```cpp
/*
 * Matrix.h
 *
 *  Created on: 21.12.2017
 *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
 */

// Header file for class Matrix - provides support for arbitrarily sized m-by-n matrices

// Declares class data type called Matrix.

// Include header guards to avoid multiple attempts at defining class.

#ifndef MATRIX_H_
#define MATRIX_H_

#include <iostream>

using namespace std;

class Matrix {

// Part 1.

protected:
        int noOfRows; //stores the number of rows
        int noOfCols; //stores the number of columns
        double* data; //stores the address to the 1-D array of the matrix entries arranged column-wise

        // getter
        int GetIndex (const int& rowIdx, const int& columnIdx) const; // determines the position (index) along 'data' of a matrix entry, in the row and column specified by 'rowIdx' and 'columnIdx', respectively.

public:

// 1. ConstructorsTesting

        // constructors
        Matrix (); // default constructor -  must be defined as other constructors have been defined here
        Matrix (const int& noOfRows_rhs, const int& noOfCols_rhs); // standard custom constructor
        Matrix (const Matrix& input); // default copy constructor

        // named static constructors:
        static Matrix Zeros(const int& noOfRows_rhs, const int& noOfCols_rhs);
        static Matrix Ones (const int& noOfRows_rhs, const int& noOfCols_rhs);

        // default destructor
        virtual ~Matrix ();

        // printing with ostream
        friend ostream& operator<< (ostream& out, const Matrix& rhs);

// 2. AssignmentTesting

        //default assignment operator
        Matrix& operator= (const Matrix& rhs);

// 3. ToeplitzTesting & ToeplitzTestingHelper

        static Matrix Print (const double* const data_rhs, const int& noOfRows_rhs, const int& noOfCols_rhs);
        static Matrix Toeplitz (const double* const column, const int& noOfRows_rhs, const double* const row, const int& noOfCols_rhs); //creates a new Toeplitz matrix

```

```
// 4. TransposeTesting

        static Matrix Transpose (const Matrix& input);  //static function - creates a new transpose matrix
        void Transpose ();  //non-static function - converts matrix into tranpose of itself

// 5. MultiplicationTesting

        friend Matrix operator* (const Matrix& lhs, const Matrix& rhs); //non-member multiplication function - creates new matrix which is result of lhs * rhs
        Matrix& operator*= (const Matrix& rhs); // member self-modifying multiplication function - converts input (lhs) matrix into result of lhs * rhs

// 6. RowColumnExchangeTesting

        Matrix& ExchangeRows (const int& row1, const int& row2); //swaps row1 and row2, for all columns in matrix
        Matrix& ExchangeRows (const int& row1, const int& row2, const int& col1, const int& col2); //swaps row1 and row2, but only between col1 and col2
        Matrix& ExchangeColumns (const int& col1, const int& col2); //swaps col1 and col2, for all rows in matrix
        Matrix& ExchangeColumns (const int& col1, const int& col2, const int& row1, const int& row2); //swaps col1 and col2, but only between row1 and row2


// 7. OtherTesting

        void Zeros (); // (sets every entry of matrix to zero)
        void Ones (); // (sets every entry of matrix to one)
        int GetNoOfRows () const; // (find out number of rows in matrix)
        int GetNoOfColumns () const; // (find out number of columns in matrix)
        double GetEntry (const int& rowIdx, const int& columnIdx) const; // (find out value of particular entry (i,j) in Matrix, starting indexing at (0,0) in upper left corner of matrix)

// Part 2.

        static Matrix Test (const double* data_rhs, const int& noOfRows_rhs, const int& noOfCols_rhs); //for constructing specified test matrices for testing in Parts 2 & 3
        static Matrix PrintMATLAB (const Matrix& rhs); //Specifically to print matrices in default MATLAB style to fixed 4.d.p precision for Parts 2 & 3

// Part 3.

// 8. Forward Substitution - Algorithm 3.1

        virtual Matrix ForwardSub (const Matrix& b) const;
                //Invoking matrix L is a lower triangular m-by-m square matrix (from SquareMatrix class function).
                //Input b is a m-by-1 column vector.
                //Returns a m-by-1 column vector y, such that Ly=b.

// 9. Backward Substitution - Algorithm 3.2

        virtual Matrix BackwardSub (const Matrix& y) const;
                //Invoking matrix U is an upper triangular m-by-m square matrix (from SquareMatrix class function).
                //Input y is a m-by-1 column vector.
                //Returns a m-by-1 column vector x, such that Ux=y.

// 10. Solving a system of linear equations in which the number of equations is equal to the number of unknowns, i.e. a uniquely determined solution exists.

        Matrix SolveUnique (const Matrix& L, const Matrix& U, const Matrix& P, Matrix& b) const;
                //Function is invoked by a m-by-m square matrix A (from SquareMatrix class function), which represents the matrix of coefficients
                //Input matrices L, U and P are produced in class SquareMatrix from LU decomposition of A using Algorithm 2.2. All 3 are m-by-m square matrices.
                //Input b is a m-by-1 column vector.
                //Returns a m-by-1 column vector x, such that Ax=b.

// 11. Solving an overdetermined system of linear equations in which the number of equations is greater than the number of unknowns, i.e. a least squares solution is produced.

        Matrix LeastSquaresHelper (const Matrix& b, const string& output) const;
                //Function is invoked by a non-square m-by-n matrix A, which represents the matrix of coefficients
                //Input b is a m-by-1 column vector.
                //Returns n-by-n square matrix A'*A if output = 'A'*A', or n-by-1 column vector A'*b if output = 'A'*b'.

};

#endif /* MATRIX_H_ */
```

```cpp
/*
 * Matrix.cpp
 *
 *  Created on: 21.12.17
 *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
 */

// Source file for class Matrix, to define implementations for functions declared in Matrix.h

#include "Matrix.h"
#include <iostream>
#include <cstdlib>
#include <string>
#include <cmath>  //for abs function
#include <iomanip> //for printing precision

using namespace std;

// Part 1.

//getter - to specify the position (index) along 'data' of a matrix entry in the row and column specified by 'rowIdx' and 'columnIdx', respectively
//assuming that the matrix indexing starts at (0,0) and the array indexing starts at (0).
int Matrix::GetIndex (const int& rowIdx, const int& columnIdx) const {
        int index = rowIdx + columnIdx*noOfRows;
        return index;
}

// 1. ConstructorsTesting

//default constructor
Matrix::Matrix () {
        noOfRows = 0;
        noOfCols = 0;
        data = 0;
}

//standard custom constructor
Matrix::Matrix (const int& noOfRows_rhs, const int& noOfCols_rhs) {
        // first check that both inputs are greater than zero
        if (noOfRows_rhs <= 0 or noOfCols_rhs <= 0) {
                cerr << endl;
                cerr << "Error:  In standard custom constructor ..." << endl;
                cerr << "        Cannot create a matrix with a non-positive dimension." << endl;
                cerr << "        Exiting program ... " << endl;
                cerr << endl;
                exit(1);
        }
        //assign dimension values
        noOfRows = noOfRows_rhs;
        noOfCols = noOfCols_rhs;
        //allocate new memory
        int product = noOfRows*noOfCols;
        data = new double [product];
        //then assign '0' to each value (this is done linearly here, rather than column-wise, as all values are the same)
        for (int i = 0; i < product; ++i) {
                data[i] = 0;
        }
}

// default copy constructor
Matrix::Matrix (const Matrix& rhs) {

        //assign dimensions
        this->noOfRows = rhs.noOfRows;
        this->noOfCols = rhs.noOfCols;

	// allocate the memory of appropriate size
	int product = noOfRows*noOfCols;
	this->data = new double[product];

	// ensure that a 'deep', rather than a 'shallow', copy is done
	for (int i = 0; i < product; ++i) {
		this->data[i] = rhs.data[i];
	}
}

// static Zeros constructor
Matrix Matrix::Zeros(const int& noOfRows_rhs, const int& noOfCols_rhs) {
	// There is no vector object attached to a static member function

	// Therefore need to create Matrix object of zeros using standard custom constructor
	Matrix Zeros (noOfRows_rhs, noOfCols_rhs);

	// return the object
	return Zeros;
}

// static Ones constructor
Matrix Matrix::Ones(const int& noOfRows_rhs, const int& noOfCols_rhs) {

	//Create Matrix object of zeros using standard custom constructor
	Matrix Ones (noOfRows_rhs, noOfCols_rhs);

	//Then assign '1' to each value (this is done linearly here, rather than column-wise, as all values are the same)
	int product = noOfRows_rhs*noOfCols_rhs;
	for (int i = 0; i < product; ++i) {
		Ones.data[i] = 1;
	}

	//Return the object
	return Ones;
}


//destructor
Matrix::~Matrix () {
	//dynamically delete all current instances of member variable data
	delete[] data;
}

//printer (remembering that the matrix has been stored columnwise in data)
ostream& operator<< (ostream& out, const Matrix& rhs)  {
	//loop through the row indices
	for (int i = 0; i < rhs.noOfRows; ++i) {
	out << "\t"; // leave a tab (8 spaces) at the beginning of the row
		//loop through the column indices
		for (int j = 0; j < rhs.noOfCols; ++j) {
			int index = i + j*rhs.noOfRows;		// GetIndex (i,j) cannot be used here as not declared in this scope
			double element = rhs.data[index];
			double tolerance = 1e-14;
			if (abs(element) < tolerance) { //set to zero if magnitude of element is less than tolerance;
				element = 0;
			}
			if (element >= 0) {
				out << " " << element; // leave an extra space if element is positive, to ensure alignment of matrix entries in console out
			}
			else {
				out << element;
```

```cpp
                        }
                        out << "         "; // leave 9 spaces between columns (tab + 1 doesn't work ...)
