Module Code & Title: CMP2801M Advanced Programming

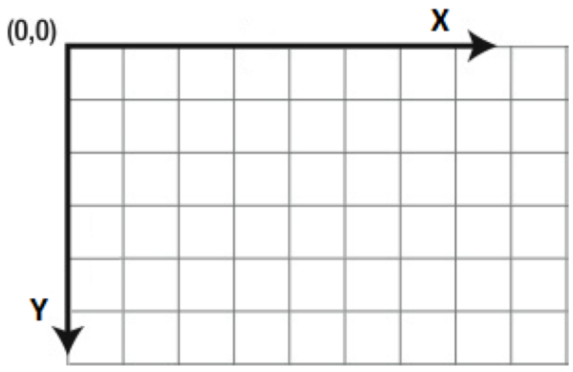
Contribution to Final Module Mark: 70%

Description of Assessment Task and Purpose:

Your task is to implement a console application in C++ for creating and handling basic geometric shapes. The purpose of this assessment is to demonstrate your understanding of programming concepts such as low-level memory management, input/output, logical/mathematical concepts, and multi-paradigm development. **You may refer to your workshop solutions in the development of your solution.**

Driver class:

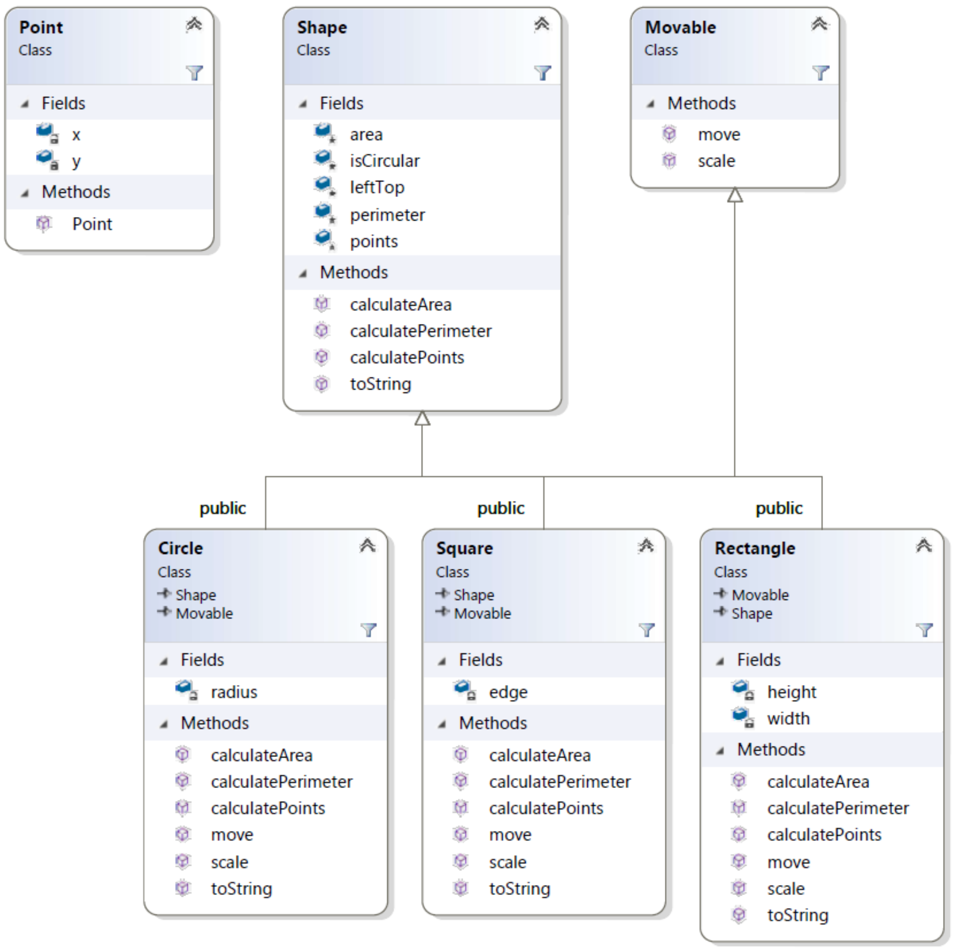
The outline of a driver class will be given to you (Driver.cpp) and you will need to complete the missing parts in this file as well as implementing the classes explained below. Driver.cpp will ask the user to enter input commands through a command line interface to create and manipulate geometric shapes. Your program will create the shapes on a two-dimensional virtual coordinate system, where X and Y coordinates will define a grid with origin (0,0) as below:



Implementation:

Your first task in this assignment will be to implement a **Point** class to represent a location in this coordinate system. Point class will have two integer member variables, denoting the x and y coordinates in the 2D plane. Your shape classes will use the Point class to represent coordinates.

