CHAPTER 8

Arrays

The variables that we have used till now are capable of storing only one value at a time. Consider a situation when we want to store and display the age of 100 employees. For this we have to do the following-

1. Declare 100 different variables to store the age of employees.
2. Assign a value to each variable.
3. Display the value of each variable.

Although we can perform our task by the above three steps, it would be difficult to handle so many variables in the program and the program would become very lengthy. The concept of arrays is useful in these types of situations where we can group similar type of data items.

An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the elements may be any valid data type like `char`, `int` or `float`. The elements of array share the same variable name but each element has a different index number known as subscript.

For the above problem we can take an array variable `age` of size 100 and of type `int`. The size of this array variable is 100 so it is capable of storing 100 integer values. The individual elements of this array are-

```
age[0], age[1], age[2], age[3], age[4], .....age[98], age[99]
```

In C the subscripts start from zero, so `age[0]` is the first element, `age[1]` is the second element of array and so on.

Arrays can be single dimensional or multidimensional. The number of subscripts determines the dimension of array. A one-dimensional array has one subscript, two dimensional array has two subscripts and so on. The one-dimensional arrays are known as vectors and two-dimensional arrays are known as matrices. In this chapter first we will study about single dimensional arrays and then move on to multi dimensional arrays.

8.1 One Dimensional Array

8.1.1 Declaration of 1-D Array

Like other simple variables, arrays should also be declared before they are used in the program. The syntax for declaration of an array is-

```
data_type array_name[size];
```

Here `array_name` denotes the name of the array and it can be any valid C identifier, `data_type` is the data type of the elements of array. The size of the array specifies the number of elements that can be stored in the array. It may be a positive integer constant or constant integer expression. Here are some examples of array declarations-

```
int age[100];
float salary[15];
char grade[20];
```

Here `age` is an array of type `int`, which can store 100 elements of type `int`. The array `salary` is a `float` type array of size 15, can hold values of type `float` and third one is a character type array of size 20, can hold characters. The individual elements of the above arrays are-

```
age[0], age[1], age[2],.....age[99]
salary[0], salary[1], salary[2],.....salary[14]
grade[0], grade[1], grade[2],.....grade[19]
```

When an array is declared, the compiler allocates space in memory sufficient to hold all the elements of the array, so the size of array should be known at the compile time. Hence we can't use variables for specifying the size of array in the declaration. The symbolic constants can be used to specify the size of array. For example-

```
#define SIZE 10
int main(void)
{
    int size = 15;
    float salary[SIZE];    /*Valid*/
    int marks[size];    /*Not valid*/
    .....
    .....
}
```

The use of symbolic constant to specify the size of array makes it convenient to modify the program if the size of array is to be changed later, because the size has to be changed only at one place, in the `#define` directive.

8.1.2 Accessing 1-D Array Elements

The elements of an array can be accessed by specifying the array name followed by subscript in brackets. In C, the array subscripts start from 0. Hence if there is an array of size 5, the valid subscripts will be from 0 to 4. The last valid subscript is one less than the size of the array. This last valid subscript is known as the upper bound of the array and 0 is known as the lower bound of the array. Let us take an array-

```
int arr[5];    /*Size of array arr is 5, can hold five integer elements*/
```

The elements of this array are-

```
arr[0], arr[1], arr[2], arr[3], arr[4]
```

Here 0 is the lower bound and 4 is the upper bound of the array.

The subscript can be any expression that yields an integer value. It can be any integer constant, integer variable, integer expression or return value(int) from a function call. For example, if `i` and `j` are integer variables then these are some valid subscripted array elements-

```
arr[3]    arr[i]    arr[i+j]    arr[2*j]    arr[i++]
```

A subscripted array element is treated as any other variable in the program. We can store values in them, print their values or perform any operation that is valid for any simple variable of the same data type. For example if `arr` and `sal` are two arrays of sizes 5 and 10 respectively, then these are valid statements-

```
int arr[5];
float sal[10];
int i;
scanf("%d",&arr[1]);    /*input value into arr[1]*/
printf("%f",sal[3]);    /*print value of sal[3]*/
arr[4]=25;              /*assign a value to arr[4]*/
arr[4]++;               /*Increment the value of arr[4] by 1*/
sal[5]+=200;            /*Add 200 to sal[5]*/
sum=arr[0]+arr[1]+arr[2]+arr[3]+arr[4];/*Add all the values of array arr[5]*/
i=2;
scanf("%f",&sal[i]);    /*Input value into sal[2]*/
printf("%f",sal[i]);    /*Print value of sal[2]*/
printf("%f",sal[i++]); /*Print value of sal[2] and increment the value of i*/
```

