11, Mat[1][0]=12, Mat[2][0]=14, Mat[3][0]=17

Mat[0][1]=0, Mat[1][1]=13, Mat[2][1]=15 Mat[3][1]=0

Mat[0][2]=0, Mat[1][2]=0, Mat[2][2]=16, Mat[3][2]=0

In memory map whether it is 1-D or 2-D, elements are stored in one contiguous manner.

We can also give the size of the 2-D array by using symbolic constant Such as

```
          #define ROW 2;
            #define COLUMN 3;        int
   mat[ROW][COLUMN];
```

Addition of two matrices

Algorithm For C Program to Add Two Matrices :

To add two matrices the number of row and number of columns in both the matrices must be same the only addition can take place.

```
Step 1: Start

Step 2: Declare matrix mat1[row][col];
        and matrix mat2[row][col];
        and matrix sum[row][col]; row= no. of rows, col= no. of columns

Step 3: Read row, col, mat1[][] and mat2[][]

Step 4: Declare variable i=0, j=0

Step 5: Repeat until i < row

      5.1: Repeat until j < col

           sum[i][j]=mat1[i][j] + mat2[i][j]

           Set j=j+1

      5.2: Set i=i+1

Step 6: sum is the required matrix after addition

Step 7: Stop
```

In the above algorithm,

- using scanf( ) will take the input from the user rows,columns,elements of both the matrix.
- we will the add the element of matrix 1 and matrix 2 in third matrix let say sum matrix.
- printf( ) will print the output on the screen
- we have to use the for loop to input the element of matrix ,sum the elements of matrices and print the final result.

Example: C Program to Add Two Matrices :

Run

```
#include <stdio.h>
int main ()
{
  int mat1[3][3] = { {0, 1, 2}, {3, 4, 5}, {6, 7, 8} };
  int mat2[3][3] = { {9, 10, 11}, {12, 13, 14}, {15, 16, 17} };
  int sum[3][3], i, j;

  printf ("matrix 1 is :\n");
  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 3; j++)
        {
          printf ("%d   ", mat1[i][j]);
          if (j == 3 - 1)
            {
              printf ("\n\n");
            }
        }
    }

  printf ("matrix 2 is :\n");
  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 3; j++)
        {
          printf ("%d   ", mat2[i][j]);
          if (j == 3 - 1)
            {
              printf ("\n\n");
            }
        }
    }
  // adding two matrices
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      {
       sum[i][j] = mat1[i][j] + mat2[i][j];
      }

  // printing the sum 0f two matrices
  printf ("\nSum of two matrices: \n");
  for (i = 0; i < 3; i++)
    {
      for (j = 0; j < 3; j++)
        {
          printf ("%d   ", sum[i][j]);
          if (j == 3 - 1)
            {
              printf ("\n\n");
            }
        }
    }

  return 0;
}
```

Output :

```
matrix 1 is :
0   1   2

3   4   5

6   7   8

matrix 2 is :
9   10   11

12   13   14

15   16   17


Sum of two matrices:
9   11   13

15   17   19

21   23   25
```

Matrix multiplication in C

Matrix multiplication in C: We can add, subtract, multiply and divide 2 matrices. To do so, we are taking input from the user for row number, column number, first matrix elements and second matrix elements. Then we are performing multiplication on the matrices entered by the user.

In matrix multiplication *first matrix one row element is multiplied by second matrix all column elements*.

Let's try to understand the matrix multiplication of **2*2 and 3*3** matrices by the figure given below:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

$$A * B = \begin{pmatrix} 1*5 + 2*8 & 1*6 + 2*9 & 1*7 + 2*10 \\ 3*5 + 4*8 & 3*6 + 4*9 & 3*7 + 4*10 \end{pmatrix}$$

$$A * B = \begin{pmatrix} 21 & 24 & 27 \\ 47 & 54 & 61 \end{pmatrix}$$

Let's see the program of matrix multiplication in C.