                }
                out << endl; // go to next line at end of row
        }

        return out;
}

// 2. AssignmentTesting

// default assignment operator
Matrix& Matrix::operator= (const Matrix& rhs) {
        // check the dimensions of the rhs input
        if (this->noOfRows != rhs.noOfRows and this->noOfCols!= rhs.noOfCols) {
                cerr << "Error:  Assignment operation failed ..." << endl;
                cerr << "        rhs input has incompatible number of rows and columns." << endl;
                cerr << endl;
        }
        else if (this->noOfRows != rhs.noOfRows) {
                cerr << "Error:  Assignment operation failed ..." << endl;
                cerr << "        rhs input has incompatible number of rows." << endl;
                cerr << endl;
        }
        else if (this->noOfCols!= rhs.noOfCols) {
                cerr << "Error:  Assignment operation failed..." << endl;
                cerr << "        rhs input has incompatible number of columns." << endl;
                cerr << endl;
        }
        //guard against copy into self
        else if (this == &rhs) {
                cerr << "Error:  Assignment operation failed..." << endl;
                cerr << "        copy into self attempted." << endl;
                cerr << endl;
        }
        else {
                // assign the data of rhs to lhs
                int product = noOfRows*noOfCols;
                for (int i = 0; i < product; ++i) {
                        this->data[i] = rhs.data[i];
                }
        }
        // return the reference to self
        return *this;
}

// 3. ToeplitzTesting & ToeplitzTestingHelper

//print function
Matrix Matrix::Print (const double* const data_rhs, const int& noOfRows_rhs, const int& noOfCols_rhs) {

        //Create Matrix object of zeros using standard custom constructor
        Matrix Print (noOfRows_rhs, noOfCols_rhs);

        //copy across the content of data_rhs
        int product = Print.noOfRows*Print.noOfCols;
        for (int i = 0; i < product; ++i) {
                        Print.data[i] = data_rhs[i];
                }

        //loop through the row indices of print
        for (int i = 0; i < Print.noOfRows; ++i) {
        cout << "\t"; // leave a tab (8 spaces) at the beginning of the row
                //loop through the column indices of print
                for (int j = 0; j < Print.noOfCols; ++j) {
                        int index = i + j*Print.noOfRows;       // GetIndex(i,j) cannot be user here without object
                        double element = data_rhs[index];
                        if (element >= 0) {
                                cout << " " << element; // leave an extra space if element is positive, to ensure alignment of matrix entries in console output
                        }
                        else {
                                cout << element;
                        }
                        cout << "         "; // leave 9 spaces between columns (tab + 1 doesn't work ...)
                }
                cout << endl; // go to next line at end of row
        }

        return Print;
}

//generate Toeplitz function
Matrix Matrix::Toeplitz (const double* const column, const int& noOfRows_rhs, const double* const row, const int& noOfCols_rhs) {

        //Create Matrix object of zeros using standard custom constructor
        Matrix Toeplitz (noOfRows_rhs, noOfCols_rhs);

        //copy across the content of data_rhs
        int product = Toeplitz.noOfRows*Toeplitz.noOfCols;

        //Display a warning message if first element of column and first element of row are not equal
        if (column[0] != row[0]) {
                cout << "Warning: In Toeplitz constructor, first element of input column does not match" << endl;
                cout << "         first element of input row! Column wins diagonal conflict." << endl;
                cout << endl;
        }

        //To construct Toeplitz matrix based on columns and row, compressed into a 1D array columnwise
        //loop through the rows indices
        for (int i = 0 ; i <Toeplitz.noOfRows; i++) {
                //loop through the column indices
                for (int j = 0 ; j <Toeplitz.noOfCols; j++) {
                        int index = (i + j*Toeplitz.noOfRows);  // GetIndex(i,j) cannot be user here without object
                        //for diagonal of matrix or below the diagonal
                        if (i>=j){
                                Toeplitz.data[index]=column[i-j];
                        }
                        //above this diagonal line
                        else if (j>i){
                                Toeplitz.data[index]=row[j-i];
                        }
                }
        }

        // return the object
        return Toeplitz;
}

// 4. TransposeTesting

//static transpose function
Matrix Matrix::Transpose(const Matrix& rhs) {
        //create an instance of Matrix object called Transpose
        Matrix Transpose(rhs);  //calls default copy constructor to replicate rhs

        //invoke the non-static transpose function (see below)
        Transpose.Transpose();

        //return the object
        return Transpose;

}

//non-static transpose function
void Matrix::Transpose () {
        //swap the number of columns and rows
        int tmp = 0;
        tmp = noOfRows;
        noOfRows = noOfCols;
        noOfCols = tmp;

        // allocate the memory for temporary holder for data
        int product = noOfRows*noOfCols;
        double *tmp_data = new double[product];

        //re-allocate members of transposed data as a tranpose of original data
        int k = 0; //use this as a counter to index original data
        //loop through the row indices of tranposed data
        for (int i = 0; i < noOfRows; ++i) {
                //looping through the column indices of transposed data
                for (int j = 0; j < noOfCols; ++j) {
                        int index = GetIndex(i,j); //to index tranposed data
                        tmp_data[index] = data[k]; //appropriate transformation to switch columns and rows (hold temporarily in tmp_data)
                        k += 1; //increment the counter
                        }
                }

        // then copy across the content from tmp_data to data in a linear fashion
        for (int i = 0; i < product; ++i) {
                        data[i] = tmp_data[i];
        }

        //delete dynamically allocated memory for tmp_data
        delete [] tmp_data;

        //empty return as void function
        return;
}

// 5. MultiplicationTesting

//non-member multiplication function - creates new matrix
Matrix operator* (const Matrix& lhs, const Matrix& rhs) {

        Matrix result(lhs); //calls default copy constructor
        result *= rhs;      //calls self-modifying member multiplication function (see below)
        return result;      //returns object

}

//member multiplication function
Matrix& Matrix::operator*= (const Matrix& rhs) {

        //check that number of cols of LHS is equal to number of rows of RHS
        if (this->noOfCols != rhs.noOfRows) {
                cerr << endl;
                cerr << "Error:  In member multiplication function ... " << endl;
                cerr << "        Inner matrix dimensions of rhs and lhs do not agree." << endl;
                cerr << "        Unmultiplied lhs matrix has been returned." << endl;
                cerr << endl;
                Matrix error;
                return *this;
        }
        // The returned matrix will have the same number of rows as LHS matrix
        // But the returned matrix will have the same number of columns as the RHS matrix
        // Therefore amend number of columns of returned matrix if need be
        if (this->noOfCols != rhs.noOfCols) {
                int noOfCols_old = this->noOfCols;
                int product_old = this->noOfRows*this->noOfCols;
                this->noOfCols = rhs.noOfCols;
                int product_new = this->noOfRows*this->noOfCols;
                if (noOfCols_old < this->noOfCols) {
                        //extend data by allocating extra columns and padding with zeros:
                        //dynamically allocate memory for array of zeros of appropriate length
                        double *tmp = new double[product_new];
                        //initialise these elements all to zero
                        for (int i = 0; i < product_new; ++i) {
                                tmp[i] = 0;
                        }
                        //then copy across the content from data to temp
                        for (int i = 0; i < product_old; ++i) {
                                tmp[i] = this->data[i];
                        }
                        //reassign data
                        this->data = tmp;
                }
        }

        // allocate the memory for temporary holder for data
        int product = this->noOfRows*this->noOfCols;
        double *tmp_data = new double[product];

        // looping through the row indices of lhs
        for (int i = 0; i < this->noOfRows; ++i) {
                //looping through the column indices of lhs
                for (int j = 0; j < this->noOfCols; ++j) {
                        double tmp = 0; //to store temporary element value of tmp_data in
                        //looping through the rows of the rhs matrix
                        for (int k = 0; k < rhs.noOfRows; ++k) {
                                int index = GetIndex(i,j); //use to index tmp_data to temporarily store result in
                                //tmp_data[index] = dot product of (lhs row i) and (rhs col j)
                                int index_LHS = i + k*this->noOfRows; // use to index lhs matrix for calculation
                                int index_RHS = k + j*rhs.noOfRows; //use to index rhs matrix for calculation
                                tmp += this->data[index_LHS] * rhs.data[index_RHS];  //increment tmp by this result
                                tmp_data[index] = tmp;
                        }
                }
        }

        // then copy across the content from tmp_data to data
        for (int i = 0; i < product; ++i) {
                        this->data[i] = tmp_data[i];
        }

        //delete dynamically allocated memory for tmp_data
        delete [] tmp_data;

        // return the reference to self
        return *this;
}
```

```

// 6. RowColumnExchangeTesting

//ExchangeRows function (overloaded with two input parameter options)

//swaps row1 and row2, across all columns
Matrix& Matrix::ExchangeRows (const int& row1, const int& row2) {

        //looping through all columns
        for (int j = 0; j < noOfCols; ++j) {
                int index_row1 = GetIndex(row1,j);    //get index for element in row1
                double tmp = data[index_row1];        //store in tmp variable
                int index_row2 = GetIndex(row2,j);    //get index for element in row2
                data[index_row1] = data[index_row2];  //swap the elements
                data[index_row2] = tmp;               //swap the elements
        }

        // return the reference to self
        return *this;
}

//swaps row1 and row2, but only between col1 and col2
Matrix& Matrix::ExchangeRows (const int& row1, const int& row2, const int& col1, const int& col2) {

        //looping through columns from col1 to col2
        for (int j = col1; j <= col2; ++j) {
                int index_row1 = GetIndex(row1,j);    //get index for element in row1
                double tmp = data[index_row1];        //store in tmp variable
                int index_row2 = GetIndex(row2,j);    //get index for element in row2
                data[index_row1] = data[index_row2];  //swap the elements
                data[index_row2] = tmp;               //swap the elements
        }

        // return the reference to self
        return *this;
}

//ExchangeColumns function (overloaded with two input parameter options)

//swaps col1 and col2, across all rows
Matrix& Matrix::ExchangeColumns (const int& col1, const int& col2) {

        //looping through all rows
        for (int i = 0; i < noOfRows; ++i) {
                int index_col1 = GetIndex(i,col1);    //get index for element in col1
                double tmp = data[index_col1];        //store in tmp variable
                int index_col2 = GetIndex(i,col2);    //get index for element in col2
                data[index_col1] = data[index_col2];  //swap the elements
                data[index_col2] = tmp;               //swap the elements
        }

        // return the reference to self
        return *this;

}

//swaps col1 and col2, but only between row1 and row2
Matrix& Matrix::ExchangeColumns (const int& col1, const int& col2, const int& row1, const int& row2)
{

        //looping through rows between row1 and row2
        for (int i = row1; i <= row2; ++i) {
                int index_col1 = GetIndex(i,col1);    //get index for element in col1
                double tmp = data[index_col1];        //store in tmp variable
                int index_col2 = GetIndex(i,col2);    //get index for element in col2
                data[index_col1] = data[index_col2];  //swap the elements
                data[index_col2] = tmp;               //swap the elements
        }

        // return the reference to self
        return *this;

}

// 7. OtherTesting

//Zeros (sets every entry to zero)
void Matrix::Zeros () {
        //Assign 0 to every element in data, in a linear fashion
        int product = noOfRows*noOfCols;
        for (int i = 0; i < product; ++i) {
                data[i] = 0;
        }

        //empty return as void function
        return;
}

//Ones (sets every entry to one)
void Matrix::Ones () {
        //Assign 1 to every element in data, in a linear fashion
        int product = noOfRows*noOfCols;
        for (int i = 0; i < product; ++i) {
                data[i] = 1;
        }

        //empty return as void function
        return;
}

//GetNoOfRows (find out number of rows)
int Matrix::GetNoOfRows () const {
        return noOfRows;
}

//GetNoOfColumns (find out number of columns)
int Matrix::GetNoOfColumns () const {
        return noOfCols;
}

//GetEntry (find out value of particular entry (i,j) in Matrix, starting indexing at (0,0) in upper left corner of matrix)
double Matrix::GetEntry (const int& rowIdx, const int& columnIdx) const {
        int index= GetIndex (rowIdx, columnIdx);
        return data[index];
}

// Part 2.

