Project 2: Doubly-Linked List Container

Due 02/15/2018

Educational Objectives: Understanding generic programming and information hiding by developing generic containers. Getting familiar with the concept of class template and its usage. Use of nested (iterator) classes. Use of namespace. Operator overloading.

Statement of Work: Implement a doubly-linked list class template List and its associated iterators

Requirements:

1. A header file List.h is provided, which 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.

2. A driver program test_list.cpp has been included. It is used to test your implementation of the doubly-linked list class template for different data types (it tests List<int> and List<string>. Similarly, you cannot change anything in the test_list.cpp file.

3. 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). You need to implement all the member functions of List<T>, List<T>::iterator, and List<T>::const_iterator, and non-class global 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. We describe the requirements of each function in the following (we may not write the function signatures in detail, please refer to the List.h file for the detailed function declaration).

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.

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.
- const_iterator(Node *p): one-parameter constructor.

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(): destructor. You should properly reclaim memory (used by head and tail nodes).
- operator=(List &rhs): copy assignment operator
- operator=(List &&rhs): move assignment operator
- 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
- 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.
- 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.
- print(ostream &os, char ofc = ' '): print all elements in the list, using character ofc as the deliminator 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.

Non-class global 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.

4. Write a makefile for your project and name your executable as proj2.x. Your program must be able to compile and run on the linprog machines.

5. Analyze the worst-case run-time complexity of the member function reverse() of the List. Give the complexity in the form of Big-O. Your analysis can be informal; however, it must be clearly understandable by others. Name the file containing the complexity analysis as "analysis.txt".

Downloads

Click here to download the tar file, which contains the following files: List.h, test_list.cpp, and proj2.x. The sample executable program proj2.x was compiled on a linprog machine. test_list.cpp is the source code for compiling proj2.x (which will need the implementation of List).

Submission

Turn in List.hpp, makefile, and analysis.txt in a tar file via the Canvas. Please do not include the List.h file. Note that you cannot change the List.h file, and we will not use your List.h even if you submit it.