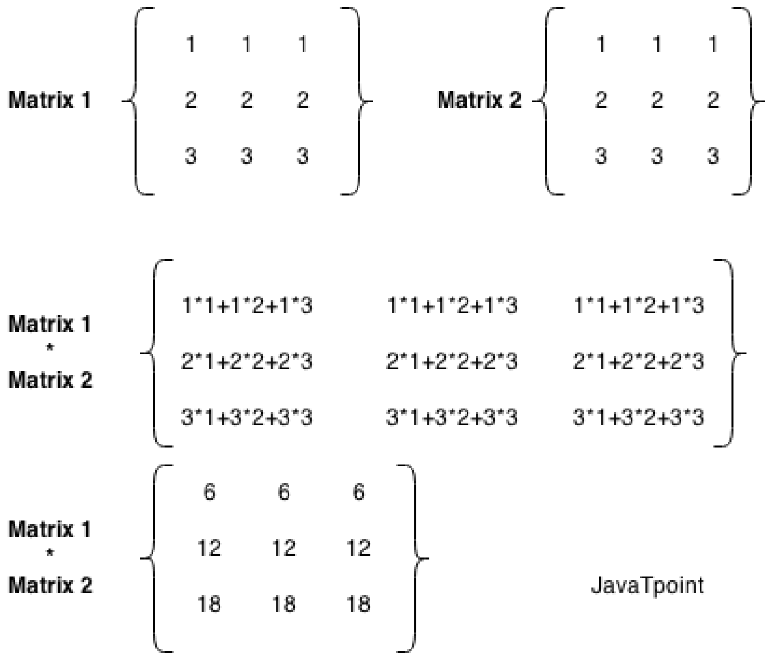
```
#include<stdio.h>
#include<stdlib.h>
int main(){
int a[10][10],b[10][10],mul[10][10],r,c,i,j,k;
system("cls");
printf("enter the number of row=");
scanf("%d",&r);
printf("enter the number of column=");
scanf("%d",&c);
printf("enter the first matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&a[i][j]);
}
}
```

```c
printf("enter the second matrix element=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
scanf("%d",&b[i][j]);
}
}

printf("multiply of the matrix=\n");
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
mul[i][j]=0;
for(k=0;k<c;k++)
{
mul[i][j]+=a[i][k]*b[k][j];
}
}
}
//for printing result
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
{
printf("%d\t",mul[i][j]);
}
printf("\n");
}
return 0;
}
```

Output:

```
enter the number of row=3
enter the number of column=3
enter the first matrix element=
1 1 1
2 2 2
3 3 3
enter the second matrix element=
1 1 1
2 2 2
3 3 3
multiply of the matrix=
6 6 6
12 12 12
18 18 18
```

Let's try to understand the matrix multiplication of **3*3 and 3*3** matrices by the figure given below:



Transpose of a square matrix

Read one matrix (called Matrix 1) using the standard input (`scanf`), and try to transpose the matrix to get the result.

Transpose matrix means the original matrix rows will be changed to columns and columns to rows.

The result is stored in the new matrix, which is the transpose matrix.

Example

Let’s first take a 2x2 matrix and try to transpose it.

Matrix 1:

[1234][1324]

The resultant matrix will look like this :

Transpose matrix:

[1324][1234]

Algorithm

Generally, the transpose of the matrix is denoted by `T`.

When reading the data, we need to use loops.

We also need loops to access the transpose of the matrix.

After transposing the elements, we store the data in the resultant matrix.

The pseudo code looks like this:

```
for(i=0;i<N; i++)

{

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

  {

    Transpose[j][i] = Matrix[i][j];

  }
}
```

Code

To run the code below, first give the input value of `n` (size), followed by values of the matrix.

After giving all the inputs, we'll get the resultant matrix which is the transpose of Matrix 1.

Note: For a 2x2 matrix, we can give the values of a matrix as follows: 2 1 2 3 4

Here, 2 is the value of `n` and we have 4 values for the matrix. Here, "2 1 2 3 4" would be the input.

```
#include<stdio.h>

int main()

{

                int i, j,  N; // Declaration of variables.

                scanf("%d", &N); // Reading the value of n which represnts size of the matrix.

                int Matrix[N][N], Transpose[N][N]; // Declaring the matrices.

                //Reading the Matrix

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

    {

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

        {

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

        }

    }

    // Transposing the matrix

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

     {

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

         {

            Transpose[j][i] =Matrix[i][j]; // In this step we interchange the rows ans columns.

         }
    }
   // Printing the matrix
   for(i=0;i<N;i++)
   {
      for(j=0;j<N;j++)
      {
         printf("%d ",Transpose[i][j]);
      }
      printf("\n");
   }
            return 0;
}
```

Enter the input below

Run

Transpose of a matrix

Explanation

- Line 4: The N represents the dimensions of the matrix.
- Line 6: We declare two matrices. The matrix Matrix[n][n] represents the original matrix and the matrix Transpose[n][n] represents the transpose of the original matrix.
    - First, we take the value of N, which represents the size of the matrix.
- Line 8: We read the first matrix using the for loop.
- Line 16: After reading the matrix, we transpose the matrix. In this step, we interchange the rows with columns or columns with rows.
- Line 24: This transposition of the matrix is put into the resultant matrix, and finally, we use the for loop to print the final resultant matrix.

String

Array of character is called a string. It is always terminated by the NULL character. String is a one dimensional array of character. We can initialize the string as

char name[]={‘j’,’o’,’h’,’n’,’\o’};

Here each character occupies 1 byte of memory and last character is always NULL character. Where ’\o’ and 0 (zero) are not same, where **ASCII** value of ‘\o’ is 0 and ASCII value of 0 is 48. Array elements of character array are also stored in contiguous memory allocation.

