# ASSIGNMENT NO. 3

# TITLE: Matrix Operations

# PROBLEM DEFINITION:

WAP in c/c++ where in a class of M students, set A play cricket, set B play badminton. Implement:

1. Set of students who play either cricket or badminton
2. Set of students who play both
3. Set of students who play only cricket
4. Set of students who play only badminton
5. Number of students who play neither cricket nor badminton

# PREREQUISITES:

Concept of arrays, n dimensional arrays and basic syntax of c++ .

# LEARNING OBJECTIVES:

To implement and analyze the concept of arrays, n dimensional arrays and operations on matrix.

# THEORY:

# *Arrays:*

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.  
  
That means that, for example, five values of type int can be declared as an array without having to declare 5 different variables (each with its own identifier). Instead, using an array, the five int values are stored in contiguous memory locations, and all five can be accessed using the same identifier, with the proper index.  
  
For example, an array containing 5 integer values of type int called foo could be represented as:  
  
http://www.cplusplus.com/doc/tutorial/arrays/arrays1.png   
where each blank panel represents an element of the array. In this case, these are values of type int. These elements are numbered from 0 to 4, being 0 the first and 4 the last; In C++, the first element in an array is always numbered with a zero (not a one), no matter its length.  
  
Like a regular variable, an array must be declared before it is used. A typical declaration for an array in C++ is:  
  
type name [elements];  
  
where type is a valid type (such as int, float...), name is a valid identifier and the elements field (which is always enclosed in square brackets []), specifies the length of the array in terms of the number of elements.  
  
Therefore, array with five elements of type int, can be declared as: int foo[5];

# *Initializing arrays:*

By default, regular arrays of *local scope* (for example, those declared within a function) are left uninitialized. This means that none of its elements are set to any particular value; their contents are undetermined at the point the array is declared.  
  
But the elements in an array can be explicitly initialized to specific values when it is declared, by enclosing those initial values in braces {}. For example:

|  |  |  |
| --- | --- | --- |
|  | int foo [5] = { 16, 2, 77, 40, 12071 }; |  |

This statement declares an array that can be represented like this:  
  
http://www.cplusplus.com/doc/tutorial/arrays/arrays2.png   
The number of values between braces {} shall not be greater than the number of elements in the array. For example, in the example above, foo was declared having 5 elements (as specified by the number enclosed in square brackets, []), and the braces {} contained exactly 5 values, one for each element. If declared with less, the remaining elements are set to their default values (which for fundamental types, means they are filled with zeroes). For example:

|  |  |  |
| --- | --- | --- |
|  | int bar [5] = { 10, 20, 30 }; |  |

Will create an array like this:  
  
http://www.cplusplus.com/doc/tutorial/arrays/arrays3.png   
The initializer can even have no values, just the braces:

|  |  |  |
| --- | --- | --- |
|  | int baz [5] = { }; |  |

This creates an array of five int values, each initialized with a value of zero:  
  
http://www.cplusplus.com/doc/tutorial/arrays/arrays4.png   
When an initialization of values is provided for an array, C++ allows the possibility of leaving the square brackets empty[]. In this case, the compiler will assume automatically a size for the array that matches the number of values included between the braces {}:

|  |  |  |
| --- | --- | --- |
|  | int foo [] = { 16, 2, 77, 40, 12071 }; |  |

After this declaration, array foo would be 5 int long, since we have provided 5 initialization values.  
  
Finally, the evolution of C++ has led to the adoption of *universal initialization* also for arrays. Therefore, there is no longer need for the equal sign between the declaration and the initializer. Both these statements are equivalent:

|  |  |  |
| --- | --- | --- |
| 1 2 | int foo[] = { 10, 20, 30 };  int foo[] { 10, 20, 30 }; |  |

Static arrays, and those declared directly in a namespace (outside any function), are always initialized. If no explicit initializer is specified, all the elements are default-initialized (with zeroes, for fundamental types).