//To construct user specified matrices for testing functions
Matrix Matrix::Test (const double* data_rhs, const int& noOfRows_rhs, const int& noOfCols_rhs) {

        // create an empty Matrix object
        Matrix Test;

        // then set up the dimensions
        Test.noOfRows = noOfRows_rhs;
        Test.noOfCols = noOfCols_rhs;

        // allocate the memory
        int product = noOfRows_rhs*noOfCols_rhs;
        Test.data = new double[product];

        //then copy across the content from data_rhs to Test.data
        for (int i = 0; i < product; ++i) {
                Test.data[i] = data_rhs[i];
        }

        // return the object
        return Test;
}

//Print function to print matrices in default MATLAB style to fixed 4.d.p precision
Matrix Matrix::PrintMATLAB (const Matrix& rhs) {
        //Need to create an empty SquareMatrix object
        Matrix print;

        // then set up the dimensions
        print.noOfRows = rhs.noOfRows;
        print.noOfCols = rhs.noOfCols;

        // allocate the memory
        int product = print.noOfRows*print.noOfCols;
        print.data = new double[product];

        //copy across the content of data_rhs
        for (int i = 0; i < product; ++i) {
                        print.data[i] = rhs.data[i];
                }

        //set to display numbers to fixed 4.d.p.
        cout << fixed << setprecision(4);

        //loop through the row indices of print
        for (int i = 0; i < print.noOfRows; ++i) {
        cout << "\t"; // leave a tab (8 spaces) at the beginning of the row
                //loop through the column indices of print
                for (int j = 0; j < print.noOfCols; ++j) {
                        int index = i + j*print.noOfRows;       // GetIndex(i,j) cannot be user here without object
                        double element = print.data[index];
                        double tolerance = 1e-14;
                        if (abs(element) < tolerance) {
                                element = 0;  //set to zero if magnitude of element is less than tolerance;
                        }
                        if (element == 0) {
                                cout << "      0";  //leave 6 spaces to ensure alignment
                        }
                        else if (element >= 0) {
                                cout << " " << element; // leave an extra space if element is positive, to ensure alignment of matrix entries in console output
                        }
                        else {
                                cout << element;
                        }
                        cout << "         "; // leave 9 spaces between columns (tab + 1 doesn't work ...)
                }
                cout << endl; // go to next line at end of row
        }
        cout << endl;

        cout.unsetf(ios::floatfield);       // unset floatfield from 'fixed' to 'not set'
        cout << setprecision(6);            // set precision back to default value of 6.d.p

        return print;
}

// Part 3.

// 8. Forward Substitution - Algorithm 3.1 - implemented here so can access and manipulate column vector b

Matrix Matrix::ForwardSub (const Matrix& b) const {

        //construct a m-by-1 column vector to hold return result y in
        int dim = this->noOfCols;
        Matrix y = Matrix(dim,1);

        //Expanding from the top row of L:
        //1. Calculate first entry of y
        y.data[0] = b.data[0]/this->data[0];

        //2. Calculate subsequent entries of y
        for (int k= 1; k <dim; ++k) {
                double tmp = 0;
                for (int i= 0; i <k; ++i) {
                        int index_ki = GetIndex(k,i);
                        tmp += (this->data[index_ki]*y.data[i]);
                }
                int index_kk = GetIndex(k,k);
                y.data[k] =  (b.data[k] - tmp)/this->data[index_kk];

        }

        return y;

}

// 9. Backward Substitution - Algorithm 3.2 - implemented here so can access and manipulate column vector y

Matrix Matrix::BackwardSub (const Matrix& y) const {

        //construct a m-by-1 column vector to hold return result x in
        int dim = this->noOfCols;
        Matrix x = Matrix(dim,1);

        //Expanding from the bottom row of U:
        //1. Calculate first entry of x
        x.data[dim-1] = y.data[dim-1]/this->data[dim*dim-1];

        //2. Calculate subsequent entries of x
        for (int k= dim-2; k >=0; --k) {
                //cout << k << endl;
                double tmp = 0;
                for (int i= k+1; i <dim; ++i) {
                        int index_ki = GetIndex(k,i);
                        tmp += (this->data[index_ki]*x.data[i]);
                }
                int index_kk = GetIndex(k,k);
                x.data[k] =  (y.data[k] - tmp)/this->data[index_kk];
        }

        return x;

}

// 10. Solving a system of linear equations in which the number of equations is equal to the number of unknowns, i.e. a unique determined solution exists.

Matrix Matrix::SolveUnique (const Matrix& L, const Matrix& U, const Matrix& P, Matrix& b) const {

        //multiply P by b
        b = P*b;                 //NB assignment operator throws an error here if number of columns in b and P do not match

        //solve for y in Ly=b using Algorithm 3.1
        Matrix y = L.ForwardSub(b);

        //solve for x in Ux=y using Algorithm 3.2
        Matrix x = U.BackwardSub(y);

        return x;
}

// 11. Solving an overdetermined system of linear equations in which the number of equations is greater than the number of unknowns, i.e. a least squares solution is produced.

Matrix Matrix::LeastSquaresHelper (const Matrix& b, const string& output) const {

        //check that number of rows in A is bigger than the number of columns in A
        if (this->noOfRows == this->noOfCols and output == "A'*A") {
                cout << "Warning: In function LeastSquaresHelper ..." << endl;
                cout << "         Matrix A is square." << endl;
                cout << "         An exact solution, not a least squares solution, will be provided." << endl;
                cout << "         Consider using SolveUnique instead." << endl;
                cout << endl;
        }
        // check that m > n - error #1
        if (this->noOfRows < this->noOfCols) {
                cerr << endl;
                cerr << "Error:  In function LeastSquaresHelper ... " << endl;
                cerr << "        In matrix A, the number of rows is less than the number of columns." << endl;
                cerr << endl;
                Matrix error;
                return error;
        }
        //check that b is of appropriate length -  error #2
        if (this->noOfRows != b.noOfRows) {
                cerr << "Error:  In function LeastSquaresHelper ... " << endl;
                cerr << "        The number of rows in b does not equal the number of rows in A." << endl;
                cerr << endl;
                Matrix error;
                return error;
        }

        //produce ATA
        Matrix A (*this);
        Matrix AT = Matrix::Transpose(A);
        Matrix ATA = AT*A;

        //produce ATb
        Matrix ATb = AT*b;

        //return ATA or ATb dependent upon output string
        if (output == "A'*A") {
                cout << "Number of rows in A'*A = " << ATA.GetNoOfRows()  << endl;
                cout << "Number of columns in A'*A = " << ATA.GetNoOfColumns() << endl;
                cout << endl;
                return ATA;
        }
        if (output == "A'*b") {
                return ATb;
        }
}
```

```
/*
 * SquareMatrix.h
 *
 *  Created on: 21.12.2017
 *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
 */

// Header file for class SquareMatrix - extends class Matrix to provide specialised support for square m-by-m matrices.

// Declares class data type called SquareMatrix.

// Include header guards to avoid multiple attempts at defining class.

#ifndef SQUAREMATRIX_H_
#define SQUAREMATRIX_H_

#include "Matrix.h"

class SquareMatrix : public Matrix { // class SquareMatrix inherits publicly from class Matrix

private:

        void checkSquare () const;  //to check if user is attempting to make square copy of unsquare matrix

public:

// Part 2.