From the above we can represent as;

J	o	h	N	‘\o’

The terminating NULL is important because it is only the way that the function that work with string can know, where string end. String can also be **initialized** as;

char name[]=”John”;

Here the NULL character is not necessary and the compiler will assume it automatically.

String constant (string literal)

A string constant is a set of character that enclosed within the double quotes and is also called a literal. Whenever a string constant is written anywhere in a program it is stored somewhere in a memory as an array of characters terminated by a NULL character (‘\o’).

Example – “m”

“Tajmahal”

“My age is %d and height is %f\n”

The string constant itself becomes a pointer to the first character in array.

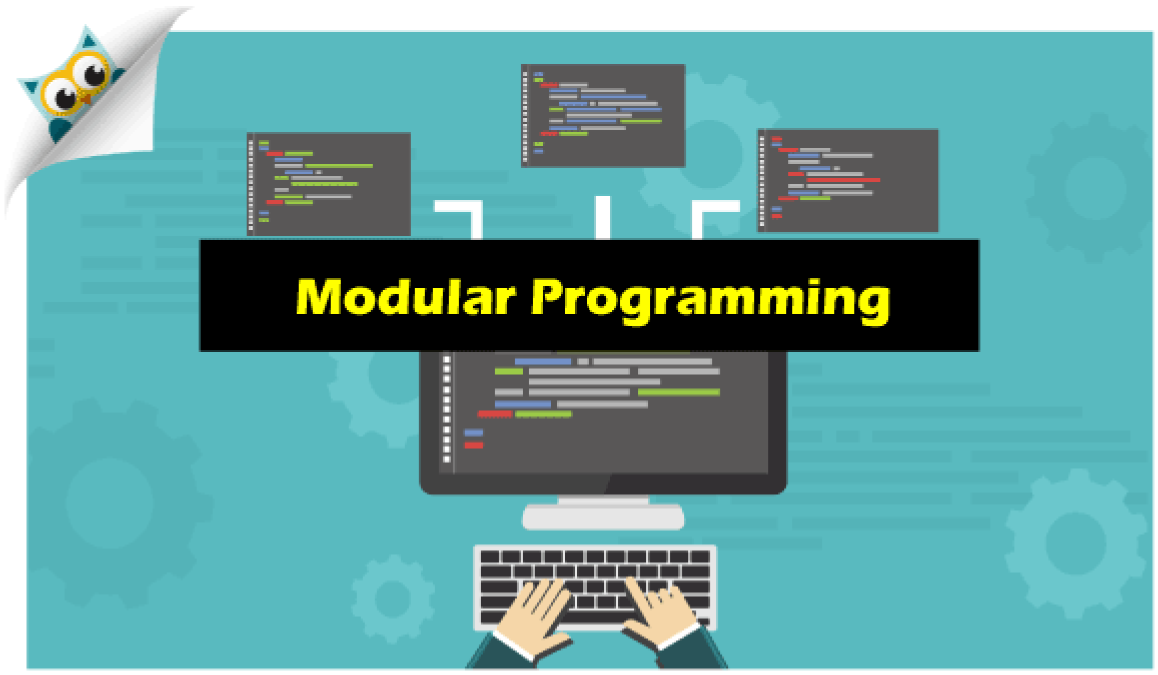
Example-char crr[20]=”Taj mahal”;

1000	1001	1002	1003	1004	1005	1006	1007	100	
1009	T	a	j	M	A	H	a	l	\o

It is called base address.

What is modular programming?

The concept of modular programming originated in the 1960s to help users. Programmers began to divide the more extensive programs into smaller parts. Though the concept of modular programming is six decades old, it is the most convenient programming method.



Definition

Modular programming is defined as a software design technique that focuses on separating the program functionality into independent, interchangeable methods/modules. Each of them contains everything needed to execute only one aspect of functionality.

Talking of modularity in terms of files and repositories, modularity can be on different levels -

- Libraries in projects
- Function in the files
- Files in the libraries or repositories

Modularity is all about making blocks, and each block is made with the help of other blocks. Every block in itself is solid and testable and can be stacked together to create an entire application. Therefore, thinking about the concept of modularity is also like building the whole architecture of the application.

Examples of modular programming languages - All the object-oriented programming languages like C++, Java, etc., are modular programming languages.

Module

A module is defined as a part of a software program that contains one or more routines. When we merge one or more modules, it makes up a program. Whenever a product is built on an enterprise level, it is a built-in module, and each module performs different operations and business. Modules are implemented in the program through interfaces. The introduction of modularity allowed programmers to reuse prewritten code with new applications. Modules are created and merged with compilers, in which each module performs a business or routine operation within the program.

