Assignment 3

Ordered Double Linked List

Purpose

You will need to use C++11 support for this exercise. For Eclipse and other GCC based compilers you will need to use the `-std=c++11` or `-std=c++0x` compiler option. This is not needed if you are using Visual Studio since it does not distinguish between the various versions of C++.

In this assignment you are going to create a double linked list and an ordered double linked list.

The bulk of the work will be building the double linked list. This will be a template classes named **ListNode<DataType>** and **DoubleLinkedList<DataType>.**

You will also be building an ordered linked list template class named **OrderedDoubleLinkedList<DataType>.**

The text book creates an ordered linked list by inheriting from the double linked list class. We are going to use composition and have our ordered double linked list use the double linked list, but not inherit from it. That means you will have to create member functions to implement the public interface for the ordered linked list and turn around and, mostly, use the capabilities of the double linked list you have already created.

In addition you will be creating two additional classes, **PhoneBook** and **PhoneBookEntry**. The **PhoneBook** class will be making use of your ordered double linked list. The ordered list will be a collection of **PhoneBookEntry** objects.

First we start with the **ListNode** and **DoubleLinkedList** class.

ListNode and DoubleLinkedList classes

The **DoubleLinkedList** class needs to implement a double linked list. Your program cannot use any existing C++ collections to implement this, you need to create the code from scratch or by adapting the code in the text book. Note the interface for your class will be different from the one used in the text book.

The **ListNode** class is going to make the **DoubleLinkedList** class a `friend` class (so it can call the private next and previous member functions that update the next and previous pointers). To make this work we are going to have a forward declaration of the **DoubleLinkedList** template class. See the sample header file for details.

You can inline the **ListNode** class if you want to. Only inline trivial member functions in the **DoubleLinkedList** class (trivial being 1 or 2 lines of code).

You must implement all of the member functions shown here. You can implement additional private, and protected member functions if you need them.

Here is the header file for class you are creating. Note this is also in file **DoubleLinkedList.h** on eLearning.

```
/*
 * BaseDoubleLinkedList.h
 *
 * Implementation of a double linked list.
 *
 * We have the ability to get the first and last entries and navigate
 * through the entries in the linked list.
 *
 * There are actually two classes here:
 *
 * ListNode<DataType>
 *
 * and
 *
 * DoubleLinkedList<DataType>
 *
 */

#ifndef DOUBLELINKEDLIST_H_
#define DOUBLELINKEDLIST_H_

#include <iostream>
#include <iomanip>
#include <cstddef>

// forward declaration of the template class DoubleLinkedList
template<class DataType>
class DoubleLinkedList;

// ListNode class
template<class DataType>
class ListNode
{
      // make DoubleLinkedList<DataType> a friend class
      friend class DoubleLinkedList<DataType>;
      private:
            // contains the actual data
            DataType dataType;
            // pointer to the next item in the list or nullptr
```

```
            // if at the end
            ListNode<DataType>* pNext;
            // pointer to the previous item in the list or nullptr
            // if at the start
            ListNode<DataType>* pPrevious;
      public:
            // default constructor
            ListNode();
            // copy constructor
            ListNode(const DataType &newItem);
            // get the next node
            ListNode* next() const;
            // get the previous node
            ListNode* previous() const;
            // return the data stored in the node as a const
            const DataType& data() const;
            // return the data stored in the node
            DataType& data();
      private:
            // update the next node
            void next(ListNode *nextNode);
            // update the previous node
            void previous(ListNode *previousNode);
};

// DoubleLinkedList class
template<class DataType>
class DoubleLinkedList
{
      private:
            // number of nodes in the list. Note that std::size_t
            // is defined in header file cstddef.
            std::size_t numberNodes;
            // point to the first node or nullptr for empty list
            ListNode<DataType>* firstNode;
            // point to the last node or nullptr for empty list
            ListNode<DataType>* lastNode;
      public:
            // default constructor
            DoubleLinkedList();
            // copy constructor
            DoubleLinkedList(const DoubleLinkedList &other);
            // destructor
            virtual ~DoubleLinkedList();
            // return the number of nodes in the list
```

```
        std::size_t size() const;
        // return true if the list is empty
        bool empty() const;
        // display the contents of the list to std::cout
        void print() const;
        // dump the contends in debug format to passed in
        // ostream - usage to cout would be:
        //   list.debug(std::cout);
        void debug(std::ostream &out) const;
        // first node in the list (or nullptr for empty)
        virtual ListNode<DataType>* first() const;
        // last node in the list (or nullptr for empty)
        virtual ListNode<DataType>* last() const;
        // add an item to the front of the list
        virtual void push_front(const DataType &newItem);
        // add an item to the back of the list
        virtual void push_back(const DataType &newItem);
        // remove an item from the front of the list
        virtual void pop_front();
        // remove an item from the back of the list
        virtual void pop_back();
        // insert newItem before the existingNode
        virtual void insert_before
              (ListNode<DataType>* existingNode,
               const DataType &newItem);
        // insert newItem after the existingNode
        virtual void insert_after
              (ListNode<DataType>* existingNode,
               const DataType &newItem);
        // find the node and return the address to the node. Return
        // nullptr if not found
        virtual ListNode<DataType>* find
                              (const DataType &existingItem);
        // remove the node equal to currentItem
        virtual bool erase(const DataType &currentItem);
        // remove the node by address existingNode
        virtual bool erase(ListNode<DataType> *existingNode);
};

// your implementation code goes here

// note the code for the debug() function is included
// display a debug version of the list
template<class DataType>
void DoubleLinkedList<DataType>::debug(std::ostream &out) const
```

```
{
      const unsigned int ADDRWIDTH = 10;
      out << "START DEBUG" << std::endl;
      out << "first  =" << std::setw(ADDRWIDTH) << firstNode;
      out << ", last=" << std::setw(ADDRWIDTH) << lastNode;
      out << ", # nodes=" << size() << std::endl;
      unsigned int count = 1;

      for (auto current=firstNode; current!= nullptr;
                                        current=current->next())
      {
            out << "node " << std::setw(2) << count;
            out << "=" << std::setw(ADDRWIDTH) << current;
            out << ", next=" << std::setw(ADDRWIDTH)
                << current->next();
            out << ", previous=" << std::setw(ADDRWIDTH)
                << current->previous();
            out << ", value=" << current->data() << std::endl;
            count++;
      }
      out << "END DEBUG" << std::endl;
}

#endif /* DOUBLELINKEDLIST_H_ */
```

