Assignment 3: Distance Class

Assigned: October 3, 2016
Due: October 17, 2016, 11:59:59pm

Purpose

The purpose of this assignment is to provide you with basic operator overloading experience. You will also gain more familiarity with comparing objects and handling conversions of internal object representations.

Project Task

You are to write a class called `Distance` using filenames `distance.h` and `distance.cpp`. This class will handle the storing and manipulating of quantifies of distance in terms of miles, yards, feet, and inches. The main interface will require you to overload some basic arithmetic, comparison, and I/O operators. These operations must work even for large distances and should not overflow the capacity of the storage variables (int) unless the actual number of miles is very close to the upper limit of int storage (~2 billion for 32bit integers).

The Distance class must allow for storage of a non-negative quantity of distance, that is all values should be non-negative. The data should always be maintained in a simplified form. Example: 14 inches should be expressed as 1 foot, 2 inches. There are 12 inches in a foot, 3 feet in a yard, and 1760 yards in a mile. The only limits on miles are the ones imposed by the hardware. All member data must be private.

The class' public interface should work exactly as specified regardless of what program might be using Distance objects. You can use whatever private functions you need, but the public interface needs to include the methods listed below.

Make sure your implementation works with `g++` on `linprog.cs.fsu.edu` before submitting. You should also add your own tests to the provided driver to fully test the functionality of your implementation.

Documentation

In addition to the project implementation, you will also need to provide a `README` file. Here you will need to document at least 5 errors you encountered and how you solved them. The errors can be some combination of compile errors, linker errors, runtime, and runtime errors. However, you can not use the same errors you used for the first two assignments. The format needs to be a numbered list where each item describes the error type, the error description, and the steps taken to fix the error. For the description, you can just describe the g++ message for compile/linker errors or the crash / wrong output for runtime errors.

You can also include documentation for places where your implementation is unfinished. If you document buggy or unfinished code and describe what the problem is and and general approach to fixing it, then point deductions will be less severe. This shows me that you are aware of problems in

your code and that you at least have an idea of how to fix them. This is completely optional, however, it can only help you as I'm not going to change my test code based on your documentation. That is, if you report an error I would have otherwise missed, you won't lose any points for it.

Part 1: Constructors

- The class should have a default constructor (no parameters), which should initialize the object so that it represents the distance 0.
- The class should also have a constructor with a single integer parameter, which represents a quantity of inches – this should be translated into the appropriate notation for a Distance object. Note that this constructor with a single parameter will be a conversion constructor that allows automatic type conversions from int to Distance. If the parameter is negative, default the Distance object to represent 0.
- The class should also have a constructor that takes 4 parameters representing the miles, yards, feet, and inches to use for initializing the object. If any of the provided values are negative, default the Distance object to represent 0. If any of the provided values are too high (but all non-negative), simplify the object to the appropriate representation).

Examples:

```
Distance t;              // 0 miles,   0 yards, 0 feet,  0 inches
Distance s(1234);        // 0 miles,  34 yards, 0 feet, 10 inches
Distance r(-123);        // 0 miles,   0 yards, 0 feet,  0 inches
Distance d(1234567);     //19 miles, 853 yards, 1 feet,  7 inches
t = 4321;                // 0 miles, 120 yards, 0 feet,  1 inch
Distance x(1, 3, 2, 7); // 1 mile,    3 yards, 2 feet,  7 inches
Distance y(2,-4, 6, 8); // 0 miles,   0 yards, 0 feet,  0 inches
Distance z(3, 5, 7, 9); // 3 miles,   7 yards, 1 feet,  9 inches
```

Part 2: Insertion Operator

Create an overload of the insertion operator << for output of Distance objects. A distance object should be printed in the format: (MILESm YARDSy FEET' INCHES") – where MILES is the number of miles, YARDS is the number of yards, FEET is the number of feet, and INCHES is the number of inches. Note that the characters 'm' and 'y' should appear after the first two values respectively, and the markers ' and " are the standard notations for feet and inches. The whole output should be inside a set of parentheses. Also, only print the first three values if they are non-zero, but always print inches.