For example - SAP(System, Applications, and Products) comprises large modules like finance, payroll, supply chain, etc. In terms of softwares example of a module is Microsoft Word which uses Microsoft paint to help users create drawings and paintings.

Advantages of modular programming

The following are advantages of modular programming -

- **Code is easier to read -** Working on modular programming makes code easier to read because functions perform different tasks as compared to monolithic codes. Sometimes modular programming can be a bit messy if we pass arguments and variables in different functions. The use of modules should be done in a sensible manner so as to avoid any problem. Functions should be neat, clean, and descriptive.
- **Code is easier to test -** In software, some functions perform fewer tasks and also functions that perform numerous tasks. If the software is easily split using modules, it becomes easier

to test. We can also focus on the riskier functions during testing and need more test cases to make it bug-free.

- **Reusability -** There are times where a piece of code is implemented everywhere in our program. Instead of copying and pasting it, again and again, modularity gives us the advantage of reusability so that we can pull our code from anywhere using interfaces or libraries. The concept of reusability also reduces the size of our program.
- **Faster fixes -** Suppose there is an error in the payment options in any application, and the bug needs to be removed. Modularity can be a great help because we know that there will be a separate function that will contain the code of payments, and only that function will only be rectified. Thus using modules to find and fixing bugs becomes much more smooth and maintainable.
- **Low-risk update -** In modular programming, a defined layer of APIs protects things that use it from making changes inside the library. Unless there is a change in the API, there is a low risk for someone's code-breaking. For example, if you didn't have explicit APIs and someone changed a function they thought was only used within that same library (but it was used elsewhere), they could accidentally break something.
- **Easy collaboration -** Different developers work on a single piece of code in the team. There are chances of conflicts when there's a git merge. This conflict can be reduced if the code is split between more functions, files, repos, etc. We can also provide ownership to specific code modules, where a team member can break them down into smaller tasks.

Disadvantages of modular programming

The following are disadvantages of modular programming -

- There is a need for extra time and budget for a product in modular programming.
- It is a challenging task to combine all the modules.
- Careful documentation is required so that other program modules are not affected.
- Some modules may partly repeat the task performed by other modules. Hence, Modular programs need more memory space and extra time for execution.
- Integrating various modules into a single program may not be a task because different people working on the design of different modules may not have the same style.

- It reduces the program's efficiency because testing and debugging are time-consuming, where each function contains a thousand lines of code.

Modular programming is an ancient concept, but it is still a buzzword among developers. For a developer, one must learn to code in modules. There are times when we need to retrieve any code, make a dummy module for testing, and minimize the risk factors. Modular programming is bagged with such features making it essential.

Standard Library of C functions

The Standard Function Library in C is a huge library of sub-libraries, each of which contains the code for several functions. In order to make use of these libraries, link each library in the broader library through the use of header files. The definitions of these functions are present in their respective header files. In order to use these functions, we have to include the header file in the program. Below are some header files with descriptions:

S No.	Header Files	Description
1	<assert.h>	It checks the value of an expression that we expect to be true under normal circumstances. If the expression is a nonzero value, the assert macro does nothing.
2	<complex.h>	A set of functions for manipulating complex numbers.
3	<float.h>	Defines macro constants specifying the implementation-specific properties of the floating-point library.
4	<limits.h>	These limits specify that a variable cannot store any value beyond these limits, for example- An unsigned character can store up to a maximum value of 255.

S No.	Header Files	Description
5	<math.h>	The math.h header defines various mathematical functions and one macro. All the Functions in this library take double as an argument and return double as the result.
6	<stdio.h>	The stdio.h header defines three variable types, several macros, and various function for performing input and output.
7	<time.h>	Defines date and time handling functions.
8	<string.h>	Strings are defined as an array of characters. The difference between a character array and a string is that a string is terminated with a special character ‘\0’.

FUNCTION PROTOTYPE IN C

Introduction:

In C programming, a ***function prototype*** is used to declare the ***signature*** of a function, which includes its ***name, return type***, and ***parameters***. Function prototypes are important because they inform the compiler of the function's interface before it is called, allowing for proper type checking and error handling. In this article, we will discuss the importance of function prototypes in C programming and how they are used.

Why use function prototypes?

Function prototypes are important in C programming for several reasons. One of the most important reasons is that they allow the ***compiler*** to check for errors before the program is actually executed. If a function is called with the wrong number or type of arguments, the compiler will generate an ***error message***, preventing the program from crashing or behaving unexpectedly at runtime.

Another important reason to use function prototypes is to enable modular programming. In C, functions are typically defined in separate files from the main program, and are linked together at compile time. By declaring function prototypes in header files that are included in both the main program and the function definition files, the function can be called from any part of the program without requiring access to the function's implementation details.