// 1. ConstructorsTesting

        //constructors
        SquareMatrix (); //default constructor
        SquareMatrix (const int& dim); // standard custom constructor
        SquareMatrix (const SquareMatrix& input); // default copy constructor #1  //returns SquareMatrix copy of SquareMatrix
        SquareMatrix (const Matrix& input); // default copy constructor #2        //returns SquareMatrix copy of Matrix

        // named static constructors:
        static SquareMatrix Zeros (const int& dim);
        static SquareMatrix Ones (const int& dim);
        static SquareMatrix Eye (const int& dim);
        static SquareMatrix Test (const double* data_rhs, const int& dim); //for constructing specified test matrices for testing

        // default destructor
        ~SquareMatrix ();

// 2. ToeplitzTesting & ToeplitzTestingHelper

        static SquareMatrix Toeplitz (const double* const row, const int& dim); //creates a new Toeplitz matrix when only first row is specified
        static SquareMatrix Toeplitz (const double* const column, const double* const row, const int& dim); //creates a new Toeplitz matrix when both first column and first row are specified

// 3. TransposeTesting

        //no new function declarations required.

// 4. TriangularExtractionTesting

        //extract upper triangular part of square matrix
        SquareMatrix TriUpper () const;  // non-static member function

        //extract lower triangular part of square matrix
        SquareMatrix TriLower () const;  // non-static member function

// 5. LUDecompositionTesting - Algorithm 2.1 - Gaussian Elimination without Pivoting

        SquareMatrix LUDecompositionOne (const char& output) const;
                //Takes a non-singular square matrix A (i.e. with non-zero determinant) as input (i.e. invoking square matrix).
                //Returns lower triangular matrix L if output = 'L', or upper triangular matrix U if output = 'U', such that A=LU.

// 6. LUDecompositionTesting - Algorithm 2.2 - Gaussian Elimination with Partial Pivoting

        SquareMatrix LUDecompositionTwo (const char& output) const;
                //Takes a non-singular square matrix A (i.e. with non-zero determinant) as input (i.e. invoking square matrix).
                //Returns lower triangular matrix L if output = 'L', or upper triangular matrix U if output = 'U', or permutation matrix P if output = 'P', such that PA=LU.

// Part 3.

// 7. Forward Substitution - Algorithm 3.1

        Matrix ForwardSub (const Matrix& b) const;
                //Invoking matrix L is a lower triangular m-by-m square matrix.
                //Input b is a m-by-1 column vector.
                //Returns a m-by-1 column vector y, such that Ly=b (via a call to the Matrix class function).

// 8. Backward Substitution - Algorithm 3.2

        Matrix BackwardSub (const Matrix& y) const;
                //Invoking matrix U is an upper triangular m-by-m square matrix.
                //Input y is a m-by-1 column vector.
                //Returns a m-by-1 column vector x, such that Ux=y (via a call to the Matrix class function).

// 9. Solving a system of linear equations in which the number of equations is equal to the number of unknowns, i.e. a uniquely determined solution exists.

        Matrix SolveUnique (Matrix& b) const;
                //Function is invoked by a m-by-m square matrix A, which represents the matrix of coefficients
                //Input b is a m-by-1 column vector.
                //Returns a m-by-1 column vector x, such that Ax=b (via a call to the Matrix class function).

};

#endif /* SQUAREMATRIX_H_ */
```

```cpp
/*
 * SquareMatrix.cpp
 *
 *  Created on: 21.12.17
 *      Author: Edward James - Medical Imaging CDT MRes, 2017-2018.
 */

// Source file for class SquareMatrix, to define implementations for functions declared in SquareMatrix.h

#include "SquareMatrix.h"
#include <cstdlib>
#include <cmath>  //for abs function

using namespace std;

// Part 2.

// 1. ConstructorsTesting

//default constructor
SquareMatrix::SquareMatrix () : Matrix () {
}

//standard custom constructor
SquareMatrix::SquareMatrix (const int& dim) : Matrix(dim,dim) {
}

//default copy constructor #1
SquareMatrix::SquareMatrix (const SquareMatrix& input) : Matrix(input) {   //returns SquareMatrix copy of SquareMatrix
}

//default copy constructor #2
SquareMatrix::SquareMatrix (const Matrix& input) : Matrix(input) {   //can return SquareMatrix copy of Matrix if Matrix is square
        //error check to see if attempting to return square copy of non-square matrix ...
        checkSquare();
}

void SquareMatrix::checkSquare () const {
        if (this->noOfRows != this->noOfCols) {
                cerr << endl;
                cerr << "Error:  In default copy constructor #2 ..." << endl;
                cerr << "       Cannot create a square matrix with unequal dimensions." << endl;
                cerr << "        Exiting program ... " << endl;
                cerr << endl;
                exit(1);
        }
}

//static zeros constructor
SquareMatrix SquareMatrix::Zeros(const int& dim) {

        //return an instance of type SquareMatrix invoked by static zeros function in class Matrix of appropriate dimensions
        return Matrix::Zeros(dim,dim);
}

//static ones constructor
SquareMatrix SquareMatrix::Ones(const int& dim) {

        //return an instance of type SquareMatrix invoked by static ones function in class Matrix of appropriate dimensions
        return Matrix::Ones(dim,dim);
}

```

```
//static identity matrix constructor
SquareMatrix SquareMatrix::Eye(const int& dim) {

        // ceate a SquareMatrix object of zeros using the standard custom constructor
        SquareMatrix Eye(dim);

        //loop through the row indices of Eye
        for (int i = 0; i < dim; ++i) {
                //looping through the column indices of Eye
                for (int j = 0; j < dim; ++j) {
                        int index = i + j*dim; //to index Eye data   (cannot use GetIndex(i,j) here)
                        if (i==j) {
                                Eye.data[index] = 1; //assign '1' to entries on main diagonal
                        }
                        else {
                                Eye.data[index] = 0; //assign '0' to other entries
                        }
                }
        }

        // return the object
        return Eye;
}

//for constructing specified test square matrices for general testing
SquareMatrix SquareMatrix::Test (const double* data_rhs, const int& dim) {

        return Matrix::Test (data_rhs, dim, dim);
}

//destructor
SquareMatrix::~SquareMatrix () {
        //will always invoke base class destructor, therefore do not delete dynamic memory here
}

// 2. ToeplitzTesting & ToeplitzTestingHelper

//creates a new Toeplitz square matrix when only first row is specified
SquareMatrix SquareMatrix::Toeplitz (const double* const row, const int& dim) {

        return Matrix::Toeplitz (row, dim, row, dim);
}

//creates a new Toeplitz square matrix when both first column and first row are specified
SquareMatrix SquareMatrix::Toeplitz (const double* const column, const double* const row, const int& dim) {

        return Matrix::Toeplitz (column, dim, row, dim);
}

// 3. TransposeTesting

//no new function declarations required.

