Swinburne University Of Technology

Faculty of Information and Communication Technologies

ASSIGNMENT COVER SHEET

Subject Code: HIT3303
Subject Title: Data Structures & Patterns
Assignment number and title: 2 – Lists and Design Patterns
Due date: **April 7, 2009, 02:30 p.m., on paper**
Lecturer: Dr. Markus Lumpe

Your name:_____

Marker's comments:

Problem	Marks	Obtained
1	27	
2	79	
Total	106	

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener:_____

Problem Set 2: Lists and Design Patterns

Preliminaries

Study or review the following concepts:

1. C++ templates
2. What is difference between value semantics and reference semantics?
3. What is a constant reference?
4. What is a constant object?
5. What is an enumeration type?
6. What is a `typedef` declaration?
7. What is `delete` and how does it work?
8. When do we need destructors?
9. What is a state machine?

Problem 1:

Define a double-linked list that satisfies the following template class specification:

```
template<class DataType>
class Node
{
private:
  const DataType& fValue;
  Node<DataType>* fNext;
  Node<DataType>* fPrevious;

public:
  Node( const DataType& aValue,
        Node<DataType>* aNext = (Node<DataType>*)0,
        Node<DataType>* aPrevious = (Node<DataType>*)0 );
  ~Node();

  const DataType& GetValue() const;
  const Node<DataType>* GetNext() const;
  const Node <DataType>* GetPrevious() const;
};
```

The template class `Node` defines the structure of a double-linked list. It uses two pointers: `fNext` and `fPrevious` to connect two adjacent list elements. The constructor takes `aValue`, `aNext`, and `aPrevious` as arguments and returns a properly initialized list node. The arguments `aNext` and `aPrevious` take a default argument `(Node<DataType>*)0`, which is a null-pointer to elements of type `Node`. The destructor destroys the list and releases all allocated resources (i.e., memory) in turn. The methods `GetValue`, `GetNext`, and `GetPrevious` and data providers that do not change the state of the corresponding object. As a result, objects of type `Node` are constant objects.

There is, however, one complication. Template classes are "class blueprints" or, better, abstractions over classes. Before we can use template classes, we have to instantiate them. But, the instantiation process, to work correctly, requires also the implementation. For this reason, when defining template classes, the implementation has to be included in the header file. There are two ways to accomplish this:

- Implement the member functions directly in the class specification (like it is done in Java or C#).
- Implement the member functions outside the class specification but within the same header file.

If you follow this scheme, working with templates is pretty straightforward.

Implement class `Node`.

Test sample 1:

```
void TestListImplementation1()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );
  string s4( "Four" );

  Node<string> n1( s1 );
  Node<string> n2( s2, (Node<string>*)0, &n1 );
  Node<string> n3( s3, (Node<string>*)0, &n2  );
  Node<string> n4( s4, (Node<string>*)0, &n3  );

  cout << "Forward:" << endl;

  for ( const Node<string>* nodes = &n1;
          nodes != (Node<string>*)0; nodes = nodes->GetNext() )
  {
    cout << nodes->GetValue() << endl;
  }

  cout << "Backward:" << endl;

  for ( const Node<string>* nodes = &n4;
          nodes != (Node<string>*)0; nodes = nodes->GetPrevious() )
  {
    cout << nodes->GetValue() << endl;
  }
}
```

Result:

```
Forward:
One
Two
Three
Four
Backward:
Four
Three
Two
One
```

Test sample 2:

```
void TestListImplementation2()
{
  string s1( "One" );
  string s2( "Two" );
  string s3( "Three" );

  Node<string>* pn1 = new Node<string>( s1 );
  Node<string>* pn2 = new Node<string>( s2, (Node<string>*)0, pn1 );
  Node<string>* pn3 = new Node<string>( s3, (Node<string>*)0, pn2  );

  cout << "Tree elements:" << endl;

  for ( const Node<string>* nodes = pn1;
            nodes != (Node<string>*)0; nodes = nodes->GetNext() )
  {
    cout << "(";
    if ( nodes->GetPrevious() != (Node<string>*)0 )
      cout << nodes->GetPrevious()->GetValue();
    else
      cout << "<NULL>";

    cout << "," << nodes->GetValue() << ",";

    if ( nodes->GetNext() != (Node<string>*)0 )
      cout << nodes->GetNext()->GetValue();
    else
      cout << "<NULL>";

    cout << ")" << endl;
  }

  delete pn2;

  cout << "Two elements:" << endl;

  for ( const Node<string>* nodes = pn1;
            nodes != (Node<string>*)0; nodes = nodes->GetNext() )
{
    cout << "(";
    if ( nodes->GetPrevious() != (Node<string>*)0 )
      cout << nodes->GetPrevious()->GetValue();
    else
      cout << "<NULL>";

    cout << "," << nodes->GetValue() << ",";

    if ( nodes->GetNext() != (Node<string>*)0 )
      cout << nodes->GetNext()->GetValue();
    else
      cout << "<NULL>";

    cout << ")" << endl;
  }

  delete pn3;
  delete pn1;
}
```

