COMS W3157 Advanced Programming, Lab #10 (150 points total)

Please read this assignment carefully and follow the instructions EXACTLY.

Submission:

Please refer to the lab retrieval and submission instruction. The requirements regarding subdirectory for each part, README.txt and Makefiles remain the same as the previous labs unless specified otherwise.

Checking memory errors with valgrind

Make sure to run your code with valgrind. You will be heavily penalized if you have any memory error.

Part 1: Wrapping up legacy code (80 points)

Your job here is to write a linked list class in C++.

You're saying, "Not again... We did that in lab 3!"

I agree. That kind of grungy pointer work should not be repeated more than once in anyone's lifetime. (I sincerely hope you'll have better luck with that than I did in your future careers...)

What we'll do is to wrap our legacy code from lab 3 in a nice shiny C++ class. A user of our class will interact with the new C++ interface, not knowing nor caring that the real work is performed by the rusty old lab 3 code.

Unlike the old linked list, our new class will provide the value semantics, much like the container classes in the standard C++ library. Data items are copied into the list and held by value. In this part, our list will be limited to hold objects of only one specific data type, MyString. This limitation will be lifted in part 2, where we will templatize our list.

Included below is the skeleton of the header file, strlist.h. You are to define the missing member functions marked by "TODO". Some helpful tips are included in the comments. Please declare the member functions in the header and write the implementations in strlist.cpp file.

I gave you strlist-test.cpp, a program that tests the member functions of StrList class. You must test your implementation with this program. This program will print the following line when it's linked with a correctly implemented StrList class:

```
{ 0 1 2 3 4 5 6 7 8 9 }
```

Make sure you test your code with valgrind. Successfully running the test program with no memory error is basically your objective.

DO NOT MODIFY strlist-test.cpp IN ANY WAY.

I recommend you start this assignment by studying the test program first. Go through the test program and try to imagine how you need to implement the StrList class to satisfy what it's doing. In general, writing a test driver even before you write the main code is an excellent programming discipline.

```
/*
 * strlist.h
 */

#ifndef __STRLIST_H__
#define __STRLIST_H__

/*
   Copy the mystring.h/cpp files from lab9/solutions directory.
 */

#include "mystring.h"

/*
   Note that extern "C" is required when you are linking with an
   object file or a library compiled in C from C++.

   Use the usual -I, -L, -l flags in your Makefile to link with
   libmylist.a in your lab3/solutions directory.  Do NOT copy over
   any of the old linked list files into lab10 directory.
 */

extern "C" {
#include "mylist.h"
}

class StrList {

    public:

        // TODO: The basic 4.
        /*
           Don't worry about efficiency in this assignment.  Do what's
           the easiest.  For example, in order to append elements from
           one List to another using the C linked list API, you can
           reverse the target list, add elements using addFront, and
           then reverse it again when you're done.

           In fact, you'll have to implement many member functions
           rather inefficiency due to the deficiency of the old list
           API.
         */


        // isEmpty() function
        /*
           I'm giving away this function to show you that you'll have
           to cast away the const-ness of the data member when
           necessary.
         */
        int isEmpty() const { return isEmptyList((List *)&list); }
```

```
        // TODO: size() function
        // returns the number of nodes in the list


        // TODO: addFront() function
        // adds a string to the front of the list
        /*
           Note that in order to call the global addFront() function
           (which has the same name with the member function that
           you're writing) you have to append "::" in front, as in
           "::addFront(&list, ......)".
         */


        // TODO: popFront() function
        // Pops a string from the front of the list and returns it.
        // The result of popping from an empty list is undefined.


        // TODO: reverse() function
        // reverse the elements in the list


        // TODO: operator+=
        // The result of "sl += sl" is undefined.


        // TODO: operator+


        // TODO: operator<<
        // Prints the content of the given list in the following
        // format:
        //
        //     { one two three }
        //
        // assuming you had the three strings ("one", "two", "three")
        // in the list.


        // TODO: operator[]
        // This function takes O(n) time.
        // The result of accessing beyond the last element is undefined.


        // TODO: operator[] const
        // This function takes O(n) time.
        // The result of accessing beyond the last element is undefined.

    private:

        // This class contains the old C list structure as its single
        // data member.  Do NOT add any data member.

        struct List list;
};
```

```
#endif
```

Part 2: No more legacy code! (50 points)

In part 2, we rewrite our linked list class in the following way:

- It will be a template class called TList, so that we can use it for data objects other than strings. See stack.h that I sent out for a similar example.

  Put all your template declarations and definitions in tlist.h (see the stack example). You do not have to create the separate tlist.cpp file as suggested in the text book.

- TList will not use the legacy linked list anymore. Instead, use the "list" container from the standard C++ library.

- Implement all operators and member functions from part 1. Name the member functions exactly the same.

  For operator+(), you should implement it using operator+=() (as you did in lab 9). This way, operator+() does not have to be a "friend" of the TList template class. I had trouble with making the op+() a friend. Friend declarations for templates are tricky and it seems that g++ compiler has not completely caught up with the language standard in this regard.

Now comes the cool part. We will use the same strlist-test.cpp file to test our new TList. By "same", I mean the exact file that you used in part 1. DO NOT MODIFY strlist-test.cpp IN ANY WAY.

How do we do this? strlist-test.cpp is full of references to StrList, which we no longer have. We turn an instance of TList into the StrList by providing a new strlist.h:

```
#ifndef __STRLIST_H__
#define __STRLIST_H__

#include <string>
#include "tlist.h"
using namespace std;

typedef string MyString;

typedef TList<string> StrList;

#endif
```

Now, to anyone #including strlist.h, MyString is a synonym for string (in the standard library) and StrList is a synonym for TList<string>. And this makes strlist-test.cpp perfectly happy without strlist.cpp or mystring.cpp.

Again, you should run strlist-test without any valgrind error.

Part 3: Perhaps it’s really not a list after all... (20 points)

Take a closer look at the member functions and operators of TList from part 2. Is "list" container the right thing to use? Specifically, is there a standard container that we can use instead of list that can give us O(1) implementation of operator[] without sacrificing any other member function or operator of TList?

Identify such a container and reimplement TList using that container. (If you think there are more than one, pick the easier one for you to implement TList with.)

Test it the same way as part 2. Don’t forget valgrind testing.

--

Good luck!