COP4530 – Project 2

Doubly-Linked List Container

Submission:

1. Create a tar file with the required files.
2. Make an electronic submission of your tarball file on Canvas.

Due Date: Oct 10, 2019, 11:50pm for all students
Late Penalty: maximum one day at the penalty of 15%.

1. Overview

This project is one of the two designated ABET assessment assignments. You need to achieve a grade of C- for at least one of these two assignments.

The objectives of this project include the following:

- Understanding generic programming and information hiding by developing generic containers;
- Getting familiar with the concept of class template and its usage;
- Getting familiar with run-time complexity analysis of functions with different inputs; and
- Use of nested (iterator) classes. Use of namespace. Operator overloading.

2. Description of Provided Source Files

For this assignment you will implement a doubly-linked list class template `List` and its associated iterators. In the tar file `proj2src.tgz` available on Canvas, the following starter files are provided.

- `List.h:` It contains the interfaces of the doubly-linked list class template List. In particular, it contains a nested Node structure, and two nested iterators class (`iterator` and `const_iterator`). You cannot change anything in the `List.h` file.
- `test_list.cpp:` It is an example test program that will run some tests on your implementation of the doubly-linked list class template for different data types (it tests `List<int>` and `List<string>`. Don't make any changes to this driver program. However, your class will be tested with more than just this sample driver. It is recommended that you write other driver programs of your own, for more thorough testing.
- `measure_reverse.cpp:` It is an example measurement program that creates several doubly-linked lists with integers and records the time to reverse the elements in these lists. The average time in microseconds for several lists were reported in the end.
- `sample_output.txt:` This file provides the output running Makefile, test_list, and measure_reverse. Again, for thoroughness in testing your implementation, you shall run more tests of your own.

You are not supposed to include these files in your final submission. Thus any change you make to these files will not be graded. To extract these files from the tar file `proj2src.tgz`, you may use the following command **on the `linprog` machines, where your code will be tested and graded**.

- `tar zxf proj2src.tgz`

3. Requirements

A. You need to implement the member functions of the doubly-linked list class template List in a file named `List.hpp`. Note that `List.hpp` has been `#included` in the header file

`List.h` (towards the end of the file). As we have discussed in class, you should not try to compile `List.hpp` (or `List.h`) by themselves. These comprise a header that will be #included into other programs you might write.

B. You need to implement all the member functions of `List<T>`, `List<T>::iterator`, and `List<T>::const_iterator`, and non-class overloaded functions `operator==()`, `operator!=()`, and `operator<<()` included in `List.h`. The design of the `List` container follows the one presented in the textbook. It has three member variables, `theSize`, `head`, and `tail`. `theSize` records the number of elements in the list. The `head` and `tail` pointers point to the sentinel nodes. They represent the beginning and end markers. They do not store any real elements in the list. It is OK for you to use the code provided in the textbook. Your implementation will contain several more features than those in the textbook's implementation. We describe the requirements of each function in the following (this specifications may not write the function signatures in detail, please refer to the `List.h` file for the detailed function declaration).

1) `Member functions of nested const_iterator class:`

- `const_iterator():` default zero-parameter constructor. Set pointer current to NULL (nullptr for C++ 2011).
- `operator*():` returns a reference to the corresponding element in the list by calling retrieve() member function.
- `operator++(), operator++(int), operator--(), operator--(int):` prefix and postfix increment and decrement operators.
- `operator==() and operator!=():` two iterators are equal if they refer to the same element.
- `retrieve():` return a reference to the corresponding element in the list.
- `const_iterator(Node *p):` one-parameter constructor. Set pointer `current` to the given node pointer `p`.

2) `Member functions of nested iterator class:`

- `iterator():` default zero-parameter constructor.
- `operator*():` returns a reference to the corresponding element in the list by calling retrieve() member function.
- `operator++(), operator++(int), operator--(), operator--(int):` prefix and postfix increment and decrement operators.
- `iterator(Node *p):` one-parameter constructor.

