```
Script started on Thu Oct 11 20:44:09 2018
[?1034hbash-3.2$ ls
Debug                    MatrixTest1Copy2.txt     MatrxiTest3.txt
        tester.cpp               vecTest2.txt
Matrix.h        MatrixTest2.txt          Vec.h
typescript
MatrixTest1.txt          MatrixTester.cpp VecTester.cpp
vecStreamOut.txt
MatrixTest1Copy.txt      MatrixTester.h           VecTester.h
        vecTest1.txt
bash-3.2$ cat MatrixTest3.txt tester.cpp.Matr[K[K[K[K[K
MatrixTest2.txt Vec.h Matric[KxTetse[K[Kerser[1Perster[C[C[Cm[K [K.cpp
VecTester.cpp vecStreamOut.txt MatrixTester.h VecTester.h Ma
trixTest1.txt Matrix.h
cat: MatrixTest3.txt: No such file or directory
/* tester.cpp drives the testing of our Vec template and Matrix class.
  * Student Name: Nana Osei Asiedu Yirenkyi
 * Date: Oct 6 2018
 * PROJECT04
 * Begun by: Joel Adams, for CS 112 at Calvin College.
 */

//#include "VecTester.h"
#include "MatrixTester.h"

int main() {
//      VecTester vt;
//      vt.runTests();
        MatrixTester mt;
        mt.runTests();





        /*displayMenu() prints out the main menu for the
         * application
         */
        cout << "\nWelcome to the matrix application." << endl;
        while (true) {
                cout << "Please choose an operation\n "
                                "1. + Addition\n "
                                "2. - Subtraction\n "
                                "3. Transpose\n "
                                "4. exit\n Option:" << flush;
                        int choice;
                        cin >> choice;

                        //exits the program if user chooses option 4
is
}
				if (choice == 4) {
					cout << "Ending...";
					break;
				}

				Matrix<double> mat1;
				cout << "Enter the file name for 1st 
Matrix:";
				string fileName;
				cin >> fileName;
				mat1.readFrom(fileName);
				mat1.writeTo(cout);


				/* outputs the value of a third matrix
				 * using Matrix + operator or the Matrix - 
operator
				 */
				if (choice == 1 || choice == 2) {

					cout << "Enter the filename for the 
2nd Matrix: ";
					Matrix<double> mat2;
					cin >> fileName;
					mat2.readFrom(fileName);
					mat2.writeTo(cout);

					cout << "The result is:\n";

					Matrix<double> mat3;
					if (choice == 1)
						mat3 = mat1 + mat2;
					else if (choice == 2)
						mat3 = mat1 - mat2;
					mat3.writeTo(cout);
				}
				/* outputs the value of a third matrix
					 * which transposes two matrices 
contained files
					 */
				else if (choice == 3) {
					cout << "Transposition is:\n";
					Matrix<double> mat4;
					mat4 = mat1.getTranspose();
					mat4.writeTo(cout);
				}

			}

}
```

```
3 4
1 2 3 4
5 6 7 8
9 10 11 12
9 10 11 12 13/* Vec.h provides a simple vector class named Vec.
  * Student Name: Nana Osei Asiedu Yirenkyi
 * Date: Oct 6 2018
 * PROJECT04
 * Begun by: Joel Adams, for CS 112 at Calvin College.
 */

#ifndef VEC_H_
#define VEC_H_

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cassert>
using namespace std;

template<class Item>
class Vec {
public:
	Vec();
	Vec(unsigned size);
	Vec(const Vec& original);
	Vec<Item>& operator=(const Vec<Item>& original);
	Vec<Item> operator-(const Vec<Item>& v2);
	Vec<Item> operator+(const Vec<Item>& v2);
	unsigned getSize() const;
	Item getItem(unsigned index) const;
	Item& operator[] (const unsigned index);
	Item operator*(const Vec& v2);
	bool operator==(const Vec<Item>& v2) const;
	bool operator!=(const Vec& v2);
	void setItem(unsigned index, const Item& it);
	void setSize(unsigned newSize);
	void writeTo(ostream& out) const;
	void readFrom(istream& in);
	void readFrom(string fileName);
	void writeTo(const string& fileName) const;
	const Item& operator[] (unsigned index) const;
	virtual ~Vec();

private:
	unsigned mySize;
	Item * myArray;
```

```
        friend class VecTester;

};


//Default constructor
template<class Item>
Vec<Item>::Vec() {
        mySize = 0;
        myArray = NULL;
}

//Explicit Constructor
template<class Item>
Vec<Item>::Vec(unsigned size) {
        mySize = size;
        if (size > 0) {
                myArray = new Item[size];
                for (unsigned i = 0; i < mySize; i++){
                        myArray[i] = 0;
                }
        }
        else myArray = NULL;
}

/*copy constructor that makes a distinct copy of the object,
 *  including its dynamically allocated memory.
 */
template<class Item>
Vec<Item>::Vec(const Vec<Item>& original) {
        mySize = original.mySize;
        if (mySize > 0) {
                myArray = new Item[mySize];
                for (unsigned i = 0; i < mySize; i++) {
                        myArray[i] = original.myArray[i];
                }
        }
        else myArray = NULL;
}


// Destructor
// returns dynamically allocated memory to the system using the delete
operation
template<class Item>
Vec<Item>::~Vec() {
         delete[] myArray;
         myArray = NULL;
         mySize = 0;
}
```

```
template<class Item>
Vec<Item>& Vec<Item>::operator=(const Vec<Item>& original) {
        if (this != &original) {
                if (mySize > 0) {
                        delete [] myArray;
                        myArray = NULL;
                }
                if (original.mySize > 0) {
                        myArray = new Item[original.mySize];
                }
                mySize = original.mySize;
        }
        for (unsigned i = 0; i < mySize; i++) {
                myArray[i] = original.myArray[i];
        }
        return *this;
   }

//obtains the size of the vector
template<class Item>
unsigned Vec<Item>::getSize() const {
        return mySize;
}

//sets a particular item in the vector to a certain value
template<class Item>
void Vec<Item>::setItem(unsigned index, const Item& it) {
        if (index >= mySize){
                throw range_error("Out of range");
        }
        else {
                myArray[index] = it;
        }
}
//
//gets a value of an item at a particular index in vector
template<class Item>
Item Vec<Item>::getItem(unsigned index) const {
        if (index >= mySize) {
                throw range_error("Out of range");
        }
        else {
                return myArray[index];
        }
}
//
//sets the size of an vector
template<class Item>
```

```
void Vec<Item>::setSize(unsigned newSize) {
	if (mySize != newSize) {
		if (newSize == 0) {
			delete [] myArray;
			myArray = NULL;
			mySize = 0;
		}
		else {
			Item* newArray = new Item[newSize];
			if (mySize < newSize) {
				for (unsigned i = 0; i < mySize; i+
+) {
					newArray[i] = myArray[i];
				}
				for (unsigned i = mySize; i <
newSize; i++) {
					newArray[i] = 0;
				}
			}
			else {
				for (unsigned i = 0; i < newSize; i+
+) {
					newArray[i] = myArray[i];
				}
			}
			mySize = newSize;
			delete [] myArray;
			myArray = newArray;
		}
	}
}
//
//finds out if items in vector are equal
template<class Item>
bool Vec<Item>::operator==(const Vec<Item>& v2) const {
	if (mySize != v2.mySize) {
		return false;
	}
	for (unsigned i = 0; i < mySize; i++) {
		if (myArray[i] != v2.myArray[i]) {
			return false;
		}
	}
	return true;
}
//
//outputs items in a vector to a text file
template<class Item>
void Vec<Item>::writeTo(ostream& out) const {
	for (unsigned i = 0; i < mySize; i++) {
```

```
                out << myArray[i] << "\n";
        }
}
//
//reads in values to indices in the vector