Examples:

```
(2m 13y 2' 9”) means  2 miles,  13 yards, 2 feet,  9 inches
(89m 175y 4”)  means 89 miles, 175 yards, 0 feet,  4 inches
(2' 10”)       means  0 miles,   0 yards, 2 feet, 10 inches
(0”)           means a distance of 0
```

Part 3: Extraction Operator

Create an overload of the extraction operator >> for reading Distance objects from an input stream. The format for the input of a Distance object is MILES,YARDS,FEET,INCHES where the user is to type

the values of a comma-separated list. You may assume that keyboard input will always be entered in this format (i.e. 4 integers separated by commas). This operator will need to do some error checking as well. If any of the input values are negative, this is an illegal Distance quantity, and the entire object should default to the value 0. If any of the values are over the allowable limit (i.e. not in simplified form), then this function should adjust the Distance object so that it is in simplified form.

Examples

- `13,5,2,8   // translates to (13m 5y 2' 8”)`
- `0,0,4,13   // translates to (1y 2' 1”)`

Part 4: Comparison Operators

Create overloads for all 6 of the comparison operators (`<  >  <=  >=  ==  !=`). Each of these operations should test two objects of type Distance and return `true` or `false`. You are testing the Distance objects for order and/or equality based on whether one quantity of distance is more than (less than, equal to, etc) the other.

Part 5: Addition Operators

Create overloads for the + operator and the - operator to allow addition and subtraction of two quantities of distance. Results should always be returned in simplified form. For subtraction, if the first quantity of distance is less than the second, return the Distance object (0”) instead

Examples:

- `(3m 7y 1' 9”) + (2m 6y 8”) = (5m 13y 2' 5”)`
- `(3m 7y 1' 9”) - (2m 6y 8”) = (1m 1y 1' 1”)`
- `(1m 6y 9”) + (2m 5y 2' 7”) = (3m 12y 4”)`
- `(1m 6y 9”) - (2m 5y 2' 7”) = (0”)`

Part 6: Multiplication Operator

Create an overload for the * operator, to allow a Distance object to be multiplied with an integer multiplier. The result as usual, should be expressed in simplified format.

Examples:

- `Distance d1(1,500,2,5); //(1m 500y 2' 5”`
- `cout << d1 * 3;         //(3m 1502y 1' 3”)`
- `cout << d1 * 5;         //(6m 744y 1”`

Part 7: Increment Operators

Create overloads for the increment and decrement operators ++ and `--`. You need to handle both the pre- and post- forms (i.e. pre-increment, post-increment, pre-decrement, post-decrement). These operators should have their usual meaning – increment will add 1 inch to the Distance value, decrement will subtract 1 inch. If the distance object is already at 0, then decrement doesn't change it (i.e. you still never will have a negative distance value).

Examples:

- `Distance d1(2,10,2,10);`
- `Distance d2(5,51,1,1);`
- `cout << d1++; //prints (2m 10y 2' 10”), d1 is (2m 10y 2' 11”)`
- `cout << ++d1; //prints (2m 11y 0”), d1 is now (2m 11y 0”)`
- `cout << d2--; //prints (5m 51y 1' 1”), d2 is now (5m 51y 1' 0”)`
- `cout << --d2; //prints (5m 51y 11”), d2 is now (5m 51y 11”)`

General Requirements

- As usual, no global variables
- All member data of the Distance class must be private
- Use the `const` qualifier whenever appropriate
- The only library that may be used is `iostream`
- Since the only output involved with your class will be in the << overload (and commands to inke it will come from some main program or other module), your output should match mine **exactly** when running test programs.

Testing

You are encouraged to create your own test programs(s), but there is a sample driver to get you started. As usual, this does not provide a comprehensive set of tests, but instead should be used to get an idea of the types of calls that can be made.

Submitting

- Remove all binary files, executables, and driver code
- Use the tar utility to archive your `distance.h`, `distance.cpp`, and `README` files
- Name the tar file according to the scheme: a3_<*last_name*>_<*first_name*>.tar
- Submit the tar file to the appropriate blackboard assignment link