Due Dec 20th Midnight

Final Lab Project

You will have to build multiple classes for this project. You are free to send me a proposal by the final lab class if you would like to do something that meets the requirements but is more individualized. The basic project involves two classes: A class derived from the `Measurement` class below and a List class.

Measurement class

Your class will be derived from the `Measurement` class. You can take the `Timer` struct from the Marathon Times lab and convert it into a class that is derived from the `Measurement` defined below. Or you can also choose to use a different class such as `Date` class or a `RealMeasurement` that uses yard, feet, inches or meters, centimeters, millimeters instead of the `Timer` class. The `Measurement` interface defines the following pure virtual functions: (all pure virtual functions must be overwritten)

```
class Measurement
{

  public:
   virtual int get_value() const = 0;
   // returns Measurement in terms of the smallest unit
   // Use this function to compare, increment and decrement.

   virtual void set_value(int value) = 0;
   // A function that sets the Measurement to the given value

   virtual int increment(int amount) = 0;
   // increments the value by a given amount, returns changed value

   virtual int decrement(int amount) = 0;
   // decreases the value by amount but does not let it go negative

   virtual string as_string() const = 0;
   // A function that creates a string representing the Measurement.
   // It should have a specific number of digits per part: mm/dd/yyyy
   // for date

   virtual int compare(const Measurement& measure) {
       return get_value() - measure.get_value();
   };
   // returns < 0 if invoking object is less than measure
   // returns > 0 if invoking object is greater than measure
   // return 0 if they are equal

};
```

Your ***derived*** Measurement class should implement the following operators (they can use your specific class name such as `Timer` because they use operator overloading which does compile time polymorphism).

- **`Timer operator +(const Timer& t, int amount);`**
  Your function can call the `increment(int amount)` function to do the math then use `set_value()` to set a temporary Measurement object that is returned. Define the function with the int parameter first as well, if you want `sum = 9 + measurement`; to work.

- **`Timer operator -(const Timer& t, int amount);`**
  Your function can call the `decrement(int amount)` function to do the math then use `set_value()` to set a temporary Measurement object that is returned.

- **`istream& operator >>(istream& ins, Timer& timer);`**
  Input an Instance of Measurement (convert your `readTimer` function) from an input stream. Validate that the input conforms to the specific format used by `to_string()` above. If it is not valid, use the `ins.setstate(ios::failbit)` to say it's an error.

- **`ostream& operator <<(ostream& outs, const Timer& timer);`**
  Output an Instance of Measurement (convert your `printTimer` function) from an output stream. Use your `to_string()` function to make it easier.

- Implement the **`bool operators ==, !=, <, >`**.
  Use `get_value()or compare(const Measurement& measure)` to compare two Measurements.

- Implement **all** accessors. For Timer, there should `be get_hours() , get_minutes(), get_seconds().` All are const member functions.

- *EXTRA CREDIT: Implement `operator ++()` and `operator --()` to return a modified Measurement. Up to 2% each.*

TimerList class

Define a class that holds a dynamic array of your `Measurement` class. The array can be an array of `Measurement*` or of your objects. You should have two other data members as part of your list class, a `size` and a `capacity`. The `size` is the number of objects in the list and the `capacity` is the size of the array.

Remember that to dynamically allocate an array, we use the `new` operator. The easier solution is an array of your object type. However, it is more efficient to implement an array of `Measurement*`. For the easier solution of a `TimerList`, allocate the array of `Timer` and keep a `Timer* list` member in your class. For an array of `Measurement*`, you'll want a `Measurement** list` member.

You must implement the following in your `TimerList` class:

- Implement ***the Big Three*** for Memory Management: because of dynamic allocation in the class, we need a destructor when the object goes out of scope, it needs to be deleted. The rest of the Big Three flows from the destructor. Therefore, we need a copy constructor and a copy assignment operator.

- **`void append(Measurement& measure);`**
  A function to add a measurement to the end of the list. It should first validate that there is room for another element. If there is not, create a bigger array by adding 10 to the current capacity and dynamically allocating it into a temporary variable. Copy the values from the old array to the new array and delete the old array. Then let it fall through and assign the object to the last index.

- **`Measurement& operator[](int index);`**
- **`Measurement& at(int index);`**
  Both of these functions are the same. Implement one and then call it from the other. Return the object at the index in the internal list.

- **`bool isEmpty() const`**
  Return is the size is greater than 0.

- **`void empty();`**
  Set the size to 0.

Extra credit: Create a template class for the TimerList class. In this case, only provide a header file with all of the template definitions. Up to 5%

Test Driver

You are responsible for writing a test driver. Notice the programs that I have assigned all semester. Most of them have allowed you to test nearly every function that you wrote in your classes. Do something similar to the Marathon Times lab or Rational Lab or ATM 2000 Lab with a loop that allows you to select different functions and combine them in different ways.

Consider a loop with strings that indicate a function should run. For instance, >> could mean you should read a new Timer, << could mean output the current Timer. ++ increments the Timer, += means you have to read an integer to use the operator. If using strings as commands, just remember that switch statements only work with individual characters and integers.

Testing all of the functions in the list class may be a little harder, particularly the big three. However, you can test them at the end of the driver by declaring new TimerList objects and using assignment and copy construction at that point.

If you need help,come ask.

Implementation Files Needed

Your classes should be stored in separate header(with proper guards) and source files: e.g.

`Measurement.h` – contains the Measurement interface declaration. Make sure this has guards `#pragma once` or `#ifndef, #define` and `#endif`. No Measurement.cpp is needed!

`Timer.h` – contains the `Timer` class declaration derived from `Measurement`. Make sure this has guards `#pragma once` or `#ifndef, #define` and `#endif`.

`Timer.cpp` - contains implementation of `Timer` functions and operators

`TimerList.h` - contains the `TimeList` class declaration, it must use dynamic allocation of an array of `Timer` instances or pointers and it must implement the Big Three. Make sure this has guards `#pragma once` or `#ifndef, #define` and `#endif`.

Extra credit: this class may use class templates for a higher grade. If using templates, there is no need for a TimerList.cpp file.

`TimerList.cpp` - contains the `TimeList` class definition, it must use dynamic allocation of an array of `Timer` instances or pointers and it must implement the Big Three.

`timer_test.cpp` – a test program that reads a bunch of `Timer` objects, tests the different functions and operators and adds the Timer to a `TimerList`. I will try and provide a model program but the Rational Lab test program is a good model as is the ATM 2000 Lab.

`test_data.txt` - contains data for a test run of your program. If you don't use File I/O in the program, you can still use test_data.txt in linux this way: (redirect a file to your program)

```
storm> ./timer_test < test_data.txt
```

Submitting the Project and Rubric

Submit all of your files including your 2 header files, 3 source files and a `test_data.txt` file. The test cases can only build and run your program. You are responsible for writing a test driver for your classes (see Test Driver above).

Having the correct files and seeing them build is worth 10%. If your test program runs, you will get an additional 10%. Your test driver quality is worth 10%.

If you implement all requirements of the `Timer` class (or your idea), it is worth 30%. The relational operators and accessors are each worth about 2%, the others are worth about 5% each.

If you implement all of the requirements in the `TimerList` class, it is worth about 30%. The bulk is in the Big Three implementations.

Documentation and style are worth 10%.