There is no check on bounds of the array. For example, if we have an array `arr` of size 5, the valid subscripts are only 0, 1, 2, 3, 4 and if someone tries to access elements beyond these subscripts, like `arr[5]` , `arr[10]`, the compiler will not show any error message but this may lead to run time errors. So it is the responsibility of programmer to provide array bounds checking wherever needed.

8.1.3 Processing 1-D Arrays

For processing arrays we generally use a `for` loop and the loop variable is used at the place of subscript. The initial value of loop variable is taken 0 since array subscripts start from zero. The loop variable is increased by 1 each time so that we can access and process the next element in the array. The total number of passes in the loop will be equal to the number of elements in the array and in each pass we will process one element. Suppose `arr` is an array of type `int`-

(i) Reading values in `arr`

```
for(i=0; i<10; i++)
        scanf("%d",&arr[i]);
```

(ii) Displaying values of `arr`

```
for(i=0; i<10; i++)
        printf("%d ",arr[i]);
```

(iii) Adding all the elements of `arr`

```
sum=0;
for(i=0; i<10; i++)
        sum+=arr[i];
```

```
/*P8.1 Program to input values into an array and display them*/
#include<stdio.h>
int main(void)
{
        int arr[5],i;
        for(i=0; i<5; i++)
        {
                printf("Enter a value for arr[%d] : ",i);
                scanf("%d",&arr[i]);
        }
        printf("The array elements are : \n");
        for(i=0; i<5; i++)
                printf("%d\t",arr[i]);
        printf("\n");
        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
```

```
/*P8.2 Program to add elements of an array*/
#include<stdio.h>
int main(void)
{
        int arr[10],i,sum=0;
        for(i=0; i<10; i++)
        {
                printf("Enter a value for arr[%d] : ",i);
                scanf("%d",&arr[i]);
                sum+=arr[i];
        }
        printf("Sum=%d\n",sum);
        return 0;
}
```

```
/*P8.3 Program to count even and odd numbers in an array*/
#include<stdio.h>
#define SIZE 10
int main(void)
{
```

```
int arr[SIZE],i,even=0,odd=0;
for(i=0; i<SIZE; i++)
{       printf("Enter a value for arr[%d] : ",i);
        scanf("%d",&arr[i]);
        if(arr[i]%2 == 0)
                even++;
        else
                odd++;

}
printf("Even numbers=%d, Odd numbers=%d\n",even,odd);
return 0;
}
```

8.1.4 Initialization of 1-D Array

After declaration, the elements of a local array have garbage value while the elements of global and static arrays are automatically initialized to zero. We can explicitly initialize arrays at the time of declaration. The syntax for initialization of an array is-

```
data_type array_name[size]={value1, value2.....valueN};
```

Here `array_name` is the name of the array variable, `size` is the size of the array and `value1, value2,.....valueN` are the constant values known as initializers, which are assigned to the array elements one after another. These values are separated by commas and there is a semicolon after the ending braces. For example-

```
int marks[5]={50,85,70,65,95};
```

The values of the array elements after this initialization are-

```
marks[0]:50, marks[1]:85, marks[2]:70, marks[3]:65, marks[4]:95
```

While initializing a 1-D array, it is optional to specify the size of the 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};
float sal[]={25.5,38.5,24.7};
```

Here the size of array `marks` is assumed to be 6 and that of `sal` is assumed to be 3.

If during initialization the number of initializers is less than the 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 array is 5 while there are only 2 initializers. After this initialization, the value of the elements are-

```
marks[0]:99,  marks[1]:78,  marks[2]:0,  marks[3]:0,  marks[4]:0
```

So if we initialize an array like this-

```
int arr[100]={0};
```

then all the elements of `arr` will be initialized to zero.

If the number of initializers is more than the size given in brackets then compiler will show an error. For example-

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

We can't copy all the elements of an array to another array by simply assigning it to the other array. For example if we have two arrays `a` and `b`, then-

```
int a[5]={1,2,3,4,5};
int b[5];
b=a;    /*Not valid*/
```

We will have to copy all the elements of array one by one, using a `for` loop.

