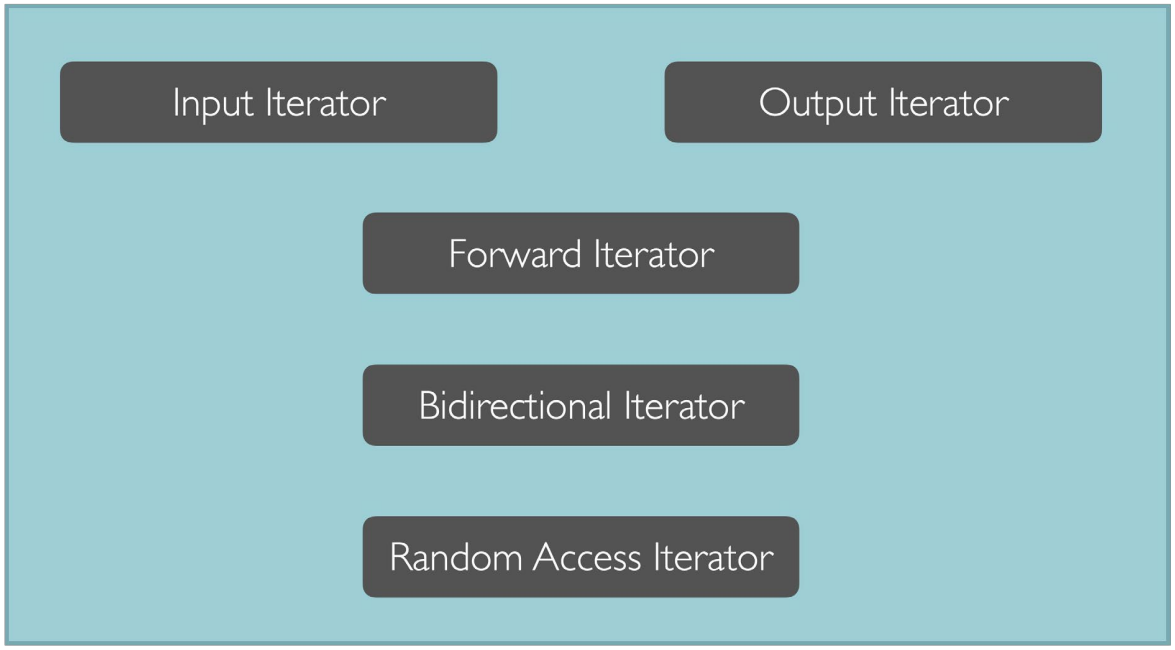
Swinburne University of Technology

School of Science, Computing and Engineering Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	6, Iterators
Lecturer:	Ms. Siti Hawa



Preliminaries

Consider the class `DataWrapper` that we developed in tutorial 5. It encapsulates an array of key-value pairs and provides an indexer to fetch the pair that corresponds to a given index:

```
#pragma once

#include "Map.h"

#include <string>

using DataMap = Map<size_t, size_t>;

class DataWrapper
{
private:
  size_t fSize;
  DataMap* fData;

public:
  DataWrapper();
  ~DataWrapper();

  // DataWrapper object are not copyable
  DataWrapper( const DataWrapper& ) = delete;
  DataWrapper& operator=( const DataWrapper& ) = delete;

  bool load( const std::string& aFileName );

  size_t size() const noexcept;

  const DataMap& operator[]( size_t aIndex ) const;
};
```

We used an object of this class to load a file named `Data.txt`, which contains a series of randomized key-value pairs. We can access every pair via `DataWrapper`'s index operator using, for example, a **for**-loop. Using a simple **for**-loop, however, produces output that is not legible:

```
.`
 `
  '\/ `    \ ,.
   o  - /    d \  _    -| _ '    \;  _ :
  / .           -   |o-     \   ' -',  "  | |     )       ` ' ,8             `-  ,
-  \ '   . ,        |  .`    `  ,_      .,'
    .        ._   `   ` .
…
```

The key-value pairs are out of order. To fix this, we defined a lambda expression that sorted the pairs. As result, we obtained output that was easy to understand or recognize.

In this tutorial, we wish to experiment with iterators. In particular, we shall create two forward iterators that provide a systematic access to all key-value pairs in a `DataWrapper` object. The first iterator simply traverses the key-value pairs in the order stored. The second iterator additionally sorts the key-value pairs on-the-fly to procedure an ordered traversal of the key-value pairs.

