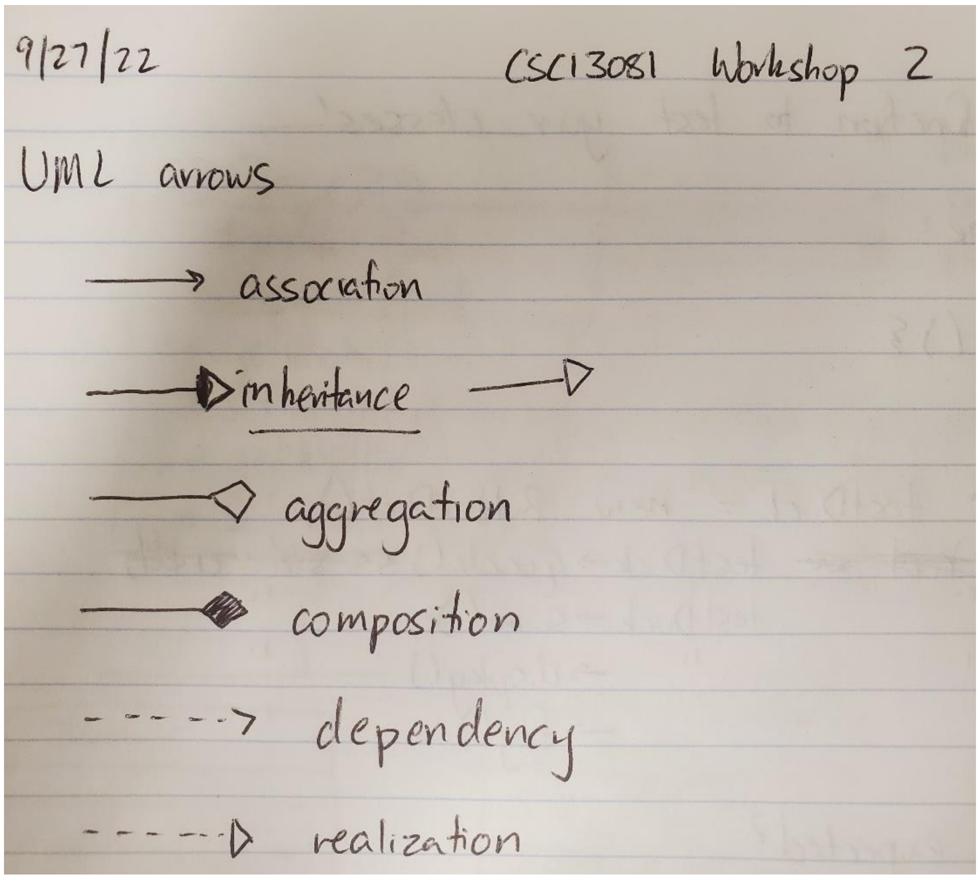
Feel free to put all of this code into one file. I know usually you would have about 16 files which is good practice, but it's hard to grade. Therefore, one file with all the classes and main() is okay.

