Chapter 8-4, Lecture notes

The ADTs List and Sorted List Revisited

1. There is an ADT list and an ADT sorted list.
   1.1 But the lists have some characteristics and operations in common.
      1.1.1 Each ADT can determine its length and say whether it is empty.
      1.1.2 You can put the commonalities into an abstract base case – and use it as a basis of other lists.
      1.1.3 For example:

```
class BasicADT  // abstract base class
{
public:
        virtual ~BasicADT();  // destructor
        virtual bool isEmpty() const = 0;
        virtual int getLenght() const = 0;
};  // end BasicADT
```

   1.2 A method would be a pure virtual method if the derived class must provide their implementations but the abstract base class does not.
   1.3 The BasicADT destructor is NOT pure and has to be implemented.

      1.3.1 The BasicADT could even be used with Stack and Queue since those ADTs also has getLength and isEmpty.
      1.3.2 We'll concentrate on List though.
      1.3.3 The following shows the ADT list using BasicADT, like so:

```
/**  @file List.h */

#include "ListNode.h"  // as specified in previous section, also specifies ListItemType

#include "ListException.h"
#include "ListIndexOutOfRangeException.h"
#include "BasicADT.h"

/** ADT list – Uses LIstNode and BasicADT */
class List : public BasicADT
{
public:
// constructors and destructor:
        List();
        List(const List& aList);
        virtual ~List();
```

```
// list operations:
        virtual bool isEmpty() const;
        virtual int getLength() const;
        virtual void insert(int index, const ListItemType& newItem)
                throw(ListIndexOutOfRangeException, ListException);
        virtual void remove(int index) throw(ListIndexOutOfRangeException);
        virtual void retrieve(int index, ListItemType& dataItem) const
                throw(ListIndexOutOfRangeException);

protected:
        void setSize(int newSize);              // sets size
        ListNode *getHead() const;              // returns head pointer
        void setHead(ListNode *newHead);  // sets head pointer

// the next two methods return the list item or next pointer of a node in the linked list
        ListItemType getNodeItem(ListNode *ptr) const;
        ListNode *getNextNode(ListNode *ptr) const;

private:
        int size;                 // number of items in the list
        ListNode *head;       // pointer to the linked list

        ListNode *find(int index) const;
}; // end List
```

2 If you do not implement a pure method in a derived class then the derived class would also be an abstract base class.

2.1 The members in private cannot be accessed by any further derived including the find method.

2.2 However, you could move the find method to protected to allow derived classes to access it.

2.3 What about using the ADT base class with Sorted List that has different names for the base methods?

```
+createSortedList()
+destroySortedList()
+sortedIsEmpty() : boolean {query}
+sortedLength() : integer {query}
+sortedInsert(in newItem : ListItemType) throw (ListException)
+sortedRemove(in anItem : ListItemType) throw (ListException)
+sortedRetrieve(in index : integer, out dataItem : ListItemType) {query}
        throw(ListIndexOutOfRangeException)
+locatePostion(in anItem : ListItemType, out isPresent : boolean) : integer {query}
```

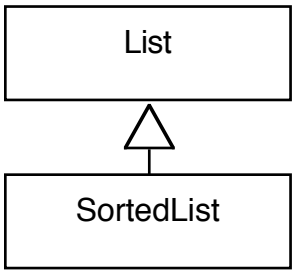
3 A sorted list IS a list. Has the same basic operations only the insert and delete would put the item in the sorted order and has the additional operation of locatePosition.
3.1 The locatePostion could be used to first locate the position to insert the new item.
3.2 Therefore the sorted list “is-a” list.
3.3 The header file looks like:

```
#include “List.h”

class SortedList : public List
{
public:
// constructors and destructor:
        SortedList();
        SortedList(const SortedList& sList)
        virtual ~SortedList();

// new operations:
        virtual void sortedInsert(const ListItemType& newItem) throw(ListException);
        virtual void sortedRemove(const ListItemType& anItem) throw(ListException);
        virtual int locatePosition(cosnt ListItemType& anItem, bool& isPresent);
}; // end SortedList
```

