EC327 - Spring 2025 - Homework 4

Background

This homework will reinforce key **object-oriented programming (OOP)** concepts, including **encapsulation, inheritance, and polymorphism**. All of the problems on this assignment will require solutions to be implemented in C++ using C++17.

Important

Please make sure your code compiles and runs as intended on the engineering grid. Code that does not compile will NOT be graded and will receive a 0.

Note on Collaboration

You are welcome to talk to your classmates about the assignment and discuss high level ideas. However, all code that you submit must be your own. We will run code similarity tools against your submissions and will reach out with questions if anything is flagged as suspicious. The closed book exams for this class will mostly cover material very similar to what you do on the homework assignments. The best way to prepare for the exams is to do the assignments on your own and internalize the learnings from that process. Cheating on the assignments will very likely result in you not doing well on the exams.

Problem 1 - Complex Number Abstract Class (add, multiply, conjugate, print) Coding Best Practices (10 points)

Submission Instructions

Your solution to this problem should contribute a single **h** file to your overall **hw4** zip. The **h** file should follow the following format:
ComplexNumber.h

Actual Problem

Given the header file `ComplexNumber.h`, add appropriate comments for all function declarations and class declarations. Your comments should clearly describe the purpose of each function, its parameters, and its return value. Use concise and professional documentation style.

Here is an example of what a comment for a function declaration might look like:

```
#define MAX_BUFFER_SIZE 1024

struct ReverseResult {
    bool success;
    char** reversedStrings;
};

/**
 * Reverses all of the input strings as long as they have an odd number of
 * characters.  If they are all odd, the resulting struct should have
 * ...
 */
struct ReverseResult reverseOdd(const char** inputStrings, int inputStringsLength);
```

Copy `ComplexNumber.h` which is given below and add comments.

```
#ifndef COMPLEXNUMBER_H
#define COMPLEXNUMBER_H

#include <iostream>

class ComplexNumber {
   protected:
       int real;
       int imag;
```

```
   public:
       ComplexNumber(int r, int i) : real(r), imag(i) {}

       int getReal() const { return real; }
       int getImag() const { return imag; }

       virtual ComplexNumber* add(const ComplexNumber& other) = 0;
       virtual ComplexNumber* multiply(const ComplexNumber& other) = 0;
       virtual ComplexNumber* conjugate() = 0;

       void print() {
           std::cout << real << (imag >= 0 ? " + " : " - ") << std::abs(imag) << "i"
<< std::endl;
       }
};
#endif
```

Problem 2 - Implement the GaussianInteger Class

Submission Instructions

Your solution to this problem should contribute a single **h** file and a single **cpp** file to your overall **hw4** zip. The **h** file should follow the following format:
GaussianInteger.h
The **cpp** file should follow the following format:
GaussianInteger.cpp

Problem 2a - GaussianInteger.h (5 points)

Note: GaussianInteger should inherit the ComplexNumber abstract class.
Write a header file defining these function prototypes:

- A no parameter constructor
- A two parameter constructor receiving the two const fields (real and imag)
- A copy constructor that receives a const reference to another GaussianInteger instance and sets all data fields to be the same as the “other” instance
- The three overridden implementations for the pure virtual methods defined in the abstract class (use the keyword ‘override’ to override the pure virtual method)
    - add
    - multiply
    - conjugate
- Two new methods divides() and norm()

- Divides should receive a const reference to another GaussianInteger instance and returns a boolean that represents whether or not the current GaussianInteger divides by the other instance
- Norm should return the norm of the current stored GaussianInteger
  - If the object is a + bi, then the norm is $a^2 + b^2$

Problem 2b - Constructors (5 points)

Provide the definitions in **GaussianInteger.cpp** for the constructors. Make sure to use initialization lists to initialize class fields and be sure to use the abstract class constructor (GaussianInteger should not define additional data fields).

```
/**
* Constructs an GaussianInteger 0
*/
GaussianInteger();

/**
* Constructs a Gaussian integer representing real + imag i
* @example GaussianInteger ex(1, 3) produces the Gaussian Integer 1 + 3i.
*/
GaussianInteger(const int real, const int imag);

/**
* Copy constructor - constructs a duplicate of [other]
* @param other The GaussianInteger to duplicate
* @example
* GaussianInteger ex(1, 3);
* GaussianInteger ex2(ex) results in ex2 being 1 + 3i as well.
*/
GaussianInteger(const GaussianInteger &other);
```

