MPHYGB24 C/C++ Coursework 2 - Part 1 - Written Report

Inspection of the skeleton class declaration and the test suite source code revealed that there were 24 class functions to be prototyped and implemented. This report therefore describes the purpose of each of these 24 functions, the appropriateness of each of the prototypes and the correctness of each of the implementations.

Tasks 1 & 2

On successful completion of Task 1 (i.e. once all the function prototypes in `Matrix.h` were correctly declared), both `Matrix.h` and `MatrixTest.cpp` could be compiled without error. This is because the compilation step merely checks that the functions declared in `Matrix.h` match the functions used in `MatrixTest.cpp` (the compilation step *does not* require the implementation of functions). However, the object files produced by this compilation step would not link at this stage, as the linking step *does* require the implementation of functions.

1. `int GetIndex (const int& rowIdx, const int& columnIdx) const;`
   This protected member function has its two constant integer arguments passed in as references (aliases to variables which share the same memory addresses). The use of keyword `const` at the end of the declaration dictates that this function cannot alter member variables, and is thus a read-only function. It has been correctly implemented assuming that the matrix values have been stored column-wise in the member variable `data`. This function cannot be invoked in static or non-member functions (even if the latter has 'friend' status).
2. `Matrix ();`
   The default constructor has been appropriately defined by declaring integer values of zero to all three member variables. Implementation of a default constructor is essential once non-default constructors have been implemented and used, and is also essential if any of the member variables have been declared using dynamic memory allocation.
3. `Matrix (const int& noOfRows_rhs, const int& noOfCols_rhs);`
   This standard custom constructor assigns a value of zero to all members of a matrix, which has dimensions which match the dimensions in the function argument. It was an important lesson here to use variable names in function arguments that did not match member variable names, so as to avoid confusion and 'double declaration' errors. It was also important to learn that *all* member variables should be initialised in a constructor.
4. `Matrix (const Matrix& input);`
   The default copy constructor produces a 'deep' copy of its input (rather than a 'shallow' copy), ensuring that both integer member variables, and all elements of the array member variable `data` are copied faithfully.
5. `static Matrix Zeros(const int& noOfRows_rhs, const int& noOfCols_rhs);`
6. `static Matrix Ones (const int& noOfRows_rhs, const int& noOfCols_rhs);`
   These two static functions are by default shared across all instances of the class Matrix, and are not associated with each object instance of the class Matrix. As such they cannot have `const` keyword modifiers at the end of their declarations as they have no member variables associated with them to alter. These two static functions behave as constructors, but have more meaningful names that elude to what they construct (i.e. a matrix of zeros or ones, respectively, of appropriate dimensions).
7. `~Matrix ();`
   There can only be one default constructor, and its definition is essential if any of the member variables have been dynamically allocated (such as the member variable `data` in this case). The use of the square brackets after the keyword `delete` denotes that the variable to be deleted is an array.
8. `friend ostream& operator<< (ostream& out, const Matrix& rhs);`
   This non-member function has been granted access to the protected member variables of the class Matrix by its 'friend' status. Non-member functions cannot have `const` keyword modifiers, as they have no member variables associated with them to alter. In this case the '<<' operator has

been overloaded to be of return type `ostream`, and to accept `out` (of type `ostream`) and an instance of class Matrix as its arguments. It is essential to include `<iostream>` as a preprocessor directive *and* to declare `using namespace std;` in `Matrix.h` for this function to work correctly. It was also important to ensure the correct spacing between columns containing positive or negative numbers.