Note that the member functions that add to the list (**push_front**, **push_back**, **insert_before** and **insert_after** take references of type **DataType**. You need to create a **LinkNode** from this and add the **LinkNode** to the list.

If you push or insert **DataType** values that contain the same values as other list nodes you will end up with duplicate values in the list. Also note that the list items are not going to be sorted in any way. They are going to be in the order you add them.

The **erase**, **pop_front** and **pop_back** member functions will remove the **ListNode** entry from the list. Make sure you delete the **ListNode**. Do not allow any memory leaks in your program. If the list is empty after the **erase** or **pop_xxxx** functions you need to set the first and last pointers to **nullptr**.

Make sure your destructor removes all of the remaining entries from the list.

The **first** and **last** member functions need to return **nullptr** if there are no entries in the list.

The **push_front, pop_front, push_back** and **pop_back** member functions need to work if there are no entries in the list. The **push_xxxx** functions need to add the entry to the list if it is currently empty. The **pop_xxxx** functions need to just return when the list is empty.

The **insert_before** and **insert_after** member functions need to insert the item into the list when the list is empty and when the **existingNode** parameter is **nullptr**. In the case where the **existingNode** parameter has a value of **nullptr** the **insert_before** member function should add the new node to the beginning of the list, and the **insert_after** member function should add the new node to the end of the list.

The `bool erase(const DataType &currentItem)` member function should return a value of `false` if the `currentItem` is not in the list.

The copy constructor needs to make a deep copy of the list. You should be able to use other member functions to do most of the work for you.

OrderedDoubleLinkedList class

The **OrderedDoubleLinkedList** class will make use of the **DoubleLinkList** class to implement a sorted linked list. Note that the ordering will depend on you **DataType** class supporting the relational operators. This will be used to determine the order the items are being added. Some of the public member functions in the **OrderedDoubleLinkedList** class will have the same signature as those in the **DoubleLinkedList** class. Some will have similar signatures. Other member functions in the **DoubleLinkeList** will not be in the **OrderedLinkedList** (such as **push_front**, **pop_front**, **push_back**, **push_front**. **Insert_before** and **insert_after**). Since the public interface of the **OrderedDoubleLinkedList** is not a superset of the public interface of the **DoubleLinkedList** class we are NOT going to use inheritance. Instead we are going to use composition.

The **insert** member function should check and see if the entry being inserted already exists. This is done my checking to see if there is an existing entry equal to the entry being inserted. If it is already in the collection the **insert** member function should replace/update the existing item with the new values. You should not end up with duplicate entries in the list.

Here is the interface you need for the **OrderedDoubleLinkedList**. This is included in the header file **OrderedDoubleLinkedList.h** on eLearning.

```
/*
 * OrderedDoubleLinkedList.h
 *
 * Store the DataType values in sorted order. This ordering
 * is determined by the comparison operators of the DataType
 * class.
 *
 */

#ifndef ORDEREDDOUBLELINKEDLIST_H_
```

```
#define ORDEREDDOUBLELINKEDLIST_H_

#include <cstddef>
#include "DoubleLinkedList.h"

template<class DataType>
class OrderedDoubleLinkedList
{
      private:
            // we are making use of the DoubleLinkedList class
            DoubleLinkedList<DataType> list;
      public:
            // default constructor
            OrderedDoubleLinkedList();
            // destructor
            virtual ~OrderedDoubleLinkedList();
            // number of items in the list
            std::size_t size() const;
            // is the list empty (true) or does it have entries (false)
            bool empty() const;
            // print the items in the list
            void print() const;
            // display debug information on the passed in ostream
            void debug(std::ostream &out) const;
            // get first node in the list (or nullptr)
            ListNode<DataType>* first() const;
            // get last node in the list (or nullptr)
            ListNode<DataType>* last() const;
            // find the entry and return its address
            // (nullptr if not found)
            ListNode<DataType>* find(const DataType &existingItem);
            // erase the item by DataType value
            bool erase(const DataType &currentItem);
            // insert the new item into the list (in sorted order)
            // a duplicate entry will be ignored
            void insert(const DataType &newItem);
};

// Your implementation code goes here

#endif /* ORDEREDDOUBLELINKEDLIST_H_ */
```