template<class Item>
void Vec<Item>::readFrom(istream& in) {
        for (unsigned i = 0; i < mySize; i++) {
                in >> myArray[i];
        }
}
//
//
                                                                            //
NANA OSEI//
/*Inequality operator != checks if myArray and v2.myArray
 *              are NOT equal
 * @param: v2, a Vec class object
 * Return: true/false
 */
template<class Item>
bool Vec<Item>::operator!=(const Vec& v2) {
        if (mySize != v2.mySize) {
                return true;
        }
        for (unsigned i = 0; i < mySize; i++) {
                if (myArray[i] != v2.myArray[i]) {
                        return true;
                }
        }
        return false;
}
//
//
//
/*Vector subtraction operator subtracts two vectors and sets a third
 *              vector as equal to them
 *@param: v2, a Vec class object
 *Return: v3, a Vec class object equal to the values in
 *              myArray minus the values in v2.myArray
 */
template<class Item>
Vec<Item> Vec<Item>::operator-(const Vec<Item>& v2) {
        Vec v3(mySize);
        if (mySize != v2.mySize) {
                throw invalid_argument("Invalid Argument!");
        }
        else if (v2.mySize == 0) {
                return v3;
        }
        for (unsigned i = 0; i < mySize; i++) {
                v3.myArray[i] = myArray[i] - v2.myArray[i];
        }
        return v3;
}
//
//
/* Vec readFrom reads values from a filename directly
 *              into myArray; the first values read in the
 *              file will be mySize
 * @filename: a string object, will be used as
 *              the file read from
 */
template<class Item>
void Vec<Item>::readFrom(string fileName) {
        ifstream fin(fileName.c_str());
        fin >> mySize;
        myArray = new Item[mySize];
        for (unsigned i = 0; i <= mySize; i++) {
                fin >> myArray[i];
        }
        fin.close();
}
//
//
//
/* Subscript to retrieve value method for constant
 *              reference values
 * @index: an unsigned int that must not be out
 *                      of range for myArray
 * Return: myArray[index]
 * (read)
 */
template<class Item>
const Item& Vec<Item>::operator[] (unsigned index) const {
        if (index >= mySize) {
                throw range_error ("Invalid Subscript.");
        }
        else {
                return myArray[index];
        }
}
//
//
//
//
                                                  //LUKE CHEN//
//
/* define a subscript operator [i] to allow user to change the value
of Vec at index i
```

```
 * @param: unsigned index i
 * postcondition: change the value of Vec at index i
 * built by: Luke Chen lc33
 */
template<class Item>
Item & Vec<Item>::operator[](unsigned index) {
    if (index >= mySize) {
        throw range_error("Index out of range");
    }
    return myArray[index];
}
//
//
//
/* define a writeTo function to write all the values in a Vec to the
file named by fileName
 * @param: string& fileName
 * postcondition: the values of the Vec are written in the file
 * built by: Luke Chen lc33
 */
template<class Item>
void Vec<Item>::writeTo(const string& fileName) const {
    ofstream fout( fileName.c_str() );
    assert( fout.is_open() );
    fout << mySize << '\n';
    for (unsigned i = 0; i < mySize; i++) {
        fout << myArray[i] << '\n';
    }
    fout.close();
}
//
//
//
/* define an addition operator + to allow addition of vector values
 * return: newArray.myArray[i] = myArray[i] + v2.myArray[i]
 * built by: Luke Chen lc33
 */
template<class Item>
Vec<Item> Vec<Item>::operator+(const Vec<Item>& v2) {
    Vec newVec(mySize);
    if (mySize != v2.mySize) {
        throw invalid_argument("vectors have different sizes");
    } else if (v2.mySize == 0) {
        return newVec;
    }
    for (unsigned i = 0; i < mySize; i++) {
        newVec.myArray[i] = myArray[i] + v2.myArray[i];
    }
    return newVec;
}
```

```
//
//
//
//
/* define a dot product operator * to allow dot production of vector
values
 * return: product += myArray[i] * v2.myArray[i]
 * built by: Luke Chen lc33
 */
template<class Item>
Item Vec<Item>::operator*(const Vec& v2) {
    Item product = 0;
    if (mySize != v2.mySize) {
        throw invalid_argument("vectors have different sizes");
    } else if (v2.mySize == 0) {
        return product;
    }
    for (unsigned i = 0; i < mySize; i++) {
        product += myArray[i] * v2.myArray[i];
    }
    return product;
}



#endif /*VEC_H_*/
/* MatrixTester.cpp defines test-methods for class Matrix.
 * Joel Adams, for CS 112 at Calvin College.
 */

#include "MatrixTester.h"
#include <iostream>        // cout, cerr, ...
#include <cstdlib>         // exit()
#include <stdexcept>       // range_error, ...
#include <cassert>
using namespace std;

void MatrixTester::runTests() {
        cout << "Running Matrix tests..." << endl;
        testDefaultConstructor();
        testExplicitConstructor();
        testCopyConstructor();
        testAssignment();
        testEquality();
        testSubscripts();
        testInequality();
        testTranspose();
        testAddition();
        testSubtraction();
//      testMultiply();
```

```
        testReadFromStream();
        testWriteToStream();
        testReadFromFile();
        testWriteToFile();
        cout << "All tests passed!" << endl;
}


void MatrixTester::testDefaultConstructor() {
        cout << "Testing default constructor..." << flush;
        Matrix<int> m1;
        assert( m1.getRows() == 0 );
        assert( m1.getColumns() == 0 );
        cout << " 0 " << flush;

        Matrix<int> m2;
        assert( m2.getRows() == 0 );
        assert( m2.getColumns() == 0 );
        cout << " 1 " << flush;

        cout << "Passed!" << endl;
}

void MatrixTester::testExplicitConstructor() {
        cout << "Testing explicit constructor..." << flush;
        Matrix<int> m1(2, 3);
        assert( m1.getRows() == 2 );
        assert( m1.getColumns() == 3 );
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        assert( m1.myVec[i][j] == 0 );
                }
        }
        cout << " 1 " << flush;

        Matrix<int> m2(3, 1);
        assert( m2.getRows() == 3 );
        assert( m2.getColumns() == 1 );
        for (unsigned i = 0; i < m2.getRows(); i++) {
                for (unsigned j = 0; j < m2.getColumns(); j++) {
                        assert( m2.myVec[i][j] == 0 );
                }
        }
        cout << " 2 " << flush;

        cout << "Passed!" << endl;
}

void MatrixTester::testCopyConstructor() {
        cout << "Testing copy constructor... " << flush;
        // copy empty matrix
        Matrix<int> m1;
        Matrix<int> m2(m1);
        assert( m2.getRows() == 0 );
        assert( m2.getColumns() == 0 );
        assert( m2.myVec.getSize() == 0 );
        cout << " 1 " << flush;

        // copy non-empty matrix
        Matrix<int> m3(2, 3);
        for (unsigned i = 0; i < m3.getRows(); i++) {
                for (unsigned j = 0; j < m3.getColumns(); j++) {
                        m3.myVec[i][j] = i+j;
                }
        }
        Matrix<int> m4(m3);
        assert( m4.getRows() == m3.getRows() );
        assert( m4.getColumns() == m3.getColumns() );
        assert( m4.myVec.getSize() == m3.myVec.getSize() );
        for (unsigned i = 0; i < m3.getRows(); i++) {