```
for(i=0; i<5; i++)
        b[i]=a[i];
```

In the following program we will find out the largest and smallest number in an integer array.

```
/*P8.4 Program to find the largest and smallest number in an array*/
#include<stdio.h>
int main(void)
{
        int i,arr[10]={2,5,4,1,8,9,11,6,3,7};
        int small,large;
        small=large=arr[0];
        for(i=1; i<10; i++)
        {
                if(arr[i] < small)
                        small=arr[i];
                if(arr[i] > large)
                        large=arr[i];
        }
        printf("Smallest=%d,Largest=%d\n",small,large);
        return 0;
}
```

We have taken the value of first element as the initial value of `small` and `large`. Inside the `for` loop, we will start comparing from second element onwards so this time we have started the loop from 1 instead of 0.

The following program will reverse the elements of an array.

```
/*P8.5 Program to reverse the elements of an array*/
#include<stdio.h>
int main(void)
{
        int i,j,temp,arr[10]={1,2,3,4,5,6,7,8,9,10};
        for(i=0,j=9; i<j; i++,j--)
        {
                temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
        }
        printf("After reversing, the array is : ");
        for(i=0; i<10; i++)
                printf("%d  ",arr[i]);
        printf("\n");
        return 0;
}
```

In the `for` loop we have used comma operator and taken two variables `i` and `j`. The variable `i` is initialized with the lower bound and `j` is initialized with upper bound. After each pass of the loop, `i` is incremented while `j` is decremented. Inside the loop, `arr[i]` is exchanged with `arr[j]`. So `arr[0]` will be exchanged with `arr[9]`, `arr[1]` with `arr[8]`, `arr[2]` with `arr[7]` and so on.

The next program prints the binary equivalent of a decimal number. The process of obtaining a binary number form decimal number 29 is given below-

	Quotient	Remainder		
29/2	14	1	a[0]	MSB
14/2	7	0	a[1]	
7/2	3	1	a[2]	
3/2	1	1	a[3]	
1/2	0	1	a[4]	LSB

We will store the remainders in an array, and then at last the array is printed in reverse order to get the binary number.

```
/*P8.6 Program to convert a decimal number to binary number*/
#include<stdio.h>
int main(void)
```

```
    int num,arr[15],i,j;
    printf("Enter a decimal number : ");
    scanf("%d",&num);
    i=0;
    while(num>0)
    {       arr[i] = num%2;
            num/=2;                 /*store the remainder in array*/
            i++;
    }
    printf("Binary number is : ");
    for(j=i-1; j>=0; j--)           /*print the array backwards*/
            printf("%d",arr[j]);
    printf("\n");
    return 0;
}
```

Output :
Enter a decimal number : 29
Binary number is : 11101

8.1.5 1-D Arrays and Functions

8.1.5.1 Passing Individual Array Elements to a Function

We know that an array element is treated as any other simple variable in the program. So like other simple variables, we can pass individual array elements as arguments to a function.

```
/*P8.7 Program to pass array elements to a function*/
#include<stdio.h>
void check(int num);
int main(void)
{
      int arr[10],i;
      printf("Enter the array elements : ");
      for(i=0; i<10; i++)
      {
            scanf("%d",&arr[i]);
            check(arr[i]);
      }
      return 0;
}
void check(int num)
{
      if(num%2==0)
            printf("%d is even\n",num);
      else
            printf("%d is odd\n",num);
}
```

8.1.5.2 Passing whole 1-D Array to a Function

We can pass whole array as an actual argument to a function. The corresponding formal argument should be declared as an array variable of the same data type.

