ECE2036 **Fall Semester, 2019**

Lab 5 –Complex Matrix Objects

Assigned: October 25, 2019 *Due Before: Nov 10, 2019 @ 11:59PM*

In this lab I would like for you to create a set of libraries that your client can use to manipulate matrices of complex numbers. The concepts that I would like for you to explore in this lab are ones we have been developing in class, which are:

1. Class Composition: You will make a `Matrix` class, and have as data members information about number of rows, number of columns, and a 1-D array of complex numbers that are the values in your matrix. You will also construct your `Complex` class similar to what we did in class. You will need to make sure that your `Complex` class has a default constructor to use it correctly in this lab.
2. `new` and `delete` Operators: The user will dynamically specify the size of the matrix; therefore, you will need to dynamically allocate the array that holds the data. You will need to manage this data accordingly, which includes preventing memory leaks.
3. The Rule of Three: Because you are dealing with dynamic memory allocation you will need to overload the assignment operator, the destructor, and the copy constructor.
4. Operator Overloading: In addition to overloading the *, +, and – operators for matrices, I would like for you to overload the insertion stream operator for printing matrices and the parenthesis operator (i.e. ()) for accessing matrix elements. You can read about this in Chapter 10.5 in the textbook. You will also overload the ~ operator to produce the transpose of a matrix, and ! unary operator to produce the conjugate transpose (see wikipedia for definition).
5. Use of the 'this' Pointer: If you are accessing private data directly, I would like for you to make explicit use of the 'this' pointer in your member functions to help understand its function.
6. The use of inclusion guards in header files. Make sure that you have separate file structures for the class interface and the member function implementation as discussed in Chapter 9 of Deitel and Deitel.
7. Basic use of Makefiles: I will provide a makefile with this lab to help you maintain this file system. Once you have all your files, all you have to do is type `make` at the command prompt.

For this lab, you will need to develop a basic `Complex` class to work with your `Matrix` Class. Therefore, you will have to manage files `matrix.cc`, `matrix.h`, `complex.cc`, `complex.h`, and `main.cc`. I will provide a `Makefile` for you to use in this lab, but can also use the following command to link these files together on the pace-ice system.

```
g++ -std=c++0x main.cc matrix.cc complex.cc –o test_matrix
```

For this lab, I am going to give you the client test code, which will be in the `main.cc` file. I will also give you *part* of the class interface for Matrix class as well. You must create your objects to be compatible with the client code and produce the desired output. You are not allowed to change the client code.

Complex Class

I would like for you to call your class Complex. It should adhere to the following guidelines:

1. All data members need to be private. Also, I would like for you to have variables of type double that store the real and imaginary associated with the complex number object. I would also like for you to have a bool variable NaN (i.e. not a number) that is true if the client tries to store the result of a divide by zero.

2. Please have "set" and "get" functions for each data member in your class. Please use the naming convention that we have used in class (i.e. setVaribleName or getVariableName).

3. I would like for you to have three different constructors.

- **a.** One that has no arguments and sets all data members to zero or false.
- **b.** One that has two double arguments that are the real and the imaginary component in rectangular format.
- **c.** One that has one double argument that is just the real component. It is assumed that if the client uses this constructor that the number is purely real and the imaginary component is zero.

4. Finally, in this lab I would like for you to start with a key concept in C++, which is "Overloading Operators."

Please notice in the client code in main.cc code that the '+' operator, for example, is sometimes used between complex objects and sometimes between matrix objects. Normally, you cannot do this automatically because the complex class, for example, is not a fundamental data type in C++; however, we can write a special member function that explicitly tells the compiler that we would like to use the '+' operator (or other operators) with our complex numbers. We might have an expression that looks like:

```
result = lhs + rhs;
```

The member function that is called with the above syntax is given below. Notice the keyword operator in the function signature.

```
Complex Complex::operator+(Complex RHS)
{
    Complex sum;
    //I am showing this for variety, but
    //I would like you to use set and get functions instead
    sum.real =  getReal()+RHS.real;
    sum.imag =  getImag()+RHS.imag;

    return(sum);
}
```

In addition, I would like for you to overload the '-', '*', and '/' operator for your `Complex` class as well. Make sure that you handle special cases (e.g. divide by zero, etc..). I will assume that your ECE2026 knowledge will help you write these operator member functions accordingly.