# *Accessing the values of an array:*

The values of any of the elements in an array can be accessed just like the value of a regular variable of the same type. The syntax is:  
  
name[index]   
Following the previous examples in which foo had 5 elements and each of those elements was of type int, the name which can be used to refer to each element is the following:  
  
http://www.cplusplus.com/doc/tutorial/arrays/arrays5.png   
For example, the following statement stores the value 75 in the third element of foo:

|  |  |  |
| --- | --- | --- |
|  | foo [2] = 75; |  |

and, for example, the following copies the value of the third element of foo to a variable called x:

|  |  |  |
| --- | --- | --- |
|  | x = foo[2]; |  |

Therefore, the expression foo[2] is itself a variable of type int.  
  
Notice that the third element of foo is specified foo[2], since the first one is foo[0], the second one is foo[1], and therefore, the third one is foo[2]. By this same reason, its last element is foo[4]. Therefore, if we write foo[5], we would be accessing the sixth element of foo, and therefore actually exceeding the size of the array.  
  
In C++, it is syntactically correct to exceed the valid range of indices for an array. This can create problems, since accessing out-of-range elements do not cause errors on compilation, but can cause errors on runtime. The reason for this being allowed will be seen in a later chapter when pointers are introduced.  
  
At this point, it is important to be able to clearly distinguish between the two uses that brackets [] have related to arrays. They perform two different tasks: one is to specify the size of arrays when they are declared; and the second one is to specify indices for concrete array elements when they are accessed. Do not confuse these two possible uses of brackets [] with arrays.

|  |  |  |
| --- | --- | --- |
| 1 2 | int foo[5]; // declaration of a new array  foo[2] = 75; // access to an element of the array. |  |

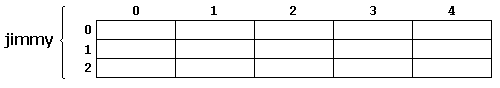
The main difference is that the declaration is preceded by the type of the elements, while the access is not.  
  
Some other valid operations with arrays:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | foo[0] = a;  foo[a] = 75;  b = foo [a+2];  foo[foo[a]] = foo[2] + 5; |  |

### *For example:*

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // arrays example  #include <iostream>  using namespace std;  int foo [] = {16, 2, 77, 40, 12071};  int n, result=0;  int main ()  {  for ( n=0 ; n<5 ; ++n )  {  result += foo[n];  }  cout << result;  return 0;  } | 12206 |  |

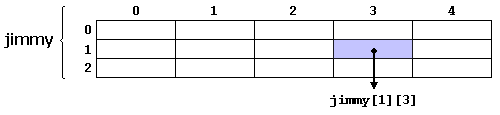
# *Multidimensional arrays:*

Multidimensional arrays can be described as "arrays of arrays". For example, a bidimensional array can be imagined as a two-dimensional table made of elements, all of them of a same uniform data type.  
  
   
jimmy represents a bidimensional array of 3 per 5 elements of type int. The C++ syntax for this is:

|  |  |  |
| --- | --- | --- |
|  | int jimmy [3][5]; |  |

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be: 

|  |  |  |
| --- | --- | --- |
|  | jimmy[1][3] |  |

   
(remember that array indices always begin with zero).  
  
Multidimensional arrays are not limited to two indices (i.e., two dimensions). They can contain as many indices as needed. Although be careful: the amount of memory needed for an array increases exponentially with each dimension. For example:

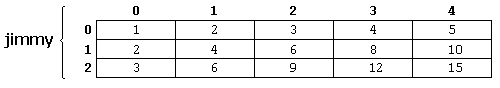
|  |  |  |
| --- | --- | --- |
|  | char century [100][365][24][60][60]; |  |

declares an array with an element of type char for each second in a century. This amounts to more than 3 billion char! So this declaration would consume more than 3 gigabytes of memory!  
  
