CSCI-1200 Data Structures — Spring 2018
Lab 13 — Multiple Inheritance & Exceptions

For this lab you will build a class inheritance structure to match the hierarchy of classic geometric shapes. The finished program will read lists of 2D point coordinates from a file, determine the shape described by each list of points. We will use a somewhat quirky method to determine the type of each shape. We will pass the list of points to each specialized shape constructor in turn, and if the constructor doesn't fail, then we know that that list of points is in fact that type of shape. Remember, the only way for a constructor to fail is to throw an exception.

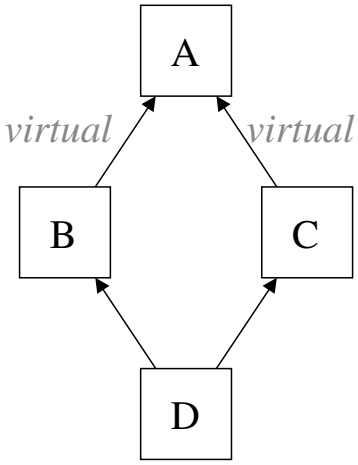
Checkpoint 1 - Shape Hierarchy

Polygon, Triangle, Quadrilateral, Isosceles Triangle, Right Triangle, Isosceles Right Triangle, Equilateral Triangle, Rectangle, and Square. Note that a particular shape may be correctly labeled by more than one of these names; e.g., a Square *is also a* Quadrilateral.

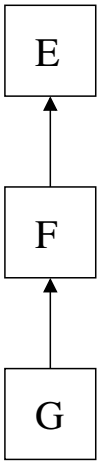
Draw the class hierarchy with arrows indicating all of the inheritance relationships (you do not need to include the member variables or member functions). Be neat, have a consistent (up or down) orientation to your arrows, and avoid messy scribbles or cross outs and arrow crossings. Sketch the shapes in the `input.txt` file and write the name of each shape next to the most specific type of shape it represents.

http://www.cs.rpi.edu/academics/courses/spring18/csci1200/labs/13_inheritance_exceptions/input.txt

The inheritance diagram of these shapes includes *multiple inheritance*, specifically in the form of the *Diamond Problem*. That is, Class D multiply inherits from Class B and Class C, and Class B and Class C each inherit from Class A. Thus when an object of type D is created, in turn instances of B and C are created, and unfortunately both will try to make their own instance of A. If two instances of A were allowed, attempts to refer to member variables or member functions of A would be ambiguous. To solve the problem, we should specify that B *virtually* inherits from A and C *virtually* inherits from A. Furthermore, when we construct an instance of D, in addition to specifying how to call constructors for B and C, we also explicitly specify the constructor for A. Note how in the single inheritance example on the right, G only explicitly calls a constructor for F.




```
class A {
public:
  A() {}
};
class B : virtual public A {
public:
  B() : A() {}
};
class C : virtual public A {
public:
  C() : A() {}
};
class D : public B, public C {
public:
  D() : A(), B(), C() {}
};
```




```
class E {
public:
  E() {}
};
class F : public E {
public:
  F() : E() {}
};
class G : public F {
public:
  G() : F() {}
};
```

Label the virtual inheritance paths in your diagram. *Hint: 2 of the inheritance arrows will be labeled virtual.*

To complete this checkpoint: Present your diagram to one of the TAs.

Checkpoint 2

To start the implementation, we'll focus on just 7 of those shapes: Polygon, Triangle, Quadrilateral, Isosceles Triangle, Equilateral Triangle, Rectangle, and Square. This subset will allow us to initially ignore the multiple

inheritance diamond property and the need for virtual inheritance it causes.

Download the code and initial example:

http://www.cs.rpi.edu/academics/courses/spring18/csci1200/labs/13_inheritance_exceptions/simple_main.cpp
http://www.cs.rpi.edu/academics/courses/spring18/csci1200/labs/13_inheritance_exceptions/utilities.h
http://www.cs.rpi.edu/academics/courses/spring18/csci1200/labs/13_inheritance_exceptions/simple.txt
http://www.cs.rpi.edu/academics/courses/spring18/csci1200/labs/13_inheritance_exceptions/output_simple.txt

The program expects 2 command line arguments, the names of the input and output files. Each line of the input file begins with a string *name* followed by 3 or more 2D coordinate *vertices*. The output categorizes each shape into one or more classes, and into groups with equal angles, equal edges, and/or a right angle.

The provided code includes code to call the constructors of the different classes, generally ordered from most specific/constrained to least specific. For example, the program will try to create a Square with the data first, and only if that constructor fails (throws an exception) then it will it try to create a Rectangle.

We also include a `utilities.h` file with a number of simple geometric operations: e.g., calculate the distance between two points, calculate the angle between two edges, and compare two distances or two angles and judge if they are sufficiently close to be called "equal". Remember that you usually don't want to check if two floating point numbers are equal; instead, check if the difference is below an appropriate tolerance.

Create a `polygons.h` file (and optionally a `polygons.cpp` file). Create the 7 classes for these shapes deriving classes from the other classes as appropriate. In each constructor write code to check whether the vertices passed in meet the requirements for that shape. Throw an exception if you find a problem. *Note: Just throw a value of type integer. The value thrown is unimportant in this program – it will be ignored.*

To complete this checkpoint: Compile, run, and debug your program. Study the output and confirm that your program is correctly labeling the shapes in simple.txt

Checkpoint 3

Now tackle the problem of multiple inheritance and the diamond property. Focus on just the triangles. Here is a revised `main.cpp` for all 9 shapes.

http://www.cs.rpi.edu/academics/courses/spring18/csci1200/labs/13_inheritance_exceptions/main.cpp
http://www.cs.rpi.edu/academics/courses/spring18/csci1200/labs/13_inheritance_exceptions/triangles.txt

In organizing your code for this lab, try to avoid unnecessarily duplicating code. For example, don't implement the `HasARightAngle` function in *every* class. Also, you don't need to check if the `RightTriangle` has 3 vertices (the constructor for its base class will do that). Instead, allow the derived class to rely on the implementation of that function in its parent class. Similarly, don't recalculate measurement data if you can deduce information from properties of that shape. For example, when a RightTriangle is asked if it `HasARightAngle`, no calculation is necessary — the answer is guaranteed to be true.

To complete this checkpoint: Be sure to ask your TAs for help if you get stuck. Near the end of lab period show a TA your progress.