                for (unsigned j = 0; j < m3.getColumns(); j++) {
                        assert( m4.myVec[i][j] == i+j );
                }
        }
        cout << " 2 " << flush;

        cout << " Passed!" << endl;
}

void MatrixTester::testAssignment() {
        cout << "Testing =... " << flush;
        // empty to empty
        Matrix<int> m0;
        Matrix<int> m1;
        m0 = m1;
        assert( m0.getRows() == 0 );
        assert( m0.getColumns() == 0 );
        assert( m0.myVec.getSize() == 0 );
        cout << " 0 " << flush;

        // nonempty to empty
        Matrix<int> m2(2,3);
        m0 = m2;
        assert( m0.getRows() == 2 );
        assert( m0.getColumns() == 3 );
        assert( m0.myVec == m2.myVec );
        cout << " 1 " << flush;

        // empty to nonempty
        m2 = m1;
```

```
	assert( m2.getRows() == 0 );
	assert( m2.getColumns() == 0 );
	assert( m2.myVec == m1.myVec );
	cout << " 2 " << flush;

	// non-empty to non-empty
	Matrix<int> m3(2, 3);
	for (unsigned i = 0; i < m3.getRows(); i++) {
		for (unsigned j = 0; j < m3.getColumns(); j++) {
			m3[i][j] = i+j;
		}
	}
	Matrix<int> m4(5, 4);
	m4 = m3;
	assert( m4.getRows() == 2 );
	assert( m4.getColumns() == 3 );
	assert( m4.myVec == m3.myVec );
	cout << " 3 " << flush;

	// chaining
	Matrix<int> m5;
	m5 = m0 = m4;
	assert( m0.getRows() == 2 );
	assert( m0.getColumns() == 3 );
	assert( m0.myVec == m4.myVec );
	assert( m5.getRows() == 2 );
	assert( m5.getColumns() == 3 );
	assert( m5.myVec == m0.myVec );
	cout << " 4 " << flush;

	// self-assignment
	m5 = m5;
	assert( m5.getRows() == 2 );
	assert( m5.getColumns() == 3 );
	for (unsigned i = 0; i < m5.getRows(); i++) {
		for (unsigned j = 0; j < m5.getColumns(); j++) {
			assert( m5[i][j] == i+j );
		}
	}
	cout << " 5 " << flush;

	cout << "Passed!" << endl;
}

void MatrixTester::testEquality() {
	cout << "Testing ==... " << flush;
	// empty
	Matrix<int> m1;
	Matrix<int> m2;
	assert( m1 == m2 );
```

```
        cout << " 1 " << flush;

        // same sized
        Matrix<int> m3(2, 3);
        Matrix<int> m4a(2, 3);
        Matrix<int> m4b(2, 3);
        Matrix<int> m5(2, 3);
        for (unsigned i = 0; i < m3.getRows(); i++) {
                for (unsigned j = 0; j < m3.getColumns(); j++) {
                        m3[i][j] = i+j;
                        m4a[i][j] = i+j;
                        m4b[i][j] = i+j;
                        m5[i][j] = i*j;
                }
        }
        assert( m3 == m4a );
        assert( !(m1 == m3) );
        assert( !(m3 == m5) );
        m4b[1][1] = 0;
        assert( !(m3 == m4b) );
        cout << " 2 " << flush;

        // different row sizes
        Matrix<int> m6(3,3);
        Matrix<int> m7(2,3);
        assert( !(m6 == m7) );
        cout << " 3 " << flush;

        // same row sizes, different column sizes
        Matrix<int> m8(2,4);
        assert( !(m8 == m7 ) );
        cout << " 4 " << flush;

        cout << "Passed!" << endl;
}

void MatrixTester::testReadSubscript(const Matrix<int>& mat) {
        for (unsigned i = 0; i < mat.getRows(); i++) {
                for (unsigned j = 0; j < mat.getColumns(); j++) {
                        assert( i*j == mat[i][j] );
                }
        }
}

void MatrixTester::testSubscripts() {
        cout << "Testing subscripts... " << flush;
        Matrix<int> m(4, 3);
        // test write-subscript
        for (unsigned i = 0; i < m.getRows(); i++) {
                for (unsigned j = 0; j < m.getColumns(); j++) {
```

```
                                m[i][j] = i*j;
                        }
                }
                cout << " 1 " << flush;

                // test read-subscript -- see above
                testReadSubscript(m);
                cout << " 2 " << flush;

                // exceptions
                // empty Matrix
                Matrix<int> m0;

                // read subscript
                try {
                        m0[0][0];
                        cerr << "successfully read from empty Matrix";
                        exit(1);
                } catch (range_error&) {
                        cout << " 3 " << flush;
                }

                // write subscript
                try {
                        m0[0][0] = 0;
                        cerr << "successfully wrote to empty Matrix";
                        exit(1);
                } catch (range_error&) {
                        cout << " 4 " << flush;
                }

                // nonempty Matrix
                Matrix<int> m2(2,3);

                // beyond last row
                try {
                        m2[2][0] = 0;
                        cerr << "successfully wrote past last Matrix row";
                        exit(1);
                } catch (range_error&) {
                        cout << " 5 " << flush;
                }

                // beyond last column
                try {
                        m2[0][3] = 0;
                        cerr << "successfully wrote past last Matrix column";
                        exit(1);
                } catch (range_error&) {
                        cout << " 6 " << flush;
```

```
		}

		cout << "Passed!" << endl;
	}


	void MatrixTester::testInequality() {
		cout << "Testing !=... " << flush;

		// empty
		Matrix<int> m1;
		Matrix<int> m2;
		assert( !(m1 != m2) );
		cout << " 0 " << flush;

		// nonempty, same size
		Matrix<int> m3(2, 3);
		Matrix<int> m4(2, 3);
		Matrix<int> m5(2, 3);
		for (unsigned i = 0; i < m3.getRows(); i++) {
			for (unsigned j = 0; j < m3.getColumns(); j++) {
				m3[i][j] = i+j;
				m4[i][j] = i+j;
				m5[i][j] = i*j;
			}
		}
		assert( !(m3 != m4) );
		assert( m1 != m3 );
		assert( m3 != m5 );
		cout << " 1 " << flush;

		// nonempty, different row sizes
		Matrix<int> m6(3,3);
		for (unsigned i = 0; i < m6.getRows(); i++) {
			for (unsigned j = 0; j < m6.getColumns(); j++) {
				m6[i][j] = i+j;
			}
		}
		assert( m3 != m6 );
		cout << " 2 " << flush;

		// nonempty, different column sizes
		Matrix<int> m7(2, 4);
		for (unsigned i = 0; i < m7.getRows(); i++) {
			for (unsigned j = 0; j < m7.getColumns(); j++) {
				m7[i][j] = i+j;
			}
		}
		assert( m3 != m7 );
		cout << " 3 " << flush;
```

```
        cout << "Passed!" << endl;
}

void MatrixTester::testTranspose() {
        cout << "Testing getTranspose()... " << flush;
        // empty
        Matrix<int> m0a, m0b(2,3);
        m0b = m0a.getTranspose();
        assert( m0b.getRows() == 0 );
        assert( m0b.getColumns() == 0 );
        assert( m0b.myVec.getSize() == 0 );
        cout << " 0 " << flush;

        // nonempty
        Matrix<int> m1(4, 3);
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        m1[i][j] = i*j;
                }
        }
        Matrix<int> m2 = m1.getTranspose();
        assert( m2.getRows() == m1.getColumns() );
        assert( m2.getColumns() == m1.getRows() );
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        assert( m2[j][i] == m1[i][j] );
                }
        }
        cout << " 1a " << flush;
        // check that m1 is unchanged
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        assert( m1[i][j] == i*j );
                }
        }
        cout << " 1b " << flush;

        cout << "Passed!" << endl;
}

void MatrixTester::testAddition() {
        cout << "Testing +... " << flush;
        // empty