// 4. TriangularExtractionTesting

//extract upper triangular part of square matrix, converts all values below the main diagonal to zero, non-static member version
SquareMatrix SquareMatrix::TriUpper () const {

        //initialise result with invoking matrix using default copy constructor
        SquareMatrix result(*this);

        int dim = this->noOfRows;

        // looping through the row indices of square matrix
        for (int i = 0; i < dim; ++i) {
                //looping through the column indices of square matrix
                for (int j = 0; j < dim; ++j) {
                        int index= GetIndex (i, j);  //to index entries
                        if (i>j) {
                                result.data[index] = 0;    //set values below the main diagonal to zero
                        }
                }
        }

        return result;
}

//extract lower triangular part of square matirx, converts all values above the main diagonal to zero, non-static member version
SquareMatrix SquareMatrix::TriLower () const {

        //initialise result with invoking matrix using default copy constructor
        SquareMatrix result(*this);

        int dim = this->noOfRows;

        // looping through the row indices of square matrix
        for (int i = 0; i < dim; ++i) {
                //looping through the column indices of square matrix
                for (int j = 0; j < dim; ++j) {
                        int index= GetIndex (i, j);     //to index entries
                        if (i<j) {
                                result.data[index] = 0; //set values above the main diagonal to zero
                        }
                }
        }

        return result;
}

// 5. LUDecompositionTesting - Algorithm 2.1 - Gaussian Elimination without Pivoting

//Non-Static version read only function
SquareMatrix SquareMatrix::LUDecompositionOne (const char& output) const {

        //initialise U with invoking matrix using default copy constructor
        SquareMatrix U(*this);

        //initialise L with an identity matrix of appropriate dimension
        int dim = this->noOfRows;
        SquareMatrix L = SquareMatrix::Eye(dim);

        //looping through the column indices of L and U
        for (int k = 0; k < dim-1; ++k) {
                //looping through the row indices of L and U
                for (int j = k+1; j < dim; ++j) {

                        int index_jk = GetIndex(j,k);   //to index jk entries
                        int index_kk = GetIndex(k,k);   //to index kk entries

                        //to check if diagonal entries in U (aka pivots) are zero
                        if (U.data[index_kk] == 0) {
                                cerr << endl;
                                cerr << "Error:  In Algorithm 2.1 - Gaussian Elimination without Pivoting ..." << endl;
                                cerr << "\tA zero pivot value has occurred." << endl;
                                cerr << "\tAlgorithm failed." << endl;
                                cerr << endl;
                                SquareMatrix error;   //use the default constructor to return the minimal version of SquareMatrix
                                return error;
                        }

                        //if not then proceed with algorithm  (NB - no tmp data holders required here)
                        else {
                                //to alter L_jk value
                                L.data[index_jk] = U.data[index_jk]/U.data[index_kk];

                                //to alter U values in row j from columns k to dim
                                for (int i = k; i < dim; ++i) {
                                        int index_ji = GetIndex(j,i);   //to index j,k:m entries
                                        int index_ki = GetIndex(k,i);   //to index k,k:m entries
                                        U.data[index_ji] -=  (L.data[index_jk]*U.data[index_ki]);
                                }
                        }
                }
        }

        if (output == 'L') {
                return L;
        }
        if (output == 'U') {
                return U;
        }

}

// 6. LUDecompositionTesting - Algorithm 2.2 - Gaussian Elimination with Partial Pivoting

//Non-static read only function
SquareMatrix SquareMatrix::LUDecompositionTwo (const char& output) const {

        //initialise U with invoking matrix using default copy constructor
        SquareMatrix U(*this);

        //initialise L and P with an identity matrix of appropriate dimension
        int dim = this->noOfRows;
        SquareMatrix L = SquareMatrix::Eye(dim);
        SquareMatrix P = SquareMatrix::Eye(dim);

        //looping through the column indices of L, U and P
        for (int k = 0; k < dim-1; ++k) {

                // select i>=k to maximise magnitude of U_ik  (i.e. for column k, determine entry, on or below the diagonal of U, with the largest absolute value)
                int i = 0;
                double max = 0;
                for (int b = k; b < dim; ++b) {                 //use b instead of i here to avoid confusion
                        int index_bk = GetIndex(b,k);           //to index bk entries
                        double tmp = abs(U.data[index_bk]);
                        if (tmp > max) {
                                max = tmp;
                                i = b;
                        }
                }

                // U_ik is now the new pivot in column k
                // Swap the row (k) containing the diagonal of the column (the default pivot) with the row (i) containing the new pivot, in all 3 matrices:

                //interchange U_(k,k:m) and U_(i,k:m) (i.e. swap row k with row i in U matrix, but only between columns k to m)
                U.ExchangeRows(k, i, k, dim-1);

```

```
			//interchange L_(k,1:k-1) and L_(i,1:k-1) (i.e. swap row k with row i in L matrix, but only between columns 1 to k-1)
			L.ExchangeRows(k, i, 0, k-1);

			//Interchange P_(k,:) and P (i,:) (i.e. swap row k with row i in P matrix, across all columns)
			P.ExchangeRows(k, i);

			//looping through the row indices of L and U
			for (int j = k+1; j < dim; ++j) {

				int index_jk = GetIndex(j,k);	//to index jk entries
				int index_kk = GetIndex(k,k);	//to index kk entries

				//to check if diagonal entries in U (aka pivots) are zero
				if (U.data[index_kk] == 0) {
					cerr << endl;
					cerr << "Error:  In Algorithm 2.2 - Gaussian Elimination with Partial Pivoting ..." << endl;
					cerr << "\tA zero pivot value has occurred." << endl;
					cerr << "\tAlgorithm failed." << endl;
					cerr << endl;
					SquareMatrix error;	//use the default constructor to return the minimal version of SquareMatrix
					return error;
				}

				//if not then proceed with algorithm  (NB - no tmp data holders required here)
				else {
					//to alter L_jk value
					L.data[index_jk] = U.data[index_jk]/U.data[index_kk];

					//to alter U values in row j from columns k to dim  (swapped i for a here to avoid ambiguity)
					for (int a = k; a < dim; ++a) {
						int index_ja = GetIndex(j,a);	//to index j,k:m entries
						int index_ka = GetIndex(k,a);	//to index k,k:m entries
						U.data[index_ja] -=  (L.data[index_jk]*U.data[index_ka]);
					}
				}
			}
		}

		if (output == 'L') {
			return L;
		}
		if (output == 'U') {
			return U;
		}
		if (output == 'P') {
			return P;
		}
}

// Part 3.

// 7. Forward Substitution - Algorithm 3.1

Matrix SquareMatrix::ForwardSub (const Matrix& b) const {

		//Check that invoking matrix is lower rectangular - error #1
		int dim = this->noOfCols;
		for (int i = 0; i < dim; ++i) {
			for (int j = 0; j < dim; ++j) {
				int index= GetIndex (i, j);
```

```
            if (i<j) {
                double tolerance = 1e-14;
                if (abs(this->data[index]) > tolerance ) {
                    cerr << endl;
                    cerr << "Error:  In Algorithm 3.1 - Forward Substitution..." << endl;
                    cerr << "\tMatrix L is not lower rectangular." << endl;
                    cerr << endl;
                    Matrix error;   //use the default constructor to return the minimal version of Matrix
                    return error;
                }
            }
        }
    }

    //Check that matrix b is of appropriate length - error #2
    int rows_b = b.GetNoOfRows();
    if (this->noOfRows != rows_b) {
        cerr << endl;
        cerr << "Error:  In Algorithm 3.1 - Forward Substitution..." << endl;
        cerr << "\tNumber of rows in b does not match dimensions of L." << endl;
        cerr << endl;
        Matrix error;   //use the default constructor to return the minimal version of Matrix
        return error;
    }

    //check that none of the diagonal entries in L are zero - error #3
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            int index= GetIndex (i, j);
            if (i==j) {
                double tolerance = 1e-14;
                if (abs(this->data[index]) < tolerance ) {
                    cerr << endl;
                    cerr << "Error:  In Algorithm 3.1 - Forward Substitution..." << endl;
                    cerr << "\tMatrix L cannot have a zero valued diagonal entry." << endl;
                    cerr << endl;
                    Matrix error;   //use the default constructor to return the minimal version of Matrix
                    return error;
                }
            }
        }
    }

    //once error checks are complete refer function to class Matrix for non-square matrix arithmetic and to access manipulate protected values in b
    return Matrix::ForwardSub(b);

}

// 8. Backward Substitution - Algorithm 3.2

Matrix SquareMatrix::BackwardSub (const Matrix& y) const {

    //Check that invoking matrix is upper rectangular - error #1
    int dim = this->noOfCols;
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            int index= GetIndex (i, j);
            if (i>j) {
                double tolerance = 1e-14;
                if (abs(this->data[index]) > tolerance ) {
```

```cpp
					cerr << endl;
					cerr << "Error:  In Algorithm 3.2 - Backward Substitution..." << endl;
					cerr << "\tMatrix U is not upper rectangular." << endl;
					cerr << endl;
					Matrix error;	//use the default constructor to return the minimal version of Matrix
					return error;
				}
			}
		}
	}

	//Check that matrix b is of appropriate length - error #2
	int rows_y = y.GetNoOfRows();
	if (this->noOfRows != rows_y) {
		cerr << endl;
		cerr << "Error:  In Algorithm 3.2 - Backward Substitution..." << endl;
		cerr << "\tNumber of rows in y does not match dimensions of U." << endl;
		cerr << endl;
		Matrix error;	//use the default constructor to return the minimal version of Matrix
		return error;
	}

	//check that none of the diagonal entries in U are zero - error #3
	for (int i = 0; i < dim; ++i) {
		for (int j = 0; j < dim; ++j) {
			int index= GetIndex (i, j);
			if (i==j) {
				double tolerance = 1e-14;
				if (abs(this->data[index]) < tolerance ) {
					cerr << endl;
					cerr << "Error:  In Algorithm 3.2 - Backward Substitution..." << endl;
					cerr << "\tMatrix U cannot have a zero valued diagonal entry." << endl;
					cerr << endl;
					Matrix error;	//use the default constructor to return the minimal version of Matrix
					return error;
				}
			}
		}
	}

	//once error checks are complete refer function to class Matrix for non-square matrix arithmetic and to access protected values in y
	return Matrix::BackwardSub(y);

}

// 9. Solving a system of linear equations in which the number of equations is equal to the number of unknowns, i.e. a uniquely determined solution exists.

