COMS W3157 Advanced Programming, Lab #9 (100 points total)

Please read this assignment carefully and follow the instructions EXACTLY.

Submission:

For part1 & part2, do NOT create part1 or part2 directories. Modify or add to the skeleton code in the top level directory. Provide a Makefile that builds all requested executables when you type "make".

If you choose to do part3, which is optional and will not be graded, create part3 subdirectory, and copy over mystring.h/cpp and other files from the top level directory as needed. This way, your attempt to implement part3 will not affect your part1 & part2.

Please refer to the lab submission instruction for other requirements.

Please check your code with valgrind:

You will be heavily penalized if you have any memory error in part 2.

Part 1: Understanding object construction and destruction in C++ (50 points)

The skeleton code contains the same MyString implementation that we learned in class, with the following additions:

- Makefile defines a macro called BASIC4TRACE. If you look at mystring.cpp, you will see that the basic 4 (i.e., constructor, destructor, copy constructor, and copy assignment) output a log message to stderr when that macro is defined.

- test4.cpp was added. It is a little test program that passes a couple of objects to a function and receives an object as a return value. Here is test4.cpp with line numbers:

```
 1      // test4.cpp
 2
 3      #include "mystring.h"
 4
 5      static MyString add(MyString a, MyString b)
 6      {
 7          MyString t(" and ");
 8          return a + t + b;
 9      }
10
11      int main()
12      {
13          MyString x("one");
14          MyString y("two");
15
16          MyString z = add(x, y);
17          cout << z << endl;
18          return 0;
19      }
```

Your job is to understand the sequence of basic 4 calls during the execution of test4 program. When you build and run test4 using the Makefile provided, you will see the log output of the basic 4 that looks something like this:

```
BASIC4TRACE: (0x7f69ac0)->MyString(const char *)    [1] 13,constructor,x
BASIC4TRACE: (0x7f69ad0)->MyString(const char *)    [2] 14,constructor,y
BASIC4TRACE: (0x7f69b00)->MyString(const MyString&) [3]
BASIC4TRACE: (0x7f69af0)->MyString(const MyString&) [4]
BASIC4TRACE: (0x7f69a70)->MyString(const char *)    [5]
BASIC4TRACE: op+(const MyString&, const MyString&)  [6] entering operator+
BASIC4TRACE: (0x7f69a20)->MyString()                [7]
BASIC4TRACE: (0x7f69a80)->MyString(const MyString&) [8] 8,copy constructor,
                                                        u1 from return temp
BASIC4TRACE: (0x7f69a20)->~MyString()               [9]
BASIC4TRACE: op+(const MyString&, const MyString&)  [10] entering operator+
BASIC4TRACE: (0x7f69a20)->MyString()                [11]
BASIC4TRACE: (0x7f69a90)->MyString(const MyString&) [12]
BASIC4TRACE: (0x7f69a20)->~MyString()               [13]
BASIC4TRACE: (0x7f69b10)->MyString(const MyString&) [14]
BASIC4TRACE: (0x7f69a90)->~MyString()               [15]
BASIC4TRACE: (0x7f69a80)->~MyString()               [16]
BASIC4TRACE: (0x7f69a70)->~MyString()               [17]
BASIC4TRACE: (0x7f69ae0)->MyString(const MyString&) [18]
BASIC4TRACE: (0x7f69b10)->~MyString()               [19]
BASIC4TRACE: (0x7f69af0)->~MyString()               [20]
BASIC4TRACE: (0x7f69b00)->~MyString()               [21]
one and two                                         [22] cout << z << endl;
BASIC4TRACE: (0x7f69ae0)->~MyString()               [23]
BASIC4TRACE: (0x7f69ad0)->~MyString()               [24]
BASIC4TRACE: (0x7f69ac0)->~MyString()               [25]
```

Answer the following questions in your README.txt:

(a) For each line of the BASIC4TRACE output, write the following information.

- The line number in test4.cpp where the output is being produced.

- Which basic 4 call is producing the output line.

- The variable name (from either test4.cpp or mystring.cpp) for the object at the address in parenthesis. If the object is a unnamed temporary object, assign a name uN (i.e. u1,u2,u3,...) when the object is created and also write which expression (from test4.cpp or mystring.cpp code) the object was created from. (The line for u1 is done for you as an example.) For subsequent lines referring to the same object, you don’t have to write the expression. Just write the uN name.

As you can see, the answers for a few lines are already filled in for you as examples.

Please prefix your answers with "[N]", where [N] is the line number in the BASIC4TRACE output, as shown in the example above. You MUST follow this format in order to get credit for this part because it will be auto-graded by a script which will look for "[N]".