3.4 That would be the inheritance of:



3.5 The implementation would be:

```
SortedList::SortedList()
{
} // end default constructor

SortedList::SortedList(const SortedList& sList) : List(sList)
{
} // end copy constructor

SortedList::~SortedList()
{
} // end destructor
```

```
void SortedList::sortedInsert(const ListItemType& newItem) throw(ListException)
{
        bool found;
        int newPosition = locatePosition(newItem, found);
        insert(newPosition, newItem);
} // end sortedInsert

void SortedList::sortedRemove(const ListItemType& anItem) throw(ListException)
{
        bool found;
        int position = locatePosition(anItem, found);
        if (found) // item actually found
                remove(position);
        else
                throw ListException("ListException: Item to remove not found");
} // end sortedRemove

int SortedList::locatePostion(const ListItemType& anItem, bool& isPresent)
{
        ListNode *trav = getHead();
        int postion = 1;
        while ((trav != NULL) && (getNodeItem(trave) < anItem))
        {
                trav = getNextNode(trav);
                position++;
        } // end while
        if ((trav != NULL) && (anItem == getNodeItem(trav))
                isPresent = true;
        else
                isPresent = false;
        return position;
} // end locatePosition
```

3.6 The class SortedList now has operations of isEmpty, getLength, and retrieve along with its sortedInsert and sortedRemove.

3.6.1 The names can be confusing.

3.6.2 The are different ways to deal with the names., such as:

3.6.2.1 You can add to the SortedList methods such as sortedGetLength like:

```
int SortedList::sortedGetLength()
{
        return getLength();
} // end sortedGetLength
```

3.6.2.2 Change the names of the SortedList operations to just insert and remove.

```
void SortedList::insert(const ListItemType& newItem) throw (ListException)
{
        bool found;
        int newPositiion = locatePosition(newItem, found);
        List::insert(newPosition, newItem);
} // end insert
```

3.7 Since the names of the insert methods for List and SortedList are the same, you have to specify the class name like "List::insert(newPosition, newItem);".

3.8 You could make sure that the insert method cannot work in SortedList by throwing an exception, like so:

```
virtual void insert(int index, ListItemType newItem) throw(ListIndexOutOfException,
ListException)
{
        throw ListException("ListException: base-class version of insert " +
                        "cannot be invoked on derived object.");
} // end insert
```

4 The problem is that the sorted list operates differently from list – the value-oriented nature of sorted list does not support all of the list's operations.

4.1 You cannot use a sorted list as a list and expect the items to remain in sorted order.

4.2 Public inheritance of list by sorted list is not appropriate.

4.3 If you do not have an "is-a" relationship between your new class and an existing class then use the "has-a" relationship.

4.4 So the SortedList can have a private data member of list or the "has-a".

4.5 In that way you can manipulate the list to be a sorted list and avoid some confusion, like so:

```
class SortedList
{
public:
// constructor and destructor:
        SortedList();
        SortedList(const SortedList& aList);
        virtual ~SortedList();

// sorted List operations
        virtual bool sortedIsEmpty const;
        virtual int sortedGetLength() const;
```

```
        virtual void sortedInsert(const ListItemType& newItem) throw(ListException);
        virtual void sortedRemove(const ListItemType& anItem) throw(ListException);
        virtual void sortedRetrieve(int index, ListItemType& anItem) const
                                                throw(ListIndexOutOfRangeException);
        virtual int locatePosition(const ListItemType& anItem, bool& isPresent);

private:
        List aList;
}; // end SortedList
```

4.6 Therefore the sortedInsert would look like:

```
void SortedList::sortedInsert(const ListItemType& newItem) throw(ListException)
{
        bool found;
        int newPostion = locatePostion(newItem, found);
        aList.insert(newPosition, newItem);
} // end sortedInsert
```

4.7 So if you do not have an "is-a" relationship, you should not use public inheritance.

4.8 However, if you wanted to access some of the functions of a base class, you can use a private inheritance, like so:

```
class SortedList : private List
{
public:
….
}; // end SortedList
```

4.9 Private inheritance is not used often.

Chapter 8-5, Class Templates

1. We've been using the following:

```
typedef double ListItemType; // type of list item
```

1.1 To allow changing the data types for the class.

1.2 However, what if you needed a list of integers, a list of doubles, AND a list of strings.

1.2.1 The typedef would not work.

1.2.2 You define a template to allow different data types for one class, like so:

```
template <typename T>
class NewClass
{
public:
        NewClass();
        NewClass(T newData);
        void setData(T newData);
        T getData();
private:
        T theData;
}; // end NewClass
```

1.2.3 So you can use it like so:

```
int main()
{
        NewClass<int>          first;
        NewClass<double>    second(4.8);

        first.setData(5);
        cout << second.getData() << endl;
…
}
```