Matrix SquareMatrix::SolveUnique (Matrix& b) const {

	//Check that matrix b is of appropriate length - error #1
	int rows_b = b.GetNoOfRows();
	if (this->noOfRows != rows_b) {
		cerr << endl;
		cerr << "Error:  In function SolveUnique..." << endl;
		cerr << "\tNumber of rows in b does not match dimensions of A." << endl;
		cerr << endl;
		Matrix error;	//use the default constructor to return the minimal version of Matrix
		return error;
```

```
        }

        //decompose A into L, U and P matrices
        SquareMatrix L = this->LUDecompositionTwo('L');
        //check that LU decomposition has occured succesfully - error #2
        if (L.noOfRows == 0) {
                //error message from LUDecompositionTwo will be displayed
                Matrix error;
                return error;
        }
        SquareMatrix U = this->LUDecompositionTwo('U');
        SquareMatrix P = this->LUDecompositionTwo('P');

        //pass these matrices to class Matrix for non-square matrix arithmetic and to access and manipulate protected values in b
        return Matrix::SolveUnique(L, U , P, b);
}
```

```cpp
        /*
 * SquareMatrixTest.cpp
 *
 *  Created on: 21.12.2017
 *      Adapted from code provided by: Gary Hui Zhang (gary.zhang@ucl.ac.uk)
 */

// A program to demonstrate and test the Matrix class and the SquareMatrix class

#include "Matrix.h"
#include "SquareMatrix.h"
#include <iostream>
#include <cstdlib>  //need this for 'system' functions


using namespace std;

void ConstructorsTesting () {
        cout << "Testing the SquareMatrix constructors:" << endl;
        cout << endl;

        cout << "Case 1: Creating a square matrix of zeros with the standard constructor:" << endl;
        {
                SquareMatrix matrix(4);  //4 is dim in example.m
                cout << matrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Creating a square matrix of zeros with the static Zeros function: " <<
endl;
        {
                SquareMatrix matrix = SquareMatrix::Zeros(4);
                cout << matrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Creating a square matrix of ones with the static Ones function: " << endl;
        {
                SquareMatrix matrix1 = SquareMatrix::Ones(4);
                cout << matrix1 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 4: Creating an identity matrix with the static Eye function:" << endl;
        {
                SquareMatrix matrix2 = SquareMatrix::Eye(4);
                cout << matrix2 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 5: Copying an identity matrix with a copy constructor:" << endl;
        {
                SquareMatrix matrix2 = SquareMatrix::Eye(4);
                cout << endl;
                cout << "The input matrix = " << endl;
                cout << matrix2 << endl;
                SquareMatrix copy_matrix2 (matrix2);
                cout << "The copy = " << endl;
                cout << copy_matrix2 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 6: Constructing a user defined matrix with the test constructor:" << endl;
        {
                double vector[16] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8};
                SquareMatrix matrix = SquareMatrix::Test(vector, 4);
                cout << matrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;
}

//'first row only' overloaded variety
void ToeplitzTestingHelper (const double* const row, const int& dim, const double* const expected) {
        cout << "The 1st row = " << endl;
        Matrix::Print(row, 1, dim);
        cout << endl;
        cout << "The matrix created by the toeplitz function in MATLAB = " << endl;
        Matrix::Print(expected, dim, dim);
        cout << endl;
        SquareMatrix matrix3 = SquareMatrix::Toeplitz(row, dim);
        cout << "The matrix created by SquareMatrix::Toeplitz = " << endl;
        cout << matrix3 << endl;

        return;
}

//'first column and first row' overloaded variety
void ToeplitzTestingHelper (const double* const column, const double* const row, const int& dim, const double* const expected) {
        cout << "The 1st column = " << endl;
        Matrix::Print(column, dim, 1);
        cout << endl;
        cout << "The 1st row = " << endl;
        Matrix::Print(row, 1, dim);
        cout << endl;
        cout << "The matrix created by the toeplitz function in MATLAB = " << endl;
        Matrix::Print(expected, dim, dim);
        cout << endl;
        SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, dim);
        cout << "The matrix created by SquareMatrix::Toeplitz = " << endl;
        cout << matrix4 << endl;

        return;
}

void ToeplitzTesting () {
        cout << "Testing the static functions SquareMatrix::Toeplitz:" << endl;
        cout << endl;

        cout << "Case 1: When only the first row is specified:" << endl;
        cout << endl;

        {
                double row[4] = {4, 3, 2, 1};  //aka vec1
                // matrix should be stored, in 1-D, column-wise
                //  4     3     2     1
                //  3     4     3     2
                //  2     3     4     3
                //  1     2     3     4
                double expected[16] = {4, 3, 2, 1, 3, 4, 3, 2, 2, 3, 4, 3, 1, 2, 3, 4};
                ToeplitzTestingHelper(row, 4, expected);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: When both the first column and the first row are specified, and the" << endl;
        cout << "        first elements are equal:" << endl;
        cout << endl;

        {
                double column[4] = {2, 1, 0, -1};
                double row[4] = {2, 3, 4, 5};
                // matrix should be stored, in 1-D, column-wise
                //  2     3     4     5
                //  1     2     3     4
                //  0     1     2     3
                // -1     0     1     2
                double expected[16] = {2, 1, 0, -1, 3, 2, 1, 0, 4, 3, 2, 1, 5, 4, 3, 2};
                ToeplitzTestingHelper(column, row, 4, expected);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: When both the first column and the first row are specified, and the" << endl;
        cout << "        first elements of each are not equal:" << endl;
        cout << endl;

        {
                double column[4] = {1, 2, 3, 4}; //aka vec2
                double row[4] = {4, 3, 2, 1};    //aka vec1
                // matrix should be stored, in 1-D, column-wise
                //  1     3     2     1
                //  2     1     3     2
                //  3     2     1     3
                //  4     3     2     1
                double expected[16] = {1, 2, 3, 4, 3, 1, 2, 3, 2, 3, 1, 2, 1, 2, 3, 1};
                ToeplitzTestingHelper(column, row, 4, expected);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;
}

void TransposeTesting () {
        cout << "Testing the class Matrix Transpose functions for SquareMatrix objects:" << endl;
        cout << endl;

        cout << "Case 1: the non-static self-modifying Transpose function:" << endl;
        cout << endl;

        {
                // the same matrix as in ToeplitzTesting case 3
                double column[4] = {1, 2, 3, 4};  //aka vec2
                double row[4] = {4, 3, 2, 1};     //aka vec1
                SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
                cout << "The original Matrix = " << endl;
                cout << matrix4 << endl;
                matrix4.Transpose();
                SquareMatrix matrix5 = matrix4;
                cout << "The transposed version = " << endl;
                cout << matrix5 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: the static Transpose function:" << endl;
        cout << endl;

        {
                // the same matrix as in ToeplitzTesting case 3
                double column[4] = {1, 2, 3, 4}; //aka vec2
                double row[4] = {4, 3, 2, 1};    //aka vec1
                SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
                cout << "The original Matrix = " << endl;
                cout << matrix4 << endl;
                SquareMatrix matrix5 = Matrix::Transpose(matrix4);
                cout << "The transposed version = " << endl;
                cout << matrix5 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;

}

void TriangularExtractionTesting () {
        cout << "Testing the triangular extraction functions of class SquareMatrix:" << endl;
        cout << endl;

        cout << "Case 1: the upper triangular extraction non-static function:" << endl;

        {
                // the same matrix as in ToeplitzTesting case 3
                double column[4] = {1, 2, 3, 4}; //aka vec2
                double row[4] = {4, 3, 2, 1};    //aka vec1
                cout << endl;
                SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
                cout << "The original square matrix = " << endl;
                cout << matrix4 << endl;
                SquareMatrix matrix6 = matrix4.TriUpper();
                cout << "The square matrix after upper triangular extraction = " << endl;
                cout << matrix6 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: the lower triangular extraction non-static function:" << endl;

        {
                // the same matrix as in ToeplitzTesting case 3
                double column[4] = {1, 2, 3, 4}; //aka vec2
                double row[4] = {4, 3, 2, 1};    //aka vec1
                cout << endl;
                SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
                cout << "The original square matrix = " << endl;
                cout << matrix4 << endl;
                SquareMatrix matrix7 = matrix4.TriLower();
                cout << "The square matrix after lower triangular extraction = " << endl;
                cout << matrix7 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

```

```cpp
        return;

}

void LUDecompositionTestingOne() {
        cout << "Testing the LU decomposition of a square matrix using Algorithm 2.1, Gaussian" << endl;
        cout << "elimination without pivoting:" << endl;
        cout << endl;

        cout << "Case 1: Matrix A is non-singular and all entries on main diagonal are non-zero:" << endl;
        cout << endl;

        {
                // the same matrix as in ToeplitzTesting case 1
                double row[4] = {4, 3, 2, 1};  //aka vec1
                SquareMatrix matrix3 = SquareMatrix::Toeplitz(row, 4);
                cout << "The input Matrix (A) = " << endl;
                cout << matrix3 << endl;
                SquareMatrix lmatrix1 = matrix3.LUDecompositionOne('L');
                cout << "The L Matrix = " << endl;
                Matrix::PrintMATLAB(lmatrix1);
                SquareMatrix umatrix1 = matrix3.LUDecompositionOne('U');
                cout << "The U Matrix = " << endl;
                Matrix::PrintMATLAB(umatrix1);
                cout << "Checking that (L*U = A):" << endl;
                cout << lmatrix1 * umatrix1 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Matrix A is non-singular and 2nd, 3rd and 4th entries on main diagonal" << endl;
        cout << "        are all zero:" << endl;
        cout << endl;

        {
                double data[16] = {6, 3, 6, 7, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
                SquareMatrix matrix = SquareMatrix::Test(data, 4);
                cout << "The input Matrix (A) = " << endl;
                cout << matrix << endl;
                SquareMatrix lmatrix = matrix.LUDecompositionOne('L');
                cout << "The L Matrix = " << endl;
                Matrix::PrintMATLAB(lmatrix);
                SquareMatrix umatrix = matrix.LUDecompositionOne('U');
                cout << "The U Matrix = " << endl;
                Matrix::PrintMATLAB(umatrix);
                cout << "Checking that (L*U = A):" << endl;
                cout << lmatrix * umatrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Matrix A is non-singular and only 1st entry on main diagonal is zero:" << endl;
        cout << endl;

        {
                double data[16] = {0, 3, 6, 7, 9, 2, 3, 8, 7, 1, 2, 9, 4, 2, 1, 5};
                SquareMatrix matrix = SquareMatrix::Test(data, 4);
                cout << "The input Matrix (A) = " << endl;
                cout << matrix << endl;
                cout << "Attempting to decompose into L and U matrices ... " << endl;