You will then implement two abstract classes called **Shape** and **Movable**, and three child classes **Circle**, **Square** and **Rectangle**, which extend both Shape *and* Movable. A simplified class diagram of the program is given below (please note that you may need to implement some functions that are not indicated below to get your program functioning):



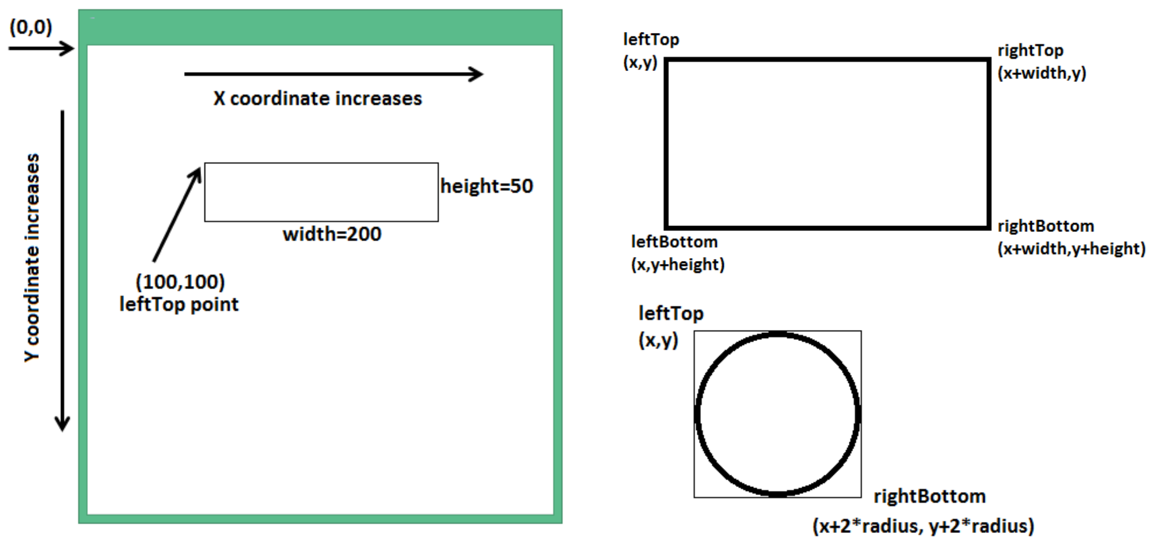
As the diagram shows, **Shape** class will have a **leftTop** variable of Point type, which keeps the left top coordinate information for a shape. **isCircular** will be used to identify circular objects. This data can be false by default and can be set to true when constructing circular objects.

Shape class will have a collection of Point objects named **points**. This collection will keep an ordered list of vertices that define the corners of a geometric shape. For each child shape, the abstract (pure virtual) **calculatePoints()** function will be overridden to compute where the corner coordinates lie and populate the list (this will be explained overleaf).

calculateArea() and **calculatePerimeter()** will also be virtual functions which should be overridden in child classes to calculate the area and perimeter of specific shapes. **toString()** function will be implemented to provide shape information consisting of type (Rectangle, Circle or Square), dimensions, coordinates of corner points, the area and perimeter. See the separate "sample execution" for details.

Calculating the points:

The figure below shows how the points should be calculated with respect to leftTop. For the Rectangle and Square classes, the order should be: leftTop, rightTop, rightBottom and leftBottom. For the Circle class, there will be only two points, leftTop and rightBottom as shown.



Movable class:
Rectangle, Square and Circle will also extend the abstract Movable class. Movable will have two functions:

- **move(int newX, int newY)** will shift a shape to a new coordinate by updating the leftTop and recalculating the other points.
- **scale(float scaleX, float scaleY)** will scale a shape in two dimensions. Two parameters will determine the scaling factors in each direction. For example a call to scale(2,1) will stretch the object in the x direction by a factor of 2, whereas the y direction will not be scaled. The scale operation will update the leftTop and recalculate the other points.