Function prototypes also make it easier to read and understand the code. By including the function's signature in the source code, other developers can easily see what the function does, its arguments, and its return type. It makes the code more self-documenting and reduces the likelihood of bugs caused by misunderstandings or misinterpretations of the code.

Syntax of function prototype:

The syntax of a function prototype in C programming is as follows:

```
return_type function_name(parameter_list);
```

The ***return_type*** is the data type that the ***function returns***, such as ***int, float***, or ***char***. The ***function_name*** is the name of the ***function***, and the ***parameter_list*** is a comma-separated list of ***parameters*** that the function takes. Each parameter in the ***parameter_list*** consists of a data type followed by the ***parameter name***.

For example, the following is a function prototype for a function that takes two ***integers*** as arguments and returns their sum:

```
int add(int num1, int num2);
```

In this example, the return type is ***int***, the function name is ***add***, and the parameter list consists of two integers named ***num1*** and ***num2***.

Some important points of function prototype in C:

Function prototypes help to catch errors:

When a ***function prototype*** is included in a C program, the compiler checks that the function is used correctly before running the program. It helps to catch errors early on before the program is executed.

Function prototypes are essential in large programs:

In large programs, it is important to separate concerns between different functions clearly. Function prototypes enable this separation by allowing each function to be developed independently without knowing the implementation details of other functions.

Function prototypes can be declared in header files:

As mentioned earlier, function prototypes are typically declared in header files. Header files are then included in both the main program and the function definition files, making the functions accessible from any part of the program.

Function prototypes can be overloaded:

C does not support function overloading like some other programming languages, but function prototypes can be overloaded by using different argument types and numbers. It enables the same function name to be used for different purposes.

Function prototypes can include default argument values:

C does not support default argument values like some other programming languages, but function prototypes can include optional arguments by using a special syntax. It enables the same function to be used with or without certain arguments.

Function prototypes can be forward-declared:

In some cases, it may be necessary to declare a function prototype before its implementation is available. It is called ***forward declaration*** and can be useful in complex programs where the implementation of a function may not be known at the time of its declaration.

Function Arguments

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

Parameter

- A parameter is a special kind of variable, used in a function to refer to one of the pieces of data provided as input to the function to utilse.
- These pieces of data are called arguments.
- Parameters are Simply Variables.

Formal Parameter

- Parameter Written in Function Definition is Called "Formal Parameter.
- Formal parameters are always variables, while actual parameters do not have to be variables.

Actual Parameter

- Parameter Written in Function Call is Called "Actual Parameter".
- One can use numbers, expressions, or even function calls as actual parameters.

Example

```
void display(int para1)

{ printf( " Number %d " , para1);

}


void main()
{          int num1;
           display(num1);

}
```

In above, para1 is called the Formal Parameter

num1 is called the Actual Parameter.

Parameter list

- A function is declared in the following manner: return-type function-name (parameter-list,...)
  { body... }
- Return-type is the variable type that the function returns.
- This cannot be an array type or a function type.

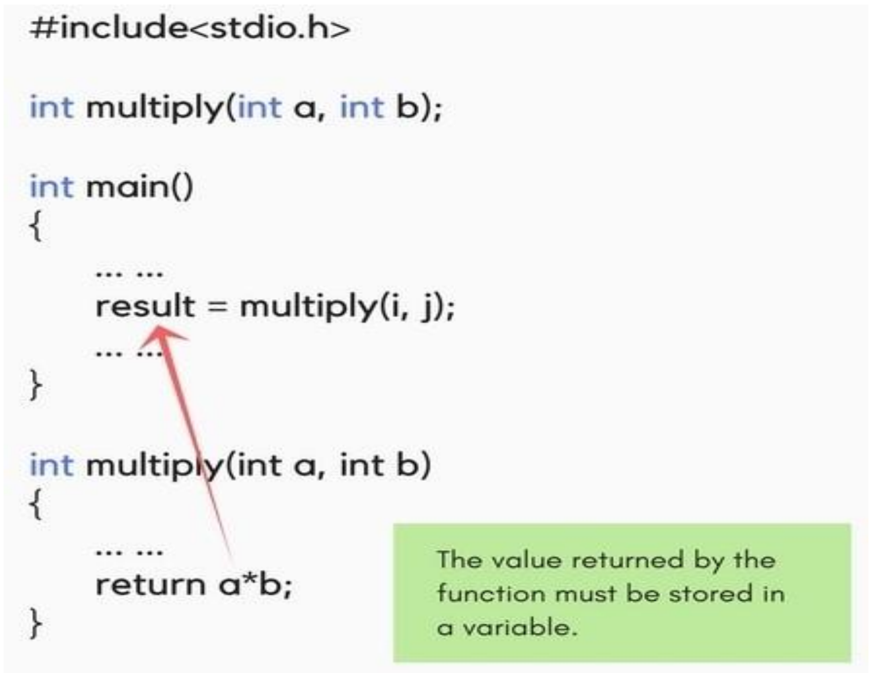
- If not given, then int is assumed.
- Function-name is the name of the function.
- Parameter-list is the list of Formal Arguments Variable.
- Given below are few examples:

  o int func1(int a, int b) /* two argument – int , int */ o float func2(int a, float b) /* two argument – int , float */ o void func3( ) /* No argument . */