At the end, multidimensional arrays are just an abstraction for programmers, since the same results can be achieved with a simple array, by multiplying its indices:

|  |  |  |
| --- | --- | --- |
| 1 2 | int jimmy [3][5]; // is equivalent to  int jimmy [15]; // (3 \* 5 = 15) |  |

With the only difference that with multidimensional arrays, the compiler automatically remembers the depth of each imaginary dimension. The following two pieces of code produce the exact same result, but one uses a bidimensional array while the other uses a simple array: 

|  |  |
| --- | --- |
| **multidimensional array** | **pseudo-multidimensional array** |
| #define WIDTH 5  #define HEIGHT 3  int jimmy [HEIGHT][WIDTH];  int n,m;  int main ()  {  for (n=0; n<HEIGHT; n++)  for (m=0; m<WIDTH; m++)  {  jimmy[n][m]=(n+1)\*(m+1);  }  } | #define WIDTH 5  #define HEIGHT 3  int jimmy [HEIGHT \* WIDTH];  int n,m;  int main ()  {  for (n=0; n<HEIGHT; n++)  for (m=0; m<WIDTH; m++)  {  jimmy[n\*WIDTH+m]=(n+1)\*(m+1);  }  } |

None of the two code snippets above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:   
  
   
Note that the code uses defined constants for the width and height, instead of using directly their numerical values. This gives the code a better readability, and allows changes in the code to be made easily in one place.

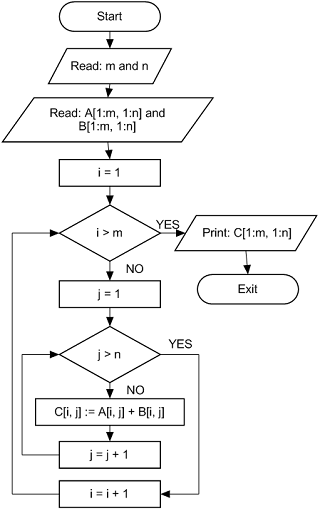
# *A c++ program can implement the following matrix operations:*

* + Addition
  + Multiplication
  + Transposition
  + Determinant
  + Singular or not
  + Upper triangular
  + Lower triangular

# ADDITION:

#### Flowchart:

### FLOWCHART FOR MATRIX ADDITION

[](http://i0.wp.com/www.bscshortnote.com/wp-content/uploads/2015/02/Flowcchart-for-Matrix-addition.png)

#### PSEUDO CODE:

Input Two matrices a and b

Output Output matrix c containing elements after addition of a and b

complexity O(n^2)

Matrix-Addition(a,b)

1 for i =1 to rows [a]

2 for j =1 to columns[a]

3 Input a[i,j];

4 Input b[i,j];

5 C[i, j] = A[i, j] + B[i, j];

6 Display C[i,j];

#### ALGORITHM:

1. To add two matrixes sufficient and necessary condition is "dimensions of matrix A = dimensions of matrix B".
2. Loop for number of rows in matrix A.
3. Loop for number of columns in matrix A.
4. Input A[i,j] and Input B[i,j] then add A[i,j] and B[i,j]
5. store and display this value as C[i,j];

#### PROGRAM PART:

//Sum Calculation

cout<<endl<<"SUM...."<<endl;

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

{

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

{

cout<<" "<<a[i][j]+b[i][j];

}

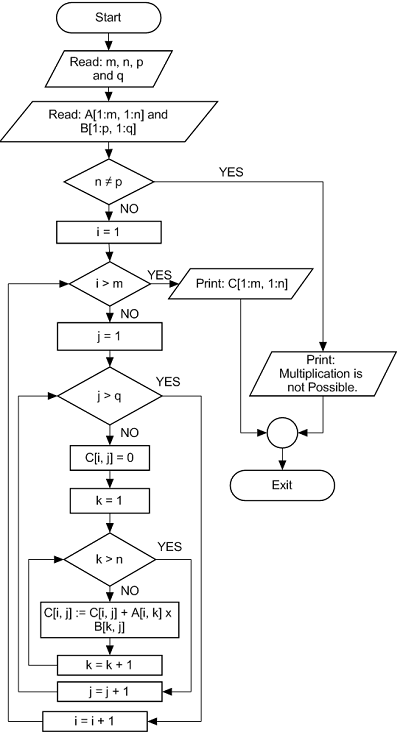
cout<<endl;

