AMS 562 Homework #3
C++ Scientific Programming Project

Joseph Schuchart

Due: October 15th, 2025

1 Objective

Develop a suite of C++ classes that model fundamental scientific entities such as `Point`, `Line`, `Rectangle`, and `Square`. These classes can be utilized in various scientific computing applications, including simulations, data analysis, and computational modeling.

Learning Outcomes:

- Implement core Object-Oriented Programming (OOP) principles including encapsulation, abstraction, inheritance, and polymorphism in C++.
- Design and organize code using best practices in C++ for scientific programming.
- Apply mathematical and geometric concepts to solve scientific problems programmatically.
- Develop and execute unit tests to ensure code reliability and correctness.
- Utilize Doxygen-style comments for comprehensive code documentation.

2 Requirements

2.1 Point Class

- Define a class `Point` with private member variables for x and y coordinates.
- Implement constructors for default initialization and parameterized initialization.
- Provide methods to:
  - Calculate the distance from the origin.
  - Calculate the distance between two points.
  - Translate the point by given x and y offsets.
  - Perform vector addition and scaling operations.
- Implement a method to represent the point as a string (e.g., `toString()`).

2.2 Line Class

- Define a class `Line` that uses two `Point` objects to represent its endpoints.
- Implement a constructor that takes two `Point` objects.
- Provide a method to calculate the length of the line.
- Include methods to calculate the slope and y-intercept of the line.
- The constructor should throw an error if both input points are the same.

2.3 Rectangle Class

- Define a class `Rectangle` that uses four `Point` objects to represent its vertices. The rectangle is assumed to be axis-aligned, meaning its edges are parallel to the x and y axes.
- Implement a constructor that takes two `Point` objects (top-left and bottom-right) to define the rectangle.
- Provide methods to calculate the area and perimeter of the rectangle.
- Implement a method to check if a given point lies inside the rectangle (excluding the edges).

2.4 Square Class

- Define a class `Square` that inherits from `Rectangle`.
- Implement a constructor that takes a `Point` object (top-left) and the side length as inputs. The constructor should throw an error if the side length is not positive, ensuring that the input represents a valid square.
- Override the `area()` method from `Rectangle` to return the side length squared.

2.5 OOP Concepts

- **Encapsulation:** Use private member variables and public member functions to protect the internal state of objects.
- **Abstraction:** Hide the implementation details of scientific calculations, exposing only necessary interfaces.
- **Inheritance:** Create a derived class `Square` that inherits from `Rectangle` to promote code reuse and hierarchical relationships.
- **Polymorphism:** Introduce virtual methods (e.g., `virtual double area() const;` in the base class) that behave differently in derived classes. Demonstrate runtime polymorphism by using base class pointers or references to interact with derived class objects.
- **Operator Overloading (Optional):** Overload operators such as `<<` for easy printing of objects or `+` for point/vector addition.
- **Exclusions:** Templates are not required for this assignment and should not be used.

2.6 Unit Testing

- **Scope:** Implement unit tests for each class and method to validate their correctness and reliability in scientific computing contexts. This includes constructors, member functions, and any utility methods.
- **Frameworks:** You may write your own test functions using assertions or utilize a testing framework such as Google Test for more structured and comprehensive testing.
- **Test Cases:**
  - **Normal Scenarios:** Typical use cases, such as creating objects with valid parameters and performing standard operations relevant to scientific applications.
  - **Edge Cases:** Handle unusual or extreme cases, such as:
    - Creating a `Line` with two identical `Point` objects.
    - Defining a `Rectangle` with zero area (where top-left and bottom-right points are the same or aligned).
    - Initializing a `Square` with a negative side length.

  - **Exception Handling:** Verify that appropriate exceptions are thrown and handled correctly when invalid inputs are provided.
  - **Performance (Optional):** Assess the efficiency of methods, especially those that might be called frequently in scientific computations.

- **Documentation:** Include descriptive messages in your tests to indicate which functionality is being tested and the nature of the test (e.g., "Test distance calculation between two points").
- **Automation:** Ensure that all tests can be run easily, preferably through the provided `Makefile`, to facilitate automated testing.

3 Project Structure

Organize your project with the following files and directory structure for clarity and maintainability:

```
project_root/
|-- include/
|   |-- Point.h
|   |-- Line.h
|   |-- Rectangle.h
|   `-- Square.h
|-- src/
|   |-- Point.cpp
|   |-- Line.cpp
|   |-- Rectangle.cpp
|   `-- Square.cpp
|-- tests/
|   `-- tests.cpp
|-- Makefile
`-- README.md
```

- `include/` - Header files for all classes, documented with Doxygen-style comments.
- `src/` - Implementation (`.cpp`) files for all classes.
- `tests/` - Contains `tests.cpp` with all unit tests.
- `Makefile` - For compilation using `g++`.
- `README.md` - Documentation about the project, compilation instructions, and any other relevant information.