1.2.4 The implementation of a templated class would look like:

```
template <typename T>
NewClass<T>::NewClass()
{
} // end default constructor

template <typename T>
NewClass<T>::NewClass(T initialData) : theData(initialData)
{
} // end constructor

template <typename T>
NewClass<T>::setData(T newData)
{
        theData = newData;
} // end setData

template <typename T>
NewClass<T>::getData()
{
        return theData;
} // end getData
```

2 There are some problems with defining all the methods using a template.
2.1 For instance:

```
template <typename T>
NewClass<T>::display()
{
        cout << theData;
} // end display
```

2.2 The cout operation would work with most basic data types such as int, double, string, and such.
2.3 However, if you substitute a structure, the cout may not work (may have to document the fact).

2.4 Now if you put templates in a header file, you will have a problem: the compiler needs to know what data types the template will use BEFORE it can compile the implementation.

2.4.1 Therefore, you cannot compile the implementation separately from the client code.
2.4.2 The easiest way to rectify that is to use an "include" at the end of the header file so that the client will compile along with the template implementation, like so:

```
/**  @file ListNodeT.h */

#include <cstddef>
template <typename T> class List;

/** ADT list – Pointer-based implementation – TEMPLATE VESION */
template <typename T>
class ListNode
{
private:
        ListNode(): next(NULL) {}
        ListNode(const T & nodeItem, ListNode *ptr) : item(nodeItem), next(ptr) {}
        T  item;                // a data item on the list
        ListNode *next;         // pointer to next node
        // friend class – can access private parts
        friend class List <T>;
}; // end ListNode



/**  @file ListT.h */
```

```
#include “ListNodeT.h”
#include “ListException.h”
#include “ListIndexOutOfRangeException.h”

/**  ADT list – Pointer-based implementation – TEMPLATE VERSION */
template <typename T>
class List
{
public:
// constructors and destructor:
        List();
        List(const List<T> & aList);
        virtual ~List();
        // List operations:
        virtual bool isEmpty() const;
        virtual int getLength() const;
        virtual void insert(int index, const T &newItem)
                throw(ListIndexOutOfRangeException, ListException);
        virtual void remove(int index) throw(ListIndexOutOfRangeException);
        virtual void retrieve(int index, T & dataItem) const
                throw(ListIndexOutOfRangeException);
protected:
        void setSize(int newSize);
        ListNode<T> *getHead() const;
        void setHead(ListNode<T> *newHead);
        T getNodeItem(ListNode<T> *ptr) const;
        ListNode<T> *getNextNode(ListNode<T> *ptr) const;

private:
        int             size;
        ListNode<T>  *head;

        ListNode<T>  *find(int position) const;
}; // end List

#include “ListT.cpp”
```

2.4.3 Note that the implementation had been included at the end of the header file.

2.4.4 The ListT.cpp file is:

```
/**  @file  ListT.cpp
 *  Excerpts from the implementation */
```

```
#include <cstddef>     // for NULL
#include <new>         // for bad_alloc

using namespace std;

template <typename T>
List<T>::List() : size(0), head(NULL)
{
} // end default constructor

template <typename T>
void List<T>::insert(int index, const T & newItem)
        throw(ListIndexOutOfRangeException, ListException)
{
        int newLength = getLength() + 1;

        if ((index < 1) || (index > newLength))
                throw ListIndexOutOfRangeException(
                        "ListIndexOutOfRangeException: insert index out of range");
        else
        { // create new node and place newItem in it
                try
                {       ListNode<T> *newPtr = new ListNode<T>;
                        size = newLength();
                        newPtr->item = newItem;

                        // attach new node to list
                        if (index == 1)
                        {       // insert new node at beginning of list
                                newPtr->next = head;
                                head = newPtr;
                        }
                        else
                        {       ListNode<T> *prev = find(index-1);
                                // insert new node after node to which prev points
                                newPtr->next = prev->next;
                                prev->next = newPtr;
                        } // end if
                }
                catch (bad_alloc e)
                {
                        throw ListException(
                           "ListException: insert connate allocate memory");
                } // end try
        } // end if
```

```
} // end insert

template <typename T>
ListNode<T> *List<T>::find(int index) const
{
        if (  (index < 1)  ||  (index > getLength()) )
                return NULL;
        else // count from the beginning of the list
        {       ListNode<T> *cur = head;
                for (int skip = 1; skip < index; ++skip)
                        cur = cur->next;
                return cur;
        } // end if
} // end find
```

