Swinburne University of Technology

Faculty of Science, Engineering and Technology

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	9, Doubly-linked List and Bidirectional Iterator
Lecturer:	Dr. Markus Lumpe

Problem 1

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

```
#pragma once

template<typename T>
class DoublyLinkedList
{
private:

  T fPayload;                                          // payload
  DoublyLinkedList* fNext;                             // next element
  DoublyLinkedList* fPrevious;                         // previous element

public:

  explicit DoublyLinkedList( const T& aPayload );      // l-value constructor
  explicit DoublyLinkedList( T&& aPayload );           // r-value constructor

  // aNode becomes previous of this
  DoublyLinkedList& push_front( DoublyLinkedList& aNode );

  // aNode becomes next of this
  DoublyLinkedList& push_back( DoublyLinkedList& aNode );

  void isolate();                                      // removes this node

  void swap( DoublyLinkedList& aNode );                // exchange payloads

  // dereference operator, payload
  const T& operator*() const;

  // returns constant reference to paylod
  const T& getPayload() const;

  // returns constant reference to next
  const DoublyLinkedList& getNext() const;

  // returns constant reference to previous
  const DoublyLinkedList& getPrevious() const;
};
```

The template class `DoublyLinkedList` defines the structure of a doubly-linked list. It uses two pointers: `fNext` and `fPrevious` to connect adjacent list elements.

The doubly-linked list supports two constructors, one for l-value references and one for r-value references. The latter “steals” the memory of the argument (this is possible as the argument is a temporary or a literal expression that goes immediately out of scope after the constructor call).

The methods **`operator*`**`()`, `getPayload`, `getNext`, and `getPrevious` define simple read-only getter functions for the corresponding fields of a `DoublyLinkedList` object.

The methods `push_front`, `push_back`, `swap`, and `isolate` provide mechanisms to manipulate objects of class `DoublyLinkedList`. The method `push_front` adds the argument `aNode` object into the list by making `aNode` the new `fPrevious` node of `this`. The method `push_back`, on the other hand, injects the argument `aNode` object into the list by making `aNode` the new `fNext` node of `this`. The method `isolate` removes `this` from the list. That is, `isolate` has to properly link the remaining list nodes adjacent to `this`. Finally, `swap` exchanges the payload of this list element and the argument.

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 to work correctly, the instantiation process requires the complete implementation of the class (see lecture notes *Class Template Instantiation*). For this reason, when defining template classes, the implementation has to be included in the header file.

There are four test drivers (Main.cpp) to allow for a staged development:

- P1:
  - constructors
  - `push_front()`
  - **`operator*`**`()`
  - `getPrevious()`
  - `getNext()`

  output:
  ```
  Test:
  	push_front()
  	operator*()
  	getPrevious()
  	getNext()
  The nodes (forwards):
  (Two,One,Four)
  (One,Four,Three)
  (Four,Three,Two)
  (Three,Two,One)
  The nodes (backwards):
  (Two,One,Four)
  (Three,Two,One)
  (Four,Three,Two)
  (One,Four,Three)
  ```

- P2:
  - `push_back()`

  output:
  ```
  Test:
  	push_back()
  The nodes (forwards):
  (Four,One,Two)
  (One,Two,Three)
  (Two,Three,Four)
  (Three,Four,One)
  The nodes (backwards):
  (Four,One,Two)
  (Three,Four,One)
  (Two,Three,Four)
  (One,Two,Three)
  ```

- P3:
  - `isolate()`

  output:
  ```
  Test:
  	isolate()
  The nodes (forwards):
  (Two,One,Four)
  (One,Four,Three)
  (Four,Three,Two)
  (Three,Two,One)
  isolate Three
  The nodes (backwards):
  (Two,One,Four)
  (Four,Two,One)
  ```

```
(One,Four,Two)
```

- P4:
  - `swap()`

output:

```
Test:
       swap()
The nodes (forwards):
(Two,One,Four)
(One,Four,Three)
(Four,Three,Two)
(Three,Two,One)
swap Three <=> One
The nodes (forwards):
(Two,Three,Four)
(Three,Four,One)
(Four,One,Two)
(One,Two,Three)
```

Problem 2

Start with the `DoublyLinkedList` template class. Define a bi-directional list iterator for doubly-linked lists that satisfies the following template class specification:

```
#pragma once

#include "DoublyLinkedList.h"

template<typename T>
class DoublyLinkedListIterator
{
private:
  enum class States { BEFORE, DATA , AFTER };                // iterator states

  using Node = DoublyLinkedList<T>;

  const Node* fRoot;                                         // doubly-linked list

  States fState;                                             // iterator state
  const Node* fCurrent;                                      // iterator position

public:

  using Iterator = DoublyLinkedListIterator<T>;

  DoublyLinkedListIterator( const Node* aRoot );             // constructor

  const T& operator*() const;                                // dereference
  Iterator& operator++();                                    // prefix increment
  Iterator operator++(int);                                  // postfix increment
  Iterator& operator--();                                    // prefix decrement
  Iterator operator--(int);                                  // postfix decrement
  bool operator==( const Iterator& aOtherIter ) const;       // equivalence
  bool operator!=( const Iterator& aOtherIter ) const;       // not equal

  Iterator begin() const;                      // first element forward
  Iterator end() const;                        // after last element forward
  Iterator rbegin() const;                     // first element backwards
  Iterator rend() const;                       // before first element backwards
};
```

The bi-directional list iterator implements the standard operators for bi-directional iterators: dereference to access the current element the iterator is positioned on, the increment operators advance the iterator to the next element, and the decrement operators take the iterator to the previous element. The list iterator also defines the equivalence predicates and the four factory methods: `begin()`, `end()`, `rbegin()`, and `rend()`. The method `begin()` returns a new iterator positioned at the first element, `end()` returns a new iterator that is positioned after the last element, `rbegin()` a new iterator positioned at the last element, and the method `rend()` returns a new list iterator positioned before the first element of the doubly-linked list.

Implement the list iterator. Please note that the constructor of the list iterator has to properly set `fRoot` and `fState`.

To guarantee to correct behavior of the `DoublyLinkedNodeIterator`, it must implement a *state machine* with three states: `BEFORE`, `DATA`, `AFTER`. See tutorial notes on state machines and the specification for the doubly-linked list iterator. Think of the iterator as a clock. The start of the list is 12 o'clock. The iterator can freely move around the clock in either direction. However, it must not go past 12 o'clock. This position marks the end for a forward or backwards iteration.

The forward iteration starts at `fRoot` and ends when the iterator tries to move onto `fRoot` again. The backwards iteration starts at `fRoot`'s previous element and stops when it moves past `fRoot`.

Implement the prefix increment first. The postfix increment just calls the prefix increment. The prefix decrement is a mirror image of the prefix increment with some minor adjustments.

There is one test driver (Main.cpp):

- P5:

  output:

  ```
  Forward iteration I:
  One
  Two
  Three
  Four
  Five
  Six
  Backward iteration I:
  Six
  Five
  Four
  Three
  Two
  One
  Forward iteration II:
  One
  Two
  Three
  Four
  Five
  Six
  Backward iteration II:
  Six
  Five
  Four
  Three
  Two
  One
  Iterator tests:
  Yes
  Yes
  Yes
  ```

Please complete this task as it provides the basis for further data structures being developed in this unit.