COMP 1633 Assignment 5 (Paint Program) - Part 3

Given: **Tuesday, March 29, 2022**
Electronic Copy Due: **Friday, April 8, 2022 at 11:59 pm <u>sharp</u>** ***NO LATES ACCEPTED***

<u>Project Scheduling</u>

Before you begin this part, from Parts 1 and 2 you should have:

- a `Grid` class test driver which illustrates that a grid can be drawn correctly
- a `Shape` class that behaves like a square and a test driver which makes use of the `Shape` class
- a complete `Shape_Collection` class and a test driver which verifies that `draw_shapes` and `write_names` operate correctly for a collection of squares
- a `makefile` with targets and rules to build each of these components

Before you begin work on Part 3, create a new sub-directory and save a copy of all your Part 2 files (including your `makefile`) in the sub-directory.

Remember the planning and time management process you should be trying to model:

- When a new part is released, write up a schedule which lists all the tasks you will need to complete to satisfy the specifications for that part.
- Then, estimate the time required for each task.
- Next, sum all the estimates to arrive at the total time estimate for the part and use a daytimer to allocate the work over the available days.
- Lastly, stick to your work plan!

<u>In part 3 make sure to change the width and height dimensions back to 80 by 22!!</u>

<u>Part 3 - The Problem:</u>

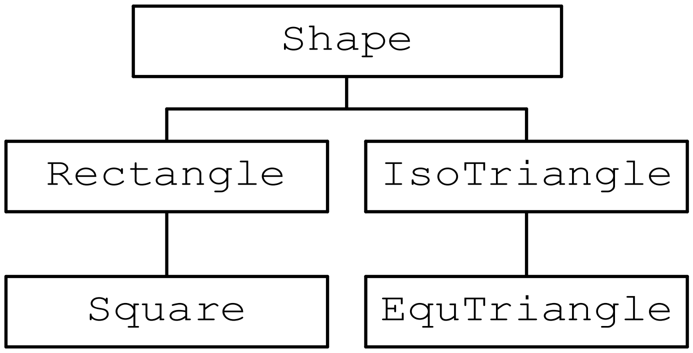
This is the final stage of the assignment. In this part, you will:

1. Add a few different shapes to the hierarchy.
2. Complete the implementation of a menu-driven interface that allows a user to paint pictures.

The Shape Hierarchy:

This part of the assignment involves modifying the design and implementation of the Shape class so that it becomes the root of a class hierarchy. The existing Shape class must be converted into a base class and a derived class for each type of polygon allowed in the program must be created.

The paint program must handle rectangles, squares, isosceles triangles and equilateral triangles. It must be possible to draw and move any of these shapes. The class hierarchy which must be implemented is:



The Base Class

Before you begin changing the Shape class, be sure to make a copy of the Part 2 code. Remember, the Part 2 version of the Shape class already behaves like a square, which is one of the derived classes you have to create, so you should be able to re-use some of the code for this part.

Three changes to the Shape class are necessary:

1. The `constructor` needs to have input parameters for all four coordinates of the bounding box. This will be the only constructor for the Shape class. Why? If you can't see why then ask your professor or an IA.
2. The `draw` method needs to be a pure virtual method. The method prototype should be as follows:

   ```
   virtual void draw (Grid& aGrid) = 0;
   ```

   The base class <u>does not provide an implementation</u> for any pure virtual method so remove the method from the Shape.cpp file.

3. A `virtual destructor` for the Shape class needs to be added to the class specification. This function will be empty in the class implementation. This is required because the `Shape_Collection`'s `destructor` will want to invoke the appropriate Shape class `destructor`. None of the derived classes will need to implement a `destructor` as the compiler supplied destructor will be sufficient. Why is this the case?

The inclusion of a pure virtual method in the `Shape` class makes it an abstract base class. A generic shape cannot exist – shapes must now belong to one of the derived classes. Furthermore, all concrete derived classes must provide their own draw implementation. This makes sense: all shapes are "drawable" but the base class is too generic to provide a meaningful interpretation that will work for the derived classes.

The `move` method is not virtual. Because it only involves manipulating a shape's bounding box, the required action will be the same no matter what derived class the shape belongs to. There is no need to override the `move` implementation in any derived class.

The Derived Classes

It is up to you to design and implement the derived classes for each of the four polygon types, subject to the following requirements:

1. The `Rectangle` constructor must take its position, its width and its height.
2. The `Square` constructor must take its position and its width.
3. The `IsoTriangle` constructor must take its position, its base width and its height.
4. The `EquTriangle` constructor must take its position and its base width.
5. The position of a shape refers to the (left,upper) coordinate of its bounding box. Width and height information allows the (right,lower) coordinate of its bounding box to be determined. For `Square` and `EquTriangle`, the height can be calculated from the width.

Each type of polygon only needs to know its bounding box information in order to draw itself properly.

Important Note:

Examine the inheritance hierarchy carefully to identify parent-child relationships. It will not be necessary to implement all methods for all derived classes. You should make proper use of inheritance and override methods only when it is essential.