3.1 Skeleton Code of Some Example Files

3.1.1 Point.h

```
/**
 * @file Point.h
 * @brief Declaration of the Point class for scientific programming.
 */

#ifndef POINT_H
#define POINT_H

#include <string>

/**
```

```
 * @class Point
 * @brief Represents a point in 2D space.
 */
class Point {
private:
    double x; /**< X-coordinate */
    double y; /**< Y-coordinate */

public:
    /**
     * @brief Default constructor initializes point at origin.
     */
    Point();

    /**
     * @brief Parameterized constructor initializes point with given coordinates.
     * @param x_val X-coordinate.
     * @param y_val Y-coordinate.
     */
    Point(double x_val, double y_val);

    // Accessors
    double getX() const;
    double getY() const;

    // Mutators
    void setX(double x_val);
    void setY(double y_val);

    /**
     * @brief Calculates the distance from the origin.
     * @return Distance from the origin.
     */
    double distanceFromOrigin() const;

    /**
     * @brief Calculates the distance to another point.
     * @param other The other Point object.
     * @return Distance between this point and the other point.
     */
    double distanceTo(const Point& other) const;

    /**
     * @brief Returns a string representation of the point.
     * @return String in the format "(x, y)".
     */
    std::string toString() const;
};

#endif // POINT_H
```

3.1.2 Point.cpp

```
/**
 * @file Point.cpp
 * @brief Implementation of the Point class.
 */

```

```cpp
#include "Point.h"
#include <cmath>
#include <sstream>

Point::Point() : x(0.0), y(0.0) {}

Point::Point(double x_val, double y_val) : x(x_val), y(y_val) {}

double Point::getX() const { return x; }

double Point::getY() const { return y; }

void Point::setX(double x_val) { x = x_val; }

void Point::setY(double y_val) { y = y_val; }

// ... other member function implementations
```

3.1.3 tests.cpp

```cpp
/**
 * @file tests.cpp
 * @brief Unit tests for the scientific programming project.
 */

#include "Point.h"
#include "Line.h"
#include "Rectangle.h"
#include "Square.h"
#include <iostream>
#include <cassert>

/**
 * @brief Tests the distanceFromOrigin method of the Point class.
 */
void test_distance_from_origin() {
    Point p(3, 4);
    assert(p.distanceFromOrigin() == 5.0);
    std::cout << "test_distance_from_origin PASSED" << std::endl;
}

/**
 * @brief Tests the distanceTo method of the Point class.
 */
void test_distance_to() {
    Point p1(0, 0);
    Point p2(3, 4);
    assert(p1.distanceTo(p2) == 5.0);
    std::cout << "test_distance_to PASSED" << std::endl;
}

// ... additional test functions for Line, Rectangle, Square

int main() {
    test_distance_from_origin();
    test_distance_to();
    // ... call other test functions
    return 0;
```

```
}
```

3.1.4 Sample Makefile

```
# Compiler and flags
CXX = g++
CXXFLAGS = -Wall -Wextra -std=c++11 -Iinclude

# Source and object files
SRCS = src/Point.cpp src/Line.cpp src/Rectangle.cpp src/Square.cpp tests/tests.cpp
OBJS = $(SRCS:.cpp=.o)

# Target executable
TARGET = hw3_tests

all: $(TARGET)

$(TARGET): $(OBJS)
        $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJS)

# Rule to compile each source file
%.o: %.cpp
        $(CXX) $(CXXFLAGS) -c $< -o $@

# Clean build artifacts
clean:
        rm -f $(OBJS) $(TARGET)
```

4 Policy on Generative AI Tools

Students are permitted to use generative AI tools such as ChatGPT or GitHub Copilot for the following purposes:

- Debugging code.
- Grammar checking and improving code comments.
- Understanding programming concepts or C++ syntax.
- Generation of Doxygen-style comments or the README file.

However, the use of these tools for generating significant portions of the code, including but not limited to class implementations, member functions, or test cases, is prohibited for this project. The emphasis is on individual understanding and application of C++ programming principles.

Academic Integrity: Any form of plagiarism or unauthorized collaboration will be subject to disciplinary action as per the university's academic policies. Ensure that all submitted work is your own.

For further clarification, please consult the university's academic integrity guidelines or reach out to the instructor.

5 Submission Guidelines

5.1 GitHub Repository

1. **Create a Private Repository:**
    - Sign in to your GitHub account and create a private repository named `AMS562_Homework3`.

