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

Please read this assignment carefully and follow the instructions
EXACTLY.

Submission:

  For this lab, do NOT create part1 or part2 directory.

  Provide a Makefile that builds all requested executables when you type
  "make".

  Otherwise please refer to the lab retrieval and submission instruction.

Checking memory errors with valgrind:

  Make sure to run your part 2 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 assignment operator) 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.

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: (0xbffff9a0)->MyString(const char *)
    BASIC4TRACE: (0xbffff998)->MyString(const char *)
    BASIC4TRACE: (0xbffff9b0)->MyString(const MyString&)
    BASIC4TRACE: (0xbffff9b8)->MyString(const MyString&)
    BASIC4TRACE: (0xbffff948)->MyString(const char *)
    BASIC4TRACE: op+(const MyString&, const MyString&)
    BASIC4TRACE: (0xbffff8f8)->MyString()
    BASIC4TRACE: (0xbffff958)->MyString(const MyString&)
    BASIC4TRACE: (0xbffff8f8)->~MyString()
    BASIC4TRACE: op+(const MyString&, const MyString&)
    BASIC4TRACE: (0xbffff8f8)->MyString()
    BASIC4TRACE: (0xbffff950)->MyString(const MyString&)
    BASIC4TRACE: (0xbffff8f8)->~MyString()
    BASIC4TRACE: (0xbffff9a8)->MyString(const MyString&)
    BASIC4TRACE: (0xbffff950)->~MyString()
    BASIC4TRACE: (0xbffff958)->~MyString()
    BASIC4TRACE: (0xbffff948)->~MyString()
    BASIC4TRACE: (0xbffff990)->MyString(const MyString&)
    BASIC4TRACE: (0xbffff9a8)->~MyString()
    BASIC4TRACE: (0xbffff9b8)->~MyString()
    BASIC4TRACE: (0xbffff9b0)->~MyString()
    one and two
    BASIC4TRACE: (0xbffff990)->~MyString()
    BASIC4TRACE: (0xbffff998)->~MyString()
    BASIC4TRACE: (0xbffff9a0)->~MyString()
```

Answer the following questions in your README:

(a) For each line of the BASIC4TRACE output, identify the expression in test4.cpp that caused that particular call. Identify the variable name if there is one, otherwise identify the action that caused the creation of a temporary object.

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

```
        static MyString add(const MyString& s1, const MyString& s2)
```

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 error 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.

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 to test your operators. 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.

Write a test driver that tests your operator+=() and the new version of operator+(). 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.

--

Good luck!