                SquareMatrix lmatrix1 = matrix.LUDecompositionOne('L');
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;

}

void LUDecompositionTestingTwo() {
        cout << "Testing the LU decomposition of a square matrix using Algorithm 2.2, Gaussian" << endl;
        cout << "elimination with partial pivoting:" << endl;
        cout << endl;

        cout << "Case 1: Matrix A is non-singular and magnitude of each entry on main diagonal is" << endl;
        cout << "        maximal within its column. Partial pivoting is not required: " << endl;
        cout << endl;

        {
                // the same matrix as in ToeplitzTesting case 1
                double row[4] = {4, 3, 2, 1};  //aka vec1
                SquareMatrix matrix3 = SquareMatrix::Toeplitz(row, 4);
                cout << "The input Matrix (A) = " << endl;
                cout << matrix3 << endl;
                SquareMatrix lmatrix1 = matrix3.LUDecompositionTwo('L');
                cout << "The L Matrix = " << endl;
                Matrix::PrintMATLAB(lmatrix1);
                SquareMatrix umatrix1 = matrix3.LUDecompositionTwo('U');
                cout << "The U Matrix = " << endl;
                Matrix::PrintMATLAB(umatrix1);
                SquareMatrix pmatrix1 = matrix3.LUDecompositionTwo('P');
                cout << "The P Matrix = " << endl;
                cout << pmatrix1 << endl;
                cout << "Checking that (L*U = P*A): " << endl;
                cout << endl;
                cout << "L*U = " << endl;
                cout << lmatrix1 * umatrix1 << endl;
                cout << "P*A = " << endl;
                cout << pmatrix1 * matrix3 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Matrix A is non-singular and magnitude of each entry on main diagonal is" << endl;
        cout << "        not maximal within its column. Partial pivoting is required: " << endl;
        cout << endl;

        {
                // the same matrix as in ToeplitzTesting case 3
                double column[4] = {1, 2, 3, 4}; //aka vec2
                double row[4] = {4, 3, 2, 1};    //aka vec1
                SquareMatrix matrix4 = SquareMatrix::Toeplitz(column, row, 4);
                cout << "The input Matrix (A) = " << endl;
                cout << matrix4 << endl;
                SquareMatrix lmatrix2 = matrix4.LUDecompositionTwo('L');
                cout << "The L Matrix = " << endl;
                Matrix::PrintMATLAB(lmatrix2);
                SquareMatrix umatrix2 = matrix4.LUDecompositionTwo('U');
                cout << "The U Matrix = " << endl;
                Matrix::PrintMATLAB(umatrix2);
                SquareMatrix pmatrix2 = matrix4.LUDecompositionTwo('P');
                cout << "The P Matrix = " << endl;
                cout << pmatrix2 << endl;
```

```cpp
                cout << "Checking that (L*U = P*A): " << endl;
                cout << endl;
                cout << "L*U = " << endl;
                cout << lmatrix2 * umatrix2 << endl;
                cout << "P*A = " << endl;
                cout << pmatrix2 * matrix4 << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Matrix A is non-singular and all diagonal entries are zero:" << endl;
        cout << endl;

        {
                double data[16] = {0, 3, 6, 7, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
                SquareMatrix matrix = SquareMatrix::Test(data, 4);;
                cout << "The input Matrix (A) = " << endl;
                cout << matrix << endl;
                SquareMatrix lmatrix = matrix.LUDecompositionTwo('L');
                cout << "The L Matrix = " << endl;
                Matrix::PrintMATLAB(lmatrix);
                SquareMatrix umatrix = matrix.LUDecompositionTwo('U');
                cout << "The U Matrix = " << endl;
                Matrix::PrintMATLAB(umatrix);
                SquareMatrix pmatrix = matrix.LUDecompositionTwo('P');
                cout << "The P Matrix = " << endl;
                cout << pmatrix << endl;
                cout << "Checking that (L*U = P*A): " << endl;
                cout << endl;
                cout << "L*U = " << endl;
                cout << lmatrix * umatrix << endl;
                cout << "P*A = " << endl;
                cout << pmatrix * matrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 4: Matrix A is non-singular, all diagonal entries are zero, and first three" << endl;
        cout << "        entries of first column are zero:" << endl;
        cout << endl;

        {
                double data[16] = {0, 0, 0, 7, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
                SquareMatrix matrix = SquareMatrix::Test(data, 4);;
                cout << "The input Matrix (A) = " << endl;
                cout << matrix << endl;
                SquareMatrix lmatrix = matrix.LUDecompositionTwo('L');
                cout << "The L Matrix = " << endl;
                Matrix::PrintMATLAB(lmatrix);
                SquareMatrix umatrix = matrix.LUDecompositionTwo('U');
                cout << "The U Matrix = " << endl;
                Matrix::PrintMATLAB(umatrix);
                SquareMatrix pmatrix = matrix.LUDecompositionTwo('P');
                cout << "The P Matrix = " << endl;
                cout << pmatrix << endl;
                cout << "Checking that (L*U = P*A): " << endl;
                cout << endl;
                cout << "L*U = " << endl;
                cout << lmatrix * umatrix << endl;
                cout << "P*A = " << endl;
                cout << pmatrix * matrix << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 5: Matrix A is singular, all diagonal entries are zero, and all entries" << endl;
        cout << "        of first column are zero:" << endl;
        cout << endl;

        {
                double data[16] = {0, 0, 0, 0, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
                SquareMatrix matrix = SquareMatrix::Test(data, 4);
                cout << "The input Matrix (A) = " << endl;
                cout << matrix << endl;
                cout << "Attempting to decompose into L, U and P matrices ... " << endl;
                SquareMatrix umatrix = matrix.LUDecompositionTwo('L');
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;
}

void ForwardSubstitution () {

        cout << "Testing Algorithm 3.1 - Forward Substitution. To solve for y in L*y = b:" << endl;
        cout << endl;

        cout << "Case 1: Successful Forward Substitution: " << endl;
        cout << endl;

        {
                double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 0, -3};
                SquareMatrix L = SquareMatrix::Test(matrix, 3);
                cout << "The lower triangular matrix, L = " << endl;
                cout << L << endl;
                double vector[3] = {16, 43, 57};
                Matrix b = Matrix::Test(vector, 3, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                cout << "The result of the forward substitution, the column vector y = " << endl;
                Matrix y = L.ForwardSub(b);
                cout << y << endl;
                cout << "Checking that (L*y = b): " << endl;
                cout << L * y << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Testing error message #1:" << endl;
        cout << endl;

        {
                double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 1, -3};
                SquareMatrix L = SquareMatrix::Test(matrix, 3);
                cout << "The input matrix, 'L' = " << endl;
                cout << L << endl;
                double vector[3] = {16, 43, 57};
                Matrix b = Matrix::Test(vector, 3, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                cout << "Attempting forward substitution ... " << endl;
                Matrix y = L.ForwardSub(b);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Testing error message #2:" << endl;
        cout << endl;

        {
                double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 0, -3};
                SquareMatrix L = SquareMatrix::Test(matrix, 3);
                cout << "The lower triangular matrix, L = " << endl;
                cout << L << endl;
                double vector[4] = {16, 43, 57, 29};
                Matrix b = Matrix::Test(vector, 4, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                cout << "Attempting forward substitution ... " << endl;
                Matrix y = L.ForwardSub(b);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 4: Testing error message #3:" << endl;
        cout << endl;

        {
                double matrix[9] = {1, 3, 4, 0, -1, 1, 0, 0, 0};
                SquareMatrix L = SquareMatrix::Test(matrix, 3);
                cout << "The lower triangular matrix, L = " << endl;
                cout << L << endl;
                double vector[3] = {16, 43, 57};
                Matrix b = Matrix::Test(vector, 3, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                cout << "Attempting forward substitution ... " << endl;
                Matrix y = L.ForwardSub(b);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;

}

void BackwardSubstitution () {

        cout << "Testing Algorithm 3.2 - Backward Substitution. To solve for x in U*x = y:" << endl;
        cout << endl;

        cout << "Case 1: Successful Backward Substitution: " << endl;
        cout << endl;

        {
                double matrix[9] = {1, 0, 0, 2, 1, 0, 3, 1, 1};
                SquareMatrix U = SquareMatrix::Test(matrix, 3);
                cout << "The upper triangular matrix, U = " << endl;
                cout << U << endl;
                double vector[3] = {16, 5, 4};
                Matrix y = Matrix::Test(vector, 3, 1);
                cout << "The column vector, y  = " << endl;
                cout << y << endl;
                cout << "The result of the backward substitution, the column vector x = " << endl;
                Matrix x = U.BackwardSub(y);
                cout << x << endl;
                cout << "Checking that (U*x = y): " << endl;
                cout << U * x << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Testing error message #1:" << endl;
        cout << endl;

        {
                double matrix[9] = {1, 0, 1, 2, 1, 0, 3, 1, 1};
                SquareMatrix U = SquareMatrix::Test(matrix, 3);
                cout << "The input matrix, 'U' = " << endl;
                cout << U << endl;
                double vector[3] = {16, 43, 57};
                Matrix y = Matrix::Test(vector, 3, 1);
                cout << "The column vector, y  = " << endl;
                cout << y << endl;
                cout << "Attempting backward substitution ... " << endl;
                Matrix x = U.BackwardSub(y);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Testing error message #2:" << endl;
        cout << endl;

        {
                double matrix[9] = {1, 0, 0, 2, 1, 0, 3, 1, 1};
                SquareMatrix U = SquareMatrix::Test(matrix, 3);
                cout << "The upper triangular matrix, U = " << endl;
                cout << U << endl;
                double vector[4] = {16, 43, 57, 29};
                Matrix y = Matrix::Test(vector, 4, 1);
                cout << "The column vector, y  = " << endl;
                cout << y << endl;
                cout << "Attempting backward substitution ... " << endl;
                Matrix x = U.BackwardSub(y);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 4: Testing error message #3:" << endl;
        cout << endl;

        {
                double matrix[9] = {0, 0, 0, 2, 1, 0, 3, 1, 1};
                SquareMatrix U = SquareMatrix::Test(matrix, 3);
                cout << "The upper triangular matrix, U = " << endl;
                cout << U << endl;
                double vector[3] = {16, 43, 57};
                Matrix y = Matrix::Test(vector, 3, 1);
                cout << "The column vector, y  = " << endl;
                cout << y << endl;
                cout << "Attempting backward substitution ... " << endl;
                Matrix x = U.BackwardSub(y);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;

}

void UniqueSolution() {

        cout << "Solving a system of linear equations where a unique solution exists:" << endl;
        cout << endl;