        Matrix<int> m0a, m0b, m0c(3,2);
        m0c = m0a + m0b;
        assert( m0c.getRows() == 0 );
        assert( m0c.getColumns() == 0 );
        assert( m0c.myVec.getSize() == 0 );
        cout << " 0 " << flush;
        // nonempty, same size
        Matrix<int> m1(3, 4);
        Matrix<int> m2(3, 4);
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        m1[i][j] = i*j;
                        m2[i][j] = i+j;
                }
        }

        Matrix<int> m3 = m1 + m2;

        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        assert( m3[i][j] == i*j + i+j );
                }
        }
        cout << " 1a " << flush;

        // check that left operand did not change
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        assert( m1[i][j] == i*j );
                }
        }
        cout << " 1b " << flush;

        // nonempty, different sized rows
        Matrix<int> m4(4,4);
        try {
                m3 = m2 + m4;
                cerr << "operator+ worked with different row sizes";
                exit(1);
        } catch (invalid_argument&) {
                cout << " 2 " << flush;
        }

        // nonempty, different sized columns
        Matrix<int> m5(3,3);
        try {
                m3 = m2 + m5;
                cerr << "operator+ worked with different column
sizes";
                exit(1);
        } catch (invalid_argument&) {
                cout << " 3 " << flush;
        }

        cout << "Passed!" << endl;
```

```
}

void MatrixTester::testSubtraction() {
        cout << "Testing -... " << flush;
        // empty
        Matrix<int> m0a, m0b, m0c(3,2);
        m0c = m0a - m0b;
        assert( m0c.getRows() == 0 );
        assert( m0c.getColumns() == 0 );
        assert( m0c.myVec.getSize() == 0 );
        cout << " 0 " << flush;

        //non-empty, valid
        Matrix<int> m1(3, 4);
        Matrix<int> m2(3, 4);
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        m1[i][j] = i*j;
                        m2[i][j] = i+j;
                }
        }

        Matrix<int> m3 = m1 - m2;
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        assert( m3[i][j] == m1[i][j] - m2[i][j] );
                }
        }
        cout << " 1a " << flush;

        // check that left operand did not change
        for (unsigned i = 0; i < m1.getRows(); i++) {
                for (unsigned j = 0; j < m1.getColumns(); j++) {
                        assert( m1[i][j] == i*j );
                }
        }
        cout << " 1b " << flush;

        // nonempty, different sized rows
        Matrix<int> m4(4,4);
        try {
                m3 = m2 - m4;
                cerr << "operator- worked with different row sizes";
                exit(1);
        } catch (invalid_argument&) {
                cout << " 2 " << flush;
        }

        // nonempty, different sized columns
        Matrix<int> m5(3,3);
        try {
                m3 = m2 - m5;
                cerr << "operator- worked with different column
sizes";
                exit(1);
        } catch (invalid_argument&) {
                cout << " 3 " << flush;
        }

        cout << "Passed!" << endl;
}

//void MatrixTester::testMultiply() {
//      cout << "Testing *..." << flush;
//      // empty
//      Matrix<int> m0a, m0b, m0c(3,2);
//      m0c = m0a * m0b;
//      assert( m0c.getRows() == 0 );
//      assert( m0c.getColumns() == 0 );
//      assert( m0c.myVec.getSize() == 0 );
//      cout << " 0 " << flush;
//
//      Matrix<int> m1(2, 3);
//      Matrix<int> m2(3, 2);
//      for (unsigned i = 0; i < m1.getRows(); i++) {
//              for (unsigned j = 0; j < m1.getColumns(); j++) {
//                      m1[i][j] = i+j+1;
//                      m2[j][i] = i+j+1;
//              }
//      }
//
//      Matrix<int> m3 = m1 * m2;
//      assert( m3.getRows() == 2);
//      assert( m3.getColumns() == 2 );
//      assert( m3[0][0] == 14 );
//      assert( m3[0][1] == 20 );
//      assert( m3[1][0] == 20 );
//      assert( m3[1][1] == 29 );
//      cout << " 1a " << flush;
//
//      // check that left operand did not change
//      for (unsigned i = 0; i < m1.getRows(); i++) {
//              for (unsigned j = 0; j < m1.getColumns(); j++) {
//                      assert( m1[i][j] == i+j+1 );
//              }
//      }
//      cout << " 1b " << flush;
//
//      // nonempty, m1.columns != m2.rows
//      Matrix<int> m4(3,3);
```

```
//	try {
//		m3 = m2 * m4;
//		cerr << "operator* worked with bad row/column sizes";
//		exit(1);
//	} catch (invalid_argument&) {
//		cout << " 2 " << flush;
//	}
//
//	cout << "Passed!" << endl;
//}

void MatrixTester::testReadFromStream() {
	cout << "Testing readFrom(istream)... " << flush;
	ifstream fin("MatrixTest1.txt");
	assert( fin.is_open() );
	unsigned rows, columns;
	fin >> rows >> columns;
	Matrix<int> m(rows, columns);
	m.readFrom(fin);
	for (unsigned i = 0; i < rows; i++) {
		for (unsigned j = 0; j < columns; j++) {
			assert( m[i][j] == i*m.getColumns()+j+1 );
		}
	}

	cout << "Passed!" << endl;
}

void MatrixTester::testReadFromFile() {
	cout << "Testing readFrom(string)... " << flush;
	Matrix<int> m;
	m.readFrom("MatrixTest1.txt");
	assert( m.getRows() == 3 );
	assert( m.getColumns() == 4 );
	for (unsigned i = 0; i < m.getRows(); i++) {
		for (unsigned j = 0; j < m.getColumns(); j++) {
			assert( m[i][j] == i*m.getColumns()+j+1 );
		}
	}

	cout << "Passed!" << endl;
}

void MatrixTester::testWriteToStream() {
	cout << "Testing writeTo(ostream)... " << flush;
	Matrix<int> m;
	// read a Matrix whose values we know
	m.readFrom("MatrixTest1.txt");
	ofstream fout("MatrixTest1Copy.txt");
	assert( fout.is_open() );
```

```
        // now write it to a file via a stream
        fout << m.getRows() << " " << m.getColumns() << "\n";
        m.writeTo(fout);
        fout.close();
        // now, read what we just wrote into a different Matrix
        Matrix<int> m1;
        m1.readFrom("MatrixTest1Copy.txt");
        // and test it
        assert( m1.getRows() == 3 );
        assert( m1.getColumns() == 4 );
        for (unsigned i = 0; i < m.getRows(); i++) {
                for (unsigned j = 0; j < m.getColumns(); j++) {
                        assert( m1[i][j] == i*m.getColumns()+j+1 );
                }
        }

        cout << "Passed!" << endl;
}

void MatrixTester::testWriteToFile() {
        cout << "Testing writeTo(string)... " << flush;
        Matrix<int> m;
        // read in a Matrix whose values we know
        m.readFrom("MatrixTest1.txt");
        // write it to a file
        m.writeTo("MatrixTest1Copy2.txt");
        // now, read what we just wrote into a different Matrix
        Matrix<int> m1;
        m1.readFrom("MatrixTest1Copy2.txt");
        // test it
        assert( m1.getRows() == 3 );
        assert( m1.getColumns() == 4 );
        for (unsigned i = 0; i < m.getRows(); i++) {
                for (unsigned j = 0; j < m.getColumns(); j++) {
                        assert( m1[i][j] == i*m.getColumns()+j+1 );
                }
        }

        cout << "Passed!" << endl;
}

/* VecTester.cpp defines the unit test-methods for Vec, a simple
vector class.
 * Student Name: Nana Osei Asiedu Yirenkyi
 * Date: October 2 2018
 * Begun by: Joel C. Adams, for CS 112 at Calvin College.
 */

#include "VecTester.h"
#include <iostream>     // cout, cerr, ...