2.5 An example of the client code out be:

```
#include "ListT.h"

int main()
{
        List<double> floatList;
        List<char> charList;

        floatList.insert(1, 1.1);
        floatList.insert(2, 2.2);

        charList.insert(1, 'a');
        charList.insert(2, 'b');
…
```

2.6 You can also have more than one data-type with a template like so:

```
template <typename T1, typename T2>
```

Chapter 8-6, Overloaded Operators

1 With many operators, they can actually have multiple meaning.
- 1.1 For instance "+" can mean add two integers, add two doubles, or concatenate strings.
- 1.2 You can overload just about any operator in C++ and attach a function to it.
- 1.3 For instance, what if you did the following to two lists:

```
if (myList == yourList)
        cout << "The lists are equal.\n";
```

1.4 The "==" could mean the lengths of the lists are the same or that every item in myList matches every item in yourList.

1.4.1 To overlaod the operator you use the "operator" keyword, like so:

operator symbol

1.4.2 So to implement the overloaded operator, you add to the List class the following:

virtual bool operator==(const List& rhs) const;

1.4.3 That would be equivalent to:

virtual bool isEqual(const List& rhs) const;

1.4.4 You can use the isEqual like so:

```
if (myList.isEqual(yourList))
        cout << "The lists are equal.\n";
```

1.4.5 You treat the operator== exactly as you do with isEqual, like so:

```
if (myList.operator==(yourList))
        cout << "The lists are equal\n";
```

1.4.6 Or the simpler way is:

```
if (myList == yourList)
        cout << "The lists are equal\n";
```

1.5 So here is the pointer-based implementation:

```
bool List::operator==(const List& rhs) const
{
        bool isEqual;
        if (size != rhs.size)
                isEqual = false; // lists have uneven lengths
        else if ( (head == NULL) && (rhs.head == NULL) )
                isEqual = true; // both lists are empty
        {       // compare items
                ListNode *leftPtr = head;
                ListNode *rightPtr = rhs.head;
                int count;
                for (cout = 1; (count <= size) &&
```

```
                    (leftPtr->item = rightPtr->item); ++count)
            {       leftPtr = leftPtr->next;
                    rightPtr = rightPtr->next;
            } // end for
            isEqual = count > size;
        } // end if
        return isEqual;
} // end operator==
```

2 You can even overload the assignment operator of "=".
2.1 However, with the operation of:

myList = yourList;

2.2 And myList already has elements, you need to deallocate the myList items before allocating new items.

Chapter 8-7, Iterators

1 You can also implement iterator operations, which can return a pointer to an item in a pointer-based list.
1.1 They would use the operator overloading also to accomplish this.

1.2 Some common iterators would be:

Operation	Description
*	Return the item that the iterator currently references
++	Move the iterator to the next item in the list
--	Move the iterator to the previous item in the list
==	Compare two iterators for equality
!=	Compare two iterators for inequality

1.3 An example on an iterator class would be:

```
/** @file ListIterator.h */

#include "ListNode.h"          // Definition of ListNode and ListItemType; ListNode
                               // declares ListIterator as a friend class

/** ListIterator – Used in the iterator version of the ADT list */
class ListIterator
{
public:
        ListIterator(const List *aList, ListNode *nodePtr);
```

```
        const ListItemType & operator*();
        ListIterator operator++();       // prefix ++

        bool operator==(const ListIterator& rhs) const;
        bool operator!=(const ListIterator& rhs) const;

        friend class List;

private:
        const List *container; // ADT associated with iterator
        ListNode *cur;         // current location in collection
}; // end ListIterator
```

1.4 The implementation is:

```
/** @file ListIterator.cpp */

#include "ListIterator.h"

ListIterator::ListIteratorconst List *aList, ListNode *nodePtr) :
                                        container(aList), cur(nodePtr)
{
} // constructor

const ListItemType& ListIterator::operator*()
{
        return cur->item;
} // end operator*

ListIterator ListIterator::operator++()
{
        cur = cur->next;
        return *this;
} // end prefix operator++

bool ListIterator::operator==(const ListIterator& rhs) const
{
        return ((container==rhs.container) && (cur == rhs.cur));
} // end operator==

bool ListIterator::operator!=(const ListIterator& rhs) const
{
        return !(*this == rhs);
} // end operator!=
```

2 Implementing the list using iterators is in the textbook.