        cout << "Case 1: Successful solution for a 4 x 4 matrix:" << endl;
        cout << endl;

        {
                double matrix[16] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8};
                SquareMatrix A = SquareMatrix::Test(matrix, 4);
                cout << "The matrix of coefficients, A = " << endl;
                cout << A << endl;
                double vector[4] = {1, 0, 0, 0};
                Matrix b = Matrix::Test(vector, 4, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                Matrix x = A.SolveUnique(b);
                cout << "The result of solving for x in (A*x = b), the column vector x  = " << endl;
                Matrix::PrintMATLAB(x);
                cout << "Checking that (A*x = b): " << endl;
                cout << A * x << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Successful solution for a 5 x 5 matrix:" << endl;
        cout << endl;

        {
                double matrix[25] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8, 4, 6, 8, 2, 0, 9, 7, 3, 4};
                SquareMatrix A = SquareMatrix::Test(matrix, 5);
                cout << "The matrix of coefficients, A = " << endl;
                cout << A << endl;
                double vector[5] = {6, 5, 3, 1, 4};
                Matrix b = Matrix::Test(vector, 5, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                Matrix x = A.SolveUnique(b);
                cout << "The result of solving for x in (A*x = b), the column vector x  = " << endl;
                Matrix::PrintMATLAB(x);
                cout << "Checking that (A*x = b): " << endl;
                cout << A * x << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Error message testing:" << endl;
        cout << endl;

        {
                double matrix[16] = {2, 4, 8, 6, 1, 3, 7, 7, 1, 3, 9, 9, 0, 1, 5, 8};
                SquareMatrix A = SquareMatrix::Test(matrix, 4);
                cout << "The matrix of coefficients, A = " << endl;
                cout << A << endl;
                double vector[5] = {1, 0, 0, 0, 2};
                Matrix b = Matrix::Test(vector, 5, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                cout << "Attempting to solve for x in (A*x = b) ... " << endl;
                Matrix x = A.SolveUnique(b);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 4: Error message testing:" << endl;
        cout << endl;

        {
                double data[16] = {0, 0, 0, 0, 9, 0, 3, 8, 7, 1, 0, 9, 4, 2, 1, 0};
                SquareMatrix A = SquareMatrix::Test(data, 4);
                cout << "When the matrix of coefficients is singular. A = " << endl;
                cout << A << endl;
                double vector[4] = {1, 0, 0, 0};
                Matrix b = Matrix::Test(vector, 4, 1);
                cout << "The column vector, b  = " << endl;
                cout << b << endl;
                cout << "Attempting to solve for x in (A*x = b) ... " << endl;
                Matrix x = A.SolveUnique(b);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        return;
}

void LeastSquareSolution() {

        cout << "Solving an overdetermined system of linear equations for a least squares" << endl;
        cout << "solution:" << endl;
        cout << endl;

        cout << "Case 1: Successful solution for a 4 x 2 matrix:" << endl;
        cout << endl;

        {
                double matrix[8] = {4, 3, 2, 1, 3, 4, 3, 2};
                Matrix A = Matrix::Test(matrix, 4, 2);
                cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
                cout << A << endl;
                double vector1[4] = {1, 2, 3, 4};
                Matrix b = Matrix::Test(vector1, 4, 1);
                cout << "The m-by-1 column vector, b  = " << endl;
                cout << b << endl;
                SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
                cout << "Converting A into A'*A, a n-by-n matrix = " << endl;
                Matrix ATb = A.LeastSquaresHelper(b, "A'*b");
                cout << ATA << endl;
                cout << "Converting b into A'*b, a n-by-1 column vector = " << endl;
                cout << ATb << endl;
                Matrix x = ATA.SolveUnique(ATb);
                cout << "The result of solving for x in (A'A*x = A'*b), the n-by-1 column vector x = " << endl;
                Matrix::PrintMATLAB(x);
                cout << "Checking against the solution given in MATLAB for x = A\\b : " << endl;
                double vector2[2] = {-1.1724, 1.7241};
                Matrix expected = Matrix::Test(vector2, 2, 1);
                Matrix::PrintMATLAB(expected);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 2: Successful solution for a 5 x 4 matrix:" << endl;
        cout << endl;

        {
                double matrix[20] = {4, 3, 2, 1, 3, 4, 3, 2, 2, 3, 4, 3, 1, 2, 3, 4, 4, 3, 2, 1};
                Matrix A = Matrix::Test(matrix, 5, 4);
                cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
                cout << A << endl;
                double vector1[5] = {1, 2, 3, 4, 5};
                Matrix b = Matrix::Test(vector1, 5, 1);
                cout << "The m-by-1 column vector, b  = " << endl;
                cout << b << endl;
                SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
                cout << "Converting A into A'*A, a n-by-n matrix = " << endl;
                Matrix ATb = A.LeastSquaresHelper(b, "A'*b");
                cout << ATA << endl;
                cout << "Converting b into A'*b, a n-by-1 column vector = " << endl;
                cout << ATb << endl;
                Matrix x = ATA.SolveUnique(ATb);
                cout << "The result of solving for x in (A'A*x = A'*b), the n-by-1 column vector x = " << endl;
                Matrix::PrintMATLAB(x);
                cout << "Checking against the solution given in MATLAB for x = A\\b : " << endl;
                double vector2[4] = {-2.7453, 7.5652, -2.9689, -1.2236};
                Matrix expected = Matrix::Test(vector2, 4, 1);
                Matrix::PrintMATLAB(expected);
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 3: Testing warning message:" << endl;
        cout << endl;

        {

                double column[4] = {1, 2, 3, 4};
                SquareMatrix A = SquareMatrix::Toeplitz(column,4);
                cout << "When the matrix of coefficients is square. A = " << endl;
                cout << A << endl;
                double vector1[4] = {1, 2, 3, 4};
                Matrix b = Matrix::Test(vector1, 4, 1);
                cout << "The m-by-1 column vector, b  = " << endl;
                cout << b << endl;
                SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
                cout << "Converting A into A'*A, a n-by-n matrix = " << endl;
                Matrix ATb = A.LeastSquaresHelper(b, "A'*b");
                cout << ATA << endl;
                cout << "Converting b into A'*b, a n-by-1 column vector = " << endl;
                cout << ATb << endl;
                Matrix x = ATA.SolveUnique(ATb);
                cout << "The result of solving for x in (A'A*x = A'*b), the n-by-1 column vector x = " << endl;
                Matrix::PrintMATLAB(x);
                cout << "Checking that (A*x = b): " << endl;
                cout << A * x << endl;
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 4: Testing error message #1:" << endl;
        cout << endl;

        {
                double matrix[8] = {1, 1, 1, 1, 1, 2, 3, 4};
                Matrix A = Matrix::Test(matrix, 2, 4);
                cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
                cout << A << endl;
                double vector1[2] = {6, 5};
                Matrix b = Matrix::Test(vector1, 2, 1);
                cout << "The m-by-1 column vector, b  = " << endl;
                cout << b << endl;
```

```cpp
                cout << "Attempting to initiate a least squares solution ... " << endl;
                SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }

        cout << "Case 5:Testing error message #2:" << endl;
        cout << endl;

        {
                double matrix[8] = {1, 1, 1, 1, 1, 2, 3, 4};
                Matrix A = Matrix::Test(matrix, 4, 2);
                cout << "The m-by-n non-square matrix of coefficients, A = " << endl;
                cout << A << endl;
                double vector1[5] = {6, 5, 7, 10, 11};
                Matrix b = Matrix::Test(vector1, 5, 1);
                cout << "The p-by-1 column vector, b  = " << endl;
                cout << b << endl;
                cout << "Attempting to initiate a least squares solution ... " << endl;
                cout << endl;
                SquareMatrix ATA = A.LeastSquaresHelper(b,"A'*A");
                cout << "Press return to continue ..." << flush;
                system("read");
                cout << endl;
        }


        return;
}

int main () {

        for (;;) {
                cout << endl;
                cout << "Choose to test one of the following:" << endl;
                cout << endl;
                cout << "  Enter \'A\' for the Ones, Eye and other Constructors Testing" << endl;
                cout << "  Enter \'B\' for the Toeplitz function Testing" << endl;
                cout << "  Enter \'C\' for the Transpose function Testing" << endl;
                cout << "  Enter \'D\' for the Triangular Extraction Testing" << endl;
                cout << "  Enter \'E\' for the LU Decomposition Testing - Algorithm 2.1" << endl;
                cout << "  Enter \'F\' for the LU Decomposition Testing - Algorithm 2.2" << endl;
                cout << "  Enter \'G\' for the Forward Substitution Testing - Algorithm 3.1" << endl;
                cout << "  Enter \'H\' for the Backward Substitution Testing - Algorithm 3.2" << endl;
                cout << "  Enter \'I\' for Solving Systems of Linear Equations in which the number of" << endl;
                cout << "            equations is equal to the number of unknowns" << endl;
                cout << "  Enter \'J\' for Solving Systems of Linear Equations in which the number of" << endl;
                cout << "            equations is greater than the number of unknowns" << endl;
                cout << endl;
                cout << ">> ";
                char choice;
                cin >> choice;
                cout << endl;
                switch (choice) {
                        case 'A':
                        case 'a':       ConstructorsTesting();
                                                break;
                        case 'B':
                        case 'b':       ToeplitzTesting();
                                                break;
                        case 'C':
                        case 'c':       TransposeTesting();
```

```
                                                break;
                        case 'D':
                        case 'd':       TriangularExtractionTesting();
                                                break;
                        case 'E':
                        case 'e':       LUDecompositionTestingOne();
                                                break;
                        case 'F':
                        case 'f':       LUDecompositionTestingTwo();
                                                break;
                        case 'G':
                        case 'g':       ForwardSubstitution();
                                                break;
                        case 'H':
                        case 'h':       BackwardSubstitution();
                                                break;
                        case 'I':
                        case 'i':       UniqueSolution();
                                                break;
                        case 'J':
                        case 'j':       LeastSquareSolution();
                                                break;
                }
                cout << "Enter \'0\' to exit or \'1\' to choose another test" << endl;
                cout << endl;
                cout << ">> ";
                cin >> choice;
                if (choice == '0') {
                        cout << endl;
                        cout << "Goodbye!" << endl;
                        cout << endl;
                        return 0;
                }
        }

}
```