```

```
#include <cassert>      // assert()
#include <cstdlib>      // exit()
#include <stdexcept>    // range_error, ...
using namespace std;

void VecTester::runTests() const {
        cout << "Testing class Vec" << endl;
        testDefaultConstructor();
        testExplicitConstructor();
        testCopyConstructor();
        testDestructor();
        testAssignment();
        testSetSize();
        testGetSize();
        testSetItem();
        testGetItem();
        testEquality();
        testWriteToStream();
        testReadFromStream();
        testSubscript();
        testInequality();
        testAddition();
        testSubtraction();
        testDotProduct();
        testReadFromFile();
        testWriteToFile();
        cout << "All tests passed!" << endl;
}

void VecTester::testDefaultConstructor() const {
        cout << "Testing default constructor... " << flush;
        Vec<double> v;
        assert( v.mySize == 0 );
        assert( v.myArray == NULL );
        cout << "Passed!" << endl;
}

void VecTester::testExplicitConstructor() const {
        cout << "Testing explicit constructor... " << flush;
        cout << " 1 " << flush;
        Vec<double> v1(3);
        assert( v1.mySize == 3 );
        assert( v1.myArray != NULL );
        for (int i = 0; i < 3; i++) {
                assert( v1.myArray[i] == 0 );
        }
        cout << " 2 " << flush;
        Vec<double> v2(8);
        assert( v2.mySize == 8 );
        assert( v2.myArray != NULL );
        for (int i = 0; i < 8; i++) {
                assert( v2.myArray[i] == 0 );
        }
        cout << "Passed!" << endl;
}

void VecTester::testCopyConstructor() const {
        cout << "Testing copy constructor..." << flush;
        cout << " 1 " << flush;
        Vec<double> v1;
        Vec<double> v2(v1);
        assert( v2.mySize == 0 );
        assert( v2.myArray == NULL);

        cout << " 2 " << flush;
        Vec<double> v3(5);
        Vec<double> v4(v3);
        assert(v4.mySize == 5);
        assert(v4.myArray != NULL);
        assert(v4.myArray != v3.myArray);
        for (unsigned i = 0; i < 5; i++) {
                assert( v4.myArray[i] == 0 );
        }

        cout << " 3 " << flush;
        Vec<double> v5(5);
        for (unsigned i = 0; i < 5; i++) {
                v5.myArray[i] = (i+1);
        }
        Vec<double> v6(v5);
        assert( v6.mySize == 5 );
        assert( v6.myArray != NULL );
        assert( v6.myArray != v5.myArray );
        for (unsigned i = 0; i < 5; i++) {
                assert( v6.myArray[i] == v5.myArray[i] );
        }
        cout << "Passed!" << endl;
}

void VecTester::testDestructor() const {
        cout << "Testing destructor... " << flush;
        Vec<double> v(5);
        v.~Vec();
        assert( v.mySize == 0 );
        assert( v.myArray == NULL );
        cout << "Passed, but make sure ~Vec() is returning the array's
memory to the system!" << endl;
}

void VecTester::testAssignment() const {
```

```
cout << "Testing =..." << flush;
// empty-to-empty
Vec<double> v, v0;
v = v0;
assert(v.mySize == 0);
assert(v.myArray == NULL);
cout << " 0 " << flush;
// empty-to-nonempty
Vec<double> v1;
Vec<double> v2(5);
v2 = v1;
assert(v2.mySize == 0);
assert(v2.myArray == NULL);
cout << " 1 " << flush;
// nonempty-to-empty
Vec<double> v3(5);
for (unsigned i = 0; i < 5; i++) {
        v3.myArray[i] = (i+1);
}
Vec<double> v4;
v4 = v3;
assert( v4.mySize == 5 );
assert( v4.myArray != v3.myArray );
for (unsigned i = 0; i < 5; i++) {
        assert( v4.myArray[i] == (i+1) );
}
cout << " 2 " << flush;
// nonempty-to-nonempty (larger into smaller)
Vec<double>  v5(2);
for (unsigned i = 0; i < 2; i++) {
        v5.myArray[i] = (i+1)*10;
}
v5 = v3;
assert(v5.mySize == 5);
assert(v5.myArray != v3.myArray);
for (unsigned i = 0; i < 5; i++) {
        assert( v5.myArray[i] == (i+1) );
}
cout << " 3 " << flush;
// nonempty-to-nonempty (smaller into larger)
Vec<double>  v6(7);
for (unsigned i = 0; i < 7; i++) {
        v6.myArray[i] = (i+1)*10;
}
v6 = v3;
assert(v6.mySize == 5);
assert(v6.myArray != v3.myArray);
for (unsigned i = 0; i < 5; i++) {
        assert( v6.myArray[i] == (i+1) );
}
```

```
        cout << " 4 " << flush;
        // nonempty-to-nonempty (equal sized)
        Vec<double>  v7(5);
        for (unsigned i = 0; i < 5; i++) {
                v7.myArray[i] = (i+1)*10;
        }
        v7 = v3;
        assert(v7.mySize == 5);
        assert(v7.myArray != v3.myArray);
        for (unsigned i = 0; i < 5; i++) {
                assert( v7.myArray[i] == (i+1) );
        }
        cout << " 5 " << flush;
        // assignment chaining
        Vec<double>  v8;
        Vec<double>  v9(4);
        v9 = v8 = v3;
        assert( v9.mySize == 5 );
        assert( v9.mySize == 5 );
        assert( v8.myArray != v3.myArray );
        assert( v8.myArray != v3.myArray );
        assert( v9.myArray != v8.myArray );
        for (unsigned i = 0; i < 5; i++) {
                assert( v8.myArray[i] == (i+1) );
                assert( v9.myArray[i] == (i+1) );
        }
        cout << " 6 " << flush;
        // self-assignment (idiotic but legal)
        v3 = v3;
        assert( v3.mySize == 5 );
        assert( v3.myArray != NULL );
        for (unsigned i = 0; i < 5; i++) {
                assert(v3.myArray[i] == (i+1) );
        }
        cout << " 7 " << flush;
        cout << "Passed!" << endl;
}

void VecTester::testSetSize() const {
        cout << "Testing setSize()..." << flush;
        // empty
        Vec<double>  v0;
        v0.setSize(3);
        assert( v0.getSize() == 3 );
        for (unsigned i = 0; i < 3; i++) {
                assert( v0.getItem(i) == 0 );
        }
        cout << " 0 " << flush;
        // non-empty, increasing
        Vec<double>  v1(5);
        for (unsigned i = 0; i < 5; i++) {
                v1.setItem(i, i+1);
        }
        v1.setSize(8);
        assert( v1.getSize() == 8 );
        for (unsigned i = 0; i < 5; i++) {
                assert( v1.getItem(i) == (i+1) );
        }
        for (unsigned i = 5; i < 8; i++) {
                assert( v1.getItem(i) == 0 );
        }
        cout << " 1 " << flush;
        // non-empty, decreasing
        Vec<double>  v2(5);
        for (unsigned i = 0; i < 5; i++) {
                v2.setItem(i, i+1);
        }
        v2.setSize(3);
        assert( v2.getSize() == 3 );
        for (unsigned i = 0; i < 3; i++) {
                assert( v2.getItem(i) == (i+1) );
        }
        cout << " 2 " << flush;
        // non-empty, equal
        Vec<double>  v3(5);
        for (unsigned i = 0; i < 5; i++) {
                v3.setItem(i, i+1);
        }
        v3.setSize(5);
        assert( v3.getSize() == 5 );
        for (unsigned i = 0; i < 5; i++) {
                assert( v3.getItem(i) == (i+1) );
        }
        cout << " 3 " << flush;
        // set size to zero
        v3.setSize(0);
        assert( v3.getSize() == 0 );
        assert( v3.myArray == NULL );
        cout << " 4 " << flush;
        cout << "Passed!" << endl;
}

void VecTester::testGetSize() const {
        cout << "Testing getSize()..." << flush;
        Vec<double>  v1;
        assert( v1.getSize() == 0 );
        cout << " 1 " << flush;
        Vec<double>  v2(5);
        assert( v2.getSize() == 5 );
        cout << " 2 " << flush;
```

```
	cout << "Passed!" << endl;
}

void VecTester::testSetItem() const {
	cout << "Testing setItem()... " << flush;
	// empty case
	Vec<double>  v0;
	try {
		v0.setItem(0, 11);
		cerr << "setItem() succeeded on empty Vec";
		exit(1);
	} catch (range_error&) {
		cout << " 0 " << flush;
	}
	// nonempty case, valid subscript
	Vec<double>  v(5);
	for (unsigned i = 0; i < 5; i++) {
		v.setItem(i, i+1);
	}
	for (unsigned i = 0; i < 5; i++) {
		assert( v.myArray[i] == (i+1) );
	}
	cout << " 1 " << flush;
	// nonempty case, invalid subscript
	Vec<double>  v2(3);
	try {
		v2.setItem(3, 33);
		cerr << "setItem() succeeded beyond end of Vec";
		exit(1);
	} catch (range_error&) {
		cout << " 2 " << flush;
	}
	cout << "Passed!" << endl;
}

void VecTester::testGetItem() const {
	cout << "Testing getItem()... " << flush;
	// empty Vec
	Vec<double>  v0;
	try {
		v0.getItem(0);
		cerr << "getItem() succeeded on empty Vec";
		exit(1);
	} catch (range_error&) {
		cout << " 0 " << flush;
	}
	// non-empty, valid access
	Vec<double>  v(5);
	for (unsigned i = 0; i < 5; i++) {
		v.setItem(i, i+1);
```

```
        }
        for (unsigned i = 0; i < 5; i++) {
                assert( v.getItem(i) == (i+1) );
        }
        cout << " 1 " << flush;
        // nonempty Vec, invalid index
        Vec<double>  v2(3);
        try {
                v2.getItem(3);
                cerr << "getItem() succeeded beyond end of Vec";
