CSC 402/502, Spring 2016 **Project 3**

Due Date: As posted in Blackboard **Instructor: Tim McCord**

Overview

This project is the first of two projects that compare two major C++ programming techniques: OO (using inheritance and polymorphism) vs. templates (generic programming).
We know how to create and manipulate vectors of ints, doubles, and other primitives. Now we want to do the same with vectors of "wrapper" objects using the first of the two approaches mentioned above (OO).
The key to our wrapper objects will be a *compare* member function: `obj1.compare(obj2)` will return -1 if obj1 < obj2, 0 if obj1 == obj2, and 1 if obj1 > obj2. The problem we are trying to solve is 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, demonstrating the OO approach, we will create an abstract base class called *IComparable* defined in a file IComparable.h. The ***partial*** definition is as follows:

```
class IComparable
{
  public:

   virtual const IComparable & operator= (const IComparable & rhs) {
      return *this;   // dummy implementation
    }

    virtual int  compare( const IComparable & icmp ) const =0 ;

    virtual void print( ostream & out= cout ) const =0 ;

    virtual bool operator<  ( const IComparable & rhs ) const {
      // Implement this using compare()
    }

    // Implement the remaining 5 relational operators
    // (>, ==, !=, <=, and >=)  similarly
    // using the compare member function
```

```
} ;
```

You will complete the definition of *IComparable* and create two derived classes, *Integer* and *Character* that act as wrappers for int and char primitives, respectively. Place the declaration and implementation for these two classes in the same header file as the abstract class, IComparable.h. In that same file, you will also provide an external overloaded << operator for IComparable objects.

SUPPLEMENT TO THE TEXTBOOK

Please read this section carefully.

In the textbook and code examples, there are no examples of a pure virtual function that takes an argument of an abstract class type. Notice that the **`compare()`** function in IComparable takes a const Icomparable reference as an argument. To instantiate a derived class, we have to override the function, but there is a type mismatch problem. To illustrate, we might first try to define compare() in the Integer class with an Integer reference as the argument (and any arbitrary implementation) :

```
int compare( const Integer & rhs ) const { return 0 ;}
```

You will get a compilation error such as this:

```
'Integer' : cannot instantiate abstract class
1>     due to following members:
1>     'int IComparable::compare(const IComparable &) const' : is abstract
```

The issue is that each concrete derived class must provide an implementation (override) for the base class member function since it is defined as a pure virtual function. By not providing a concrete implementation, Integer inherits the abstract function definition and thus Integer is also abstract. Yet, we have to be able to instantiate Integer and Character objects, which in turn implement the IComparable interface.

The solution can be illustrated with the theoretical implementation of a base class function called add that takes a const IComparable reference as an argument:

```
// Abstract base class declaration:
// Adds the value stored in rhs to this object's value
virtual void add( const IComparable & rhs ) =0 ;

// Derived class declaration and implementation:
void add( const IComparable & rhs ) {
```

```
    value += dynamic_cast<const Integer &>(rhs).value ;
  }
```

Notice that we dynamically cast the IComparable reference to a reference of the type of our derived class and then access the value data member via the dynamically-cast object.

File Layout

1) Submit one **and only one file**, IComparable.h, which will contain the definitions and implementations of IComparable, Integer, Character, and the << overload for IComparable objects. [In a real-world application, the IComparable interface definition would likely exist alone in one header file, while Integer and Character might be defined in their own header files, and the implementations for one or all of the classes would typically be found in .cpp files matching the names of the classes. Collapsing all into one file is simply for convenience.]

2) testIComparable.cpp will be provided with the assignment and will contain a main function that creates and manipulates objects of type Integer and Character, as well as IComparable pointers, and vectors of IComparable pointers which will be populated with Integers and Characters. I have also provided three external functions that take vectors of pointers to IComparable objects are used to view, sort, and check to see if they are sorted:

```
// forward declarations
void print( const vector<IComparable *>  & v ) ;
void sort( vector<IComparable *> & v ) ;
bool isSorted( const vector<IComparable *> & v ) ;
```

HINT: As you begin to test your classes, you will need to 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 the header file.

R2. Provide declarations/implementations for the derived classes Integer and Character, where Integer wraps a primitive int and Character wraps a primitive char. Note that the data members are not pointers, static const members, references, etc.—just simple int and char primitive types.

R3. Define a private data member of the appropriate type called *value* for each of the two derived classes.

R4. Provide implementations of the print() function for the Integer and Character classes that will be called by the external << operator overload. [print () will be a simple public accessor in the derived classes.]

R5. Each of the derived classes must have a single-arg constructor with a default parameter to serve as the no-arg constructor as well. For the Integer class, the default value is 0. For the Character class, use the ? character.

R6. Provide the "typical" destructor for the abstract base class.

R7. Based on the member data types, neither of the derived classes *should* require an explicit implementation for the Big Three (copy constructor, the copy assignment operator, and the destructor). However, to enable polymorphism, we need to be able to work with IComparable pointers and references. Notice that the IComparable partial definition provides a default

implementation for operator= which is not necessarily sufficient to satisfy the test cases, but it will make the entire project simpler, as opposed to trying to implement operator= as a pure virtual function. Determine if any or all of the Big Three need to be explicitly implemented in the derived classes based on the test results and implement only what is necessary to pass the test cases in the test driver.

R8. Provide declarations/implementations of the compare() function for each of the two derived classes.

R9. Provide declarations/implementations of the <, **>, ==, !=, <=,** and >= operators implemented in terms of the compare() function. Think very carefully as to whether these operators can be implemented in the base class and inherited by the derived classes.

R10. Upload only the IComparable.h file to the BlackBoard drop box associated with Project 3. **DO NOT submit entire VS projects, additional files, etc.**

CSC 502 Additional Requirement/10 points Extra Credit for 402

R11. Enable the code that is surrounded by #ifdef-#endif in the main method by uncommenting the line that reads `#define CSC502` in main.

```
#ifdef CSC502
  std::sort( charVector.begin(), charVector.end(), LessThan() );
#endif
```

Comment out the lines in main that call the local sort() function such that the vectors are unsorted before the call to std::sort. Notice that, unless we do something extra to allow it to function properly, std::sort will not sort a vector of Integer or Character objects accessed as IComparable pointers.

Yet, at the bottom of main is an example of std::sort with vector of Integer objects, and it seems to work perfectly. How do we get std::sort to handle a vector of IComparable objects accessed via pointers?

Notice the syntax of the call to std::sort in lines 157 and 180 in the test driver. The key is the reference to LessThan, which is a struct that overloads the () operator as seen in the text, slides, and code samples. LessThan's () operator returns a bool and takes two const IComparable pointers as parameters. It must be implemented without using Integer or Character types. Provide the struct in the IComparable.h file and verify that the std::sort function works as expected.