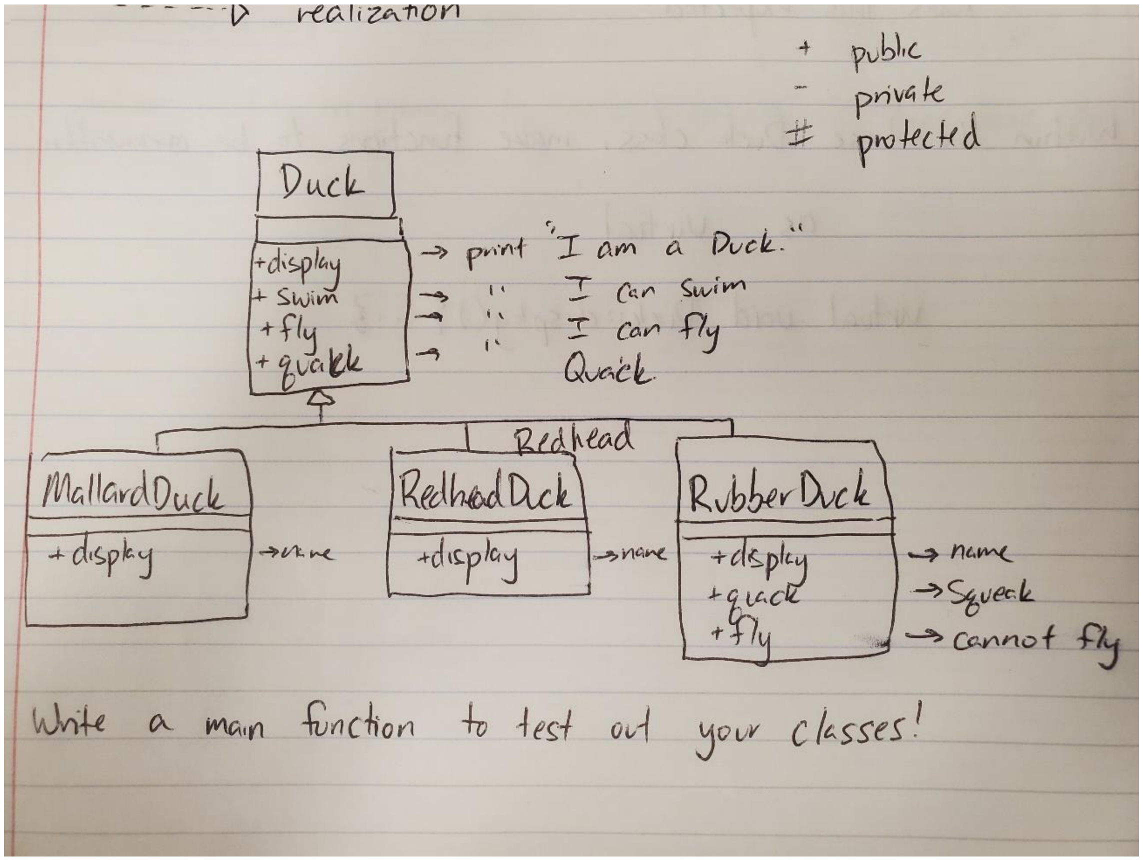
Part A - Ducks

1. UML Arrows – there are a lot of arrows used in UML, we will be focusing on only one today: inheritance; we will focus on the others in workshop 3



2. In UML, objects are different boxes. The box is divided into three sections: the name of the class, the variables, and the functions.

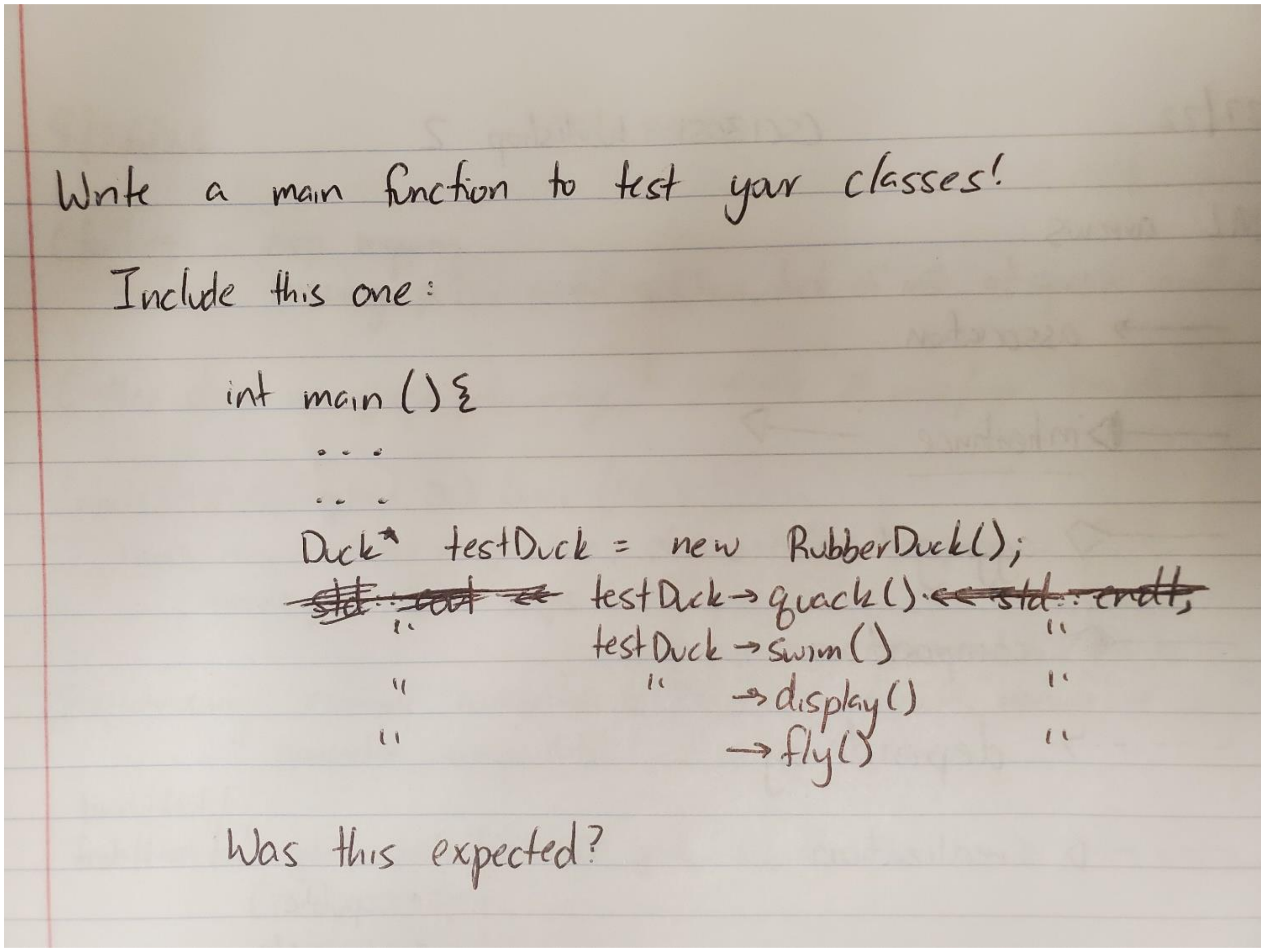
3. +, -, and # denote public, private, and protected