                exit(1);
        } catch (range_error&) {
                cout << " 2 " << flush;
        }
        cout << "Passed!" << endl;
}


void VecTester::testEquality() const {
        cout << "Testing ==..." << flush;
        // empty case
        Vec<double>  v1;
        Vec<double>  v2;
        assert( v1 == v2 );
        cout << " 1 " << flush;
        // nonempty, same size, default values
        Vec<double>  v3(5);
        Vec<double>  v4(5);
        assert( v3 == v4 );
        cout << " 2 " << flush;
        // nonempty, same size, set values
        Vec<double>  v5(5);
        Vec<double>  v6(5);
        for (unsigned i = 0; i < 5; i++) {
                v5.setItem(i, i+1);
                v6.setItem(i, i+1);
        }
        assert( v5 == v6 );
        cout << " 3 " << flush;
        // empty vs nonempty
        Vec<double>  v7;
        Vec<double>  v8(5);
        assert( !(v7 == v8) );
        cout << " 4 " << flush;
        // nonempty, same size, first value different
        Vec<double>  v9(5);
        Vec<double>  v10(5);
        Vec<double>  v11(5);
        Vec<double>  v12(5);
        v10.setItem(0, 1);
        assert( !(v9 == v10) );
        cout << " 5 " << flush;
        // nonempty, same size, middle value different
        v11.setItem(2, 1);
        assert( !(v9 == v11) );
        cout << " 6 " << flush;
        // nonempty, same size, last value different
        v12.setItem(4, 1);
        assert( !(v9 == v12) );
        cout << " 7 " << flush;

        cout << "Passed!" << endl;
}

void VecTester::testWriteToStream() const {
      cout << "Testing writeTo(ostream)... " << flush;
      Vec<double> v1(5);
      for (unsigned i = 0; i < 5; i++) {
              v1.setItem(i, i+10);
      }
      // write to an ofstream instead of cout, to automate the test
      ofstream fout("vecStreamOut.txt");
      assert( fout.is_open() );
      fout << v1.getSize() << "\n";
      v1.writeTo(fout);
      fout.close();
      // now read in what we just wrote...
      ifstream fin("vecStreamOut.txt");
      assert( fin.is_open() );
      unsigned size;
      fin >> size;
      assert( size == 5 );
      double value;
      for (unsigned i = 0; i < 5; i++) {
              fin >> value;
              assert( value == i+10 );
      }
      cout << "Passed! See 'vecStreamOut.txt' for values..." <<
      endl;
}

void VecTester::testReadFromStream() const {
      cout << "Testing readFrom(istream)... " << flush;
      // an ifstream is-an istream, so use one to automate the test
      ifstream fin("vecStreamOut.txt");
      assert( fin.is_open() );
      // get the size and build the Vec
      unsigned size;
      fin >> size;
      assert( size == 5 );
      Vec<double> v(size);
      // test readFrom()
      v.readFrom(fin);
      for (unsigned i = 0; i < 5; i++) {
              assert( v.getItem(i)== i+10 );
      }
      fin.close();
      cout << "Passed!" << endl;
}


void testConstSubscript(const Vec<double>& v) {
        assert( v[0] == 11 );
        assert( v[1] == 22 );
        assert( v[2] == 33 );
}

void VecTester::testSubscript() const {
        cout << "Testing subscript... " << flush;
        // empty case
        Vec<double> v0;
        try {
                v0[0];
                cerr << "Subscript worked on empty Vec";
                exit(1);
        } catch (const range_error& re) {
                cout << " 0 " << flush;
        }
        // non-empty case, write version
        Vec<double> v1(3);
        v1[0] = 11;
        v1[1] = 22;
        v1[2] = 33;
        assert( v1.getItem(0) == 11 );
        assert( v1.getItem(1) == 22 );
        assert( v1.getItem(2) == 33 );
        cout << " 1 " << flush;
        // non-empty case, read version
        testConstSubscript(v1);
        cout << " 2 " << flush;
        cout << " Passed!" << endl;
}


void VecTester::testInequality() const {
        cout << "Testing !=... " << flush;

        // empty
        Vec<double>  v0;
        Vec<double>  v1;
        assert( !(v0 != v1) );
        cout << " 0 " << flush;

        // empty vs nonempty
        Vec<double>  v2(3);
        assert( v1 != v2 );
        cout << " 1 " << flush;

        // equal sized, same values
        Vec<double>  v3(3);
        Vec<double>  v4(3);
        assert( !(v3 != v4) );
        cout << " 2 " << flush;

        // equal sized, different values
        for (unsigned i = 0; i < 3; i++) {
                v4.myArray[i] = i+1;
        }
        assert( v3 != v4 );
        cout << " 3 " << flush;

        // equal sized, same except first
        for (unsigned i = 0; i < 3; i++) {
                v3.myArray[i] = i+1;
        }
        v3.myArray[0] = 0;
        assert( v3 != v4 );
        cout << " 4 " << flush;

        // equal sized, same except middle
        v3.myArray[0] = 1;
        v3.myArray[1] = 0;
        assert( v3 != v4 );
        cout << " 5 " << flush;

        // equal sized, same except last
        v3.myArray[1] = 2;
        v3.myArray[2] = 0;
        assert( v3 != v4 );
        cout << " 6 " << flush;

        // equal sized, equal
        v3.myArray[2] = 3;
        assert( !(v3 != v4) );
        cout << " 7 " << flush;

        cout << "Passed!" << endl;
}

void VecTester::testAddition() const {
	cout << "Testing +... " << flush;
	// nonempty
	Vec<double>  v1(3);
	Vec<double>  v2(3);
	v1.setItem(0, 1);
	v1.setItem(1, 2);
	v1.setItem(2, 3);
	v2.setItem(0, 2);
	v2.setItem(1, 4);
	v2.setItem(2, 6);
	Vec<double>  v3 = v1 + v2;
	assert( v3.getItem(0) == 3 );
	assert( v3.getItem(1) == 6 );
	assert( v3.getItem(2) == 9 );
	cout << " 1 " << flush;
	// empty
	Vec<double>  v4, v5;
	v3 = v4 + v5;
	assert( v3.getSize() == 0 );
	assert( v3.myArray == NULL );
	cout << " 2 " << flush;
	// different sizes
	try {
		v3 = v2 + v4;
		cerr << "v2 + v4 succeeded for Vecs of different 
sizes";
		exit(1);
	} catch (invalid_argument&) {
		cout << " 3 " << flush;
	}
	cout << "Passed!" << endl;
}

void VecTester::testSubtraction() const {
	cout << "Testing -... " << flush;
	// nonempty
	Vec<double>  v1(3);
	Vec<double>  v2(3);
	v1.setItem(0, 1);
	v1.setItem(1, 2);
	v1.setItem(2, 3);
	v2.setItem(0, 2);
	v2.setItem(1, 4);
	v2.setItem(2, 6);
	Vec<double>  v3 = v1 - v2;
	assert( v3.getItem(0) == -1 );
	assert( v3.getItem(1) == -2 );
	assert( v3.getItem(2) == -3 );
	cout << " 1 " << flush;
	// empty
```

```
        Vec<double>  v4, v5;
        v3 = v4 - v5;
        assert( v3.getSize() == 0 );
        assert( v3.myArray == NULL );
        cout << " 2 " << flush;
        // different sizes
        try {
                v3 = v2 - v4;
                cerr << "v2 - v4 succeeded for Vecs of different
sizes";
                exit(1);
        } catch (invalid_argument&) {
                cout << " 3 " << flush;
        }
        cout << "Passed!" << endl;
}

void VecTester::testDotProduct() const {
        cout << "Testing *... " << flush;
        Vec<double>  v1(3);
        Vec<double>  v2(3);
        v1.setItem(0, 1);
        v1.setItem(1, 2);
        v1.setItem(2, 3);
        v2.setItem(0, 2);
        v2.setItem(1, 4);
        v2.setItem(2, 6);
        double product = v1 * v2;
        assert( product == 28 );
        cout << " 1 " << flush;
        // empty
        Vec<double>  v4, v5;
        product = v4 * v5;
        assert( product == 0 );
        cout << " 2 " << flush;
        // different sizes
        try {
                product = v2 * v4;
                cerr << "v2 * v4 succeeded for Vecs of different
sizes";
                exit(1);
        } catch (invalid_argument&) {
                cout << " 3 " << flush;
        }
        cout << "Passed!" << endl;
}

void VecTester::testReadFromFile() const {
      cout << "Testing readFrom()... " << flush;
      Vec<double> v1;
      v1.readFrom("vecTest1.txt");
      assert( v1.getSize() == 3 );
      assert( v1.myArray != NULL );
      assert( v1.getItem(0) == 1 );
      assert( v1.getItem(1) == 2 );
      assert( v1.getItem(2) == 3 );
      cout << " 1 " << flush;

      Vec<double> v2(5);
      double* oldAddr = v2.myArray;
      v2.readFrom("vecTest2.txt");
      assert( v2.getSize() == 4 );
      cout << " 2 " << flush;
      assert( v2.myArray != NULL );
      assert( v2.getItem(0) == 2 );
      assert( v2.getItem(1) == 4 );
      assert( v2.getItem(2) == 6 );
      assert( v2.getItem(3) == 8 );
       /* the following assertion assumes that the new array's base
        *  address is different from the old array's base address,
        *  which may not be true. If your method seems to be right,
        *  is correctly using delete to deallocate the old array,
        *  and passes all assertions except this one, you may
        *  comment out this assertion.