```
int main(void)
{

      int arr[10]
      .....
      .....
      func(arr); /*In function call, array name is specified without brackets*/
      return 0;
}
```

```
func(int val[10])
{
      ------
      ------
}
```

It is optional to specify the size of the array in the formal argument, for example we may write the function definition as-

```
func(int val[])
{
      ------
      ------
}
```

We have studied that changes made in formal arguments do not affect the actual arguments, but this is not the case while passing an array to a function. The mechanism of passing an array to a function is quite different from that of passing a simple variable. We know that in the case of simple variables, the called function creates a copy of the variable and works on it, so any changes made in the function do not affect the original variable. When an array is passed as an actual argument, the called function actually gets access to the original array and works on it, so any changes made inside the function affect the original array. Here is a program in which an array is passed to a function.

```
/*P8.8 Program to understand the effect of passing an array to a function*/
#include<stdio.h>
void func(int val[]);
int main(void)
{
      int i,arr[6]={1,2,3,4,5,6};
      func(arr);
      printf("Contents of array are : ");
      for(i=0; i<6; i++)
            printf("%d ",arr[i]);
      printf("\n");
      return 0;
}
void func(int val[])
{
      int sum=0,i;
      for(i=0; i<6; i++)
      {
            val[i]=val[i]*val[i];
            sum+=val[i];
      }
      printf("Sum of squares=%d\n",sum);
}
```

Output :
Sum of squares=91
Contents of array are : 1 4 9 16 25 36

Here we can see that the changes made to the array inside the called function are reflected in the calling function. The name of the formal argument is different but it refers to the original array.

Since it is not necessary to specify the size of array in the function definition, we can write general functions that can work on arrays of same type but different sizes. For example in the next program, the function `add()` is capable of adding the elements of any size of an integer array.

```
/*P8.9 Program that uses a general function which works on arrays of different sizes*/
#include<stdio.h>
int add(int arr[],int n);
int main(void)
{
      int a[5]={2,4,6,8,10};
      int b[8]={1,3,5,7,9,11,13,15};
      int c[10]={1,2,3,4,5,6,7,8,9,10};
```

```
    printf("Sum of elements of array a : %d\n",add(a,5));
    printf("Sum of elements of array b : %d\n",add(b,8));
    printf("Sum of elements of array c : %d\n",add(c,10));
    return 0;

}
int add(int arr[],int n)
{
    int i,sum=0;
    for(i=0; i<n; i++)
      sum+=arr[i];
    return sum;
}
```

Output :
Sum of elements of array a : 30
Sum of elements of array b : 64
Sum of elements of array c : 55

8.2 Two Dimensional Arrays

8.2.1 Declaration and Accessing Individual Elements of a 2-D array

The syntax of declaration of a 2-D array is similar to that of 1-D arrays, but here we have two subscripts.

```
data_type array_name[rowsize][columnsize];
```

Here `rowsize` specifies the number of rows and `columnsize` represents the number of columns in the array. The total number of elements in the array are `rowsize*columnsize`. For example suppose we have an array `arr` declared as-

```
int arr[4][5];
```

Here `arr` is a 2-D array with 4 rows and 5 columns. The individual elements of this array can be accessed by applying two subscripts, where the first subscript denotes the row number and the second subscript denotes the column number. The starting element of this array is `arr[0][0]` and the last element is `arr[3][4]`. The total number of elements in this array is 4*5 = 20.

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
Row 1	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
Row 2	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]
Row 3	arr[3][0]	arr[3][1]	arr[3][2]	arr[3][3]	arr[3][4]

8.2.2 Processing 2-D Arrays

For processing 2-D arrays, we use two nested `for` loops. The outer `for` loop corresponds to the row and the inner `for` loop corresponds to the column.

```
int arr[4][5];
```

(i) Reading values in `arr`

```
for(i=0; i<4; i++)
      for(j=0; j<5; j++)
            scanf("%d",&arr[i][j]);
```

(ii) Displaying values of `arr`

```
for(i=0; i<4; i++)
      for(j=0; j<5; j++)
            printf("%d ",arr[i][j]);
```

This will print all the elements in the same line. If we want to print the elements of different rows on different lines then we can write like this-

```
for(i=0; i<4; i++)
{
    for(j=0; j<5; j++)
        printf("%d ",arr[i][j]);
```

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

Here the `printf("\n")` statement causes the next row to begin from a new line.

```
/*P8.10 Program to input and display a matrix*/
#define ROW 3
#define COL 4
#include<stdio.h>
int main(void)
{
        int mat[ROW][COL], i, j;
        printf("Enter the elements of the matrix(%dx%d) row-wise :\n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat[i][j]);

        printf("The matrix that you have entered is :\n");
        for(i=0; i<ROW; i++)
        {
                for(j=0; j<COL; j++)
                        printf("%5d",mat[i][j]);
                printf("\n");
        }
        printf("\n");
        return 0;
}
```

8.2.3 Initialization of 2-D Arrays

2-D arrays can be initialized in a way similar to that of 1-D arrays. 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[0][1] : 12	mat[0][2] : 13
mat[1][0] : 14	mat[1][1] : 15	mat[1][2] : 16
mat[2][0] : 17	mat[2][1] : 18	mat[2][2] : 19
mat[3][0] : 20	mat[3][1] : 21	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} };
int mat[4][3] = {
                        {11,12,13},     /*Row 0*/
                        {14,15,16},     /*Row 1*/
                        {17,18,19},     /*Row 2*/
                        {20,21,22}      /*Row 3*/
                };
```

Here the values in the first inner braces will be the values of Row 0, values in the second inner braces will be values of Row 1 and so on. Now consider this array initialization-

```
int mat[4][3] = {
                        {11},           /*Row 0*/
                        {12,13},        /*Row 1*/
                        {14,15,16},     /*Row 2*/
                        {17}            /*Row 3*/
                };
```

The remaining elements in each row will be assigned values 0, so the values of elements will be-

mat[0][0] : 11	mat[0][1] : 0	mat[0][2] : 0
mat[1][0] : 12	mat[1][1] : 13	mat[1][2] : 0
mat[2][0] : 14	mat[2][1] : 15	mat[2][2] : 16
mat[3][0] : 17	mat[3][1] : 0	mat[3][2] : 0

In 2-D arrays, it is optional to specify the first dimension while initializing but the second dimension should always be present. For example-

```
int mat[][3] = {
                {1,10},
                {2,20,200},
                {3},
                {4,40,400}
              };
```

Here first dimension is taken 4 since there are 4 rows in initialization list.

A 2-D array is also known as a matrix. The next program adds two matrices; the order of both the matrices should be same.

```
/*P8.11 Addition of two matrices*/
#define ROW 3
#define COL 4
#include<stdio.h>
int main(void)
{
    int i,j,mat1[ROW][COL],mat2[ROW][COL],mat3[ROW][COL];
    printf("Enter matrix mat1(%dx%d)row-wise :\n",ROW,COL);
    for(i=0; i<ROW; i++)
        for(j=0; j<COL; j++)
            scanf("%d",&mat1[i][j]);
    printf("Enter matrix mat2(%dx%d)row-wise :\n",ROW,COL);
    for(i=0; i<ROW; i++)
        for(j=0; j<COL; j++)
            scanf("%d",&mat2[i][j] );
    /*Addition*/
    for(i=0; i<ROW; i++)
        for(j=0; j<COL; j++)
            mat3[i][j] = mat1[i][j] + mat2[i][j];
    printf("The resultant matrix mat3 is :\n");
    for(i=0; i<ROW; i++)
    {
        for(j=0; j<COL; j++)
            printf("%5d",mat3[i][j]);
        printf("\n");
    }
    return 0;
}
```

Output :

```
Enter matrix mat1(3x4) row-wise :
1 2 8 4
5 6 7 8
3 2 1 4
Enter matrix mat2(3x4) row-wise :
2 5 4 2
1 5 2 6
9 4 7 2
The resultant matrix mat3 is :
  3  7  12  6
  6 11   9 14
 12  6   8  6
```

