CSCI-1200 Data Structures — Spring 2014
Homework 10 — Multiple Inheritance & Exceptions

For this assignment you will build a class inheritance structure to match the hierarchy of classic geometric shapes. Your finished program will read lists of 2D point coordinates from a file, determine the shape described by each list of points, and then output various statistics about the shapes to a file. 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.

Shape Hierarchy

You are required to recognize 17 different shapes: Polygon, Triangle, Quadrilateral, Isosceles Triangle, Right Triangle, Isosceles Right Triangle, Obtuse Triangle, Isosceles Obtuse Triangle, Equilateral Triangle, Trapezoid, Kite, Arrow, Parallelogram, Isosceles Trapezoid, Rectangle, Rhombus, 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.

For the first part of the homework you must draw a detailed diagram of the class hierarchy. The diagram must include all 17 different shapes. You should draw arrows indicating all of the inheritance relationships. To receive full credit, the diagram should be legible, neat, and well organized, with no messy scribbles or cross outs, a consistent (up or down) orientation to the edges, and few or no arrow crossings. You do *not* need to include any of the member variables or member functions in this diagram. We are just looking for the high level relationships between class names. You should label the virtual inheritance paths (described on the next page). *Hint: 9 of the inheritance arrows will be labeled virtual.*

Important Note: There is some debate about the exact definitions of the relationships of some of these shapes. One of the more contentious is the definition of an Isosceles Trapezoid. Is a Rectangle an Isosceles Trapezoid? Is a Rhombus an Isosceles Trapezoid? Some say "no" to both questions, and insist that a Trapezoid has exactly one pair of parallel edges. Others say "yes" to both, as long as one pair is parallel and the other pair is equal length, it's an Isosceles Trapezoid. We will apply the following definition: The two base angles of an Isosceles Trapezoid must be equal, which means that the shape has bilateral symmetry around the perpendicular bisector of the base. Thus, a Rectangle *is* an Isosceles Trapezoid and a Rhombus *is not* an Isosceles Trapezoid. Furthermore, we settle a different but similar dispute by declaring, for this homework, that an Arrow is *not* a Kite because one diagonal does not bisect the other.

Since many of you will draw this using pen/pencil & paper, this part of the homework is due in hardcopy to your graduate TA ***in your normal lab section on Wednesday, April 30th***. Even if you choose to draw the diagram electronically, you are still required to print it out and submit the paper in your normal lab section. You may not use late days for this portion of the assignment.

Provided Code

We provide code that implements all of the I/O for this homework assignment. The executable expects 2 command line arguments, the names of the input and output files. The input file contains a number of lines of data. Each line begins with a string *name* followed by 3 or more 2D coordinate *vertices*. The output, which is written to a file, includes basic data on the 17 different shape classes: how many of the input shapes are members of each class, and what are the names (in alphabetically sorted order) associated with those members? Also, the output lists data on the shapes with all equal angles, all equal edges, at least one right angle, obtuse angle, or acute angle, and which shapes are convex vs. concave.

The provided code includes all of the code to call the constructors of the 17 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) will the program try to create a Quadrilateral.

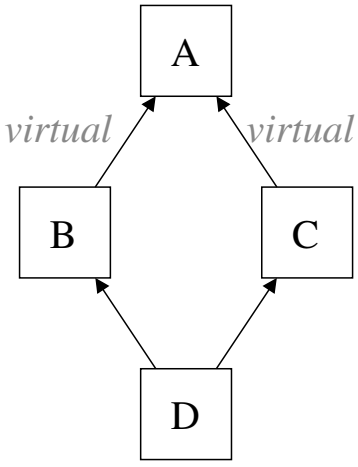
We also include a `utilities.h` file with a number of simple geometric operations: e.g., calculate the distance between two points, calculate the angle between two edges, and compare two distances or two angles and judge if they are sufficiently close to be called "equal". Remember that you usually don't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

Important Note: You should not modify the provided code.

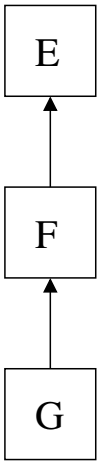
Your Implementation

Your task for this assignment is to implement the 17 shape classes. You should break up the code into at least one new `.h` file and at least one new `.cpp` file. Some hints about the implementation:

- In our sample solution, only one class has member variables, the base class. The two variables are the `std::string` *name* and an `std::vector<Point>` of the *vertices*.
- In our sample solution, we have implemented one constructor for each class, with specific arguments: the *name* and the *vertices*. We do not need to define the default constructor, the copy constructor, or the destructor.
- In organizing your code for this assignment, try to avoid unnecessarily duplicating code. For example, don't implement the `HasARightAngle` function in *every* class. 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 Rectangle is asked if it `HasARightAngle`, no calculation is necessary — the answer is guaranteed to be true.
- The inheritance diagram of these shapes includes *multiple inheritance*. Furthermore, the multiple inheritance is 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() {}
};
```

You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages, etc. with anyone, please list their names in your `README.txt` file. When you are finished please zip up your folder exactly as instructed for the previous assignments and submit it through the course webpage.