CS 3210, Lab #5, SHAPES CLASSES AND SHAPES CONTAINER

INTRODUCTION

The purpose of this lab is to create a class hierarchy and take advantage of the polymorphism features of C++, and an STL-based container class for the polymorphic shapes.

BACKGROUND

The field of computer graphics lends itself quite well to the use of base and derived classes. The ability of derived classes to inherit properties from a base class so that they do not have to be rewritten is quite powerful. In addition, the ability to override and customize methods from the base class and to add new methods allows a derived class to customize itself for its own needs.

For this lab we will be creating a small hierarchy of classes to represent graphics entities. Inheritance will be key in that it will allow us to **reuse** various properties and behaviors, as well as use polymorphism to treat all entities consistently regardless of their actual type.

As an example, consider the concept of a geometric "shape." It is easy to envision that the shape has some properties, such as color, location on the screen, etc. However, how do you draw a shape?

We will create an abstract base class, shape, placing all common properties and behaviors in this class (color, location), and derive a concrete class for each actual shape we wish to represent. The concrete classes will contain any additional properties needed by that shape (i.e. a circle needs a radius, but a line doesn't) as well as the behaviors specific to that concrete shape, such as how to draw itself.

Once we have a Shapes-class hierarchy working, we need a way to store a collections of shapes that will form a drawing or an image. Since any given image will contain an unknown number of shapes, an array would not be appropriate. Rather, we should turn to a more advanced data structure. The C++ standard library offers a variety of templated data structures for our use. This portion of the C++ standard library is often referred to as the "Standard Template Library," or, STL.

Templates in C++ have a syntax similar to generics in Java. While the mechanism is quite different, the syntax and end results are similar. The easiest STL data structure to understand and use is std::vector, which presents as a growable array similar to Java's ArrayList.

The next issue is what does the vector store? As seen in lecture, we cannot store Shape objects because vectors store by value. Thus slicing would occur when adding Lines or Circles. Rather, our only choice is to store Shape*, which will allow all of our Shapes to be stored and handled polymorphically. Since we are storing pointers, all memory management falls back on us. Remember, you will need to delete each object when it goes out of scope.

LAB ACTIVITIES

You will be designing an abstract base class along with a few concrete classes. **Starter code is not provided**, the design is up to you per the requirements listed below. Read all of the requirements closely and use them to generate your design. Generating a UML diagram is suggested, but not required. Our design will support 3D, but we will initially be working in 2D, so the Z dimension can be ignored for awhile – just set to 0 when not in use.

SHAPE BASE CLASS

This will be the abstract base class. This class will house properties common to all shapes as well as public methods (accessors and mutators - only as needed) for interacting with those properties. These properties must be initialized via Shape's constructor(s). The Shape class should implement the assignment operator as protected. Since the Shape class is abstract, we would not want users to do something like this: *shape_ptr1 = *shape_ptr2, but, we would like to keep the responsibility of managing Shape's properties within the Shape class. Thus, by making the assignment operator protected, derivatives of the Shape class can call it when needed, but we do not need to worry about a partial copy. This is **your** design, but you have to be able to defend your design. That is, be sure you think about access specifiers, const-ness, minimize redundant code, etc. You may be asked questions like "why did you make the color public/protected/private and will need to defend your decisions.

As a minimum, your Shape class must contain the following properties:

- **color**
  All shapes will have a color property. You can represent this as three different integers (R, G, B), or a single integer with the three colors combined (24-bits minimum suggested).