Notice that the base class here is Duck and the others are inherited from it. We can tell by the use of the inheritance arrow. Also, there are no member variables for any of the classes, only functions, all of which are public. For each function, just have them print out something. An example of all the prints for Duck are above. Notice how for RubberDuck, the quack function prints Squeak and the fly function prints that it cannot fly. For display, each one should print I am a ____ where ____ is the class name.

4. Code the four classes described in UML above.

5. Write a main function to test out your four classes.



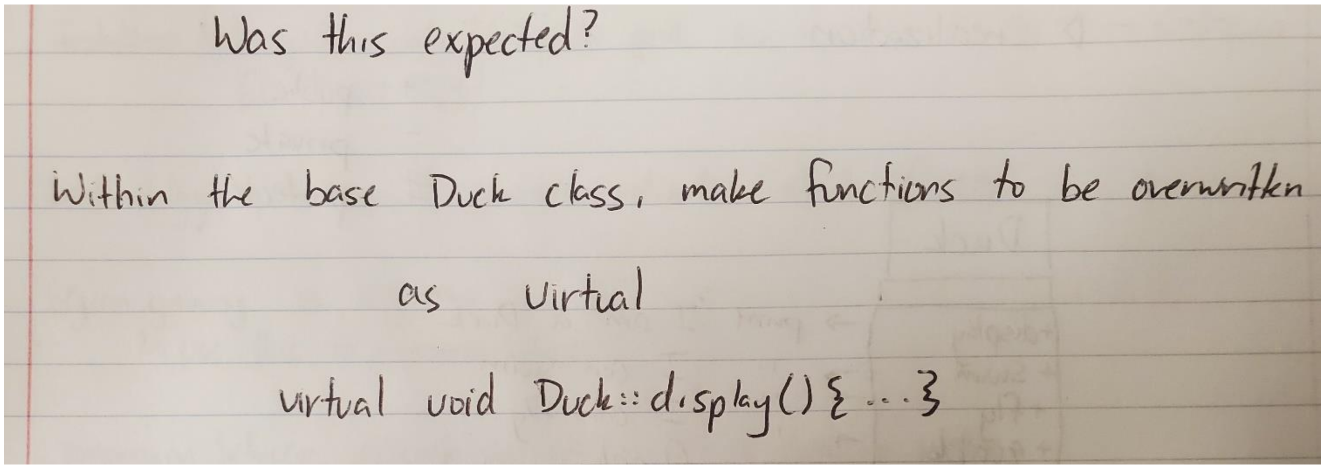
Write a main function to test your classes!

Include this one:

```
int main(){
    . . .
    . . .
    Duck* testDuck = new RubberDuck();
    ~~std::cout <<~~ testDuck→quack()~~<< std::endl;~~
          "        testDuck→swim()                 "
          "           "    →display()              "
          "                →fly()                  "
```

Was this expected?

You probably noticed that when we do it this way, it was using the definition from Duck instead of the definition from RubberDuck.



Was this expected?

