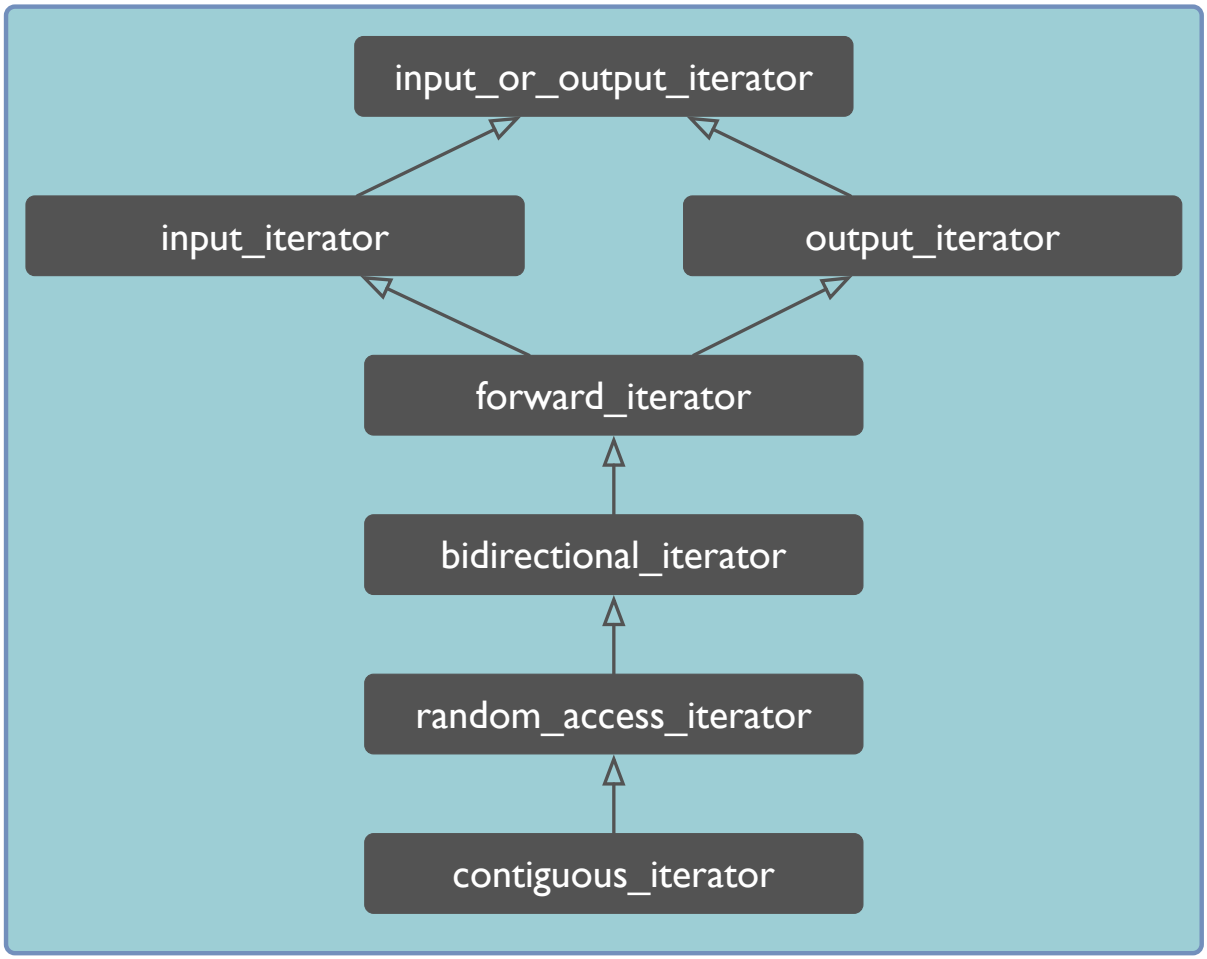
Swinburne University of Technology

School of Science, Computing and Emerging Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	5, Iterators
Lecturer:	Dr. Markus Lumpe



Preliminaries

Consider the `DataWrapper` class we created in Tutorial 4. It encapsulates an array of key-value pairs and provides an indexer to retrieve the pair corresponding 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 objects 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 populated the state of a `DataWrapper` object using the `load()` method. The input file "`Data.txt`" contained a series of randomized key-value pairs. To access a specific pair, we can use `DataWrapper`'s index operator in a **`for`** loop, for instance, and output its value component rendered as a character to the console. Without ordering the key-value pairs, the result is not legible:

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

The key-value pairs are out of order. To fix this, we must order the pairs first. In Tutorial 4, we employed an on-the-fly process whose behavior is similar to *Bubble Sort*. The result 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 systematic access to all key-value pairs in a `DataWrapper` object. The first iterator 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 implementation of the iterators requires approx. 60-80 lines of C++ code each.

Conditional Compilation

The test drivers for this tutorial task use conditional compilation controlled via preprocessor directives. This allows you to focus only on the task you are working on.

The conditional code is enclosed in a

```
#ifdef PX
…
#endif
```

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

This tutorial comprises three problems. The test driver (i.e., `Main.cpp`) uses `P1`, `P2`, and `P3` as variables to enable/disable the test associated with a corresponding problem. To enable a test uncomment the respective `#define` line. For example, to test problem 2 only, enable `#define P2`:

```
//#define P1
#define P2
//#define P3
```

In Visual Studio and Xcode, 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.