In the class interface, you need to specify the operator function in the following way:

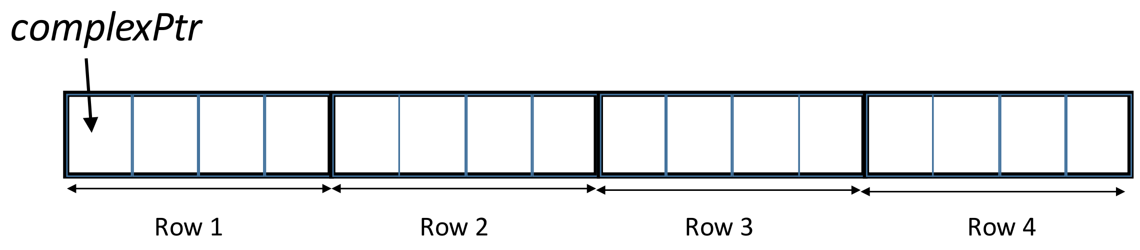
```
Complex operator+(Complex);
```

Please extend out example for the '+' operator to the other operators that you must create.

Matrix Class

Please see the header file on the last page (before Appendix A) of this for some specifications for the Matrix class. You must adhere to the following guidelines:

1. I would like for you have at least one constructor that takes as arguments the number of rows and number of columns in the matrix. You will need to use the `new` operator to allocate memory accordingly for the matrix. Even though the matrix is a 2-D array, I would like for you to store the contents in a one-dimensional C++ dynamically allocated array. You will need to "unroll" the matrix into a linear array by storing the first row, then the second row, then the third row, etc.. For a 4x4 matrix might look something like the following...



2. If there is an error in matrix operations, for example the matrices may not agree in a multiplication, addition, or subtraction, then you can return a matrix with both `rows` and `cols` be equal to zero. The `complexPtr` for this "not-a-matrix" should just point to null.

3. You will need to overload the ( ) operator so that you can access different elements in the matrix. The function prototype in the class interface would look like:

```
Complex & operator()(int,int);
```

You will also need to figure out how to overcome the zero indexing issue. For example, I would refer to each element in a 2x2 matrix, called A, in the following way.

```
A(1,1) = Complex(3,0);
A(1,2) = Complex(3,7);
A(2,1) = Complex(4,0);
A(2,2) = Complex(7,10);
```

4. You will need to overload the *, +, and – operator for matrix operations. You will also need to overload the unary operator ~ to indicate the regular transpose of the matrix and ! to indicate the conjugate transpose. Just like in the book, I would like for you to overload the << operator. You will notice in the output code that if the number is real then this operator just prints out the real number only. In addition, if the number is pure imaginary this operator just prints out the imaginary number only. Finally, I would also like for you to overload the * operator with a complex number on *either* size of a matrix object.

5. I would like for you to also have `transpose` and `printMatrix` non-operator member functions. These are a little redundant with the operators, but I would like for you to implement them nonetheless. Note, however, that this transpose member function will actually change object when called. Applying the ~ operator will not change the operand, but instead it will return a copy of an object that contains the transposed matrix.

6. Make sure you have no memory leaks. Be careful in the assignment operator! Also make sure your destructor deallocates memory accordingly. To verify this you can run the utility valgrind.

To get this to work you will need to compile your code with the option flag –g

```
g++ -g -std=c++0x main.cc matrix.cc complex.cc –o ./test_matrix
```

Then you can run valgrind using the following statement

```
valgrind --leak-check=yes ./test_matrix
```

You can see the results of possible memory leaks in the `valgrind` "Heap Summary" report as discussed in class. Your TAs will look at this as well to evaluate this part of the rubric.

