Swinburne University of Technology

Faculty of Science, Engineering and Technology

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 2, Indexers, Method Overriding, and Lambdas
Due date: April 7, 2022, 14:30
Lecturer: Dr. Markus Lumpe

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

Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30

Marker's comments:

Problem	Marks	Obtained
1	48	
2	30+10= 40	
3	58	
Total	146	

Extension certification:

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

Signature of Convener:_____

Problem Set 2: Indexers, Method Overriding, and Lambdas

Problem 1

In this problem set, we define a simple integer vector class, called `IntVector`, that provides us with a container type for integer arrays. The class `IntVector` shares some similarities with the standard vector class `std::vector`, but we only define those features here that would allow us to practice indexers, method overriding, and lambda expression (being used latter to implement basic sorting algorithms).

The class `IntVector` defines the following interface:

```
#pragma once

// include for size_t (unsigned integral type)
#include <cstddef>

class IntVector
{
private:
  int * fElements;
  size_t fNumberOfElements;

public:

  // Constructor: copy argument array
  IntVector( const int aArrayOfIntegers[], size_t aNumberOfElements );

  // Destructor: release memory
  // Destructor is virtual to allow inheritance
  virtual ~IntVector();

  // size getter
  size_t size() const;

  // element getter
  const int get( size_t aIndex ) const;

  // swap two elements within the vector
  void swap( size_t aSourceIndex, size_t aTargetIndex );

  // indexer
  const int operator[]( size_t aIndex ) const;
};
```

The class `IntVector` may be extended by inheritance. For this reason, the destructor `~IntVector()` is marked virtual. This will allow for a proper dynamic method being of the destructor to prevent memory leaks.

The class `IntVector` just defines a wrapper for an array of integers. The wrapper adds, however, range checks so that index errors can be caught. The class does not expose the underlying array to clients or subclasses. Access to elements is read only. Nevertheless, we can change to order of elements via method `swap()`.

The constructor for `IntVector` takes an integer array and its size as parameters. The constructor copies this array as shown below:

```
IntVector::IntVector( const int aArrayOfIntegers[], size_t aNumberOfElements )
{
  fNumberOfElements = aNumberOfElements;
  fElements = new int[fNumberOfElements];

  for ( size_t i = 0; i < fNumberOfElements; i++ )
  {
    fElements[i] = aArrayOfIntegers[i];
  }
}
```

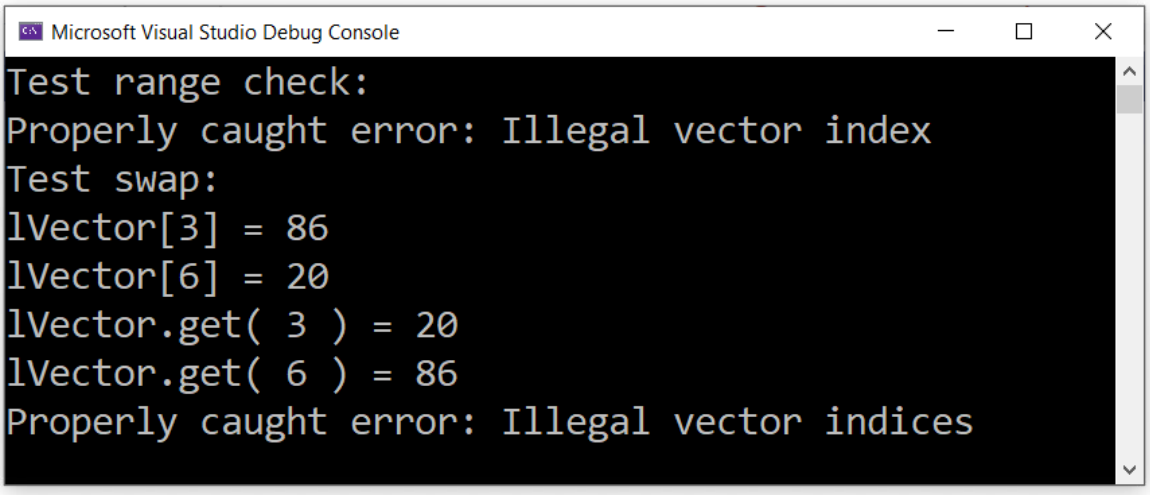
The destructor has to free the allocated memory and the member function `size()` has to return the number of elements in the array.

The member function `swap()` takes two indices and, if they are within range, swaps the corresponding array elements in an `IntVector` object. We need `swap()` for sorting.

The indexer **`operator[]`** and the method `get()` both return the value that corresponds to `aIndex`, if this is possible. Please note that both return a read-only value copy by design. This has not impact on performance, but requires us to use member function `swap()` when we wish to exchange array elements.

You should implement method `get()` using **`operator[]`**. This approach requires you to refer to "this object" explicitly. In C++, we write **`*this`** to mean "this object." You need to enclose **`*this`** is parentheses, that is, **`(*this)`**, to avoid any issues with operator priority.

You can use the test driver in Main.cpp (available on Canvas) to test your implementation. Please uncomment `#define P1` for this purpose. Running the program should produce the following output:



The test driver uses exception handling in order to verify range checks. The two error messages are expected here. No other error message should appear in the output though.

Problem 2

Implement *Bubble Sort*, that is, implement class `SortableIntVector` which is a public subclass of `IntVector`:

```
#pragma once

#include "IntVector.h"

#include <functional>

using Comparable = std::function<bool(int, int)>;

class SortableIntVector : public IntVector
{
public:
  SortableIntVector( const int aArrayOfIntegers[], size_t aNumberOfElements );

  virtual void sort( Comparable aOrderFunction );
};
```

Bubble Sort is simple quadratic-time complexity sorting algorithm. See Canvas for a pseudo code implementation. There is no need for a flag "is-sorted" even though some sources suggest so. There is limited if any improvement on the performance of the algorithm. Worse, it may even slow it down due to the extra tests necessary. See D.E. Knuth's comments on this matter.

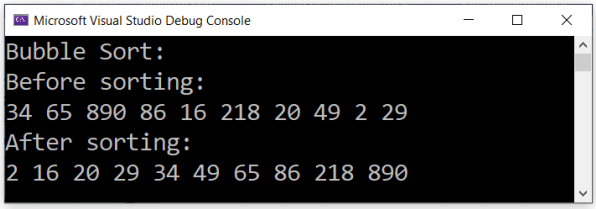
Class `SortableIntVector` is a subclass of `IntVector`. It defines a constructor and we need to use proper class-chaining to initialize objects of class `SortableIntVector`. Remember, the initialization of the super class requires a super class constructor call defined as member initializer in C++. Please note that you need to use `IntVector`'s `swap()` member function to exchange elements.

The method `sort()` implements *Bubble Sort*. We can sort in increasing or decreasing order. Here we wish to sort in increasing order. The method `sort()` takes as parameter a `Comparable` function. `Comparable` is a type alias for the standard function template `std::function<bool(int, int)>`. That is, `Comparable` is a Boolean function that takes two integer values and returns true, if the left integer goes before the right integer. Programmatically, the left integer goes before the right integer if the value of the left integer does not exceed the value of the right integer.

To provide a matching function for `Comparable`, you need to define a lambda expression, an anonymous function object representing a callable unit of code, when calling the `sort()` method in `main()` for Problem 2:

```
// Use a lambda expression here that orders integers in increasing order.
// The lambda expression does not capture any variables of throws any exceptions.
// It has to return a bool value.
lVector.sort( /* lambda expression */ );
```

You can use the test driver in Main.cpp (available on Canvas) to test your implementation. Please uncomment `#define P2` for this purpose. Running the program should produce the following output:



Problem 3

Implement *Cocktail Shaker Sort*, that is, implement class `ShakerSortableIntVector` which is a public subclass of `SortableIntVector`:

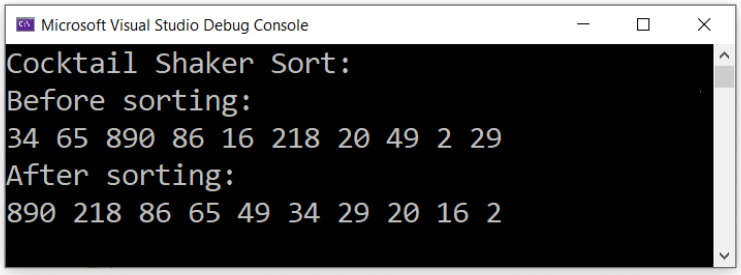
```
#pragma once

#include "SortableIntVector.h"

class ShakerSortableIntVector : public SortableIntVector
{
public:

  ShakerSortableIntVector( const int aArrayOfIntegers[], size_t aNumberOfElements );

  void sort( Comparable aOrderFunction = [] (int aLeft, int aRight)
                                          { return aLeft <= aRight; } ) override;
};
```

Cocktail Shaker Sort is bidirectional *Bubble Sort*. See Canvas for a pseudo code implementation. There is no need for a flag "is-sorted" even though some sources suggest one. There is limited if any improvement on the performance of the algorithm. Worse, it may even slow it down due to the extra tests necessary. See D.E. Knuth's comments on this matter.

The implementation of *Cocktail Shaker Sort* can be achieved solely by implementing the `sort()` method and using its default implementation for `aOrderFunction`. Please note that you need to use `IntVector`'s `swap()` member function to exchange elements.

There is only one `Comparable` function. However, it suffices to implement the bidirectional sorting process. Analyze carefully its behavior and interaction with *Cocktail Shaker Sort* to devise a proper solution. The implementation must sort the elements in decreasing order.

You can use the test driver in Main.cpp (available on Canvas) to test your implementation. Please uncomment `#define P3` for this purpose. Running the program should produce the following output:



The solution for all problems requires 120-140 lines of low density C++ code.

Submission deadline: Thursday, April 7, 2022, 14:30.

Submission procedure: PDF of printed code for **`IntVector`**, **`Main_PS2`**, and **`SortableVector`**, and **`ShakerSortableVector`**.