Problem 1

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

```
#pragma once

#include <concepts>

#include "DataWrapper.h"

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

public:

  // C++20 iterator type properties
  using iterator = SimpleForwardIterator;
  using difference_type = std::ptrdiff_t;
  using value_type = DataMap;

  // Iterator must be default initializable
  SimpleForwardIterator( const DataWrapper* aCollection = nullptr ) noexcept;

  // Iterator must be const
  const value_type& operator*() const noexcept;

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

  // Iterator must implement operator==. The operation != is synthesized.
  bool operator==( const iterator& aOther ) const noexcept;

  // Auxiliary methods not part of std::forward_iterator.
  iterator begin() const noexcept;
  iterator end() const noexcept;
};

static_assert(std::forward_iterator<SimpleForwardIterator>);
```

The iterator uses two member variables: `fCollection`, a raw pointer to an object of type `DataWrapper`, and `fIndex`, an unsigned integer capturing the 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 manage explicitly ownership of objects. In this tutorial, however, we are not concerned about ownership and raw pointers do not cause any issues.

The definition of this iterator is standard. Remember, an iterator behaves like a pointer. It refers to an element and can be changed using pointer arithmetic.

There is no need for assertion checking. Iterators must be tested before using the dereference operation on iterators. However, there may not be an underlying collection in the current scenario. The pointer `fCollection` can be **`nullptr`**. In method `end()`, we must inspect this pointer to determine the correct end position.

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 test your program further.

Problem 2

We now focus on a forward iterator for `DataWrapper` objects that orders the data elements. Again, the design of this iterator follows the principles shown in the lecture, except that we define the iterator as a standard class rather than a template. The specification of the simple forward iterator is given below:

```
#pragma once

#include <concepts>
#include <functional>

#include "DataWrapper.h"

class OrderingForwardIterator
{
public:

  using Sorter = std::function<const DataMap&(const DataWrapper&, size_t)>;

  // C++20 iterator type properties
  using iterator = OrderingForwardIterator;
  using difference_type = std::ptrdiff_t;
  using value_type = DataMap;

  // Iterator must be default initializable
  OrderingForwardIterator( const DataWrapper* aCollection = nullptr,
                           Sorter aOrdering = [] ( const DataWrapper& aCollection,
                                                   size_t aIndex ) -> const DataMap&
                                    {
                                        return aCollection[aIndex];
                                    } ) noexcept;

  // Iterator must be const
  const value_type& operator*() const noexcept;

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

  // Iterator must implement operator==. The operation != is synthesized.
  bool operator==( const iterator& aOther ) const noexcept;

  // Auxiliary methods not part of std::forward_iterator.
  iterator begin() const noexcept;
  iterator end() const noexcept;

private:

  const DataWrapper* fCollection;
  Sorter fOrdering;
  size_t fIndex;
};

static_assert(std::forward_iterator<OrderingForwardIterator>);
```

This declaration is identical to the one in Problem 1, except for two aspects. First, the **`private`** member variable section occurs at the end of the class declaration. We require an extra member variable, `fOrdering`. Its type is `Sorter` which must be defined first in the public interface of `OrderingForwardIterator`. The member variable fOrdering represents a lookup function that maps the iterator position, `fIndex`, to a corresponding key-value pair in `fCollection`.

Second, the class `OrderingForwardIterator` defines a function type `Sorter`. Functions of this type map the iterator position `aIndex` in the underlying collection `aCollection` to a

corresponding key-value pair of type `DataMap`. We must use the template `std::function` to define the signature of the ordering function. The term

```
const DataMap&(const DataWrapper&, size_t)
```

defines a function that takes two arguments, a constant reference or a `DataWrapper` object and a value of type `size_t`, and returns a constant reference to a `DataMap` object. More precisely, we use call-by-reference to pass the collection and call-by-value to pass the iterator position to an ordering function that returns a constant l-value reference to a `DataMap` object.

A suitable ordering function is passed as a second argument to the constructor. We define a default function, as a lambda expression, that performs the identity mapping. The lambda expression

```
[] ( const DataWrapper& aCollection,
     size_t aIndex ) -> const DataMap&
{
     return aCollection[aIndex];
}
```

returns simply the key-value pair corresponding to `aIndex` in `aCollection`. The function does not perform any ordering. It defines a simple lookup of values.

The function `fOrdering` is used in the dereference operator to return the mapped element.

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

Problem 3

Enable the third test by uncommenting the line **`#define`** `P3` in `Main.cpp`.

In the `main` function, the test driver for Problem 3 becomes available. Look for the text

```
auto lOrdering = [] () -> const DataMap&
                    {

                    };

testP3( lWrapper, lOrdering );
```

This code fragment defines a lambda expression, `lOrdering`, and call function `testP3()`.

Correct the signature of the lambda expression and define a linear search within its body that maps the iterator index to the corresponding key-value pair in the collection. We have found the match if the iterator index equals the key of a key-value pair.

Compile your solution. Using `Data_1.txt`, the output is legible. You can also use `Data_2.txt` and `Data_3.txt` to test your program further.