Please note that move and scale also affect the **area** and **circumference** of the objects, which should be handled by your code.

User commands:
In the driver class, the user should be able to create and manipulate arbitrary shapes. These should be stored in a vector: **vector<Shape*> shapes.** You will need to implement the various commands provided in Driver.cpp, which should behave according to the sample execution provided as a separate attachment. You do not need to reproduce the output formatting exactly as shown, but it should be similar.

Marking guidelines:
The marker **must be able to compile and run your code** to test its behaviour. You must also ensure that your **class and function names match the class diagram**, so that automated testing can be performed. The marker will also examine your program's structure and observe how it demonstrates the following:

- Good practice (user prompts, input validation, defensive programming, comments)
- OO concepts (appropriate use of constructors/destructors, access modifiers, etc.)
- Correct implementation of the inheritance hierarchy/polymorphic types
- Use of virtual/pure virtual functions where appropriate
- Use of advanced techniques such as operator overloading, templates & lambdas

Top marks can be achieved in the following ways:

- Use of STL/contemporary C++ components in your solution
- Innovative use of advanced techniques in your implementation (as above)
- Implementation of additional commands (the others must still work as requested), e.g.
  - **display**: show details of all shapes currently in memory
  - **clear**: delete all shapes (make sure the memory is freed!)

While you may discuss your solution with your peers, please note that this an individual assessment and therefore all code must be your own. Code will undergo checking for plagiarism.

Learning Outcomes Assessed:

[LO 2] Use advanced object-oriented principles and programming techniques in software development;
[LO 3] Apply advanced logical and mathematical techniques in the development of software solutions.

Knowledge & Skills Assessed:

- Input/output streams and C-strings
- Low-level memory management/pointers
- Evaluation strategies (call-by-value/call-by-reference)
- Effective use of encapsulation and access modifiers
- Inheritance, virtual functions and abstract classes
- Operator overloading and functional programming

Assessment Submission Instructions:
A ZIP file should be uploaded to *Assessment 2 – Supporting Documentation Upload* and should only contain your source code. *DO NOT include this briefing document with your submission.*

Date for Return of Feedback: TBC

Format for Assessment:
Your submission should consist of a zipped, complete Visual Studio solution. You must first run the "Clean Solution" command to remove any build files and remove the hidden ".vs" folder prior to submission. Be sure to re-download your Blackboard submission to verify it still builds.

Other compressed formats (tar.gz, rar, etc.) will **not** be accepted. The zipped directory should be compressed using Windows' *Send to → Compressed (zipped) folder* option, or in UNIX-based operating systems: `zip -r compressed.zip project_folder/`.

Feedback Format:
Feedback will be provided via Blackboard, where commentary will be provided on:

- Overall program structure and adherence to C++ conventions
- Missing or incomplete functionality, and areas where it may be improved
- Effective use of object-oriented design principles in accordance with the brief
- Any attempts at the additional tasks and use of STL/contemporary C++ components
- How well comments were used to provide explanations of the program's logic

Additional Information for Completion of Assessment:
Please make sure you have a clear understanding of the grading principles for this component as detailed in the accompanying Criterion Reference Grid. If you are stuck on any component of the assessment, please consult the recommended reading lists, lecture materials (slides,

recorded lectures) and workshop tasks in the first instance. Please be advised that the delivery team is not here to debug/fix your code, but to give general advice and further explanation of concepts you may be finding difficult.

Assessment Support Information:

For general enquiries about the assessment strategy, please contact the module coordinator.

For other queries about the module content, please contact a member of the delivery team.

Details of the delivery team’s office hours can be found on the module site on Blackboard.

Important Information on Dishonesty & Plagiarism:

University of Lincoln Regulations define plagiarism as 'the passing off of another person's thoughts, ideas, writings or images as one's own...Examples of plagiarism include the unacknowledged use of another person's material whether in original or summary form. Plagiarism also includes the copying of another student's work'.

Plagiarism is a serious offence and is treated by the University as a form of academic dishonesty. Students are directed to the University Regulations for details of the procedures and penalties involved.

For further information, see www.plagiarism.org