CSC 402/502, Spring 2016 **Project 4**
Due Date: As posted in Blackboard **Instructor: Tim McCord**

Overview

This is the second of two projects that compare two C++ programming techniques: OO (using inheritance) and templates (generic programming).
We know how to create and manipulate vectors of ints, doubles, and other primitives. As in Project 3, we want to do the same with vectors of "wrapper" objects using the second of the two approaches mentioned above (generic programming).

The key to our *class template* for Comparable wrapper objects is no longer a *compare* member function. Essentially, we don't need it. Instead, we can implement the required relational operators directly in terms of the wrapped value which is of type T. [Think about the assumptions we must make to do this as they will need to be documented block in your Comparable class template.]

The problem we are trying to solve is the same as in Project 3: how to create a framework by which several types of wrapper objects, each wrapping a different primitive (or even a simple object) type, can be "forced" to behave according to certain rules.

In this project, we will create a template class called *Comparable* defined in a file *Comparable.h*. The **partial** definition is as follows:

```
template <typename T>
class Comparable
{
  /* You will document your assumptions about T here:
  */
  public:
    // There is no suitable built-in type such as int, double, etc.,
    //   to serve as the default parameter, but we can assume
    //   something and create a default (no-arg) constructor.
    //   Document that assumption along with others above.
    Comparable( const T & initialValue = T() )
      :  value( initialValue )    { }

    // Implement the 6 relational operators: <, >, ==, !=, <=, and >=
    //   as member functions without using any other member function
```

```
    // print is again used to assist with the external << operator
    void print( ostream & out= cout ) const
    { out << value ; }

private:
  T value ;
} ;
```

This is the only *class* to be defined in the Comparable.h file. However, it needs one more thing to be a complete implementation. The textbook shows a technique for defining external functions such as the overloaded << operator wherein each generated class used in the program is anticipated by defining a function with a specific signature. The disadvantage it that it limits the types we can wrap. For example, if we define the following two specific overloads, we could only use those two types of Comparable objects with the << operator:

```
ostream & operator<< ( ostream & out, const Comparable<int> & i ) ;
ostream & operator<< ( ostream & out, const Comparable<char> & d ) ;
```

We know that other primitive types such as doubles ought to be compatible with our Comparable template class. Instead of providing specific versions of the << operator, you will provide a template version of the << operator (a function) which again assumes something about Comparable<T> objects. You will need to document any assumptions in the function template just as you did in the class template.

Finally, we will determine if the Integer and Character objects created in Project 3 are good candidates to act as the internal values for Comparable objects. To do that, add your version of IComparable.h to your project. [If you have not yet completed Project 3 when you start on Project 4, obviously, you would need to comment out the `#include "IComparable.h"` in the test driver and any code that depends on IComparable-derived Integer and Character types.]

File Layout

1) Submit two files: Comparable.h, which will contain the definition and implementation of the Comparable class and the << overload for Comparable objects; and IComparable.h, which is the deliverable from Project 3 .

2) testComparable.cpp will be provided and will contain a main function that creates and manipulates objects of type Comparable<int>, Comparable<char>, and other built-in types. Similar to Project 3, you will see vectors of Comparable<int> and Comparable<char> that are processed using three external template functions which are provided for you:

```
// forward declarations
template <typename T> void print( const vector<T> & v )  ;
template <typename T> bool isSorted( const vector<T> & v ) ;
template <typename T> void sort( vector<T> & v ) ;
```

HINT: As you begin to test your class, comment out portions of the test driver that you have not yet implemented using the block comment symbols (/* and */) .

Standing Requirements for all Projects

SR1. Provide a standard comment block at the top of each file documenting the file name and other details listed below. Failure to provide the information will result in a 5-point deduction per file.

```
/*
  File Name: <filename>.cpp or <filename>.h
  Author: your first and last name
  Course: CSC 402 (502 for MSCS students)
  Date: date completed
*/
```

SR2. All code you submit should compile without errors, and should be free of runtime errors for typical or expected input values. For instance, if a function is written to expect an integer from 1 to 1000, any number in that range should be handled gracefully. Each syntax or runtime error will result in a 5-point deduction with a maximum of 25 points total.

SR3. Observe C++ stylistic conventions demonstrated in class or in the textbook and use standard C++ constructs, as opposed to C language features that may be supported in C++ due to its origin. For example, prefer *cout* to *printf*. Also, be consistent in your approach to indentation, block structure, etc. Gross violations may result in a 5-point deduction.

SR4. This requirement, re: test case coverage is not applicable for this project, since I provided the test driver.

Project-specific Requirements

R1. Use standard header guards or #pragma once directives in all header files.

R2. Complete the implementation of the Compare template class, including the missing relational operator overloads, provided in one file, Comparable.h. Include the requested comment block listing your assumptions. [With this and most templates, in general, there are assumptions, so don't leave it blank.]

R3. Provide the << operator overload as a template function and include it in the Comparable.h file. Be sure to include the requested comment block listing your assumptions, or state positively that you make no assumptions.

R4. As in Project 3, determine if the Comparable class requires an explicit implementation for the copy constructor, the copy assignment (=) operator, or the destructor. Determine if any or all of the three needs to be explicitly implemented based on the test results and implement them **only if necessary**. [Notice that one set of test cases uses the C++ *string* object as the wrapped "value," but you have to test to see whether the compiler can or cannot construct a Comparable<string> object.]

R5. Add your version of IComparable.h from Project 3 into this project and make sure the test cases identified as "Tests with Project 3 IComparables:" run properly. [If you were unable to complete Project 3, my version will be made available after the late period has expired for Project 3.]

R6. Upload the 2 files, Comparable.h and IComparable.h, to the BlackBoard drop box associated with Project 4. [For this project, I *do not want* a zip archive.] **DO NOT submit entire VS projects, additional files, etc.**

CSC 502 Additional Requirement/10 points Extra Credit for 402

R7. Enable the code that is surrounded by preprocessor directives in the main method by uncommenting the line that reads `#define CSC502` in main. That will control which sort function is invoked:

```
ifndef CSC502
  cout << "Using local sort" << endl ;
  sort(charVector) ;
#else
  cout << "Using std::sort" << endl ;
  std::sort(charVector.begin(), charVector.end()) ;
#endif
```

a) Test to see if std::sort exactly as I have written it will sort a vector of your Comparable<T> objects, and make the fixes in your code to make it work if not. (5 pts.)

b) Explain in a comment block at the bottom of the Comparable.h file why the addition of a functor is not required to allow std::sort to function properly. (5 pts.)