Problem 2c - Implementations for Pure Virtual Methods (add, multiply, conjugate) (10 points)

Write implementations for the following overridden methods in **GaussianInteger.cpp**:

- Add method should add the current stored gaussian integer with the other gaussian integer received from the parameters and return a new gaussian integer
- Multiply method should multiply the current stored gaussian integer with the other gaussian integer received from the parameters and return a new gaussian integer
- Conjugate method should return a new gaussian integer representing the conjugate of the current stored gaussian integer

Problem 2d - Division (divides, norm) (15 points)

Write two methods in **GaussianInteger.cpp**, divides() and norm().

$$\frac{a+bi}{c+di} = \frac{a+bi}{c+di} * \frac{c-di}{c-di}$$

$$= \frac{(a+bi)(c-di)}{c^2+d^2}$$

$$= \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}\,i$$

Note: The result is only a gaussian integer if ac + bd and bc - ad both evenly divide c^2 + d^2.

```
/**
* @param other The GaussianInteger by which to divide this object
* @return true if and only if this object divides by [other]
* (or, in other words, other is a factor of this object)
* @example 1
* GaussianInteger four(4, 0);
* GaussianInteger two(2, 0);
* four.divides(two) is true, because 2*2=4
*
* @example 2
* GaussianInteger two(2, 0); // i.e., 2 + 0i
* GaussianInteger onePlusI(1, 1); // i.e., 1 + i
* two.divides(onePlusI) is true because (1 + i) * (1 - i) == 2
*
* @example 3
* GaussianInteger two(2, 0); // i.e., 2 + 0i
* GaussianInteger twoPlusI(2, 1); // i.e., 2 + i
* two.divides(twoPlusI) is false because 2 does not divide 2 + i
*/
bool divides(const GaussianInteger &other) const;

/**
```

```
* @return the norm of this object. If this object is a + bi, then the norm is a^2 +
b^2.
* @example
* GaussianInteger onePlusI(1, 1);
* onePlusI.norm() == 2, corresponding to the norm of 1 + i
*/
int norm() const;
```

Problem 2e - Operators (add, multiply, equality, stdout) (10 points)

Provide some operators to make operations on gaussian integers easier.

```
/**
* @param other The GaussianInteger to add to this object
* @return this object added to [other].
* Does not modify this object.
*
* @example
* GaussianInteger one(1, 0);
* GaussianInteger i(0, 1);
* (one + i) produces the Gaussian Integer 1 + i
*/
GaussianInteger operator+(const GaussianInteger &other) const;

/**
* @param other The GaussianInteger to multiply to this object
* @return this object multiplied by [other]
* Does not modify this object.
*
* @example
* GaussianInteger one(1, 0);
* GaussianInteger i(0, 1);
* (one * i) produces the Gaussian Integer i (1 * i)
*/
GaussianInteger operator*(const GaussianInteger &other) const;

/**
*
* @param other The Gaussian integer to which this should be compared
* @return true if and only if this object is equal to [other]
```

```
* Does not modify this object.
* @example
* GaussianInteger one(1, 0);
* GaussianInteger i(0, 1);
* one == one is true, but one == i is false
*/
bool operator==(const GaussianInteger &other) const;

/**
* @param os A reference to the ostream object
* @param other The GaussianInteger being printed
* @return The reference to the ostream object to allow for cout chaining
* friend function
*
* @example
* GaussianInteger two(2, 0);
* std::cout << two << std::endl;
* produces 2 + 0i
*/
std::ostream& operator<<(std::ostream& os, const GaussianInteger& other);
```

Problem 3 - Extending the GaussianInteger Class (Quaternion class) (45 points)

Submission Instructions

Your solution to this problem should contribute a single **cpp** file to your overall **hw4** zip. The **cpp** file should follow the following format:
Quaternion.cpp

Actual Problem

Quaternions are an extension of the Gaussian integer with four components a+bi+cj+dk where a,b,c,d are real numbers. They are an extension of complex numbers where components i, j, and k are interpreted as unit vectors pointing along the three spatial coordinates.