2. **Push Your Code:**
   - Include all C++ source and header files, organized as per the project structure guidelines.
   - Add a `README.md` file that includes:
     - **Project Overview:** A brief description of the project and its objectives.
     - **File Structure:** Explanation of the directory and file organization.
     - **Compilation Instructions:** Step-by-step guide on how to compile the project using the provided Makefile.
     - **Usage Instructions:** How to run the tests and any example commands.
     - **Doxygen Documentation:** Instructions or links on how to generate and view the Doxygen documentation.
3. **Share the Repository:**
   - Add the TA (`yuxuanye1`) as a collaborator to your private repository:
     (a) Navigate to your repository on GitHub.
     (b) Go to **Settings >Manage Access >Invite a collaborator**.
     (c) Enter the TA's GitHub username: `yuxuanye1`.
   - Ensure that your last commit is made before the project deadline.

5.2 Code Documentation

- **Doxygen-Style Comments:**
  - Document all classes, member functions, and important code segments using Doxygen-style comments.
  - Ensure that the documentation is comprehensive and follows the examples provided in the skeleton code.
- **Code Quality:**
  - Follow consistent coding standards and naming conventions.
  - Ensure proper indentation and code formatting for readability.
- **Comments:**
  - Provide meaningful comments that explain the purpose and functionality of classes, methods, and complex code blocks.
  - Avoid redundant comments that do not add value.
- **Documentation Tools:**
  - Generate HTML or PDF documentation using Doxygen and include instructions in the README on how to access it.

5.3 Report Submission

1. **Report Document:**
   - Create a 1–2 page report using Microsoft Word, Google Docs, or LaTeX.
   - Format the report with:
     - Font: Times New Roman
     - Font Size: 12
     - Line Spacing: Single

- Margins: 1-inch margins on all sides

2. **Report Content:**
   - **Repository URL:**
     - Include the URL of your GitHub repository shared with the TA.
   - **Classes and OOP Concepts:**
     - Provide a brief description of each class implemented (`Point`, `Line`, `Rectangle`, `Square`).
     - Explain the OOP principles demonstrated, such as encapsulation, inheritance, and polymorphism.
   - **Testing and Verification:**
     - Discuss how you tested your code, including the types of tests performed.
     - Summarize the results of your tests and any issues encountered.
   - **Lessons Learned and Challenges:**
     - Reflect on what you learned during the project.
     - Describe any challenges faced and how you overcame them.
3. **Submission:**
   - Convert your report to PDF format.
   - Upload the PDF file to Brightspace under the designated assignment submission link before the deadline.

5.4 Deadlines

- **Project Deadline:** October 18th, 2024, by 11:59 PM EDT.

6 Grading Rubric

Your assignment will be evaluated based on the following criteria:

- **Code Structure (30%)**
  - **Organization (10%):** Files and directories are logically organized as per the project structure guidelines.
  - **Naming Conventions (10%):** Consistent and descriptive naming for classes, methods, variables, and files.
  - **C++ Best Practices (10%):** Proper use of language features, memory management, and adherence to modern C++ standards.
- **Implementation (30%)**
  - **Functionality (15%):** All required classes and methods are implemented correctly and perform as specified.
  - **Error Handling (5%):** Proper handling of invalid inputs and exceptional cases using exceptions or other mechanisms.
  - **Efficiency (5%):** Solutions are optimized for performance without unnecessary complexity.
  - **Code Quality (5%):** Clean, readable code with minimal redundancy and appropriate use of OOP principles.
- **Unit Tests (20%)**
  - **Coverage (10%):** Tests cover all classes and major functionalities, including edge cases.

  - **Effectiveness (5%):** Tests correctly identify valid and invalid scenarios, ensuring reliability.
  - **Implementation (5%):** Tests are well-structured, use assertions appropriately, and are easy to run.

- **Documentation (10%)**
  - **Doxygen Comments (5%):** Comprehensive and correctly formatted Doxygen-style comments for all classes and methods.
  - **Report Quality (5%):** The report is well-written, concise, and covers all required sections comprehensively.

- **Submission (10%)**
  - **GitHub Repository (5%):** Properly set up repository with all required files, a comprehensive README, and correct sharing permissions.
  - **Report Submission (5%):** The report is submitted on Brightspace in the correct format by the deadline.

6.1 Partial Credit and Deductions

- **Incomplete Features:** Points will be deducted for missing classes, methods, or functionalities as specified.
- **Incorrect Implementations:** Errors in logic or calculations will result in partial credit based on the severity and impact on overall functionality.
- **Poor Code Quality:** Lack of readability, inconsistent formatting, or deviation from best practices will negatively affect the Code Structure and Implementation scores.
- **Insufficient Testing:** Inadequate unit tests or failure to cover edge cases will impact the Unit Tests score.
- **Incomplete Documentation:** Missing or improperly formatted comments will affect the Documentation score.
- **Late Submissions:** Assignments submitted after the deadline may incur a penalty as per the course's late submission policy.