The implementations of the iterators require approx. 60-80 lines of low density C++ code each.

Conditional Compilation

The test driver provided for this tutorial task makes use of conditional compilation via preprocessor directives. This allows you to focus only on the task you are working on.

Conditional code is enclosed in

```
#ifdef PX
…
#endif
```

preprocessor conditional directive. If the variable `PX` is defined, then the enclosed code becomes part of the solution and. Otherwise, it is ignored.

This tutorial comprises of two problems. 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

We start with a basic forward iterator for `DataWrapper` objects. The design of this iterator follows the principles shown in class with the exception that we define the iterator as a standard class rather than a template. The specification of the basic forward iterator is given below:

```
#pragma once

#include "DataWrapper.h"

class SimpleForwardIterator
{
private:
  const DataWrapper* fCollection;
  size_t fIndex;

public:
  SimpleForwardIterator( const DataWrapper* aCollection ) noexcept;

  const DataMap& operator*() noexcept;

  SimpleForwardIterator& operator++() noexcept;       // prefix
  SimpleForwardIterator operator++(int) noexcept;     // posfix

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

  SimpleForwardIterator begin() const noexcept;
  SimpleForwardIterator end() const noexcept;
};
```

The iterator uses two member variables: `fCollection` which is a raw pointer to an object of type `DataWrapper` and `fIndex` which is an unsigned integer that represents to current iterator position. We use a raw pointer for `fCollection` to avoid value copies and to facilitate the comparison of two iterators. Using raw pointers is not considered a good practice in general as raw pointers are non-specific about ownership of objects. We will revisit this issue later in the unit and learn how to use smart pointers to explicitly manage ownership of objects. In this tutorial, however, we are not concerned about ownership and the use of a raw pointer does not cause any issues.

The implementation of `SimpleForwardIterator` is standard in the sense that we use and update the member variable `fIndex` to access the current element and advance the iterator position, respectively.

The auxiliary methods are required to use the iterator in a **for**-range loop. The iterator has reached the end when its index equals the number of elements in the collection.

To test your implementation of `SimpleForwardIterator`, uncomment **`#define`** `P1` and compile your solution. Using `Data_1.txt`, the output should match that obtained in Problem 2 of tutorial 4.

You can also use `Data_2.txt` and `Data_3.txt` to further test your program.

Problem 2

We now focus our attention on a forward iterator for `DataWrapper` objects that orders the data elements. Again, the design of this iterator follows the principles shown in class with the exception that we define the iterator as a standard class rather than a template. However, we include an extra member variable, `fMapPosition`, and a private member function `setMapPosition()` to map the current iterator index to the correct, ordered datum in the sequence of the underlying collection (i.e., a `DataWrapper` object). The specification of the ordering forward iterator is given below:

```
#pragma once

#include "DataWrapper.h"

class OrderingForwardIterator
{
private:
  const DataWrapper* fCollection;
  size_t fIndex;
  size_t fMapPosition;

  void setMapPosition();

public:
  OrderingForwardIterator( const DataWrapper* aCollection ) noexcept;

  const DataMap& operator*() noexcept;

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

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

  OrderingForwardIterator begin() const noexcept;
  OrderingForwardIterator end() const noexcept;
};
```

This iterator implicitly orders the data elements. The "sorting process" is implemented in the private member function `setMapPosition()`. Similar to the lambda expression we implemented in tutorial 5, the function `setMapPosition()` has to find the key-value pair whose key component equals the current iterator index and sets the `fMapPosition` to the index of that pair. This has to happen three times: in the constructor, in the prefix increment operator, and in `begin()`. Also, when accessing the current element, the iterator is positioned on, we use `fMapPosition` to select the element now.

The implementation of `OrderingForwardIterator` is standard in the sense that we use and update the member variable `fIndex` and `fMapPosition` to access the current element and advance the iterator position, respectively.

The auxiliary methods are required to use the iterator in a **for**-range loop. The iterator has reached the end when its index equals the number of elements in the collection.

To test your implementation of `OrderingForwardIterator`, uncomment **`#define`** `P2` and compile your solution. Using `Data_1.txt`, the output should match that obtained in Problem 3 of tutorial 5.

You can also use `Data_2.txt` and `Data_3.txt` to further test your program.