The Function Return Type

- The function return type specifies the data type that the function returns to the calling program.
- The return type can be any of C's data types: char, int , long , float , or double .
- One can also define a function that doesn't return a value by using a return type of void.
- Given below are few examples:

  o int func1(...) /* Returns a type int. */ o float func2(...) /* Returns a type float. */ o void func3(...) /* No Returns . */



```
#include<stdio.h>

int multiply(int a, int b);

int main()
{
      ... ...
      result = multiply(i, j);
      ... ...
}

int multiply(int a, int b)
{
      ... ...
      return a*b;
}
```

The value returned by the function must be stored in a variable.

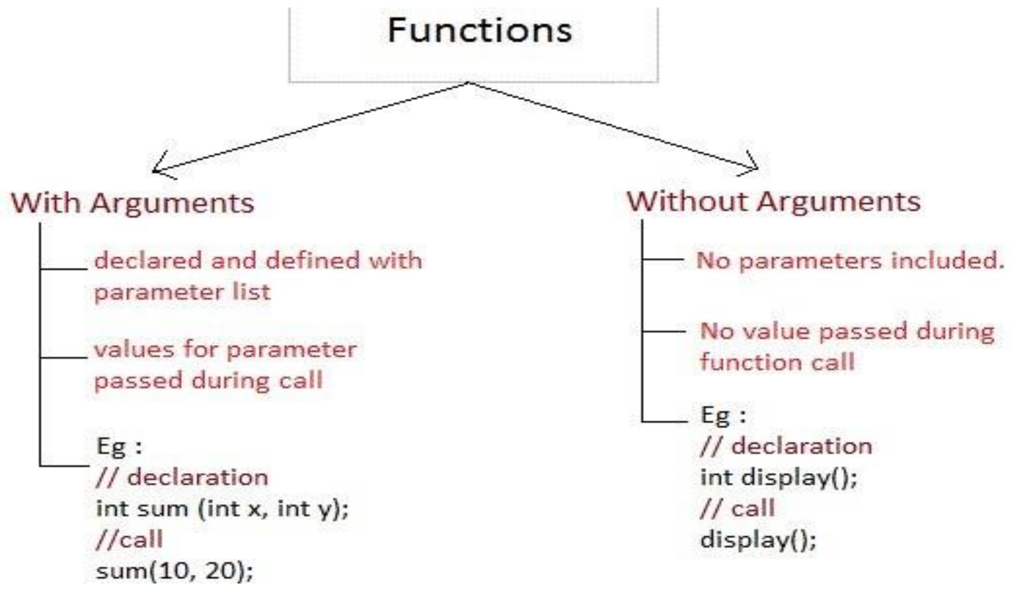
Program – A function that return the maximum between two numbers int max(int num1, int num2) { /* local variable declaration */ int result; if (num1>num2)

```
                    result=num1;

                else

                    result=num2;


            return result;
}

void  main()
{           int r;

             r=max( 10,15); print("Result =
            %d " , r);

}
```

Calling Functions

There are two ways to call a function.

1) Any function can be called by simply using its name and argument list alone in a statement, as i n the following example.
    i) If the function has a return value, it is discarded. wait(12);
2) The second method can be used only with functions that have a return value.
    i) Because these functions evaluate to a value (that is, their return value), they are valid C expressions and can be used anywhere a C expression can be used.
    ii) An expression with a return value used as the right side of an assignment statement.



Passing arguments to C functions

By default, LotusScript® passes arguments to functions and subs by reference. If the argument is an array, a user-defined data type variable, or an object reference variable, you must pass it by reference. In most other cases, you use the ByVal keyword to pass variables by value.

Passing arguments by reference

When an argument is passed by reference, the C function receives a 4-byte pointer to the value area.

In some cases, the actual stack argument is changed to a publicly readable structure. In all cases, the data may be changed by the called function, and the changed value is reflected in LotusScript® variables and in the properties of product objects. For such properties, this change occurs directly after the call has returned.

Data type	**How it is passed to a C function**
String	A 4-byte pointer to the string in the platform-native character set format
Product object (including a collection)	A 4-byte product object handle
Array	A 4-byte pointer to the array stored in the LotusScript® internal array format

Data type	How it is passed to a C function
Type	A 4-byte pointer to the data in the type instance (This may include strings as elements)
User-defined object	A 4-byte pointer to the data in the object (this data may include strings, arrays, lists, product objects, etc., as elements)

Note: Lists cannot be passed by reference. They also cannot be passed by value. Using a list as an argument produces a run-time error.