For this problem you will implement `Quaternion.cpp` based off the description in `Quaternion.h` and will do the following:

- Inherit from GaussianInteger, but add two more data fields

- Implement no parameter constructor, parameterized constructor, and copy constructor (15 points)
- Override addition for Quaternions (same as Gaussian Integer, but with 4 fields) (10 points)
- Override conjugate for Quaternions (5 points)
- Override multiply to implement quaternion multiplication rule (10 points)
- Override print to display four components (5 points)

Note: Use initializer lists for constructors

```
#ifndef QUATERNION_H
#define QUATERNION_H

#include "GaussianInteger.h"

class Quaternion : public GaussianInteger {
private:
   int j;
   int k;

public:

   /**
   * Default constructor
   *
   * @return a quaternion with zeros as fields
   */
   Quaternion();

   /**
   * Parameterized constructor
   *
   * @param real real part of Quaternion
   * @param imag imaginary component of Quaternion i
   * @param j imaginary component of Quaternion j
   * @param k imaginary component of Quaternion k
   * @return newly created Quaternion with the given values.
   */
   Quaternion(const int real, const int imag, const int j, const int k);

   /**
   * Copy constructor
   *
   * @param other reference of quaternion to copy from
```

```
 * @return new Quaternion that is a copy of other
 */
Quaternion(const Quaternion& other);

/**
 * Parameterized constructor
 *
 * @param real real part of Quaternion
 * @param imag imaginary component of Quaternion i
 * @param j imaginary component of Quaternion j
 * @param k imaginary component of Quaternion k
 * @return newly created Quaternion with the given values.
 */

/**
 * Adds the current quaternion with another quaternion.
 *
 * This method performs component-wise addition of two quaternions, adding
 * their real and imaginary parts separately.
 *
 * @param other The other quaternion to add to the current quaternion.
 * @return A new Quaternion that represents the sum of the two quaternions.
 *         The result is a quaternion with the summed real and imaginary parts.
 *         If the other object is not a Quaternion, an error message is printed,
 *         and nullptr is returned.
 */
ComplexNumber* add(const ComplexNumber& other) override;

/**
 * Multiplies the current quaternion with another quaternion.
 *
 * This method performs quaternion multiplication based on the following formula:
 *
 * q1 * q2 = (w1 * w2 - x1 * x2 - y1 * y2 - z1 * z2) +
 *           (w1 * x2 + x1 * w2 + y1 * z2 - z1 * y2)i +
 *           (w1 * y2 - x1 * z2 + y1 * w2 + z1 * x2)j +
 *           (w1 * z2 + x1 * y2 - y1 * x2 + z1 * w2)k
 *
 * @param other The other quaternion to multiply with the current quaternion.
 * @return A new Quaternion that represents the product of the two quaternions.
 *         If the other object is not a Quaternion, an error message is printed,
 *         and nullptr is returned.
```

```
    */
    ComplexNumber* multiply(const ComplexNumber& other) override;

    /**
    * Returns the conjugate of the current quaternion.
    *
    * The conjugate of a quaternion q = w + xi + yj + zk is defined as:
    *
    * q* = w - xi - yj - zk
    *
    * The real part remains unchanged and the signs of the imaginary components are
flipped.
    *
    * @return A new Quaternion that represents the conjugate of the current quaternion.
    */
    ComplexNumber* conjugate() override;

    /**
    * Prints the current quaternion in the format:
    *
    * w + xi + yj + zk
    *
    * Example output for a quaternion q = 3 + 2i - 4j + 0k:
    * "3 + 2i - 4j + 0k"
    */
    friend std::ostream& operator<<(std::ostream& os, const Quaternion& other);

    // Getter methods for j and k
    int getJ() const { return j; }
    int getK() const { return k; }
};

#endif // QUATERNION_H
```