CMPT 125 Week 5 Self Reading Material

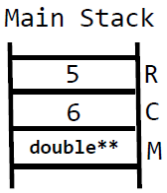
Multi Dimensional Arrays and Pointer Arithmetic

Multi Dimensional Arrays

Multidimensional dynamic arrays are created using pointer of pointers. In this course, we restrict our attention to two dimensional (2D) dynamic arrays. A 2D array is a rectangular arrangement of data in rows and columns. Thus a 2D array represents a matrix data. A 2D dynamic array is created using a pointer to pointer.

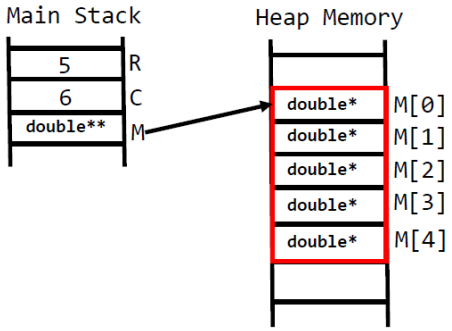
In order to create a matrix **M**, say with 5 rows and 6 columns, so that to store double data type elements; we first specify the number of rows and columns and then declare a double data type pointer to pointer as shown below. Schematic memory diagrams are also presented to describe pointers better. In addition, the data type of the pointer **M** is shown inside the memory space reserved for **M** for clarity purposes.

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
       int R = 5, C = 6;
       double ** M;
```



Next, we create a dynamic array of as many as **R** pointers and assign **M** the memory address of the first element of the dynamic array as follows. Arrows are used to show where pointers are pointing to instead of displaying the actual memory address values. The data type of each element of **M** is also shown inside the memory space reserved for the element for clarity purposes.

```
int main()
{
   int R = 5, C = 6;
   double ** M;
   M = new double*[R];
```



Now we observe that

- **M** is a dynamic array of pointers with **R** elements
- Elements of **M** can be accessed via indexing
- The elements of **M** are pointers
- The data type of elements of **M** is **double***
- Each element of **M** can be made a dynamic array of double data type values. For example, we can make the **M[0]** pointer a dynamic array of as many as **C** double data type elements as follows:

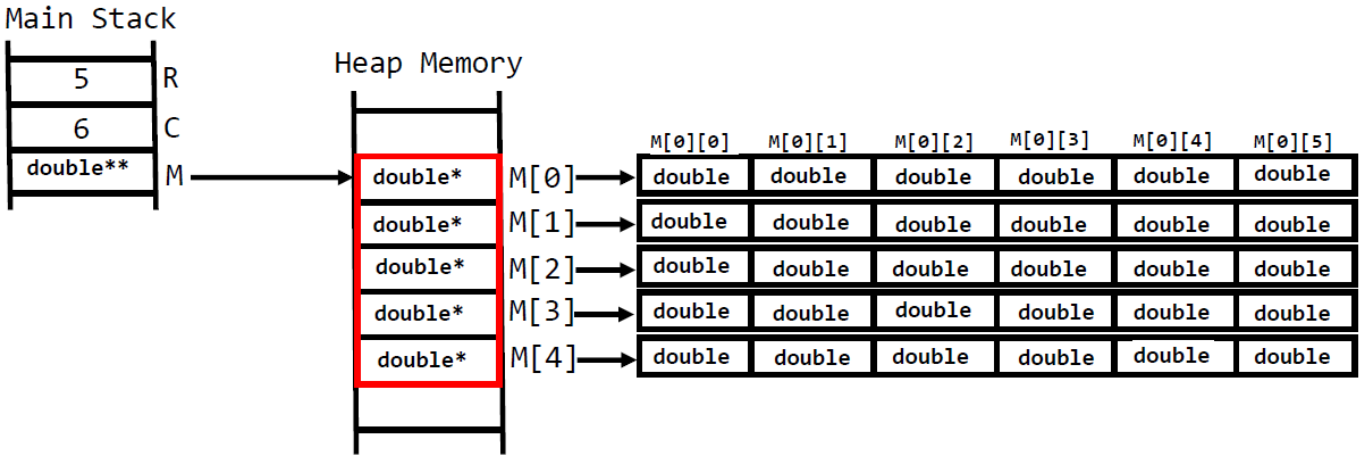
  M[0] = new double[C];

  then the elements of **M[0]** dynamic array will be **M[0][0], M[0][1], M[0][2],…, M[0],[C-1]**

- We can also make each of the pointers **M[1], M[2],…, M[R-1]** a dynamic array of as many as **C** double data type elements in order to create a matrix as shown below.

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
```

```
int R = 5, C = 6;
double ** M;
M = new double*[R];
for (int i = 0; i < R; i++)
	M[i] = new double[C];
```