3) `Basic member functions of List class template`

- `List():` Default zero-parameter constructor. Call init() to initialize list member variables.
- `List(const List &rhs):` Copy constructor. Create the new list using elements in existing list `rhs`.
- `List(List &&rhs):` move constructor.
- `List(int num, const T & val = T()):` Construct a list with num elements, all initialized with value `val`.
- `List(const_iterator start, const_iterator end):` construct a List

with elements from another list between start and end. Including the element referred to by the start iterator, but not the end iterator, that is [`start, end`).

- `List(std::initializer_list<T> iList):` construct a List with elements from the initializer list that is passed in. You can refer to the C++ reference website ((http://www.cplusplus.com/reference/initializer_list/) for its member functions: `size, begin and end.` Note that this will allow declarations like this:

  ```
  List<int> myList {2, 4, 6, 8, 10, 12, 14, 16};
  ```
- `~List():` destructor. You should properly reclaim memory (used by head and tail nodes).
- `operator=(List &rhs):` copy assignment operator
- `operator=(List &&rhs):` move assignment operator
- `operator=(std::initializer_list<T> iList)` : assign the initializer list data to be the calling object's new data. Example call:

  ```
  list1 = {1, 3, 5, 7, 9, 11, 13, 15};
  ```
- `size():` return the number of elements in the List.
- `empty():` return true if no element is in the list; otherwise, return false.
- `clear():` delete all the elements in the list
- `front() and back():` return reference to the first and last element in the list, respectively.
- `push_front() and push_back(),` insert the new object as the first and last element into the list, respectively; and their move versions.
- `pop_front() and pop_back(),` delete the first and last element in the list, respectively.
- `remove(const T & val):` delete all nodes with value equal to val from the list.
- `remove_if(PREDICATE pred):` delete all nodes for which `pred` returns true. PREDICATE is a template type, allowing a function object to be passed. (i.e. a true/false condition/function can be passed in via the template parameter).
- `print(ostream &os, char ofc = ' '):` print all elements in the list, using character ofc as the deliminator (`' '` by default) between elements in the list.
- `begin():` return iterator to the first element in the list.
- `end():` return iterator to the end marker of the list (tail).
- `insert(iterator itr, const T & val):` insert value `val` ahead of the node referred to by itr; and its move version
- `erase(iterator itr):` delete node referred to by `itr`. The return value is an iterator to the following node.
- `erase(iterator start, iterator end):` delete all nodes between `start` and `end` (including start but not end), that is, all elements in the range [start, end).
- `init():` initialize the member variables of list.

4) Special member functions of List class template

- `reverse():` reverse the order of the elements in the list. That is, the original first element becomes the last, while the original last becomes the first. For example,

  ```
  list2 = {1, 5, 3, 5, 1, 11, 5, 7}
  ```

  will be changed to be the following after calling reverse().

    `list2 = {7, 5, 11, 1, 5, 3, 5, 1}`

- `deduplicate():` traverse the list and remove all duplicates of any node, i.e., all nodes should be unique are the duplicates removed. For example, the list below will have only unique nodes after calling deduplicate().

    `list3 = {1, 5, 3, 5, 1, 11, 5, 7};`

  will be changed to a list with unique nodes.

    `list3 = {1, 5, 3, 11, 7}`

5) Non-class functions

- `operator==(const List<T> & lhs, const List<T> & rhs):` check if two lists contain the same sequence of elements. Two lists are equal if they have the same number of elements and the elements at the corresponding position are equal.
- `operator!=(const List<T> & lhs, const List<T> & rhs):` opposite of operator==().
- `operator<<(ostream & os, const List<T> & l):` print out all elements in list `l` by calling List<T>::print() function.

C. Write a `Makefile` for your project, to compile the `test_list.cpp` driver into an executable called `test_list`. Your program must be able to compile and run on the `linprog` machines.