Now we will write a program to multiply two matrices. Multiplication of matrices requires that the number of columns in first matrix should be equal to the number of rows in second matrix. Each row of first matrix is multiplied with the column of second matrix and then added to get the element of resultant matrix. If we multiply two matrices of order m x n and n x p then the multiplied matrix will be of order m x p. For example-

$$A_{2x2} = \begin{bmatrix} 4 & 5 \\ 3 & 2 \end{bmatrix} \qquad B_{2x3} = \begin{bmatrix} 2 & 6 & 3 \\ -3 & 2 & 4 \end{bmatrix}$$

$$C_{2x3} = \begin{bmatrix} 4*2+5*(-3) & 4*6+5*2 & 4*3+5*4 \\ 3*2+2*(-3) & 3*6+2*2 & 3*3+2*4 \end{bmatrix} = \begin{bmatrix} -7 & 34 & 32 \\ 0 & 22 & 17 \end{bmatrix}$$

```
/*P8.12 Multiplication of two matrices*/
#include<stdio.h>
#define ROW1 3
#define COL1 4
#define ROW2 COL1
#define COL2 2
int main(void)
{
        int mat1[ROW1][COL1],mat2[ROW2][COL2],mat3[ROW1][COL2];
        int i,j,k;
        printf("Enter matrix mat1(%dx%d)row-wise :\n",ROW1,COL1);
        for(i=0; i<ROW1; i++)
                for(j=0; j<COL1; j++)
                        scanf("%d",&mat1[i][j]);
        printf("Enter matrix mat2(%dx%d)row-wise :\n",ROW2,COL2);
        for(i=0; i<ROW2; i++)
                for(j=0; j<COL2; j++)
                        scanf("%d",&mat2[i][j] );
        /*Multiplication*/
        for(i=0; i<ROW1; i++)
                for(j=0; j<COL2; j++)
                {
                        mat3[i][j] = 0;
                        for(k=0; k<COL1; k++)
                                mat3[i][j] += mat1[i][k] * mat2[k][j];
                }
        printf("The Resultant matrix mat3 is :\n");
        for(i=0; i<ROW1; i++)
        {
                for(j=0; j<COL2; j++)
                        printf("%5d",mat3[i][j]);
                printf("\n");
        }
        return 0;
}
```

Output :

```
Enter matrix mat1(3x4)row-wise :
2 1 4 3
5 2 7 1
3 1 4 2
Enter matrix mat2(4x2)row-wise :
1 2
3 4
2 5
6 2
The Resultant matrix mat3 is :
  31  34
  31  55
  26  34
```

The next program finds out the transpose of a matrix. Transpose matrix is defined as the matrix that is obtained by interchanging the rows and columns of a matrix. If a matrix is of mxn order then its transpose matrix will be of order nxm.

```
/*P8.13 Tranpose of matrix*/
#include<stdio.h>
#define ROW 3
#define COL 4
int main(void)
{
        int mat1[ROW][COL], mat2[COL][ROW],i,j;
        printf("Enter matrix mat1(%dx%d) row-wise : \n",ROW,COL);
        for(i=0; i<ROW; i++)
                for(j=0; j<COL; j++)
                        scanf("%d",&mat1[i][j]);
```

```
        for(i=0; i<COL; i++)
                for(j=0; j<ROW; j++)
                        mat2[i][j]=mat1[j][i];

        printf("Transpose of matrix is:\n");
        for(i=0; i<COL; i++)
        {
                for(j=0; j<ROW; j++)
                        printf("%5d",mat2[i][j]);
                printf("\n");
        }
        return 0;
}
```

Output :
Enter matrix mat1(3x4) row-wise :
3 2 1 5
6 5 8 2
9 3 4 1
Tranpose of matrix is:
3 6 9
2 5 3
1 8 4
5 2 1

8.3 Arrays with more than two Dimensions

We will just give a brief overview of three-d arrays. We can think of a three-d array as an array of 2-D arrays. For example if we have an array-

```
int arr[2][4][3];
```

We can think of this as an array which consists of two 2-D arrays where each of those 2-D arrays has 4 rows and 3 columns.

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

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

The individual elements are-
arr[0][0][0], arr[0][0][1], arr[0][0][2], arr[0][1][0].....arr[0][3]2]
arr[1][0][0], arr[1][0][1], arr[1][0][2], arr[1][1][0].....arr[1][3]2]

Total number of elements in the above 3-d array are-

= 2 * 4 * 3
= 24

This array can be initialized as-

```
int arr[2][4][3]={
                {
                    {1,2,3},        /*Matrix 0,Row 0*/
                    {4,5},          /*Matrix 0,Row 1*/
                    {6,7,8},        /*Matrix 0,Row 2*/
                    {9}             /*Matrix 0,Row 3*/
                },
                {
                    {10,11},        /*Matrix 1,Row 0*/
                    {12,13,14},     /*Matrix 1,Row 1*/
                    {15,16}         /*Matrix 1,Row 2*/
                     {17,18,19}     /*Matrix 1,Row 3*/
                }
            };
```