Now **M** is a matrix of **5** rows and **6** columns. Moreover

- The first row elements are M[0][0], M[0][1], M[0][2], M[0][3], M[0][4], and M[0][5] (as shown above)
- The second row elements are M[1][0], M[1][1], M[1][2], M[1][3], M[1][4], and M[1][5]
- The third row elements are M[2][0], M[2][1], M[2][2], M[2][3], M[2][4], and M[2][5]
- The fourth row elements are M[3][0], M[3][1], M[3][2], M[3][3], M[3][4], and M[3][5]
- The fifth row elements are M[4][4], M[4][1], M[4][2], M[4][3], M[4][4], and M[4][5]

Therefore we may initialize the elements of the matrix M say for example with random double values in the range [0.0, 1.0) as follows

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
	int R = 5, C = 6;
	double ** M;
	M = new double*[R];
	for (int i = 0; i < R; i++)
		M[i] = new double[C];

	//Initialize the elements of the matrix
	srand(time(0));
	for (int i = 0; i < R; i++)
		for (int j = 0; j < R; j++)
			M[i][j] = 1.0*rand()/RAND_MAX;
```

Similarly, we may print the elements of the matrix M in a rectangular format as follows:

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
	int R = 5, C = 6;
	double ** M;
	M = new double*[R];
```

```
	for (int i = 0; i < R; i++)
		M[i] = new double[C];

	//Initialize the elements of the matrix
	srand(time(0));
	for (int i = 0; i < R; i++)
		for (int j = 0; j < R; j++)
			M[i][j] = 1.0*rand()/RAND_MAX;

	//Print the matrix
	for (int i = 0; i < R; i++)
	{
		for (int j = 0; j < R; j++)
			cout << M[i][j] << "\t";
		cout << endl;
	}
```

Before we finish the program, we need to free (delete) all the memory spaces allocated on the heap memory. In order to do so, we first delete each row (which is a dynamic array of double values) with the statement **`delete[] M[i]`** and then delete the dynamic array of pointers with the statement **`delete[] M`** as shown below.

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
	int R = 5, C = 6;
	double ** M;
	M = new double*[R];
	for (int i = 0; i < R; i++)
		M[i] = new double[C];

	//Initialize the elements of the matrix
	srand(time(0));
	for (int i = 0; i < R; i++)
		for (int j = 0; j < R; j++)
			M[i][j] = 1.0*rand()/RAND_MAX;

	//Print the matrix
	for (int i = 0; i < R; i++)
	{
		for (int j = 0; j < R; j++)
			cout << M[i][j] << "\t";
		cout << endl;
	}

	//Delete the heap memory
	for (int i = 0; i < R; i++)
		delete[] M[i];
	delete[] M;

	system("Pause");
	return 0;
}
```

We may also re-write the program in order to organize it better using functions such as initializing the matrix in a function, printing the matrix in a function, and deleting the heap memory in a function. This is possible because we can pass a matrix to a function as an argument. Passing a matrix to a function as an argument is akin to passing a pointer (of pointers) to a function and therefore follows the rule of parameter passing by

pointer. All that is needed is to make sure the parameter of the function corresponding to the matrix is a pointer of pointers.

```
#include <iostream>
#include <ctime>
using namespace std;
void initializeMatrix(double **M, const int &R, const int &C)
{
	for (int i = 0; i < R; i++)
		for (int j = 0; j < R; j++)
			M[i][j] = 1.0*rand()/RAND_MAX;
}
void printMatrix(double **M, const int &R, const int &C)
{
	for (int i = 0; i < R; i++)
	{
		for (int j = 0; j < R; j++)
			cout << M[i][j] << "\t";
		cout << endl;
	}
}
void deleteMatrix(double **M, const int &R, const int &C)
{
	for (int i = 0; i < R; i++)
		delete[] M[i];
	delete[] M;
}
int main()
{
	int R = 5, C = 6;
	double ** M;
	M = new double*[R];
	for (int i = 0; i < R; i++)
		M[i] = new double[C];

	//Initialize the elements of the matrix
	initializeMatrix(M, R, C);

	//Print the matrix
	printMatrix(M, R, C);

	//Delete the heap memory
	deleteMatrix(M, R, C);

	system("Pause");
	return 0;
}
```

We may also create the matrix space inside a function and return the matrix from the function. This is possible because returning a matrix from a function is akin to returning a pointer of pointers from a function.