4. Computational complexity analysis and time measurement.

Analyze the worst-case run-time complexities of the member functions `deduplicate(), and reverse()`.You need to report the input size and the execution time of your tests in your report file. Give the complexities in the form of Big-O. Name the report file containing the complexity analysis as "`analysis.txt`".

Your analysis should be performed with three different lists, e.g., lists with `100, 1000, and 10,000` random elements. Too many elements may cause your program to hang due to memory or other resource constraints. You can generate random numbers and strings using the random number generator (http://www.cplusplus.com/reference/cstdlib/rand/). You may decide a smaller range of elements in each list to create more duplicates.

Time measurement: You need to modify the measure_list.cpp program to include the measurement for both reverse() and deduplicate(). Also the number of elements need to be changed. Name the new file as `measure.cpp`. Take at least three measurements and report the average of reported time.

5. Programs and files you need to create.

You need to submit a total of three files for the completion of this assignment: `List.hpp`, `Makefile, measure.cpp` and `analysis.txt.` Create a tarball file from these four files using the following command, where `fsuid` is your FSU ID. An initial Makefile is included in the tar file. You need to modify it to compile the additional file(s).

- `tar zcf <fsuid>_proj2.tgz List.hpp Makefile analysis.txt measure.cpp`

6. General Requirements

- Document your code appropriately so that it is readable and easy to navigate

- You may use standard C++ I/O libraries, as well as class libraries like string. You need to use `initializer_list` for the operator= with a parameter initializer_list in the List class. You may NOT use other container class libraries from the STL for the implementation of doubly linked lists.
- Make sure your implementation compile and run on linprog.cs.fsu.edu with g++, using the C++11 standard compilation flag. Your Makefile should function in a similar manner to produce the driver programs, as shown in the file sample_output.txt.

7. Breakdown of points

Note that we will use more thorough tests than the test driver program, test_list.cpp. We have designed four different ABET assessments for this project.

ABET Rubric	Details of ABET Assessments
A1	Effective understanding of C++ class/function declarations, effective creation of C++ utility functions; and effective creation and use of UNIX commands and Makefile for program creation and execution.
A2	Effective understanding, design and implementation of nested classes and their member functions for smooth program solutions.
A3	Effective understanding, design and implementation of container classes and their member functions for complex program solutions.
A4	Effective understanding of problems of different complexities, design and implementation of solutions with different asymptotic complexities.
A5	Effective understanding of asymptotic notations; successful tests and run-time analysis of functions with different computational complexities.

The grade breakdown of points for these assessments is provided as follows:

ABET	Points	Requirements
A1	5	Have completed the required functions in List.hpp in a reasonably correct manner.
	5	The submitted files List.hpp can compile via a Makefile and produce a driver program for the provided test.
	5	Correct implementation and successful passing of non-class functions in the namespace cop4530, i.e., Group 5.
A2	20	Correct implementation and successful passing of tests on the member functions, i.e., Groups 1) and 2), for the nested iterator class (10 points) and the nested const_iterator class (10 points).
A3	25	Correct implementation and successful passing of member functions, i.e., Group 3, for the template List class.
A4	20	Correct implementation and successful passing of special member functions, i.e., Group 4, for the template List class.
A5	15	Have performed complexity tests on reverse() and deduplicate(), with three different lists of 100,1000, 10000 random members. Sample execution outputs are included in the file "analysis.txt". 5 points for reverse() and 10 for deduplicate().
	5	The input sizes and execution time are described and analyzed in a reasonable manner in the file "analysis.txt".

8. Miscellaneous

The first person to report any compilation or execution errors in the provided materials will be given 3% extra credit (maximum 15% per person). We have explicitly separated your code from the provided source code files. Thus Automatic plagiarism detection software will be used on all submissions – any cases detected will result in a grade of 0 for those involved.