The value of elements after this initialization are-

```
arr[0][0][0] : 1    arr[0][0][1] : 2    arr[0][0][2] : 3
arr[0][1][0] : 4    arr[0][1][1] : 5    arr[0][1][2] : 0
arr[0][2][0] : 6    arr[0][2][1] : 7    arr[0][2][2] : 8
arr[0][3][0] : 9    arr[0][3][1] : 0    arr[0][3][2] : 0

arr[1][0][0] : 10   arr[1][0][1] : 11   arr[1][0][2] : 0
arr[1][1][0] : 12   arr[1][1][1] : 13   arr[1][1][2] : 14
arr[1][2][0] : 15   arr[1][2][1] : 16   arr[1][2][2] : 0
arr[1][3][0] : 17   arr[1][3][1] : 18   arr[1][3][2] : 19
```

Remember that the rule of initialization of multidimensional arrays is that the last subscript varies most frequently and the first subscript varies least frequently.

8.3.1 Multidimensional Array and Functions

Multidimensional arrays can also be passed to functions like 1-D arrays. When passing multidimensional array the first(leftmost) dimension may be omitted but all other dimensions have to be specified in the functio definition. For example it would be invalid to write a function like this-

```
func(int a[],int b[][],int c[][][])   /*Invalid*/
{
        ..............
}
```

In arrays `b` and `c` we can't omit all the dimensions. The correct form is-

```
func(int a[],int b[][4],int c[][3][5])        /*Valid*/
{
        ..............
}
```

8.4 Introduction to Strings

We'll discuss strings in detail in a separate chapter; here is just a brief introduction to strings. In C, strings treated as arrays of type `char` and are terminated by a null character('\0'). This null character has ASCII v zero.

These are the two forms of initialization of a string variable-

```
char str[10]={'I','n','d','i','a','\0'};
char str[10]="India";  /*Here the null character is automatically placed at the end*/
```

8.4.1 Input and output of strings

```
/*P8.14 Input and output of strings using scanf() and printf()*/
#include<stdio.h>
int main(void)
{
        char str[10]="Anpara";
        printf("String is : %s\n",str);
        printf("Enter new value for string  : ");
        scanf("%s",str);
        printf("String is : %s\n",str);
        return 0;
}
```

Output :

```
String is : Anpara
Enter new value for string : Bareilly
String is : Bareilly
```

The next program uses the functions `gets()` and `puts()` for the input and output of strings.

```
/*P8.15 Program for input and output of strings using gets() and puts()*/
#include<stdio.h>
```

```
int main(void)
{
    char str[10];
    printf("Enter a string  : ");
    gets(str);
    printf("String is : ");
    puts(str);
    return 0;
}
```

Output :
Enter a string : New Delhi
String is : New Delhi

Some Additional Problems

Problem 1

Write a program using arrays, that reads a decimal number and converts it to (1) Binary (2) Octal or (3) Hexadecimal depending on user's choice.

```
/*P8.16 Program to convert a decimal number to Binary, octal or hexadecimal*/
#include<stdio.h>
void func(int num,int b);
int main(void)
{
    int num,ch;
    printf("Enter a decimal number : ");
    scanf("%d",&num);
    printf("1.Binary\n2.Octal\n3.Hexadecimal\n");
    printf("Enter your choice : ");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1:
            printf("Binary equivalent is : ");
            func(num,2);
            break;
        case 2:
            printf("Octal equivalent is : ");
            func(num,8);
            break;
        case 3:
            printf("Hexadecimal equivalent is : ");
            func(num,16);
            break;
    }
    printf("\n");
    return 0;
}
void func(int num,int b)
{
    int i=0,j,rem;
    char arr[20];
    while(num>0)
    {
        rem=num%b;
        num/=b;
        if(rem>9 && rem<16)
            arr[i++]=rem-10+'A';
        else
            arr[i++]=rem+'0';
    }
    for(j=i-1; j>=0; j--)
```