You can also use `valgrind` for debugging as well. If you add the -v option it will look at other possible memory errors (like not initializing your variables, etc). That is:

```
valgrind --leak-check=yes ./test_matrix
```

Turning in AND setting up Lab5

1. Make a text file with your output and call it `output.txt`. The easiest way to do this is with a redirect (>) in Linux.

   ```
   ./test_matrix > output.txt
   ```

   If you would like to easily see the difference between the `outputExample.txt` file that I have given to you and the file you create, you can use the follow command in Linux.

   ```
   diff output.txt outputExample.txt
   ```

   Please note that small differences due to roundoff error is okay.

2. I will provide you a tarfile with a `Makefile`, sample output, and the main.cc called lab5Provided.tar.gz. You can download the file from canvas. To extract a Lab5 directory, type the following in your HOME directory.

   ```
   tar -xvzf lab5Provided.tar.gz
   ```

3. After you have filled your Lab5 with YOUR code, please type the following command while in your HOME directory to create a compressed `tarball` of your work

   ```
   tar   -cvzf     yourusernameLab5.tar.gz   ./Lab5
   ```

4. As I showed you in class, you will need to download this file to your local machine. You will then submit this `yourusernameLab5.tar.gz` to canvas so that the TAs can grade it.

Code in main.cc (Also provided to you on canvas in main.cc)

```
#include <iostream>
#include <iomanip>
#include "matrix.h"
#include "complex.h"
using namespace std;

int main()
{
//first let's declare an array of complex numbers
Complex a[5];
Complex result[5];
Complex test1(4.5,6.5);

//I will fill these with data
a[0].setComplex(3,4);
a[1].setComplex(-1,3);
a[2].setComplex(-1.23,-9.83);
a[3].setComplex(3.14,-98.3);
a[4].setComplex(2.71,1.61);

//Now print the data in polar format
cout << "\nPrint Array of Complex Numbers in Polar Format" << endl;
for (int i = 0; i < 5; i++)
{
  a[i].displayPolar();
}

//Now test the add function
cout << "\nTesting add operator a[0] + a[1]" << endl;
result[0] = a[0] + a[1];
result[0].displayRect();

//Now test the sub function
cout << "\nTesting subtract operator a[1] - a[2]" << endl;
result[1] = a[1] - a[2];
cout << result[1] << endl;

//Now test the multiply function
cout << "\nTesting multiply operator a[2] * a[3]" << endl;
result[2] = a[2] * a[3];
cout << result[2] << endl;
result[2].displayRect();

//Now test the divide function
cout << "\nTesting divide operator a[3] / a[4]" << endl;
result[3] = a[3] / a[4];
result[3].displayRect();

//Now test the divide by zero
cout << "\nTesting divide by zero a[4] / (0)" << endl;
result[4] = a[4] / Complex(0,0);
cout << result[4] << endl;
```

```
//Now display the results array in polar format
cout << "\nNow display the results array in polar format" << endl;
for (int i = 0; i < 5 ; i++)
{
      result[i].displayPolar();
}

//Now start to test the matrix

Matrix A(3,3);
Matrix C(4,4);
Matrix D(2,3);
Matrix E(2,3);
Complex num(1,1);
int counter =0;

for (int i = 1; i<=3; i++)
  for (int j = 1; j<=3; j++)
      { A(i,j) = Complex(counter++,0); }

for (int i = 1; i<=2; i++)
  for (int j = 1; j<=3; j++)
  {
       D(i,j) = Complex(counter,counter);
       counter++;
   }

Matrix B(A);

cout << "A Matrix" << endl;
A.printMatrix();
cout << endl;

cout << "B transpose" << endl;
B.transpose();
cout << B << endl;
cout << endl;

cout << "The C matrix " << endl;

C(1,1) = num;
C(2,2) = Complex(4,2);
C(3,3) = Complex(1,1);
C(4,4) = Complex(0,1);

cout << C << endl;

cout << "D Matrix" << endl;
D.printMatrix();
cout << endl;

A = B*B;
```

```
cout << "The A = B*B matrix is " << endl;
cout << A << endl;
cout << endl;

cout << "The transpose of A is then" << endl;
(~A).printMatrix();
cout << endl;

cout << "The matrix A is still the following" << endl;
cout << A << endl;

B = B+B;
cout << "The B = B+B matrix is " << endl;
cout << B << endl;

B = Complex(6,7)*B;
cout << "The B = (6+7j)*B gives B as " << endl;
cout << B << endl;

B = B*Complex(7,7);
cout << "The B = B*(7+7j) gives B as " << endl;
cout << B << endl;

B = &B;
cout << "The conjugate transpose of B is " << endl;
cout << B << endl;

A = C;

cout << "The A = A-A matrix is " << endl;
A = A - A;
cout << A << endl;

cout << "Multiply with rectanglur matrix D: E=D*B" << endl;
E = D*B;
cout << E << endl;

cout << "Try multiplying mismatched matrices" << endl;
C = A*B;
cout << C << endl;

cout << "Try adding mismatched matrices" << endl;
A = A+B;
cout << A << endl;

cout << "Try subtracting mismatched matrices" << endl;
A = A-B;
cout << A << endl;

return 0;

}
```