PhoneBook and PhoneBookEntry classes

You are going to be creating a phone book using the **OrdererDoubleLinkList** class. The entries being added are going to be of type **PhoneBookEntry**. This will contain three **std::string** values for name, phone number, and e-mail address.

Here are the requirements for the **PhoneBookEntry** class.

You will need four constructors for the **PhoneBookEntry** class as follows:

```
PhoneBookEntry();
PhoneBookEntry(const std::string &name, const std::string &number);
PhoneBookEntry(const std::string &name, const std::string &number,
               const std::string &email);
PhoneBookEntry(const PhoneBookEntry &copyFrom);
```

Note that the last one above is a copy constructor.

In addition you will need the following accessors. Note that we are using more of a C++ style for these and not the getXxxxx format used in the text book:

```
std::string name() const {return sName;
std::string phoneNumber() const;
std::string email() const;
```

You can also allow updates to the phoneNumber and email values. Note you cannot change the name member data.

```
void phoneNumber(const std::string &newNumber);
void email(const std::string &newEmail);
```

Finally you need to overload the operators ==, !=, <, <=, > and >= for the **PhoneBookEntry** class.

You need to check only the name values and not all of the other values for the comparison operations. For example two PhoneBookEntry objects would be == if the name of one was equal to the name of the other, even if the phone numbers are different.

You will also need the following stand-alone function. The prototype of this needs to be in the **PhoneBookEntry.h** file (but NOT in the class declaration). You should implement it in the **PhoneBookEntry.cpp** file. This function will display the contents of the **PhoneBookEntry** on the out **ostream** object. Note that your **operator<<** function needs to return **out** when finished. See the section "Overloading the Stream Insertion (<<) and Extraction (>>) Operators" starting on page 924 of your text book.

std::ostream& operator<<(std::ostream &out, const PhoneBookEntry &entry);

Here are the requirements for the **PhoneBook** class.

The **PhoneBook** class will be using the **OrderedLinkedListClass** you created above.

The **PhoneBook** class needs to have a default constructor.

In addition it will need the following public member functions:

The insert operations will add the **PhoneBookEntry** to the collection. They are being added in sorted order, ordered by the name field. This is done by the **OrderedDoubleLinkedList** class when it uses the relational operators to determine where to insert the phone book entry. For the 2nd and 3rd versions of the insert you need to create the **PhoneBookEntry** that is to be added to the collection. These can be temporary **PhoneBookEntry** objects.

```
void insert(const PhoneBookEntry &entry);
void insert(const std::string &name, const std::string &number,
            const std::string &email);
void insert(const std::string &name, const std::string &number);
```

The **erase** operation will remove the phone book entry with the associated name from the collection. This will need to create a temporary **PhoneBookEntry** that is passed to the appropriate **OrderedDoubleLinkedList** operation.

```
bool erase(std::string name);
```

The find operation will determine if there is a phone book entry with the specified name in the collection. It will return `true` if it is found and `false` if it is not found.

```
bool find(std::string name);
```

The rest of the member functions will delegate the work to the corresponding **OrderedDoubleLinkedList** member function.

```
void print() const;
std::size_t size() const;
ListNode<PhoneBookEntry>* first() const;
ListNode<PhoneBookEntry>* last() const;
void debug(std::ostream &out) const;
```

Sample testing programs are available on eLearning in file **phoneBookTest.cpp**. You may have to comment out some of the code while you are building the various classes.

Note that you will be using the **DoubleLinkedList** and **ListNode** classes in Assignment 4.

Submit all of your header and source files to eLearning to get credit for this project.

This is going to take a while to get working, so get started right away.

You should write stubs for all of the functions as you are creating the classes. Start with the **ListNode** class and then the **DoubleLinkedList** class. Write the stubs so they compile and return a reasonable default value (**nullptr** or **false** for example). Then write the code for one member function and get it working before you go to the next member function. Try writing the constructors first (except the copy constructor, you may want to do that after you have completed the rest of the class). Then get **push_front** working. Then get **pop_front** working. Next try **push_back** and then **pop_back**. Continue to develop and test until you have the whole class working. Use the **debug()** member function to display your list and make sure the pointers are correct in both versions. Also make sure the count of items is always correct.

By doing this incremental development you will have a better idea which function is failing (because it is probably the code you just wrote). Also, make sure you test and fix a member function before you go onto the next one.

Use good formatting in your programs. Use meaningful variable names and comment your code - what do you want to remember about the code when you do assignment 4?

Let me know if you have any questions.