Result:

```
Tree elements:
(<NULL>,One,Two)
(One,Two,Three)
(Two,Three,<NULL>)
Two elements:
(<NULL>,One,Three)
(One,Three,<NULL>)
```

Problem 2:

Define a bi-directional list iterator for double-linked lists that satisfies the following template class specification:

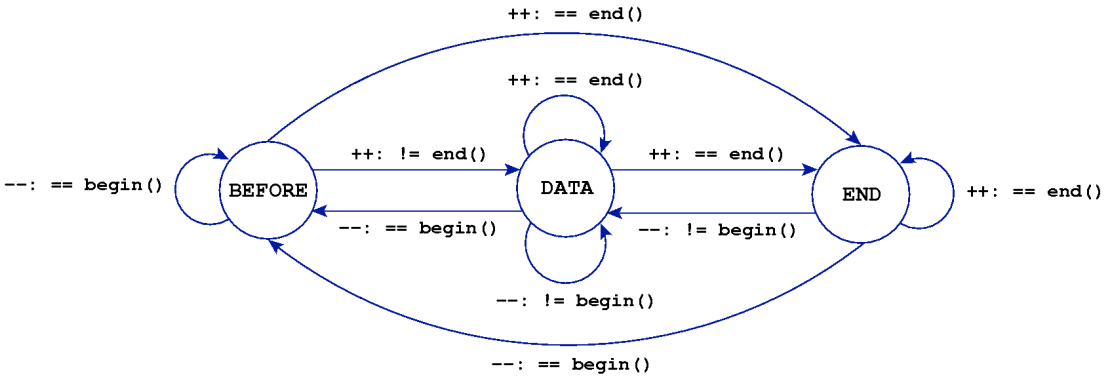
```
template<class DataType>
class NodeIterator
{
private:

  enum IteratorStates { BEFORE, DATA , END };

  const Node<DataType>* fTop;
  const Node<DataType>* fLast;
  const Node<DataType>* fCurrent;
  IteratorStates fState;

public:
  NodeIterator( Node<DataType>* aList );

  DataType operator*() const;    // dereference

  NodeIterator& operator++();    // prefix increment
  NodeIterator operator++(int);  // postfix increment
  NodeIterator& operator--();    // prefix decrement
  NodeIterator operator--(int);  // postfix decrement

  bool operator==( const NodeIterator& aOtherIter ) const;
  bool operator!=( const NodeIterator& aOtherIter ) const;

  NodeIterator begin();
  NodeIterator end();
};
```

The bi-directional list iterator implements the standard operators for iterators: dereference to access the current iterator element, both versions of increment to advance the iterator to the next element, and both versions of decrement to go backwards. The list iterator also defines the equivalence predicates and the two factory methods `begin()` and `end()`. The method `begin()` returns a new list iterator positioned before the first element of the double-linked list, whereas `end()` returns a new list iterator that is positioned after the last element of the double-linked list.

Implement the list iterator. Please note that the constructor of the list iterator has to properly set `fTop`, `fLast`, and `fCurrent`. In particular, the constructor has to position the iterator on the first element of the list or yield an iterator equivalent to `end()` if the list is empty.

An iterator must not change the underlying collection. However, in the case of `NodeIterator` we need a special marker to denote, whether the iterator is "before" the first list element or "after" the last list element. Since we cannot change the underlying list, we need to add "state" to the iterator. Using the iterator state (i.e., `fState`) we can now clearly mark when the iterator is before the first element, within the first and the last element, or after the last element.

To guarantee to correct behavior of the `NodeIterator`, it must implement a "state machine" with three states: `BEFORE`, `DATA`, `END`. The following state transition diagram illustrates, how `NodeIterator` works:



All increment and decrement operators have to test, whether the iterator is still positioned within the collection. In this case the current iterator is different from both `begin()` and `end()`. If the iterator is positioned before the first element, then it is equivalent to `begin()`. If the iterator is positioned past the last element, then it is equivalent to `end()`. Please note that the iterator can in one step become equivalent to `begin()` or `end()`.

Implement class `NodeIterator`.

Test sample 3:

```
void TestListIterator()
{
  Node<int> n1( 1 );
  Node<int> n2( 2, (Node<int>*)0, &n1 );
  Node<int> n3( 3, (Node<int>*)0, &n2 );
  Node<int> n4( 4, (Node<int>*)0, &n3 );
  Node<int> n5( 5, (Node<int>*)0, &n4 );
  Node<int> n6( 6, (Node<int>*)0, &n5 );

  cout << "Forward iteration:" << endl;
  NodeIterator<int> iter1( &n1 );

  for ( ; iter1 != iter1.end(); iter1++ )
  {
    cout << *iter1 << endl;
  }

  cout << "Backward iteration:" << endl;
  NodeIterator<int> iter2( &n6 );

  for ( ; iter2 != iter2.begin(); iter2-- )
  {
    cout << *iter2 << endl;
  }
}
```

Result:

```
Forward iteration:
1
2
3
4
5
6
Backward iteration:
6
5
4
3
2
1
```

Submission deadline: Tuesday, April 7, 2009, 2:30 p.m.

Submission procedure: on paper.