Partial Class Interface in matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include "complex.h"
#include <string>
#define MATRIX_FIELD 30

//This is a class prototype to let the compiler know
//that I intend to define a class Matrix. It is needed
//for the global function definition that I put before
//the class Matrix as an example in this lab.

class Matrix;
std::ostream& operator<< (std::ostream &, const Matrix &);
Matrix operator*(Complex, Matrix &);

class Matrix
{
friend std::ostream& operator<< (std::ostream &, const Matrix &);
friend Matrix operator*(Complex, Matrix &);

//you put stuff in here!

};

#endif
```

APPENDIX A: ECE 2036 Lab Grading Rubric

If a student's program runs correctly and produces the desired output, the student has the potential to get a 100 on his or her lab; however, TA's will **randomly** look through this set of "perfect-output" programs to look for other elements of meeting the lab requirements. The table below shows typical deductions that could occur.

In addition, if a student's code does not compile, then he or she will have an automatic 30% deduction on the lab. Code that compiles, but does not match the sample output can incur a deduction from 10% to 30% depending on how poorly the output matches the output specified by the lab. This is in addition to the other deductions listed below or due to the student not attempting the entire assignment.

AUTOMATIC GRADING POINT DEDUCTIONS

Element	**Percentage Deduction**	**Details**
Does Not Compile	30%	Program does not compile on pace-ice cluster!
Does Not Match Output	10%-30%	The program compiles but doesn't match all output exactly

ADDITIONAL GRADING POINT DEDUCTIONS FOR RANDOMLY SELECTED PROGRAMS

Element	**Percentage Deduction**	**Details**
Correct file structure	10%	Does not use both .cc and .h files, implementing class prototype correctly
Explicit this pointer use	5%	If accessing private in a member function, does the student explicitly use the 'this' pointer
Const Correctness	5%	Correctly uses const to apply the principle of least priviledge.
Encapsulation	10%	Does not use correct encapsulation in object-oriented objects
Setters/Getters	10%	Does not use setters and getters for each data member.
No Memory Leaks	10%	Passes with a clean valgrind report
Constructors	10%	Does not implement constructors with the correct functionality.
Member or Global Function Specifications	10%	Does not implement each member function or global function specified in the lab assignment
Clear Self-Documenting Coding Styles	5%-15%	This can include incorrect indentation, using unclear variable names, unclear comments, or compiling with warnings. (See next page)

LATE POLICY

Element	Percentage Deduction	Details
Late Deduction Function	score - (20/24)*H	H = number of hours (ceiling function) passed deadline note : Sat/Sun count has one day; therefore H = 0.5*Hweekend

Appendix B: Good Programming Practices

Indentation

When using *if/for/while* statements, make sure you indent 2 to 4 spaces for the content inside those. For example…

```
for(int i; i < 10; i++)
  j = j + i;
```

If you have nested statements, you should use multiple indentions. Your *if/for/while* statement brackets *{ }* can follow two possible conventions. Each both getting their own line (like the *for* loop) OR the open bracket on the same line as the statement (like for the *if/else* statement) and closing bracket its own line. If you have *else* or *else if* statements after your *if* statement, they should be on their own line.

