COMSC-200 Lab 6
Self-Reference

GOOD PROGRAMMING PRACTICES *show / hide*

ABOUT THIS ASSIGNMENT
In this lab session, you begin solving the **ShortestRoute** problem -- a problem that will be completed in the final project. You will also practice programming with classes by doing exercises from our textbook.

As you complete this assignment, post the required files to the COMSC server. You may post them all at once, or one-at-a-time as you complete them, as you prefer. The individual labs are graded in order, starting with "a", which must be fully correct before "b" is graded, and so on. This assignment is worth 50-points, to be awarded after all labs are successfully completed. Use the "Submit your SP2015 work" link on the class website to post your file for this lab, using the **lab6** folder provided.

Lab 6a [`Time.h and Time.cpp`]
Write the class as presented in figures 10.17 and 10.18 of our Deitel textbook, with modifications per the universal requirements listed in lab 1a, and the grading policy. Name the files **Time.h** and **Time.cpp**, and fully test them with your own test CPP. Name the functions exactly as Dietel does.

Note that the class has functions that return self references. Your testing should include "cascading" calls that check to see if the self-referencing actually works right. It should compile without error, and give the expected results. For eample, `t.setHour(0).setMinute(0);` -- that's "cascading".

*Post **Time.h** and **Time.cpp** to the COMSC server for credit.*

Lab 6b Using Dynamic Memory Allocation [`ShortestRoute2.cpp`]
In this exercise, you will build and test the second of the three classes needed to solve the shortest-route problem. Copy **ShortestRoute1.cpp** from your **lab5** folder into your **lab6** folder, and rename it as **ShortestRoute2.cpp**.

For this assignment, add a class named **Route**, that represents a route between two non-adjacent cities in a network. The route is to consist of multiple connecting `Legs`. Here are the specifications for the `Route` class:

1. MEMBER DATA:
Create a private member variable for saving a dynamically-sized array of const `Leg` pointers. The data type should be `const Leg**`. You will have to include some way of marking the end of the array, by using an extra `int` to track the number of `Legs` in the array.

NOTE. If you use any "static" members for either the Leg or Route classes, you are not doing this assignment correctly!

2. TWO CONSTRUCTORS:
Include two **`public`** *constructors* -- (1) one to create a simple route consisting of only one leg, and (2) another to create a new route by adding a leg to the end of an existing route. The first constructor's

parameter list should include a `Leg` object only. The parameter should be *const references*.

The second should include (1) a `Route` object and (2) a `Leg` object, whose *starting city matches the ending city* of the `Route` object. All parameters should be *const references*. (At this time you do *not* need to enforce the matching of cities in the second constructor. We will add this next week by using `if` statements and throwing "exceptions".)

We will use the second constructor to build new routes from existing ones by adding one more leg to the end. The first constructor will be used as the first step in building a route, by adding the first leg.

The constructors should (1) create the `Leg` pointer array of the size needed to contain pointers to these `Legs`, (2) populate the `Leg` pointer array with the memory addresses of the `Leg` objects that form the route, and (3) have some way of tracking the number of `Leg` pointers in the array.

3. ONE MORE CONSTRUCTOR:
This class uses dynamic memory, so include a copy constructor. It is always a good idea to have a copy constructor when the "`new`" operator is used in the class. Without it, you risk having memory errors when the "`delete`" operator is used in the destructor.

4. A DESTRUCTOR:
Since there is dynamically-allocated memory for the class (in the array of Leg pointers), you need to include a destructor to deallocate the memory, using the `delete []` operator.

5. A FRIENDLY OUTPUT FUNCTION:
For output, include a void *friend* **`outputRoute`** function. It should output a textual description of the route, including understandable labeling and summarizing all of its member variables. List each city in the route *once only* and output the only the *total* distance. (For example, "San Francisco to Sacramento to Reno to Salt Lake City to Denver to Omaha, 1490 miles"). The prototype should be as follows: `void outputRoute(ostream&, const Route&)`, where the first parameter is to be used to pass `cout` -- name it `out` in the function. By including the **`outputRoute`** function as a friend of your class, you can send it to `cout` as follows:

```
Leg a("San Francisco", "Sacramento", 70.0);
Leg b("Sacramento", "Reno", 120.0);
...

outputLeg(cout, a);
...

Route ra(a); // route from San Francisco to Sacramento
Route rb(ra, b); // route from San Francisco to Reno
...

outputRoute(cout, ra);
...
```

In `outputRoute`, traverse the route's `Leg` pointer array in order to list all of the cities in the route. For this reason, the function should be declared as a friend in both the `Leg` class and the `Route` class. This may present a problem, because you will have to refer to the `Route` class in the `Leg` class definition (presuming that you define `Leg` first). You'll have to find a way to solve this...

6. TESTING THE CLASSES:
Include a **main** function to test the class. In **main**, instantiate at least 5 objects of the `Leg` class. You may use either (1) 5 separate `Leg` variable declarations, or (2) one declaration of an array of `Leg` objects. Name the cities and set their distances as you wish (different from the above example), so that when put together, they represent a route -- so the city name at the end of a leg must be the same as the starting city of the next leg. Build at least 5 corresponding objects of the `Route` class, starting with a one-leg object, and building on it by adding one leg at a time.

In main, create ALL Leg objects first, then output them, then create ALL Route objects, then create and output the Route object copy (see below), then output all of the Route objects.

7. TESTING THE ROUTE CLASS COPY CONSTRUCTOR:
Between the code block that creates the Route objects and the code block that outputs them, add a code block like this, INCLUDING the local scope curly-brace container:

```
{ // start local scope
  const Route rCopy = ra;
  outputRoute(cout, rCopy);
} // end local scope
```

This will test the copy constructor. If you include the above line, *without* a copy constructor, you WILL get a memory error, or garbage results. (Why? Think about it...) Only if your program is very simple is there a chance that you could be spared this error.

NOTE: Do not do so, but if you were to include in **main** an assignment statement like this: **`rCopy = rb;`** , you would invite a memory error, or garbage results. (Why? Think about it...) We will solve this eventually by *overloading the assignment operator*.

NOTE: Output for this program should include ALL Leg and Route objects.

Do *not* use more than one source file -- that means no **.h** files, and just a single **.cpp**. *Post **ShortestRoute2.cpp** to the COMSC server for credit.* Save this file, because modifications will be made to this program in lab 7 and the final project. The ShortestRoute series is the subject of three (3) lab assignments, and the final project.

How to pause a console program: *show / hide*

GRADING POLICY *show/hide*