(b) Change the add() function in test4.cpp as follows:

```
static MyString add(const MyString& a, const MyString& b)
```

Explain the changes in the BASIC4TRACE output.

(c) The Makefile uses a compiler flag: -fno-elide-constructors. What does this flag do? (See g++ man page.) Rebuild test4 without the flag and examine the BASIC4TRACE output. Describe the changes from (b).

Part 2: Fleshing out MyString class (50 points)

For part 2, make sure you do valgrind testing. Not having any memory errors will be a big part of the grade.

(a) Implement the following operators for MyString class:

```
<, >, ==, !=, <=, >=
```

The comparison should be lexicographical (i.e., what strcmp() does). The == and != operators should evaluate to 1 if the condition is true, 0 if false. (Actually, C++ now has bool type. Feel free to use it if you'd like.)

You're welcome to implement some of them using the others (for example, you can easily implement != using ==).

Make sure that you can invoke the operators with string literal on either side. That is, both of the following expressions should be valid:

```
str == "hello"
"hello" == str
```

where str is a MyString object.

Write a test driver program, named "test5", to test your operators. (And name your source file test5.cpp.) The assert() C library function might be useful for writing a test driver. See the man page for how to use it.

In a C source file, you would #include <assert.h> in order to use assert() function. You have to do things a little differently in C++. You #include <cassert> instead.

(b) Implement += operator that appends a string given on the right-hand side to the one on the left-hand side. For example,

```
MyString s("hello");
s += " world";
```

```
    cout << s << endl;
```

will print out "hello world".

Once you have operator+=(), reimplement operator+() using +=. Using operator+=(), you can implement operator+() without accessing the data members directly. Un-friend operator+().

Write some test code that tests your operator+=() and the new version of operator+(). Add your test code to test5.cpp. Your test driver MUST include the following statements:

```
    // test op+=() and op+()

    MyString sp(" ");
    MyString period(".");
    MyString str;

    str += "This" + sp + "should" + sp
        += "work" + sp + "without"
        += sp + "any" + sp + "memory"
        += sp + "leak"
        += period;

    cout << str << endl;
```

You can test more statements if you'd like. Don't forget the valgrind testing.

Part 3: Move operations (0 points)

This part is optional and will not be graded.

In this part, we explore the move operations -- move constructor and move assignment -- which is a new addition to C++ as of C++11.

Recall from part 1 that, in certain cases, the compiler can optimize your code by eliding certain operations involving temporary objects. Returning a stack-allocated class object by value is a good example. A temporary object is copy-constructed out of the stack object before the stack object gets destructed. Furthermore, the returned temporary object is often copied again to copy-construct yet another object. (This was the case in test4.cpp: MyString z = add(x,y)). Knowing that the temporary object serves no purpose other than to carry the content of the stack object out of the function so that it can become an input to a copy constructor, the compiler can tweak the generated code so that there is no need to create a temporary object.

Unfortunately, there are cases where the compiler cannot apply such optimizations even when we are copying from a temporary object. In many cases, we would have to fully copy-construct a new object out of a temporary object, only to have the temporary object then destroy all the inner data structure that was the source of copying. It would be a lot more efficient if we had a way to transfer or "move" the internals from one object to another, instead of copying it and then destroying the original.

This is the motivation for move constructor and move assignment operations in C++11. The programmer can supply move operations, which will then be used by the compiler, instead of copy operations, when the compiler determines that it's safe to do so, like copying out of a temporary object which will soon go away, for example. If you provide only copy operations, and no move operations, the compiler will just resort to copy in all cases, unless of course the compiler can optimize them away entirely.

(3.1)

Read 4.6.2 of "A Tour of C++" by Bjarne Stroustrup.

(3.2)

Implement move constructor and move assignment for MyString class. Make sure that all you test drivers work correctly and valgrind-clean.

For full C++11 support, use Clang 3.3 or later with "-std=c++11" option:

```
clang++  -std=c++11  program.cpp
```

Depending on your platform, you may have to swap in Clang's own C++ standard library libc++ (as opposed to gcc's libstdc++) like this:

```
clang++  -std=c++11  -stdlib=libc++  program.cpp
```

GCC 4.7 or later supports C++11 as well, albeit not as completely as Clang:

```
g++  -std=c++11  program.cpp
```

Earlier versions GCC have been tracking C++0x, the draft standard before it was finalized as C++11, so they might support some C++11 features:

```
g++  -std=c++0x  program.cpp
```

(3.3)

Repeat the experiment in part 1 with the move operations in place. Which copy calls are replaced with move calls? Can you see why those were safely replaced? For the copy calls not replaced with move calls, can you see why they were not replaced?

--

Good luck!