Lab 10 - Multiple files and Overloading

Directions

- The labs are marked based on attendance and effort.
- It is your responsibility to ensure the TA records your progress by the end of the lab.
- Do each step of the lab in order.
- While completing these labs, you are encouraged to help your classmates and receive as much help as you like. Assignments, however, are individual work.
  You may not work on assignments during your lab time.
- If you complete the lab early, you should experiment with C++; however, you may leave if you prefer.
- If you do not finish the lab exercises during your lab time, you are encouraged to complete them later to finish learning the material. You will still receive full marks if you arrived on-time and put in your best effort to complete the lab.

1. Overloaded Moped

Remember that it's bad to overload a truck, but perfectly OK to overload with C++. So, now we are going to combine the two ideas: an overloaded moped!

1. Create a new file named `mopedTest.cpp`.
2. In the file, create the class `Moped`.
   - Give it the following attributes: **a name** (string), **a year** (int), and **a colour** (string).
   - Write the (non-inline) **default constructor** for the class which initializes all the attributes to reasonable default values.
   - Write a (non-inline) **destructor** for the class which outputs some text to the screen (using cout) which states the object is being destroyed, and shows the values of all attributes. (Sample below).
3. Create a new test function in the client code (outside of the `Moped` class) to test your class so far.
   - Place your new test function **below** `main();` you will need to add a prototype above `main()` in order to call it.
   - Have the new test function instantiate a `Moped`.
   - Run your program. Your output may look similar to the following:

```
I'm sorry; your white 2000 moped named unnamed was just destroyed!
```

4. Create the following two overloaded `Moped` constructor:
   - Parameters accepted: just the name.
   - Parameters accepted: a name, a year, and a colour.
5. Change your test function to create one additional `Moped` for each of the overloaded

constructor options (i.e., test each new option).

- When all the objects go out of scope, the output may look similar to the output below. (Order of destruction may be different).
  - Note that the order in which they are destroyed may be different (opposite) to the order you created the objects.

```
I'm sorry; your Ferrari-red 2011 moped named Lightning was just
destroyed!
I'm sorry; your white 2000 moped named Go-go-stop was just destroyed!
I'm sorry; your white 2000 moped named unnamed was just destroyed!
```

6. **Understanding:**
   - Why could you not add another constructor to the Moped class which accepts only the colour?
   - Why are we not overloading the destructor as well?
   - Understand and be able to explain when the constructor is called, and when the destructor is called.

2. Multiple files

1. Convert your `mopedTest.cpp` file into multiple files.
   - Extract the `Moped` class into `moped.h` and `moped.cpp` files.
   - Include the new header file in both `moped.cpp`, and in `mopedTest.cpp`.
   - Put an include-guard in `moped.h`. See the notes for details as needed.
   - Include `<iostream>` and `<string>` in the appropriate files.
   - In VC++, have all three (3) files set to be compiled (i.e., do not exclude any of the three from the build).
2. Re-run your program. Its output should look identical to before.
3. **Understanding:**
   - What to put into each of the different files for class declaration, class implementation, and client code.
   - What #include ... statements are needed to make the program work once it's been split into multiple files.

3. Testing with asserts

1. Add `getYear()` and `setYear()` functions to the `Moped` class.
   - Make `setYear()` check that the new year is between 1950 and 2050 inclusive. If the new year is valid, change the member variable and return `true`; if it is not valid, just return `false`.
2. Assert statements are used to help programmers debug their code. They are used to trigger run-time errors when they "detect" a logic error.
   - Include `<cassert>` in `mopedTest.cpp` in order to use `assert()`.
   - In `main()`, add the following statements:
     ```
     int x = 100;
     cout << "First"  << endl;
     assert( x > 0 );
     cout << "Second" << endl;
     assert( x < 50 );
     cout << "Third"  << endl;
     ```
   - Read this code; which `assert()` should fail? (i.e, which one's condition is false?)
   - Run the program. It should generate a run-time error. Track down which line failed.
   - Comment out these 6 lines of code (they are just to experiment with).
3. In `mopedTest.cpp`, create a new test function for testing the `getYear()` and `setYear()` member functions. Have `main()` call this new function.
   - Create a `Moped` object, and use an `assert()` statement to verify it's year. Ex:
     ```
     Moped myMoped("Test 1", 2011, "blue");
     assert(myMoped.getYear() == 2011);
     ```
   - Run the program and prove your test passes.
4. Add additional tests to the function, testing `getYear()` and `setYear()` for some interesting values:
   - Each test will involve:
     - Calling `setYear()` with the new value, and checking its return value.
     - Calling `getYear()` and ensuring it returns the correct value.
   - Each value tested will have the following form (example is for the year 2000):
     ```
     assert(myMoped.setYear(2000) == true);   // Should pass.
     assert(myMoped.getYear() == 2000);       // Test for new value.
     ```
     - The first assert statement checks that `setYear()` succeeds, and returns the correct value. If you are passing in an invalid argument (such as the year 1234), then `setYear()` will fail and should return false.
     - The second assert statement checks that `getYear()` returns the correct year. If `setYear()` should pass, then check that `getYear()` returns the new value; if `setYear()` should fail, then check for the previously set year. Ex:

     ```
     assert(myMoped.setYear(1234) == false);  // Should fail.
     ```



```
assert(myMoped.getYear() == 2000);       // Test for old value.
```

- Pick five or more values to test the full range of values that `setYear()` could be passed. Suggested values to test:
  - The smallest number it should accept.
  - The largest number it should accept.
  - The smallest number (above its acceptance range) it should reject.
  - A negative number, zero, ...

5. Run the program and fix any bugs.
   - Note that often the test-code end up being longer than the code being tested.
   - When an assert fails (run-time error), be sure to double check the assert's code as well. It's very common to have errors in the test code to start with!
6. **Understanding:**
   - What does an `assert()` do when it fails?
   - What is the advantage of using an assert over dumping text to the screen? Consider writing tests for 200 functions. Would it be easy to look through 1000 lines of output to make sure it is all correct? How could asserts help?

4. Challenges

- Add a `ride()` method to the `Moped` class which, when called, displays to the screen a randomly selected message (out of three or four pre-programmed messages). For example, it might say "Wheeee! I can barely walk this fast!" or "Riding in style!", or "My other car's a Pinto".
- When a Moped is created with a name, make it output the name like a bumper sticker.
  - For example, given the name "Lightning", the constructor might output the following to the screen.

```
*****************
* ! Lightning ! *
*****************
```

  - Make your code scale to large and small names (handle names like "Go" and "I think I can, I think I can... Nope. I can't").
- Add a cargo-capacity feature to the `Moped` class.
  - Allow each instance of the `Moped` class to store a string which represents the cargo it is carrying.
  - Add a constructor, accessor and mutator function for this string. However, a `Moped` is small and not terribly stable. Therefore, ensure that the user only gives you strings to store which:
    - are less than 20 characters long, and
    - only contain an even number of characters (after all, it has to be balanced!).
  - See if you can make it enforce that the string must have an even number of upper-case characters (don't want an odd number of big letters making it unstable).

5. Skills and Understanding

You should now be able to answer all the "understanding" questions in the previous sections. Complete the following to get credit for the lab:

- Show the TA the following:
  - Your operational programs which complete all of the above tasks.
  - The TA may ask you to explain any section of the lab, or answer any of the "Understanding" questions.
- **Nothing** is to be submitted electronically or in hard-copy for this lab.

