Project 1: fsu::List<T>

Educational Objectives: After completing this assignment, the student should be able to accomplish the following:

- Use linked structures to implement a dynamically sized data structure
- Test dynamic data structures for functionality
- Test dynamic data structures for resource leaks
- Implement the ADT List using linked structures
- Explain the use of single and double links and the advantages of each in an implementation
- Define and implement iterators for a data structure

Operational Objectives: Supply source code implementing the template classes `List<T>` and `ListIterator<T>`. Code should be thoroughly tested for functional correctness, robustness, and memory management. The supplied code should function correctly and be free of memory leaks, and your tests should provide evidence of both. This evidence should be summarized in a test report.

Deliverables: Three files: `list.cpp`, `name.com`, `log.txt`

Procedural Requirements

1. Begin with understanding the chapters on Lists and Deques and a working knowledge of the techniques involved in creating linked structures dynamically.

2. The official development/testing/assessment environment is specified in the Course Organizer.

3. Make sure you understand the implementation plan for `List<T>` described in the references above.

4. Work within your subdirectory called `cop4530/proj1`. Keep in mind that with all assignments *it is a violation of course policy and the FSU Honor Code to give or receive help on assignments from anyone other than the course instruction staff or to copy code from any source other than those explicitly distributed in the course library.*

5. Copy the following files from the course `LIB` into your `proj1` directory:

```
LIB/tests/flist.cpp
LIB/tests/mlist.cpp
LIB/tcpp/list.h
LIB/proj1/list.cpp.partial
LIB/proj1/makefile
LIB/proj1/deliverables.sh
LIB/scripts/submit.sh
```

6. Create three more files:

    a. A source code file `list.cpp` implementing the template classes `fsu::List<T>`, `fsu::ListIterator<T>`, and

`fsu::ConstListIterator<T>` that are defined in `tcpp/list.h`. There are only a few missing implementations.

b. A text file `name.com` that is a command file for flist.cpp [ElementType = char] such that:
   i. The characters of your first name are inserted into x1 in alphabetical order, with the first letter capitalized
   ii. The characters of your last name are inserted into x2 in alphabetical order, with the first letter capitalized
   iii. Finishing with the commands to accomplish `x3 = x1; x3 += x2; x3.Display(std::cout)` results in printing your name to screen, with first and last name separated by the underscore character '_'.
   iv. Note character insert order is alphabetical, and traversal order spells your name.

   For example, the file `name.com` depicted here:

```
#
# name.com
#
# input order:     Chirs acehLr
# traversal order: Chris Lacher
#

11C
12h
12i
1a
1++
1++
1ir
12s
21a
12_
22c
22e
2a
2++
2++
2ih
21L
22r
3=1
3+=2
3d
q
```

   results in "Chris_Lacher" to screen.

c. A text file `log.txt` consisting of a log of all development activity, *including documentation for all testing*. All three files should be placed in the `proj1` directory.

7. Also in the log, keep detailed notes on procedures and results as you test your implementation `list.cpp`. Use these testing notes to create a summary testing report to conclude the file `log.txt`.

8. Turn in the files `list.cpp`, `name.com`, and `log.txt` using the script `submit.sh`.

   Warning: *Submit scripts do not work on the `program` and `linprog` servers. Use `shell.cs.fsu.edu` to submit this assignment. If you do not receive the second confirmation with the contents of your project, there has been a malfunction.*

Technical Requirements and Specifications

1. Your file `list.cpp` is a "slave" file to `list.h`. (See the chapter on Vectors for an explanation of "slave" file.)

2. Your implementation should follow the plan detailed above.

3. Much of the implementation is given in the file `list.cpp.partial`. You supply the missing implementations.

4. Your implementation of `List<T>`, `ConstListIterator<T>`, and `ListIterator<T>` should be tested for both *functionality* and *memory containment* using at least the classes `T = char` and `T = fsu::String`. Two test programs, clients of `fsu::List<T>`, are supplied. Specific instructions for testing for memory leaks are included as comment at the top of the file `mlist.cpp`. *DO NOT TEST FOR MEMORY LEAKS WITHOUT FOLLOWING THESE INSTRUCTIONS.*

5. Document all testing in your `log.txt`, which will be collected by the submit script.

Hints:

- The following files are used directly from the course library:

```
tcpp/list.h          // defines fsu::List<T> and list iterator classes
cpp/xran.h           // the fsu::xran family of random object generators
cpp/xran.cpp         // ... these are used by mlist.cpp
cpp/xranxstr.h
cpp/xranxstr.cpp
```

- The file `tcpp/list.h` is in the course library. Note that the file `list.cpp` is included into this file near the bottom but *inside the namespace* `fsu`. Therefore the code in `list.cpp` is automatically in the correct namespace.

- The file `flist.cpp` contains a typical "functionality" test program. The idea is to provide access to the entire public interface of class `List<>` and `ListIterator<>` so that you can perform operations on three distinct `List` objects and associated iterators. It is up to you to use this test program effectively.

- The file `mlist.cpp` contains a dynamic test for correct memory and pointer management in the implementation of `List<>` and `ListIterator<>`. In contrast to the functionality test, this one runs without user input. It sets up three `List<>` objects and associated iterators, much as is done in `flist`, but the operations are called randomly in a loop

that runs until Ctrl-C is entered or until the program crashes. This is a very dangerous program that will crash the entire server on which it runs if a defective implementation of `List` is tested without careful containment of the runspace for the program. *Therefore it is imperative that the precautions delineated in the documentation be followed.*

- The file `LIB/tests/fqueue.cpp` contains a functionality test for the `Queue<>` adaptor class. Run this test using `List` as a basis for ADT **queue** to be sure that your `List` implements **queue** correctly.

- A `makefile` for your project that compiles separate executables for the client programs is supplied. You can compile the two supplied test programs using this makefile by entering the command "`make all`". You can compile individual test programs or any other target in the makefile by entering "`make xxx`" where "`xxx`" is the target. For example, "`make flist.x`" creates the executable for `flist.cpp` and "`make mlist.o`" creates the object code for `mlist.cpp`.

- Be sure to follow the recommendations in Notes Chapter 1 on incremental implementation of complex classes. In particular: create the file `list.cpp` with all method headers and minimalist non-functional bodies (empty or returning a simple value of the appropriate type). This should result in no compile or link errors. Then proceed to implement the methods one at a time. Test the methods as you build them. Think about, understand, plan, and design each method before proceeding to its implementation.

- Sample executables for `flist` and `mlist` are available in `LIB/area51`.