```
for(int i; i < 10; i++)
{
  if(i < 5) {
    counter++;
    k -= i;
  }
  else {
    k += i;
  }
  j += i;
}
```

Camel Case (Suggested But Not Required)

This is simply the naming convention of each new word in a variable should be capitalized: firstSecondThird. This applies for functions and member functions as well! The main exception to this is class names, where the first letter should also be capitalized.

Variable and Function Names

Your variable and function names should be clear about what that variable or function is. Do not use one letter variables, but use abbreviations when it is appropriate (for example: "imag" instead of "imaginary"). The more descriptive your variable and function names are, the more readable your code will be. This is an idea of self-documenting code.

Clear Comments

Some good opportunities to use comments are…

- Introducing a member function or class
- Introducing a section of code in a long function implementation
- Your name, class information, etc. at the beginning of the file

Some bad times to use comments are…

- To clarify what a variable represents (`double x // imaginary component`)
- To explain confusing code (you should probably rewrite it instead!)

Appendix C: GDB Debugger Tutorial

Once you start to manipulate pointers and the heap, the errors that you get can be a little more difficult to find. The key idea with debugging is to ISOLATE YOUR ERROR!

A popular debugger in Linux is gdb. This helps for you to identify areas where your code failed, if for example you have a core dump. You can see how variables are changing as you step through a program.

Here are a few things for you to try. There are a lot on online tutorials that you can find as well.

1. You must compile with a special option to create a "Debugging Symbol Table." Please use the following command that is just like for `valgrind`.

   ```
   g++ -g -std=c++0x main.cc matrix.cc complex.cc –o ./test_matrix
   ```

2. Now you need to run the debugger in the following way.

   ```
   gdb ./test_matrix
   ```

3. This does not automatically run the program, but you could run the entire program use the r command at the prompt, i.e.

   ```
   r
   ```

4. If you want the program to stop at a certain point in the program, you can set a "breakpoint." For example, you can set a breakpoint at main using the following command at the prompt.

   ```
   b main
   ```

   then you can run the program

   ```
   r
   ```

   It will stop at the beginning of the program. If you want to go to the next instruction you can use `next` (executes entire functions) or `step` (much finer granularity)

   ```
   next
   ```

   or

   ```
   step
   ```

   You can also use the following command to go to the next breakpoint or finish program

   ```
   continue
   ```

5. You can set break points in functions or by line number in the program. If there are multiple files then you can use the following command

```
b filename:linenumber
b filename:functionName
```

6. You can also view variables during runtime inside the debugger using the following

   ```
   print variableName
   ```

   or if you want it to display a certain variable as you step through the program you can use

   ```
   display variableName
   ```

   You can use `undisplay` to turn this off for each variable.

7. You can determine where you are in a function by using where or `bt`

   ```
   bt
   ```

   or

   ```
   where
   ```

8. Here is a list of commands that you can play around with as well.

   b main - Puts a breakpoint at the beginning of the program

   b - Puts a breakpoint at the current line

   b N - Puts a breakpoint at line N

   b +N - Puts a breakpoint N lines down from the current line

   b fn - Puts a breakpoint at the beginning of function "fn"

   d N - Deletes breakpoint number N

   info break - list breakpoints

   r - Runs the program until a breakpoint or error

   c - Continues running the program until the next breakpoint or error

   f - Runs until the current function is finished

   s - Runs the next line of the program

   s N - Runs the next N lines of the program

   n - Like s, but it does not step into functions

   u N - Runs until you get N lines in front of the current line

   p var - Prints the current value of the variable "var"

   bt - Prints a stack trace

   u - Goes up a level in the stack

   d - Goes down a level in the stack

   q - Quits gdb