ECE 2036: Lab 5: Dynamic Matrix Object

Due: April 5, 2019 at 11:59 PM

Directions: In this lab you will be creating a set of objects that a client can use to manipulate matrices of complex numbers. The following concepts will be explored:

1. Class composition: You will be creating a **matrix** class that will contain a 2-dimensional array of complex numbers. You will also construct a **complex** class.
2. **new** and **delete** operators: The client will dynamically specify the size of the matrix; therefore, you will need to dynamically allocate the 2-dimensional 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 inside of your **matrix** class, you will need to overload the assignment operator, the destructor, and the copy constructor.
4. Operator Overloading: You will be overloading several operators for your **matrix** class and your **complex** class. The implementation for the arithmetic operators will following their definition from mathematics. You will also need to overload the insertion stream operator for printing matrices and complex numbers and the parenthesis operator for indexing your matrix. The following links may be helpful:
    a. https://www.learncpp.com/cpp-tutorial/93-overloading-the-io-operators/
    b. https://www.learncpp.com/cpp-tutorial/99-overloading-the-parenthesis-operator/

5. Output Formatting: You will be provided with the sample program that will be used to test the classes you create in this lab. You will also be provided with the correct output. The final output your program produces must match this **exactly** in order to receive full credit on the assignment.

Details: You have been provided a skeleton implementation for the complex number and matrix classes. You will need to add data members and methods to these classes for them to function with the test program provided in main.cc of the skeleton code. You should **not** change any of the code in main.cc. When we grade your assignment, we will be copying in a fresh copy of the test program in main.cc to ensure this.

You **must** implement the values of the matrix as a 2-dimensional array of complex objects. The data member **m_vals** in the provided skeleton should be used to refer to it. The following link will clarify how to dynamically allocate and deallocate the required space:

https://stackoverflow.com/questions/936687/how-do-i-declare-a-2d-array-in-c-using-new

The first two answers are the most useful for this assignment. The third answer provides a good argument on why this may not always be the best way to structure this type of object. Regardless, for this assignment it is good enough and it is a requirement.

The output format that is required for this program is like the default style used by MATLAB. All the following cases can be found in the output your program produces. You will need to program this behavior in the overloaded insertion stream operator of your classes.

For complex numbers

- Print both the real component and the imaginary component with 4 decimal places separated by the sign of the imaginary component.

```
6.0000 + 3.0000j
```

For matrices

- Print them all in the imaginary format.

```
6.0000 + 3.0000j, 8.0000 + 4.0000j
6.0000 + 0.0000j, 8.0000 + 0.0000j
```

Points will be deducted from your score if your implementation leaks memory while executing the provided test program. If you are not careful, it is extremely easy to leak memory on this assignment. Therefore, it is highly recommended that you check this frequently. How you do that depends on which compiler you are using.

For submission, compress your solution into lab5.zip (NOT .TAR.GZ, or .RAR or anything else). zip is installed on the pace login nodes if you are using pace. Submit via canvas.

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 on **`pace-ice.pace.gatech.edu`**, 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 on PACE-ICE System	30%	Program does not compile on pace ice cluster!
Does Not Match Output	10%-30%	The program compiles but doesn't match output.txt.

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
Does not implement classes	10%	Does not implement classes for complex and matrix objects
Encapsulation	10%	Does not use correct encapsulation in object-oriented objects
Constructors	10%	Does not implement constructors with the correct functionality.
Clear Self-Documenting Coding Styles	5%-15%	This can include incorrect indentation, using unclear variable names, unclear comments, or compiling with warnings. (See Appendix C)

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: Coding Standards

Indentation:

When using *if/for/while* statements, make sure you indent 4 spaces for the content inside those. Also make sure that you use spaces to make the code more readable.
For example:

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

If you have nested statements, you should use multiple indentions. Each { should be on its own line (like the *for* loop) 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 +=1;
   }
   j += i;
}
```

Camel Case:

This naming convention has the first letter of the variable be lower case, and the first letter in each new word be capitalized (e.g. 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 the idea behind self-documenting code.

File Headers:

Every file should have the following header at the top

```
/*
Author: <your name>
Date Last Modified: <date last modified>
Organization: ECE2036 Class

Description:

Describe what is done in this file.

*/
```

Code Comments:

1. Every function must have a comment section describing the purpose of the function, the input and output parameters, the return value (if any).
2. Every class must have a comment section to describe the purpose of the class.
3. Comments need to be placed inside of functions/loops to assist in the understanding of the flow of the code.