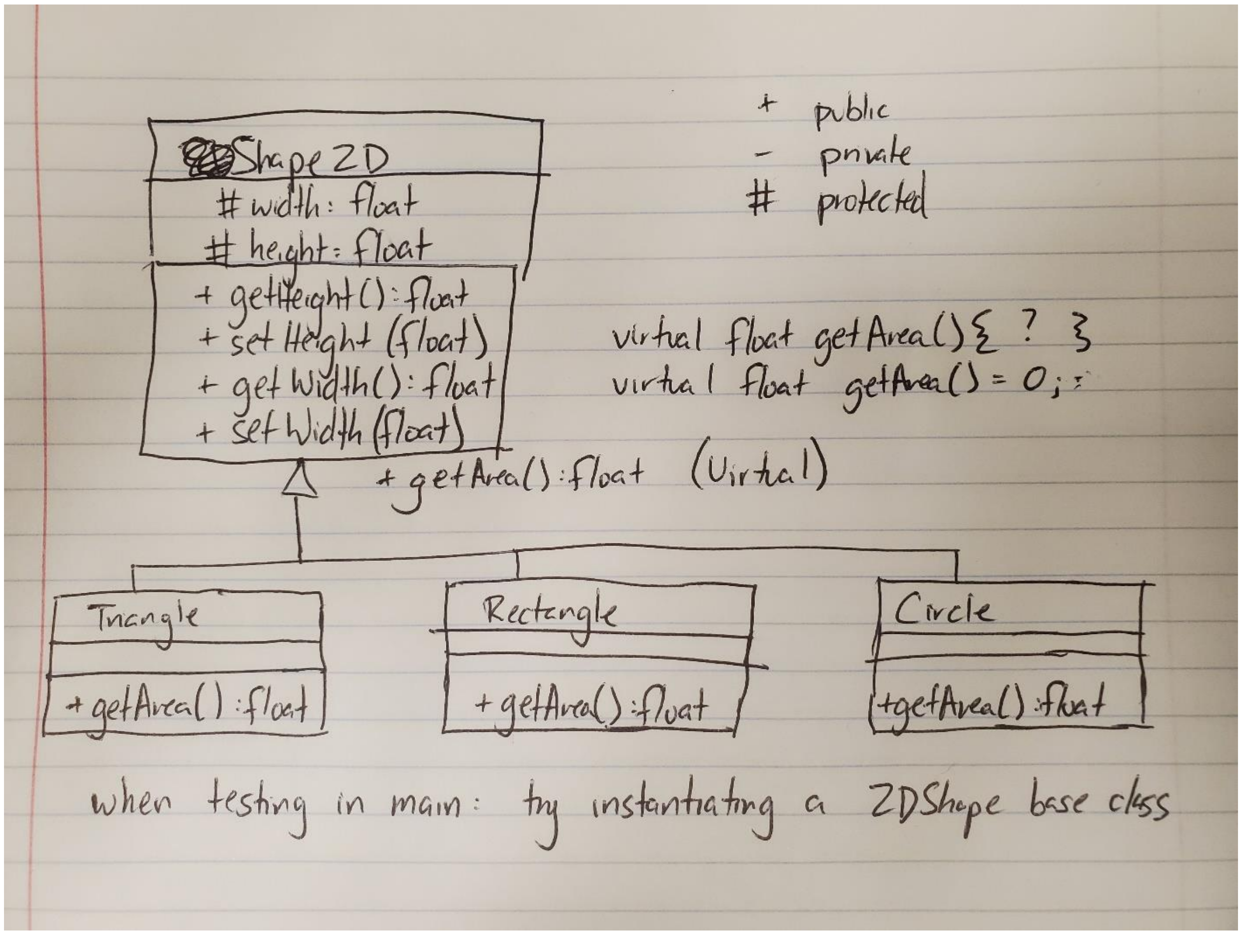
Within the base Duck class, make functions to be overwritten as virtual

```
virtual void Duck::display(){ ... }
```

Now when we run the test code from above, we can see that it is in fact using the definition from RubberDuck instead of Duck. That is because we included the virtual keyword in the base class's definitions that were to be used differently in the inherited classes.

Part B – Shapes

Make this:



If we put a getArea function inside the base class, it is impossible to define correctly. (There is no formula to calculate area for all 2D shapes). We do require that every inherited class from Shape2D has a getArea function. Not only that, we want it such that if we hand this off to someone else, they are also required to make a getArea function. Now we could just verbally tell them that it is a requirement, or we could make a **pure virtual** function. What a pure virtual function does is that it doesn't get defined and then every inherited class must define it. It also makes the class abstract. Once a class is abstract, it is impossible to instantiate. The syntax for the pure virtual method in Shape2D is as follows: `virtual float getArea() = 0;`

Now code the above system based on the UML. Remember to make the base class abstract by having that pure virtual function.

Submission:

The submission is on Gradescope. Feel free to either zip your files and upload a zipped folder, upload files individually, or just upload one file that contains all your code (probably 100+ lines). This is due Thursday, September 27th at 11:59 pm.