        */
      assert( v2.myArray != oldAddr );
      cout << " 3 " << flush;
      cout << "Passed! Make sure your method closed the file..." <<
endl;
}

void VecTester::testWriteToFile() const {
      cout << "Testing writeTo()... " << flush;
      // read a vector we know into an empty Vec...
      Vec<double> v1;
      v1.readFrom("vecTest1.txt");
      // make a copy of it in a different file
      v1.writeTo("vecTest1Copy.txt");
      cout << " 1 " << flush;
      cout << "vecTest1Copy.txt created " << flush;
      // read in the copy, and check it
      Vec<double> v3;
      v3.readFrom("vecTest1Copy.txt");
      assert( v3.getSize() == 3 );
      assert( v3.myArray != NULL );
      assert( v3.getItem(0) == 1 );
      assert( v3.getItem(1) == 2 );
      assert( v3.getItem(2) == 3 );
      cout << " 2 " << flush;
      // read a vector we know into a nonempty Vec...
      Vec<double> v2(5);
      v2.readFrom("vecTest2.txt");
      // make a copy of it in a different file
      v2.writeTo("vecTest2Copy.txt");
      cout << " 3 " << flush;
      cout << "vecTest2Copy.txt created " << flush;
      // read in the copy and check it
      v3.readFrom("vecTest2Copy.txt");
      assert( v3.getSize() == 4 );
      assert( v3.myArray != NULL );
      assert( v3.getItem(0) == 2 );
      assert( v3.getItem(1) == 4 );
      assert( v3.getItem(2) == 6 );
      assert( v3.getItem(3) == 8 );
      cout << " 4 " << flush;
      cout << "Passed!  Make sure you closed the file..." << endl;
}
```

```
5
10
11
12
13
14
```

```
/* MatrixTester.h declares test-methods for class Matrix.
 * Joel C. Adams, for CS 112 at Calvin College.
 */

#ifndef MATRIXTESTER_H_
#define MATRIXTESTER_H_

#include "Matrix.h"

class MatrixTester {
public:
        void runTests();
        void testDefaultConstructor();
        void testExplicitConstructor();
        void testCopyConstructor();
        void testAssignment();
        void testEquality();
        void testInequality();
        void testSubscripts();
        void testReadSubscript(const Matrix<int>& mat);
        void testTranspose();
        void testAddition();
        void testSubtraction();
        void testMultiply();
        void testReadFromStream();
        void testReadFromFile();
        void testWriteToStream();
        void testWriteToFile();
};

#endif /*MATRIXTESTER_H_*/
/* VecTester.h provides unit tests for Vec, a simple vector class.
 * Student Name:Nana Osei Asiedu Yirenkyi
 * Date: Oct 2 2018
 * Begun by: Joel C. Adams, for CS 112 at Calvin College.
 */

#ifndef VECTESTER_H_
#define VECTESTER_H_

#include "Vec.h"

class VecTester {
public:
        void runTests() const;
        void testDefaultConstructor() const;
        void testExplicitConstructor() const;
        void testDestructor() const;
        void testGetSize() const;
        void testSetItem() const;
        void testGetItem() const;
        void testSetSize() const;
        void testCopyConstructor() const;
        void testAssignment() const;
        void testEquality() const;
        void testWriteToStream() const;
        void testReadFromStream() const;
//      void testConstSubscript(const Vec& v);
        void testSubscript() const;
        void testInequality() const;
        void testAddition() const;
        void testSubtraction() const;
        void testDotProduct() const;
        void testReadFromFile() const;
        void testWriteToFile() const;
};

#endif /*VECTESTER_H_*/
3 4
1 2 3 4
5 6 7 8
9 10 11 12

/* Matrix.h provides a class for manipulating 2-dimensional vectors.
 * Student Name: Nana Osei Asiedu Yirenkyi
 * Date: Oct 6 2018
 * PROJECT04
 * Begun by: Joel Adams, for CS 112 at Calvin College.