```
#include <iostream>
#include <ctime>
using namespace std;
double** createMatrix(const int &R, const int &C)
{
	double ** M = new double*[R];
	for (int i = 0; i < R; i++)
		M[i] = new double[C];
	return M;
}
void initializeMatrix(double **M, const int &R, const int &C)
{
	for (int i = 0; i < R; i++)
```

```
                for (int j = 0; j < R; j++)
                        M[i][j] = 1.0*rand()/RAND_MAX;
}
void printMatrix(double **M, const int &R, const int &C)
{
        for (int i = 0; i < R; i++)
        {
                for (int j = 0; j < R; j++)
                        cout << M[i][j] << "\t";
                cout << endl;
        }
}
void deleteMatrix(double **M, const int &R, const int &C)
{
        for (int i = 0; i < R; i++)
                delete[] M[i];
        delete[] M;
}
int main()
{
        //Set the required matrix dimensions (number of rows and number of columns)
        int R = 5, C = 6;

        //Create a matrix with R rows and C columns
        double **M = createMatrix(R, C);

        //Initialize the elements of the matrix
        initializeMatrix(M, R, C);

        //Print the matrix
        printMatrix(M, R, C);

        //Delete the heap memory
        deleteMatrix(M, R, C);

        system("Pause");
        return 0;
}
```

Self Test Practice Questions

1. Define a function that takes a matrix of double data type, a row index **r**, and its column size **C** and returns the sum of the elements of the matrix at row index **r**.
2. Define a function that takes a matrix of double data type, its row size **R**, and a column index **c** and returns the sum of the elements of the matrix at column index **c**.
3. Define a function that takes a matrix of double data type, its column size **C**, and two row indexes **r1** and **r2** and swaps the rows of the matrix at row index **r1** with the row of the matrix at row index **r2**.
4. Define a function that takes a matrix of double data type, its row size **R**, and two column indexes **c1** and **c2** and swaps the column of the matrix at column index **c1** with the column of the matrix at column index **c2**.
5. Define a function that takes a matrix of double data type and its row and column sizes and returns true if all the rows of the matrix have the same sum and returns false otherwise.
6. Define a function that takes a matrix of double data type and its row and column sizes and returns true if all the columns of the matrix have the same sum and returns false otherwise.
7. Define a C++ function that takes a two dimensional dynamic array (matrix) of int data type and its row and column sizes and returns true if the matrix is an identity matrix and returns false otherwise.

Remark: A matrix is an identity matrix if it is a square matrix (that is **row size = column size**) and all its diagonal elements are exactly equal to 1 while every other element is 0.

8. Define a C++ function that takes a two dimensional dynamic array (matrix) of int data type and its row and column sizes and returns true if the matrix is a lower matrix and returns false otherwise.

   Remark: A matrix M is a lower matrix if it is a square matrix (row size = column size) and that all the elements of the matrix above the main diagonal are 0. The remaining elements don't matter.

9. Define a C++ function that takes a two dimensional dynamic array (matrix) of int data type and its row and column sizes and returns true if the matrix is an upper matrix and returns false otherwise.

   Remark: A matrix M is an upper matrix if it is a square matrix (row size = column size) and that all the elements of the matrix below the main diagonal are 0. The remaining elements don't matter.

10. Define a function that takes a matrix of double data type and its row and column sizes and returns the transpose of the matrix.

11. **[Challenge: Beyond your scope]** Define a function named **`dotProduct`** that takes two same size dynamic arrays **A** and **B** of double data type and their sizes as arguments and returns the dot product of the dynamic arrays. Remark:- The dot product of two vectors (that is arrays) **A** and **B** of equal length **size** is given by

$$dotProduct(A, B) = \sum_{i=0}^{size-1} A[i] * B[i]$$

12. **[Challenge: Beyond your scope]** Define a function that takes two matrices M1 and M2 of double data types and their sizes and returns the product of the matrices. Make sure you first understand the mathematics of matrix multiplication before wasting time trying to write the code. Two matrices are compatible for multiplication if the number of columns of M1 is equal to the number of rows of M2. Assume the argument matrices are compatible for multiplication. Use the **`dotProduct`** function when you answer this question. The function declaration is given below:

```
double** multiply(const double** M1, const int &R1, const int &C1, const double** M2, const int &R2, const int &C2)
```

13. **[Challenge: Beyond your scope]** Define a function that takes a matrix of double data type and its row and column sizes and returns the determinant of the matrix.

14. **[Challenge: Beyond your scope]** Define a function named **`isInvertible`** that takes a matrix of double data type and its row and column sizes and returns true if the matrix is invertible (that is its inverse exists) otherwise returns false.

15. **[Challenge: Beyond your scope]** A set of simultaneous linear equations is often written in a matrix equation form as **MX = Y** where **M** is called the augmented matrix and is created by arranging the coefficients of the variables in the linear equations, **X** is a column vector of the variables in the linear equations, and **Y** is a column vector of the right hand side values of the linear equations. Multiplying both sides of the matrix equation by $\mathbf{M}^{-1}$ (where $\mathbf{M}^{-1}$ is the inverse of the matrix **M**) from the left gives $\mathbf{X} = \mathbf{M}^{-1}\mathbf{Y}$ which gives the solution set of the simultaneous linear equations. Write a C++ program that reads a set of linear equations appropriately and prints the solution of the set of linear equations. If no solution is found, your program should print no solution exists. Use your **`isInvertible`** function to see if the augmented matrix is invertible and therefore if the set of linear equations has a solution.