9. `Matrix& operator= (const Matrix& rhs);`
   The default assignment operator was implemented such that it first checked if the dimensions of the matrices on both sides of the assignment matched (it would not make sense to assign a matrix to another matrix if their dimensions did not match). An error message is displayed in the event that there is a mismatch in dimensions, or if the user has attempted to copy an object into itself. If no error message is displayed, then the assignment operation proceeds. The return type of this function is of type Matrix, and the function returns a reference to `this` (i.e. a reference to the instance of the class associated with the function).
10. `static Matrix Print (const double* const data_rhs, const int& noOfRows_rhs, const int& noOfCols_rhs);`
    The use of the first function argument here means that `data_rhs` is being passed in by reference using a pointer, and that both the address in the pointer and the variable being pointed to are constant and cannot be changed. This is the most restrictive and rigorous method of passing a pointer into a function. Again this static function has no object attached to it, and so an instance of class matrix called `Print` is generated in the implementation using the standard custom constructor. The input arguments are copied across to this new instance of class Matrix, which is printed in an identical fashion to function 8 above.
11. `static Matrix Toeplitz (const double* const column, const int& noOfRows_rhs, const double* const row, const int& noOfCols_rhs);`
    This static member function again uses the standard custom constructor to produce a blank instance of class Matrix of appropriate dimensions. It uses the code developed in Worksheet 5 to then populate `data` with the correct entries, dependent upon the first column and the first row of the Toeplitz matrix given in the function argument. A warning message is shown if the first element of the input row and the input column are not equal.
12. `static Matrix Transpose (const Matrix& input);`
    In order to maximise code reuse and minimise code duplication, this function merely generates an instance of class Matrix which is a copy of the input, and then invokes function 13 (see below) using this object, before returning the object.
13. `void Transpose ();`
    This function is of return type void, as it does not actually return a variable, but alters the variable by which it is invoked. It makes use of the declaration of temporary variables to hold swapped values in. These swapped values are then assigned to the appropriate member variable. The transpose operation involves swapping the number of columns and rows of the matrix, and then storing the entries of the matrix row-wise (rather than column-wise) in `data`. The temporary data holder array is then dynamically deleted.
14. `friend Matrix operator* (const Matrix& lhs, const Matrix& rhs);`
    This non-member function has `friend` status, and thus has access to the protected member variables of class Matrix. This function overloads the '*' operator, and creates a new instance of the input argument `lhs` using the default copy constructor, and then uses the member function 15 to multiply itself by the input argument `rhs`, before returning the result.
15. `Matrix& operator*= (const Matrix& rhs);`
    This member function was the most complex to implement. It overloads the '*=' operator to modify the instance of class Matrix associated with the function by multiplying itself by the function argument. An error message is displayed if the inner dimensions of the `rhs` matrix and the `lhs` matrix are not equal. The result of a matrix multiplication will have the number of rows of the `lhs` matrix, and the number of columns of the `rhs` matrix. Therefore the number of columns of the `lhs` matrix is decreased or increased as required (the extra vales are padded with zeros in the latter case). Each element of the result matrix is then calculated by the dot product of the respective row of the `lhs` matrix with the respective column of the `rhs` matrix. This was done using three

nested for loops. A temporary data variable was used to hold these calculated values in until all values had been calculated. The calculated values were then reassigned to `data`, before it was returned as one of the member variables of the object associated with the function.

16. `Matrix& ExchangeRows (const int& row1, const int& row2);`
17. `Matrix& ExchangeRows (const int& row1, const int& row2, const int& col1, const int& col2);`
    These two functions both have return types of Matrix references, and represent overloaded variants of each other. As they are non-static member functions, the `GetIndex` function is used to determine the index of the elements to exchange. Again a temporary data holder is used to facilitate the exchange, and complete rows are exchanged by looping across all columns for function 16, or partial rows are exchanged by looping between certain columns specified in the argument of function 17.
18. `Matrix& ExchangeColumns (const int& col1, const int& col2);`
19. `Matrix& ExchangeColumns (const int& col1, const int& col2, const int& row1, const int& row2);`
    These two functions were declared and implemented in a very similar fashion to functions 16 and 17, with appropriate adjustments made. It was important to note here that any functions which are invoked in the following way: `cout << object.function() <<;` are required to have a return type (i.e. they cannot be void). Examples of where this is used in this test suite include type Matrix (functions 16, 17, 18 and 19), type double (function 24), and type integer (functions 23 and 24).
20. `void Zeros ();`
21. `void Ones ();`
    These two functions are both very similar: they have no input arguments and no return type. They both simply alter the values of `data` belonging to the object by which they are invoked, and change all of the values to zeros or ones respectively.
22. `int GetNoOfRows () const;`
23. `int GetNoOfColumns () const;`
    These two functions are also very similar: they are both read-only functions, they have no input arguments, and return a variable of type integer. They were both very straightforward to implement by returning the relevant member variable of the object by which they were invoked.
24. `double GetEntry (const int& rowIdx, const int& columnIdx) const;`
    This is a read-only function of return type double. It uses the `GetIndex` function to return the entry at position `(i,j)` of the matrix belonging to the object by which it was invoked.

Task 3

As explained above for the default assignment operator (function 9), it makes sense that two matrices should have the same dimensions if assigning one to the other. There are four different cases which could occur when using the assignment operator in this context:

1. both the number of rows and the number of columns are not equal;
2. the number of rows are equal, but the number of columns are not equal;
3. the number of columns are equal, but the number of rows are not equal;
4. both the number of columns and the number of rows are equal.

Therefore each of these four situations is tested in the test suite. In situations 1, 2 and 3 the appropriate error message is shown, and it is clear that the assignment operation has not taken place as the matrix of ones (shown before and after attempted assignment) has not been altered by the attempted assignment. In situation 4 no error message is shown, and it is clear that assignment has taken place as the matrix of ones is visibly reassigned to become a matrix of zeros. Additionally, the default assignment operator also checks that the user has not attempted to copy an object into itself, and displays an error message accordingly if this occurs (as is tested in case 5).