}

# MULTIPLICATION:

#### FLOWCHART:

### FLOWCHART FOR MATRIX MULTIPLICATION

[](http://i0.wp.com/www.bscshortnote.com/wp-content/uploads/2015/02/Flowchart-for-Matrix-Multiplication.png)

#### Pseudo code:

Input two matrixes.

Output Output matrix C.

Complexity O(n^3)

Matrix-Multiply(A, B)

1 if columns [A] ≠ rows [B]

2 then error "incompatible dimensions"

3 else

4 for i =1 to rows [A]

5 for j = 1 to columns [B]

6 C[i, j] =0

7 for k = 1 to columns [A]

8 C[i, j]=C[i, j]+A[i, k]\*B[k, j]

9 return C

#### Algorithm:

1. To multiply two matrixes sufficient and necessary condition is "number of columns in matrix A = number of rows in matrix B".
2. Loop for each row in matrix A.
3. Loop for each columns in matrix B and initialize output matrix C to 0. This loop will run for each rows of matrix A.
4. Loop for each columns in matrix A.
5. Multiply A[i,k] to B[k,j] and add this value to C[i,j]
6. Return output matrix C.

#### Program part:

{

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

{

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

{

c[i][j]=0;

for(z=0;z<l;z++)

{

c[i][j]=c[i][j]+(a[i][z]\*b[z][j]);

}

}

}               }

else

{

cout<<"invalid data....";

}

|  |  |  |
| --- | --- | --- |
|  | int foo [5]; |  |
|  |  |  |
|  |  |  |

# TRANSPOSE:

ALGORITHM:S p1: Start.

Step2: Read: m and n

Step3: Read: Take inputs for Matrix A[1:m, 1:n].

Step4: If m == n then:

Repeat for i = 1 to m by 1

Repeat for j = 1 to n by 1

B[i, j] = A[j, i]

[End of for loop]

[End of for loop]

Else:

temp = m

m = n

n = temp

Repeat for i = 1 to m by 1

Repeat for j = 1 to n by 1

B[i, j] = A[j, i]

[End of for loop]

[End of for loop]

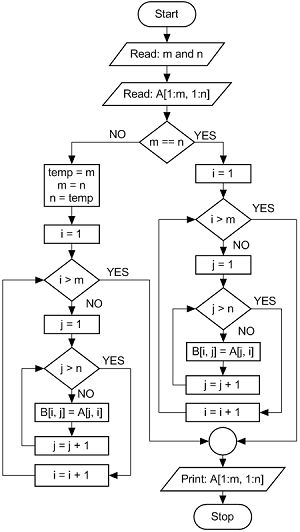
[End of If structure]

Step5: Print: B[1:m, 1:n]

Step6: Exit.

#### FLOWCHART:

### FLOWCHART TO FIND TRANSPOSE OF A MATRIX

[](http://i1.wp.com/www.bscshortnote.com/wp-content/uploads/2015/02/FLOWCHART-TO-FIND-TRANSPOSE-OF-A-MATRIX.png)

#### PROGRAM PART:

//Transpose

cout<<endl<<"transpose..."<<endl;

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

{

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

{

t[i][j]=a[j][i];

}

}

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

{

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

{

cout<<" "<<t[i][j];

}

cout<<endl;

}

# CONCLUSION:

We implemented and analyzed the concept of arrays, n dimensional arrays and operations on matrix.

# FAQ: