# Swinburne University of Technology

*Faculty of Science, Engineering and Technology*

## ASSIGNMENT COVER SHEET

---

**Subject Code:** COS30008
**Subject Title:** Data Structures & Patterns
**Assignment number and title:** 6 – Copy Control & Container Types
**Due date:** May 21, 2019, 10:30
**Lecturer:** Dr. Markus Lumpe

---

**Your name:**_____ **Your student id:**_____

| Check Tutorial | Wed 08:30 | Wed 10:30 | Wed 12:30 | Wed 14:30 | Thurs 08:30 | Thurs 10:30 | Thurs 12:30 | Thurs 14:30 | Fri 10:30 |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |

---

Marker's comments:

| Problem | Marks | Obtained |
|---|---|---|
| 1 | 14+24=38 | |
| 2 | 17 | |
| 3 | 18 | |
| 4 (Optional) | 17 | |
| 5 (Optional) | 18 | |
| Total excluding 4&5 | 73(108) | |

---

**Extension certification:**

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

Signature of Convener:_____

# Problem Set 6: Copy Control & Container Types

## Preliminaries

Review the solution of tutorial 9 & 10 and problem set 5, the lecture material regarding the construction of an abstract data type, and how copy control is achieved for a given data type.

## Problem 1:

Add proper copy control to the template class `List` that we developed in problem set 5.

```
#pragma once

#include "DoublyLinkedNode.h"
#include "DoublyLinkedNodeIterator.h"

#include <stdexcept>

template<class T>
class List
{
private:
  // auxiliary definition to simplify node usage
  typedef DoublyLinkedNode<T> Node;

  Node* fFirst;            // the first element in the list
  Node* fLast;             // the last element in the list
  unsigned int fCount;     // number of elements in the list

public:
  // auxiliary definition to simplify iterator usage
  typedef DoublyLinkedNodeIterator<T> Iterator;

  List();                  // default constructor - creates empty list
  ~List();                 // destructor - frees all nodes

  // PS6: copy control features (destructor is given)
  List( const List& aOtherList );              // copy constructor
  List& operator=( const List& aOtherList );   // assignment operator

  bool isEmpty() const;    // Is list empty?
  int size() const;        // list size

  void push_front( const T& aElement ); // adds a node aElement at front
  void push_back( const T& aElement );  // adds a node aElement at back
  void remove( const T& aElement );     // removes first node that matches aElement

  const T& operator[]( unsigned int aIndex ) const;  // list indexer

  Iterator begin() const;         // return a forward iterator
  Iterator end() const;           // return a forward end iterator
  Iterator rbegin() const;        // return a backwards iterator
  Iterator rend() const;          // return a backwards end iterator
};
```

## Test harness

```
string s1( "AAAA" );
string s2( "BBBB" );
string s3( "CCCC" );
string s4( "DDDD" );
string s5( "EEEE" );

List<string> lList;

lList.push_front( s4 );
lList.push_front( s3 );
lList.push_front( s2 );

List<string> copy( lList );

// iterate from the top
cout << "A - Top to bottom: " << copy.size() << " elements" << endl;

for ( const string& element : copy )
{
  cout << "A list element: " << element << endl;
}

// override list
lList = copy;

lList.push_front( s1 );
lList.push_back( s5 );

// iterate from the top
cout << "B – Bottom to top: " << lList.size() << " elements" << endl;

for ( List<string>::Iterator iter = lList.rbegin();
                iter != iter.rend(); iter-- )
{
  cout << "A list element: " << *iter << endl;
}
```

The result should look like this:

```
A - Top to bottom: 3 elements
A list element: BBBB
A list element: CCCC
A list element: DDDD
B – Bottom to top: 5 elements
A list element: EEEE
A list element: DDDD
A list element: CCCC
A list element: BBBB
A list element: AAAA
```

## Problem 2:

Using the template class `List` with proper copy control, implement the template class `Stack` as specified below:

```
#pragma once

#include "List.h"
#include <stdexcept>

template<class T>
class Stack
{
private:
  List<T> fElements;

public:
  bool isEmpty() const;
  int size() const;
  void push( const T& aItem );
  void pop();
  const T& top() const;
};
```

That is, `Stack` is a stack container type that can grow in size on demand.

Complete the implementation of the template class `Stack`.

Test harness:

```
Stack<string> lStack;

lStack.push( "AAAA" );
lStack.push( "BBBB" );
lStack.push( "CCCC" );
lStack.push( "DDDD" );

cout << "top: " << lStack.top() << endl;
lStack.pop();
cout << "top: " << lStack.top() << endl;
lStack.pop();
cout << "top: " << lStack.top() << endl;
cout << "size: " << lStack.size() << endl;
cout << "is empty: " << (lStack.isEmpty() ? "T" : "F" ) << endl;
lStack.pop();
cout << "top: " << lStack.top() << endl;
lStack.pop();
cout << "is empty: " << (lStack.isEmpty() ? "T" : "F" ) << endl;
```

Result:

```
top: DDDD
top: CCCC
top: BBBB
size: 2
is empty: F
top: AAAA
is empty: T
```

## Problem 3:

Using the template class `Stack`, define a `StackIterator` that is initialized with a `Stack` and provides a sequential (forward) access to all elements contained in the stack.

```
#pragma once

#include "Stack.h"

template<class T>
class StackIterator
{
private:
  Stack<T> fStack;

public:
  StackIterator( const Stack<T>& aStack );

  const T& operator*() const;              // dereference
  StackIterator& operator++();             // prefix increment
  StackIterator operator++(int);           // postfix increment
  bool operator==( const StackIterator& aOtherIter ) const;
  bool operator!=( const StackIterator& aOtherIter ) const;

  StackIterator begin() const;      // new iterator (at first element)
  StackIterator end() const;        // new iterator (after last element)
};
```

This problem requires some extra considerations. We cannot compare `Stack` objects directly without destroying the stacks. This is not really a problem. We just demand that our stack iterator is being used consistently. That is, we do not mix stack iterators for different stacks. As a result, we only have to compare the respective stack sizes when defining **`operator==`** and **`operator!=`**. (We have used a similar approach when defining the `CharacterCounterIterator` and the `FibonacciIterator`).

What does it mean for a stack iterator to be positioned after the last element? The answer is straightforward. However, the solution must be consistent with the implementation of **`operator==`** and **`operator!=`**.

Complete the implementation of the template class `StackIterator`.

## Test harness:

```
Stack<string> lStack;

string s1( "One" );
string s2( "Two" );
string s3( "Three" );
string s4( "Four" );
string s5( "Five" );
string s6( "Six" );

lStack.push( s1 );
lStack.push( s2 );
lStack.push( s3 );
lStack.push( s4 );
lStack.push( s5 );
lStack.push( s6 );

cout << "Traverse elements" << endl;
StackIterator<string> iter( lStack );

for ( StackIterator<string> i = iter.begin(); i != i.end(); ++i )
{
  cout << "value: " << *i++ << endl;
}
```

## Result:

```
Traverse elements
value: Six
value: Four
value: Two
```

## Problem Extra Credit 1:

Using the template class `List` with proper copy control, implement the template class `Queue` as specified below:

```
#pragma once

#include "List.h"
#include <stdexcept>

template<class T>
class Queue
{
private:
  List<T> fElements;

public:
  bool isEmpty() const;
  int size() const;
  void enqueue( const T& aElement );
  void dequeue();
  const T& top() const;
};
```

That is, `Queue` is a queue container type that can grow in size on demand.

Complete the implementation of the template class `Queue`.

Test harness:

```
Queue<int> lQueue;

lQueue.enqueue( 1 );
lQueue.enqueue( 2 );
lQueue.enqueue( 3 );
lQueue.enqueue( 4 );
lQueue.enqueue( 5 );
lQueue.enqueue( 6 );

cout << "Queue elements:" << endl;

for ( ;!lQueue.isEmpty(); lQueue.dequeue() )
{
cout << "value: " << lQueue.top() << endl;
}
```

Result:

```
Queue elements:
value: 1
value: 2
value: 3
value: 4
value: 5
value: 6
```

## Problem Extra Credit 2:

Using the template class `Queue`, define a `QueueIterator` that is initialized with a `Queue` and provides a sequential (forward) access to all elements contained in the queue.

```
#pragma once

#include "Queue.h"

template<class T>
class QueueIterator
{
private:
  Queue<T> fQueue;

public:
  QueueIterator( const Queue<T>& aQueue );

  const T& operator*();                    // dereference
  QueueIterator& operator++();             // prefix increment
  QueueIterator operator++(int);           // postfix increment
  bool operator==( const QueueIterator& aOtherIter ) const;
  bool operator!=( const QueueIterator& aOtherIter ) const;

  QueueIterator begin() const;      // new iterator (at first element)
  QueueIterator end() const;        // new iterator (after last element)
};
```

The `QueueIterator` requires some extra considerations. First, we cannot compare `Queue` objects directly without destroying the queues. This is not really a problem. We just demand that our queue iterator is being used consistently, that is, we do not mix queue iterators for different queues and inspect the respective queue sizes (compare `top: DDDD`

```
top: CCCC
top: BBBB
size: 2
is empty: F
top: AAAA
is empty: T
```

Problem 3: `StackIterator`).

What does it mean for a queue iterator to be positioned after the last element? The answer is straightforward. However, the solution must be consistent with the implementation of **`operator==`** and **`operator!=`**.

Complete the implementation of the template class `QueueIterator`.

Test harness:

```
Queue<string> lQueue;

string s1( "One" );
string s2( "Two" );
string s3( "Three" );
string s4( "Four" );
string s5( "Five" );
string s6( "Six" );

lQueue.enqueue( s1 );
lQueue.enqueue( s2 );
lQueue.enqueue( s3 );
lQueue.enqueue( s4 );
lQueue.enqueue( s5 );
lQueue.enqueue( s6 );

cout << "Traverse queue elements" << endl;
QueueIterator<string> iter( lQueue );

for (QueueIterator<string> i = iter.begin(); i != i.end(); ++i )
{
  cout << "value: " << *i++ << endl;
}
```

Result:

```
Traverse queue elements
value: One
value: Three
value: Five
```

**Submission deadline:** **Tuesday, May 21, 2019, 10:30**

**Submission procedure:** **on paper (printout of list copy control and template classes for container types).**