Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	11, Linked Lists
Lecturer:	Dr. Markus Lumpe

Lab 10: Linked Lists

In this tutorial, we are experimenting with doubly-linked lists and a corresponding iterator.

The test driver (i.e., main.cpp) uses `P1` and `P2` as variables to enable/disable the test associated with a corresponding problem. To enable a test just uncomment the respective **`#define`** line. For example, to test problem 2 only, enable **`#define`** `P2`:

```
// #define P1
#define P2
```

In Visual Studio, the code blocks enclosed in **`#ifdef`** `PX` … **`#endif`** are grayed out, if the corresponding test is disabled. The preprocessor definition **`#ifdef`** `PX` … **`#endif`** enables conditional compilation. XCode does not use this color coding scheme.

Problem 1

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

```
#pragma once

#include <memory>
#include <algorithm>

template<typename T>
class DoublyLinkedList
{
public:

  using Node = std::shared_ptr<DoublyLinkedList<T>>;
  using Next = std::shared_ptr<DoublyLinkedList<T>>;
  using Previous = std::weak_ptr<DoublyLinkedList<T>>;

  T fData;
  Node fNext;
  Previous fPrevious;

  // factory method for list nodes
  template<typename... Args>
  static Node makeNode( Args&&... args );

  DoublyLinkedList( const T& aData ) noexcept;

  DoublyLinkedList( T&& aData ) noexcept;

  void isolate() noexcept;

  void swap( DoublyLinkedList& aOther ) noexcept;
};
```

Refer to the lecture slides to guide your implementation.

The method `swap()` follows the practice that we studied when defining copy and move semantics for abstract data types in C++.

You can use `#define P1` in `Main.cpp` to enable the corresponding test driver, which should produce the following output:

```
Test DoublyLinkedList:
Traverse links in forward direction:
Value: DDDD
Value: EEEE
Value: FFFF
Value: AAAA
Value: BBBB
Value: CCCC
Traverse links in backwards direction:
Value: DDDD
Value: CCCC
Value: BBBB
Value: AAAA
Value: FFFF
Value: EEEE
Traverse links in forward direction (Four <==> Six):
Value: FFFF
```

```
Value: EEEE
Value: DDDD
Value: AAAA
Value: BBBB
Value: CCCC
Traverse links in forward direction (isolate Three):
Value: FFFF
Value: EEEE
Value: DDDD
Value: AAAA
Value: BBBB
Test complete.
```

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 <cassert>

#include "DoublyLinkedList.h"

template<typename T>
class DoublyLinkedListIterator
{
public:
  using Iterator = DoublyLinkedListIterator<T>;
  using Node = typename DoublyLinkedList<T>::Node;

  enum class States { BEFORE, DATA, AFTER };

  DoublyLinkedListIterator( const Node& aHead, const Node& aTail ) noexcept;

  const T& operator*() const noexcept;

  Iterator& operator++() noexcept;              // prefix
  Iterator operator++(int) noexcept;            // postfix

  Iterator& operator--() noexcept;              // prefix
  Iterator operator--(int) noexcept;            // postfix

  bool operator==( const Iterator& aOther ) const noexcept;
  bool operator!=( const Iterator& aOther ) const noexcept;

  Iterator begin() const noexcept;
  Iterator end() const noexcept;
  Iterator rbegin() const noexcept;
  Iterator rend() const noexcept;

private:
  Node fHead;
  Node fTail;
  Node fCurrent;
  States fState;
};
```

The doubly-linked list iterator implements a standard bi-directional iterator. The dereference operator provides access to 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 doubly-linked 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.

To guarantee the correct behavior of the `DoublyLinkedListIterator`, it must implement a *state machine* with three states: `BEFORE`, `DATA`, `AFTER`. See the additional tutorial notes on state machines and the specification for the doubly-linked list iterator.

Think of the iterator as an analog 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.

Please note that the iterator uses a head and a tail to record the respective ends of the doubly-linked list. The ends of the doubly-linked list are not connected (i.e., `fPrevious` of the head is **`nullptr`** and `fNext` of tail is a **`nullptr`**). Together with the state of the iterator we can hence always determine whether there are more elements to be visited or if the iterator has reached the end. The direction (i.e., increment or decrement) tells us whether we are past the last element or whether we are prior the first element in the doubly-linked list.

The constructor for the iterator has to test a crucial precondition. Head and tail must both either point to a proper list node or be **`nullptr`**.

You can use `#define P2` in `Main.cpp` to enable the corresponding test driver, which should produce the following output (sorting in increasing order and counting the number of exchanges):

```
Test DoublyLinkedListIterator:
Forward iteration:
AAAA
BBBB
CCCC
DDDD
EEEE
FFFF
Backwards iteration:
FFFF
EEEE
DDDD
CCCC
BBBB
AAAA
Test complete.
```

There should be no memory leaks. The smart pointers handle memory management automatically when a list object goes out of scope.

Please note, however, that the doubly-linked list uses a weak smart pointer. To access the corresponding handle (i.e., the pointer to the object stored on the heap), the weak smart pointer must be locked first to obtain a shared smart pointer.