As a minimum, your Shape class should declare the following methods:

- **Constructor(s):**
  Initialize properties that reside in Shape.
- **Copy Constructor:**
  Copy Shape's properties during a copy operation.
- **Destructor:**
  Probably no work for it in Shape, but good to put in just in case (and make virtual - why?).
- **Assignment operator:**
  This will handle assignment of Shape's properties only. Since Shape is abstract, this method should never be called by "users." See note below.
- **draw:**
  This will be the method called to draw the shape. Since Shape itself is abstract, we do not have anything for this method to do in the base class, thus, it will likely be "pure virtual." Alternatively, you could make this method just virtual, and chain to it from over-ridden versions in order to set the color or other properties prior to the concrete shape drawing itself. **This method must accept a 'GraphicsContext*'** in order to know where to draw the shape. The functions gc->setColor(…) and gc->drawLine(…) will be called from inside the draw function.
- **out:**
  This method will be like the Matrix::out method. It will accept a std::ostream& and dump the Shape properties to the ostream. Note, this method should be virtual, meaning the derived class (concrete) shape's version will get called first. Be sure to chain to this version to have Shape dump its own properties, that is, for example, if Line's out method is called, Line's out should call Shape's out method to output Shape's properties, and then Line's out should only output Line's properties.
- **in:**
  This is the counterpart to out, and will accept a std::istream&. It will be used to read shape properties from a text file. The format will be whatever you specified for out.
- **clone:**
  This is the "virtual constructor" that will be discussed in lecture. You will need this to make copies of Shapes. Since Shape will be abstract, this method should be pure virtual.

CONCRETE SHAPE CLASSES

Required Shapes:

- **Line**
  - Needs at least two coordinates
  - Constructor:

    line(int x0, int y0, int x1, int y1, unsigned int color);
- **Triangle**
  - Needs at least three coordinates
  - Constructor:

    triangle(int x0, int y0, int x1, int y1, int x2, int y2, unsigned int color);

Optional Shapes:

- **Circle Class**

  We need to decide the geometric interpretation of the origin being held by the Shape class. In addition, how do we specify the circle - radius, diameter? Actually, a circle will turn out to be somewhat difficult to deal with later on when we are performing transformations. If you wish circle to work with 2D transformations, you will likely want to represent the circle as an origin and a point on the circle.
- **Rectangle Class**

  While we have a number of options for constructing the rectangle in terms of what information we pass to the constructor (origin, height and width; origin and opposite corner) we will need to be sure we store all four coordinate pairs for the corners, and have at least one constructor that can accept all four coordinate pairs. This is necessary for our Rectangle to work properly later on when we are doing transformations.
- **Polygon Class**

  This one will be a bit tricky. Our polygon can have any number of vertices, so the question becomes how do we store data of unknown length? In addition, how do we construct it in the first place. As a guide, consider how you draw a polygon in a typical drawing program. Each mouse-click creates a new vertex until entry is done. You will need your Polygon to behave similarly. For example, the constructor could establish the origin/location, and then an additional method (addVertex()?) should add each additional vertex to completely specify the polygon. Definitely worth extra credit…

As a minimum, all concrete shape classes must contain the following properties:

- **Coordinates**

  All shapes will have coordinates. The geometric meaning of location may vary by concrete shape, but all shapes will have at least one. This must be a floating-point value(s) (float or double) for a 2-D Cartesian system. I suggest you use a **Matrix** object to store your coordinates. To be ready for future labs, I suggest you actually use a 4×1 Matrix (four rows, one column). The first row will be the X value, the second Y, the third Z, and the fourth to **1.0**.
- **Clone**

  All shapes will have a clone function. This is discussed in lecture.

- **Constructor**

  All shapes will have a constructor. The arguments for each shape are listed above.

- **Others**

  You must decide what other functions your shapes will implement. Each function **must** have a description of what it does/its purpose in the .h file.

IMAGE CLASS

Once you have some shapes, design, document, and implement an image class that will act as a container for a variable number of Shapes. A description of the methods you will need to implement in this class are described below.