Example Client Code

A client of the class hierarchy must be able to include code such as the following:

```
Grid aGrid;

Square aSquare (2, 3, 5);               // square of width 5 at (2,3)
IsoTriangle aTriangle (33, 1, 12, 17);  // isosceles triangle of base
                                        // 12 and height 17 at (33,1)

aSquare.draw(aGrid);                    // render both polygons
aTriangle.draw(aGrid);

cout << aGrid;                          // OUTPUT #1: write the grid

aSquare.move(77, -2);                   // move square to (77,-2)
aTriangle.move(15,13);                  // move triangle to (15,13)

aTriangle.draw(aGrid);                  // re-render both polygons
aSquare.draw(aGrid);

cout << aGrid;                          // OUTPUT #2
```

OUTPUT #1 produces this picture (or something very close to it):

```
--------------------------------------------------------------------------------

                                      *
                                      *
  *****                              * *
  *   *                              * *
  *   *                              * *
  *   *                             *   *
  *****                             *   *
                                    *    *
                                    *    *
                                   *     *
                                   *      *
                                   *      *
                                  *       *
                                  *        *
                                  *        *
                                 *          *
                                 ************




--------------------------------------------------------------------------------
```

OUTPUT #2 produces this picture (or something very close to it – WITHOUT THE LINE NUMBERING!):

```
            1         2         3         4         5         6         7
  01234567890123456789012345678901234567890123456789012345678901234567890123456789
  --------------------------------------------------------------------------------
 0                                                                              *
 1                                        *                                     *
 2                                        *                                     ***
 3  *****                              * *
 4  *   *                              * *
 5  *   *                              *  *
 6  *   *                             *   *
 7  *****                             *   *
 8                                    *    *
 9                                    *    *
10                                   *     *
11                                   *      *
12                                   *      *
13                    *             *        *
14                    *             *        *
15                   * *            *        *
16                   * *           *          *
17                   *  *          ************
18                  *   *
19                  *   *
20                  *    *
21                  *    *
  --------------------------------------------------------------------------------
```

The Client Program:

The final step of the assignment is to provide a menu-driven user interface so the user can paint pictures. The program supports the following commands:

```
a – add a shape to the picture
d – delete a shape from the picture
m – move a shape in the picture
q – quit
```

When executed, it repeatedly:

1. lists the names of all shapes in the picture
2. renders the picture
3. prompts the user for a command choice
4. responds to the choice

The **add** command will prompt the user for the name of the shape to add. If the name already exists, a message will be displayed and the command terminated. Otherwise, the user is prompted for the type of shape to create (`r, s, i or e`) plus its position (the (left, upper) coordinate of its bounding box), width and, if necessary, height.

The **`delete`** command will prompt the user for the name of the shape to remove. If it doesn't exist, a message will be displayed and the command terminated.

The **`move`** command will prompt the user for the name of the shape to move. If it exists, it will also prompt the user for a new position. Otherwise, a message will be displayed and the command terminated.

The **`quit`** command will prompt the user for confirmation and either display a cancellation message or exit the program.

Error Handling in the Client Program

The program must handle all possible kinds of user input error. It is your job to categorize the possible kinds of input error and ensure that they are handled.

You should make special note of these rules governing semantics:

1. shape names must be unique. The characteristics of a name are the same as specified in part 2 and unique means case-sensitive. Trying to add a shape with a duplicate name is an error.
2. width and height values must be integers greater than zero; any other value is an error.
3. the x-coordinate of a position may be outside the range 0-79. This must be handled normally.
4. the y-coordinate of a position may be outside the range 0-21. This must be handled normally.
5. partly or wholly clipped shapes are legal.

Error handling should be performed by the *client*. There are no error cases to be handled by the `Shape_Collection` class, the `Shape` class, or any of the derived classes in the `Shape` hierarchy.

Design and Implementation

The client program has been partially completed. The file containing the skeleton main program is available in the course directory:

```
/users/library/comp1633/assignments/a5/p3/main.cpp
```

Your job is two-fold:

- complete the implementation of the command handlers for the `add`, `remove` and `move` operations.
- complete the implementation of the data validation function stubs which are already in the main program file and add any other functions needed for proper error handling.

User Input

All input in `main.cpp` utilizes the supplied `ioutil`. This code is not to be altered.

Strings

The client program uses the C++ standard string class for handling the Shape names. All ioutil functions still use c-strings. Chapter 8 of your text book has more information about strings. In particular, string objects have dynamic size so there is no need to handle possible overflow situations when reading strings.

The regular input operator >> skips leading whitespace and halts on (and discards) the first trailing whitespace. Thus, when reading using >>, an empty string cannot be read.

In addition, the assignment operator = is defined for string objects.

Documentation

For this program you need the following documentation:

- an ID block for all of your files
- for all of the public functions you create, write appropriate class documentation in the corresponding `.h` file, i.e. `Shape`, `Rectangle`, etc
- for any private member functions you create, write appropriate class documentation in the corresponding `.cpp` file.
- DO NOT provide documentation for any code that you didn't create, i.e. `Grid`, `Shape_Collection` and `main`.
- While you should comment the functions you add to the `Shape_Collection` and `main` these will not be marked so if they are omitted there will not be any mark deductions.

Part 3 Deliverables and Milestones:

For this part of the assignment, you need to do the following:

1. Modify the `Shape` class definition and implementation modules so that `Shape` serves as an abstract base class.

2. Design and implement four derived classes (`Rectangle`, `Square`, `IsoTriangle` and `EquTriangle`) according to the specified `Shape` hierarchy.

3. Finish the implementation of the `main` program.

4. Add appropriate targets and rules to your `makefile` so that these new components of the final project can be built.

Coding Style and Documentation

Write well structured, readable and self-documenting code. Concentrate on high quality documentation which explains the code clearly, while remaining concise. As always, follow the course coding and documentation standards.

Submission and Grading

Details submission instructions will be available on April 5nd. As previously stated you will be required to run the `Grid` test driver from Part 1 and the `Shape` test driver from Part 2. In addition, there will be instructions for using the program to draw a picture.