 */

#ifndef MATRIX_H_
#define MATRIX_H_

#include "Vec.h"
#include "MatrixTester.h"

template<class Item>

class Matrix {
public:
	Matrix();
	Matrix(unsigned rows, unsigned columns);
	unsigned getRows() const;
	unsigned getColumns() const;
	const Vec<Item>& operator[](unsigned index) const;
	Vec<Item>& operator[](unsigned index);
	bool operator==(const Matrix<Item>& m2) const;
	bool operator!=(const Matrix<Item>& m2) const;
	void readFrom(istream& in);
	void writeTo(ostream& out) const;
	void readFrom(const string& fileName);
	void writeTo(const string& fileName);
	Matrix<Item> operator + (const Matrix<Item>& rhs) const;
	Matrix<Item> operator - (const Matrix<Item>& rhs) const;
	Matrix<Item> getTranspose() const;


private:
	unsigned myRows;
	unsigned myColumns;
	Vec< Vec<Item> > myVec;
	friend class MatrixTester;
	friend class Application;
};


//Default constructor
template<class Item>
Matrix<Item>::Matrix() {
	myRows = myColumns = 0;
}

//Explicit Constructor
template<class Item>
Matrix<Item>::Matrix(unsigned rows, unsigned columns) {
        myRows = rows;
        myColumns = columns;
        myVec.setSize(rows);
        for (unsigned i = 0; i < rows; i++) {
                myVec[i].setSize(columns);
        }
}


//Returns the number of Rows in the matrix
template<class Item>
unsigned Matrix<Item>::getRows() const {
        return myRows;
}


//Returns the number of Rows in the matrix
template<class Item>
unsigned Matrix<Item>::getColumns() const {
        return myColumns;
}


/* Subscript to retrieve value method for constant
 *              reference values
 * @index: an unsigned int that must not be out
 *                      of range for myVec
 * Return: myVec[index]
 */
template<class Item>
const Vec<Item>& Matrix<Item>::operator[](unsigned i) const {
        if (i >= myRows) {
                throw range_error("Bad Subscript");
        }
        return myVec[i];
}


/* Subscript to retrieve value method
 * @index: an unsigned int that must not be out
 *                      of range for myArray
 * Return: myVec[index]
 */
template<class Item>
Vec<Item>& Matrix<Item>::operator[](unsigned i) {
        if (i >= myRows) {
                throw range_error("Bad Subscript");
        }
        return myVec[i];
```

```
}


/*Inequality operator != checks if myArray and v2.myArray
 *               are NOT equal
 * @param: m2, a Vec class object
 * Return: true/false
 */template<class Item>
bool Matrix<Item>::operator==(const Matrix<Item>& m2) const {
      if ( myRows != m2.getRows() || myColumns != m2.getColumns() ) {
           return false;
      } else {
           return myVec == m2.myVec;
      }
   }


 //--------------------------------------------
PROJECT04----------------------

/*Inequality operator != checks if myArray and v2.myArray
 *               are NOT equal
 * @param: m2, a Vec class object
 * Return: true/false
 */
template<class Item>
bool Matrix<Item>::operator != (const Matrix<Item>& m2) const {
           if ( myRows != m2.getRows() || myColumns !=
m2.getColumns() ) {
                return true;
           } else {
                return !(myVec == m2.myVec);
           }
        }


/* Matrix readFrom reads values from stream directly
 *               into myArray; the first values read in the
 *               stream will be mySize
 */
template<class Item>
void Matrix<Item>::readFrom(istream& in) {
        for (unsigned i = 0; i < myRows; i++) {
                for (unsigned j = 0; j < myColumns; j++)
                        in >> myVec[i][j];
                }
}


/* Matrix writeTo writes values to of myArray into
```

```
 *              out stream; the first values read in the
 *              stream will be mySize
 */
template<class Item>
void Matrix<Item>::writeTo(ostream& out) const {
        for (unsigned i = 0; i < myRows; i++) {
                for (unsigned j = 0; j < myColumns; j++) {
                        out << myVec[i][j] << '\t';
                }
                out << endl;
        }
}


/* Matrix readFrom reads values from a filename directly
 *              into myArray; the first values read in the
 *              file will be mySize
 * @filename: a string object, will be used as
 *              the file read from
 */
template<class Item>
void Matrix<Item>::readFrom(const string& fileName) {
        ifstream in(fileName.c_str());
        in >> myRows;
        in >> myColumns;
        myVec.setSize(myRows);
        for (unsigned i =0; i < myRows; i++) {
                myVec[i].setSize(myColumns);
        }
        readFrom(in);
        in.close();
}

/* Matrix writeTo writes values to of myArray into
 *              a file filename; the first values read in the
 *              file will be mySize
 * @filename: a string object, will be used as
 *              the file written to
 */
template<class Item>
void Matrix<Item>::writeTo(const string& fileName) {
        ofstream fout(fileName.c_str());
        fout << myRows << ' '<< myColumns << endl;
        writeTo(fout);
        fout.close();
}


/*Matrix addition operator adds two matrices and sets a third
 *              matrix as equal to them
 *@param: m2, a Matrix class object
 *Return: m3, a Matrix class object equal to myArray + v2.myArray
 */
template<class Item>
Matrix<Item> Matrix<Item>::operator + (const Matrix<Item>& rhs) const
{
        if(myRows != rhs.myRows || myColumns != rhs.myColumns)
                throw invalid_argument("Can only add matrices of the
same size.");
        Matrix<Item> newM(myRows, myColumns);
        for (unsigned i = 0; i < myRows; i++) {
                for (unsigned j = 0; j < myColumns; j++)
                        newM.myVec[i][j] = myVec[i][j] + rhs.myVec[i]
[j];
        }
        return newM;
}

/*Matrix subtraction operator subtracts two matrices and sets a third
 *               matrix as equal to them
 *@param: m2, a Matrix class object
 *Return: m3, a Matrix class object equal to the values in
 *               myArray minus the values in v2.myArray
 */
template<class Item>
Matrix<Item> Matrix<Item>::operator - (const Matrix<Item>& rhs) const
{
        if(myRows != rhs.myRows || myColumns != rhs.myColumns)
                throw invalid_argument("Can only subtract matrices of
the same size.");
        Matrix<Item> newM(myRows, myColumns);
        for (unsigned i = 0; i < myRows; i++) {
                for (unsigned j = 0; j < myColumns; j++)newM.myVec[i]
[j] = myVec[i][j] - rhs.myVec[i][j];
        }
        return newM;
}

/*Matrix transpose switches the row and column elements
 *@param: m1,m2 Matrix class object
 *Return: m3, a Matrix class object equal to the values in
 *               myArray and v2.myArray with i and j switched around
 */
template<class Item>
Matrix<Item> Matrix<Item>::getTranspose() const {
        Matrix<Item> newM(myColumns, myRows);
        if (myRows == 0) {
                return newM;
        }
        else {
                for (unsigned i = 0; i < myRows; i ++) {
                        for (unsigned j = 0; j < myColumns; j ++) {
                                newM[j][i] = myVec[i][j];
                        }
                }
                return newM;
        }
}
#endif
bash-3.2$ cd Debug
bash-3.2$ ls
MatrixTester.d   VecTester.d      makefile project04       subdir.mk
        tester.o
MatrixTester.o   VecTester.o      objects.mk       sources.mk
tester.d
bash-3.2$ make all
make: Nothing to be done for `all'.
bash-3.2$ cd ..
bash-3.2$ .Debug/.Debug[1P.Debug[C/Debug[C[C[C[C[Cm[K/project04
Running Matrix tests...
Testing default constructor... 0  1 Passed!
Testing explicit constructor... 1  2 Passed!
Testing copy constructor...  1  2  Passed!
Testing =...  0  1  2  3  4  5 Passed!
Testing ==...  1  2  3  4 Passed!
Testing subscripts...  1  2  3  4  5  6 Passed!
Testing !=...  0  1  2  3 Passed!
Testing getTranspose()...  0  1a  1b Passed!
Testing +...  0  1a  1b  2  3 Passed!
Testing -...  0  1a  1b  2  3 Passed!
Testing readFrom(istream)... Passed!
Testing writeTo(ostream)... Passed!
Testing readFrom(string)... Passed!
Testing writeTo(string)... Passed!
All tests passed!

Welcome to the matrix application.
Please choose an operation
 1. + Addition
 2. - Subtraction
 3. Transpose
 4. exit
 Option:1
Enter the file name for 1st Matrix:MatrixTets  st1.txt
1       2       3       4
5       6       7       8
9       10      11      12
Enter the filename for the 2nd Matrix: MatrixTest2.txt
1       2       3       4
5       6       7       8
```

```
9       10      11      12
The result is:
2       4       6       8
10      12      14      16
18      20      22      24
Please choose an operation
 1. + Addition
 2. - Subtraction
 3. Transpose
 4. exit
 Option:2
Enter the file name for 1st Matrix:MatrixTest2.txt
1       2       3       4
5       6       7       8
9       10      11      12
Enter the filename for the 2nd Matrix: MatrixTest1.txt
1       2       3       4
5       6       7       8
9       10      11      12
The result is:
0       0       0       0
0       0       0       0
0       0       0       0
Please choose an operation
 1. + Addition
 2. - Subtraction
 3. Transpose
 4. exit
 Option:3
Enter the file name for 1st Matrix:Matrix  Test2.txt
1       2       3       4
5       6       7       8
9       10      11      12
Transposition is:
1       5       9
2       6       10
3       7       11
4       8       12
Please choose an operation
 1. + Addition
 2. - Subtraction
 3. Transpose
 4. exit
 Option:4
Ending...bash-3.2$ exit

Script done on Thu Oct 11 21:55:43 2018
```