- **Constructors**
  No-argument - creates a default image (empty, containing no shapes)
  Copy - Creates a duplicate image from one that already exists; the newly-created image contains deep copies of the earlier image's shapes.
- **Destructor**
  Delete all the shape objects from the container (the STL destructor will not do this automatically since it only holds pointers to the shapes unless you use std::unique_ptr) and delete all other dynamic objects.
- **operator=**
  Assigns an image to contain the same shapes as another image. The image object must first completely destroy its current collection of shapes, and then create deep copies of the shapes it is being assigned from the other image. Be aware of self-assignment.
- **add**
  Add a new shape (via a pointer) to the container.
- **draw**
  Ask all the shapes to draw themselves by iterating through the image's collection of shapes, and invoking each shape's draw() method via the pointer. This method will need to accept a GraphicsContext*.
- **out**
  Output all the shapes to an ostream (console or data file). This routine will need to accept an ostream&.
- **in**
  Read a set of shapes from an istream (console or data file). This routine will need to accept an istream&.
- **erase**
  Remove all the shapes from an image and return any dynamic memory they occupy.

Of all these methods the **in** method may turn out to be the most difficult to implement. In most cases C++ can automatically pick the proper virtual method for derived class objects. All of this presupposes that the type of object is already known and created, but when reading an object from a file or the terminal we do not know what kind of object we have until after we begin reading it. This presents somewhat of a dilemma. There have been a number of suggestions on how to solve this particular problem and often revolve around a special set of classes called factory or foundry classes. This is a nice abstract way to do this, but you may be able to approach the problem in a simpler fashion. Essentially each shape will need to write out a unique character that identifies its type. The container class's in method in this case will have to first input the type field and then use a construct like a switch-case to determine which type of shape it is to create and then input the remaining data. This requires that the container class in method be updated every time you create a new kind of shape. (Typically a factory class's job would be to do this for the container class and thus isolate the container class from this level of detail.) A better location for this logic might be as a static method in the Shape class, but location is not as important as functionality at this point.

IN SUMMARY

- Design and implement the abstract base class, Shape.
- Design and document the following concrete Shape derivatives: Line and Triangle. Circle, Rectangle, and Polygon are optional.
    - Have each derived shape contain different data members as appropriate (that parametrically define the object).
        - Common properties should be pushed to base classes.
        - Use floating point types (double or float) for all coordinates (preferably instances of Matrix as discussed above).
    - Implement all of the member functions. Use your code from Lab 2 to draw lines and circles. Rectangles and polygons should use line drawing.
    - Your Triangle class **must** store its three coordinates – not width and height, bounding box, etc.
- Implement the image container class.
- Develop a driver program for testing purposes. The driver program should
    - Be able to create multiple image objects (so you can thoroughly test copy and assignment) and add an arbitrary number of the derived shapes to the container.
    - Be able to make a new copy of an existing image (via the copy constructor). Note, in order to perform a deep copy, you will need to invoke the clone() method(s) in the shape derived classes.
    - Be able to assign the contents of one image to another (using the assignment operator). The contents of the target image must be destroyed correctly.
    - Be able to write an arbitrary image to file (using « or in/out via istream/ostream). Note - one image per file.
    - After closing your program and restarting it, be able to read an image from a file.
    - Thoroughly test all of the methods in the image class to make sure they work correctly. Pay special attention to the cases of when you copy an image (either through the copy constructor or operator=) and then destroy the original, because if you don't correctly implement deep copying, you'll see big problems. This will help test your clone() methods (in the shape derived classes); i.e., to make sure you didn't just copy a pointer (shallow copy), but rather copied the entire object (deep copy).

DELIVERABLES

This lab will span two weeks:

- **Due Week 6:**

  .h files for all classes must be complete. The code must compile, but the function definitions do not need to be finished. The Canvas submission shall consist of a pdf of all code.
- **Due Week 7:**

  Completed lab due. The Canvas submission shall consist of a pdf of all code, and a zip file with all files needed to build the complete project, including a makefile.

GRADING CHECKLIST

- All requested methods implemented per descriptions with reasonable selections of parameters
- All methods have thorough descriptions in the .h files
- All common attributes present in base class
- Proper handling of dynamic memory
- Virtual constructor used to duplicate shapes when duplicating images
- All methods tested in some fashion, particular care taken to test copying of container