Passing arguments by value

When an argument is passed by value, the C function receives a copy of the actual value of the argument.

- To specify that the argument should always be passed by value, use the keyword ByVal preceding the parameter declaration for that argument in the Declare statement for the C function.
- To specify that the argument should be passed by value in a particular call to the function, use parentheses around the argument in the call.

The C routine cannot change this value, even if the C routine defines the argument as passed by reference.

Data type	Keyword	How it is passed to a C function
Boolean		A 2-byte Integer, of value 0 or -1, is pushed on the call stack.
Byte		A 1-byte Integer value is pushed on the call stack.
Integer		A 2-byte Integer value is pushed on the call stack.
Long		A 4-byte Long value is pushed on the call stack.
Single		A 4-byte Single value is pushed on the call stack.
Double		An 8-byte Double value is pushed on the call stack.
Currency		An 8-byte value, in the LotusScript® internal Currency format, is pushed on the call stack.
String		A 4-byte pointer to the characters is pushed on the call stack. The C function should not write to memory beyond the end of the string. If the call is made with a variable, changes to the string by the C function are reflected in the variable. This is not true for a string argument to a LotusScript® function declared as ByVal.
Variant		A 16-byte structure, in the LotusScript® format for Variants, is pushed on the call stack.
Product object		A 4-byte product object handle is pushed on the call stack.

Data type	Keyword	How it is passed to a C function
Any		The number of bytes of data in the argument is pushed on the call stack. For example, the argument contains a Long value, then the called function receives 4 bytes. The function may receive a different number of bytes at run time.

No other data types--arrays, lists, fixed-length strings, types, classes, or voids--can be passed by value. It is a run-time error to use these types as arguments.

Any of the data types that can be passed by value can also be passed by reference.

The argument ByVal 0& specifies a null pointer to a C function, when the argument is declared as Any.

Example

```
Declare Sub SemiCopy Lib "mylib.dll" _
   (valPtr As Long, ByVal iVal As Long)
Dim vTestA As Long, vTestB As Long
vTestA = 1
vTestB = 2

SemiCopy vTestA, vTestB
' The C function named SemiCopy receives a 4-byte pointer to a
' 2-byte integer containing the value of vTestA, and a 2-byte
' integer containing the value of vTestB.
' Since vTestA is passed by reference, SemiCopy can dereference
' the 4-byte pointer and assign any 2-byte integer to that
' location. When control returns to LotusScript, vTestA
' contains the modified value. Since vTestB was passed by
' value, any changes made by the C function are not reflected
' in vTestB after the function call.
```

Recursion and Recursive Function in C

In C language, recursion refers to the process in which a function repeatedly calls its multiple copies when working on some very small problem. Here, the functions that keep calling themselves repeatedly in a program are known as the *recursive functions*, and such types of functions in which the recursive functions call themselves are known as recursive calls.

The process of recursion in the C language consists of multiple recursive calls. And, it is also a prerequisite that you choose a termination condition for the recursive function- or else, the program will be stuck in a loop.

Pseudocode for Recursive Functions in C

Let us check the pseudocode used for writing the recursive function in any code:

```
if (base_test)

{

return given_value;

}

else if (another_base_test)

{

return other_given_value;

}

else

{

// Giving a Statement here;

recursive call;

}
```

Working of a Recursive Function in C

It is simple to understand how a recursive function works in the C language. It involves certain tasks and divides them into various subtasks. Some of the subtasks consist of termination conditions/conditions. These subtasks need to satisfy these conditions to terminate the program. Else, as discussed above, a never-ending loop will be created.

Next, the process of recursion finally stops, and we then get to derive the final result from the recursive function. Here, we also have the concept of the *base case*. A base case refers to the case where the function doesn't recur, and thus, we have the *base case*. There are various instances when the recursive function tries to perform a subtask by repeatedly calling itself. It is known as the *recursive case*.

Let us take a look at the format used for writing the recursive function in C.

Examples

Here, we will write a C program that prints the 10th values of the Fibonacci series in the form of output.

```
#include<stdio.h>

int numberfibonacci(int);

void main ()

{

int a,b;

printf("Please enter the value of the n number here : ");

scanf("%d",&a);

b= numberfibonacci(a);

printf("%d",b);

}

int numberfibonacci(int a)

{

if (a==0)

{

return 0;
}

else if (a == 1)

{

return 1;

}

else

{

return numberfibonacci(a-1) + numberfibonacci(a-2);

}

}
```

The output generated here from the code mentioned above would be:

