CISP 400 - Lab Assignment #9
Due Thursday, November 8, 2018 at 11:59 PM

This assignment is designed to let you work more with inheritance. Specifically, it is intended for you to work with:

- inheritance hierarchies
- overriding
- virtual functions
- dynamic casting

Overall:

We are going to be defining three classes – HoldsString, BiggerString and SmallerString.

HoldString:

The HoldsString class will be the parent class/superclass/base class for BiggerString and SmallerString. HoldsString holds, literally, two strings. The actual name you give these does not matter much, but we'll call them stringOne and stringTwo. The constructor accepts these two strings and assigns them to protected member variables.

HoldsString defines two member functions. The first, getString(), is ***virtual***, and returns a string. The second, getLength(), is not virtual, and returns an int.

HoldsString::getString() takes stringOne and stringTwo and concatenates them (HINT: use the + operator). It then returns that concatenation. It does not store this concatenation. Every time that getString() is called it does the computation again (this is, for many applications, not very efficient, but we are exploring concepts, not building a real system).

HoldsString::getLength() takes the result of getString(), and returns the length.

BiggerString:

The BiggerString class is the subclass of the HoldsString class. It will not add any private or protected member variables. However, we will override HoldString code.

BiggerString has BiggerString::getString(), with the exact same return type and parameters as HoldsString::getString(). Remember, HoldsString::getString() is marked virtual and can be replaced by overriding it. Because getString() is a virtual function, if a BiggerString object is constructed, the getString() function will use the BiggerString version of the function, regardless of what object type we may cast it into. It will also have a new member function, announceBigger().

BiggerString::getString() will take the two strings, stringOne and stringTwo, and it will return whichever string is considered larger. Remember, we can use greater-than and lesser-than comparison operators with string objects. If they are equal (the same contents in each string object), BiggerString::getString() can return either one.

BiggerString::announceBigger() will return a single string: “Bigger”. Literally just that word. The function does nothing else other than that.

SmallerString:

The SmallerString class is extremely similar to BiggerString. The following differences are relevant:

- SmallerString::getString() returns the smaller string object. Rermember, both greater-than and lesser-than operators are available for strings. If the contents are equivalent, then the function returns either one.
- There will be another member function, string SmallerString::announceSmaller(), which returns the string “Smaller”. It does nothing other than that.

Program:

The main function should declare three shared pointers, each of the HoldsString type. It should then generate three new objects, one of each type we have made (HoldsStrings, BiggerString, SmallerString).

These should each be passed to a function named outputContents(), in the order: BiggerString, SmallerString, HoldsString.

The outputContents() function must accept one parameter, which is a reference to a shared pointer, of type HoldsString. This function must do the following:

Get the output of getLength() and getString() from the object passed in the parameter. It must do this without casting.

Use dynamic_pointer_cast to determine which type it is, and store the resulting shared pointer as a local variable (if one was successfully generated). The cast pointer from this should be used to get the result from the announceBigger() or announceSmaller() function, as is appropriate depending on the true type of the class, and store it into a string, which we’ll call announceString. If the object is neither of these classes, then you should put “Concatenate” in announceString.

It should then output, each on its own line, the following values: the type of the object, the length of the string, and the result of getString().

STARTING CODE:

```
#include <iostream>
#include <memory>

using namespace std;

// Forward class declarations

class HoldsStrings;
class BiggerString;
class SmallerString;
```

```
// Function prototypes

void outputContents(shared_ptr<HoldStrings> &str);
void printSeparator();

const string BIGGER_STR_1 = "Cat", BIGGER_STR_2 = "Dog";
const string SMALLER_STR_1 = "House", SMALLER_STR_2 = "Car";
const string HOLDER_STR_1 = "Length", HOLDER_STR_2 = "Width";

int main()
{
        shared_ptr<HoldsStrings> bigger, smaller, holder;

        // Construct the three objects here.  HINT: try to
        // use make_shared!  Also remember to specify the
        // type that you are building.

        outputContents(bigger);
        printSeparator();
        outputContents(smaller);
        printSeparator();
        outputContents(holder);

        return 0;
}

void outputContents(shared_ptr<HoldStrings> &str)
{
        // You write the output logic here
}

void printSeparator()
{
        cout << "**********************" << endl;
}
```

SAMPLE OUTPUT:

I have provided sample output. Your program should provide similar output:

```
Type: Bigger
Result: Dog
Length: 3
**********************
Type: Smaller
Result: House
Length: 5
**********************
Type: Concatenate
Result: LengthWidth
Length: 11
```

NOTES AND TIPS:

- Do not do any calculations or complex computation in the constructor for any of the classes. Just store the variables.
- All string comparisons, concatenation, etc. must occur in the getString() class. Additionally, length calculations should be done in getLength(). Specifically, getLength() as is written for HoldsString. Do NOT use casting with this, or any class-specific code – if you use virtual functions you ought to be able to write very brief, accurate code that will work for both the base class and the derived classes. HINT: use getString(), and remember what happens to in-class calls to member virtual functions that have been overridden. It is not difficult to write getLength() so that it will take advantage of this – in fact it might not require more than one line of code.
- Pay attention to the output. You are going to be doing some slightly unusual things in this lab, and that's part of the point.
- The derived classes do not require additional member variables, so don't worry about adding those.
- No user input is required.
- You will probably not need a fancy destructor, but remember that you will probably need to include a destructor that is set to be virtual to make sure the compiler does not complain.

DELIVERABLES:

Upload the completed .cpp file to Canvas by the due date.