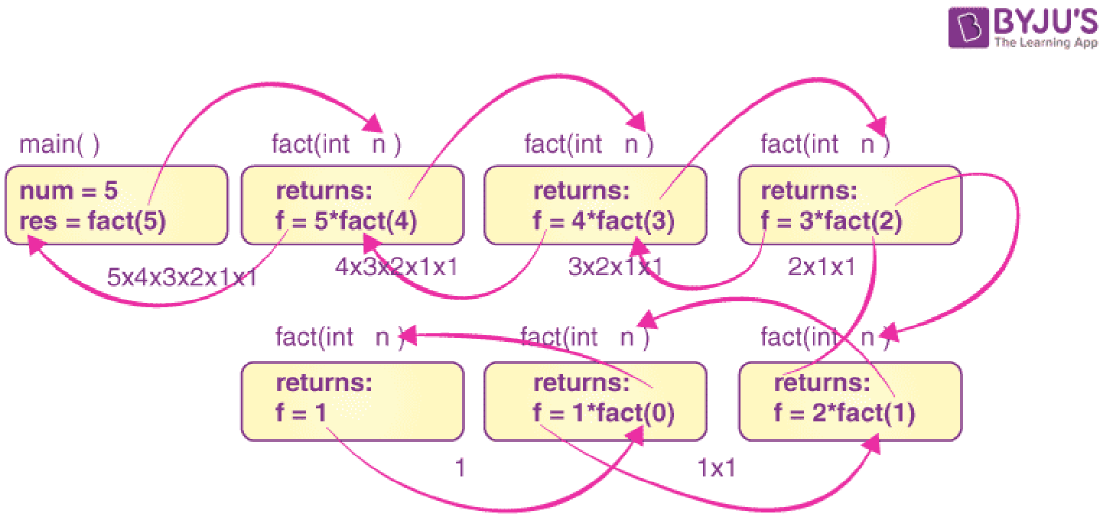
```
Please enter the value of the n number here : 10

55
```

Let us take a look at another example. We will write a program in C that finds the factorial of an available number using the recursive function and not loops.

```
#include<stdio.h>

#include<conio.h>

int fact(int a); /* Definition of Function */

void main()

{

int number, result;

clrscr();
printf(“Please enter a non-negative number here: “);
scanf(“%d”,&number);
result = factorial(number); /* Function Calling in a Normal way */
printf(“%d! = %d” ,number ,result);
getch();
}
int factorial(int a) /* Definition of Function */
{
int x=1;
if(a <= 0)
{
return(1);
}
else
{
x = a * factorial(a-1); /* Function Call Recursively as the fact() calls itself in the program */
return(x);
}
}
```

The output generated here from the code mentioned above would be: Please enter a non-negative number here: 5

5! = 120



Pros and Cons of Using Recursive Functions in C

The recursive functions lead to a very short recursion code in the C language. These are much shorter as compared to the iterative codes- and thus, pretty confusing and tricky to understand. Therefore, we generally avoid using the recursive functions unless in need.

Every problem that occurs in C can't be solved by recursion. This process only comes in handy when we perform certain tasks that need to be defined with similar types of subtasks. Not every problem needs to do that. For example, searching, sorting, and traversal problems become easy to solve with recursion in the C language.

The iterative solutions are much easier to understand and use, as well as efficient when compared to the process of recursion. On top of that, any function that we generally solve recursively also has the scope to be solved iteratively. Thus, iteration is more preferable. Yet, some programs are better solved by recursive functions in C, such as the Fibonacci series, Factorial finding, Tower of Hanoi, etc.

Arrays as Function Arguments

If you want to pass a single-dimension array as an argument in a function, you would have to declare a formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received. Similarly, you can pass multi-dimensional arrays as formal parameters.

Way-1

Formal parameters as a pointer −

```
void myFunction(int *param) {
  .
  .
  .
}
```

Way-2

Formal parameters as a sized array −

```
void myFunction(int param[10]) {
  .
  .
  .
}
```

Way-3

Formal parameters as an unsized array −

```
void myFunction(int param[]) {
  .
  .
  .
}
```

Example

Now, consider the following function, which takes an array as an argument along with another argument and based on the passed arguments, it returns the average of the numbers passed through the array as follows −

```
double getAverage(int arr[], int size) {

  int i;
  double avg;
  double sum = 0;
  for (i = 0; i < size; ++i) {
    sum += arr[i];
  }

  avg = sum / size;

  return avg;
}
```

Now, let us call the above function as follows −

```
#include <stdio.h>

/* function declaration */
double getAverage(int arr[], int size);

int main () {

  /* an int array with 5 elements */
  int balance[5] = {1000, 2, 3, 17, 50};
  double avg;

  /* pass pointer to the array as an argument */
  avg = getAverage( balance, 5 ) ;

  /* output the returned value */
  printf( "Average value is: %f ", avg );

  return 0;
}
```

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

```
Average value is: 214.400000
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C performs no bounds checking for formal parameters.