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:	6, Templates, Inheritance, and Algorithm Analysis
Lecturer:	Dr. Markus Lumpe

Time cools, time clarifies; no mood can be maintained quite unaltered through the course of hours.

Mark Twain

Templates, Inheritance, and Algorithm Analysis

In this tutorial, we will implement two sorting algorithms and analyze their complexity based on the number of element exchanges these algorithms generate for a given input of size n. In particular, we use *Bubble Sort* and *Sorting by Fisher & Yates* (Sorting known as *BogoSort*). Concerning their running time, the former is $O(n^2)$, whereas the latter is O(n*n!). But what does this mean in practice? Donald Knuth's assessment of *Bubble Sort* is: "*… bubble sort seems to have nothing to recommend it, except a catchy name…*". This is a damning assessment. What about *Sorting by Fisher & Yates*? Would it fare any better? The *Sorting by Fisher & Yates* algorithm repeatedly shuffles a given collection until it is sorted. In other words, *Sorting by Fisher & Yates* permutates a collection until it discovers a sorted permutation and keeps shuffling until it finds one, possibly even checking the same permutation more than once. We can think of this process as repeatedly throwing a deck of cards into the air and, at one point, it lands back into our hands as a sorted deck. Suddenly, *Bubble Sort* looks appealing.

To run the experiments, we define three data types in this tutorial:

- A template class `ArraySorter` that provides the core infrastructure for sorting algorithms,
- A template class `BubbleSorter`, a subclass of `ArraySorter` that implements *Bubble Sort*, and
- A template class `FisherAndYatesSorter`, a subclass of `ArraySorter` that implements *Sorting by Fisher & Yates*.

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 **`#if defined`** `(PX)` … **`#endif`** are grayed out if the corresponding test is disabled. The preprocessor definition **`#if defined`** `(PX)` … **`#endif`** enables conditional compilation.

Templates and Inheritance

Using inheritance while defining template classes can result in subtle compile time issues that arise when we wish to call base class methods. In particular, we must explicitly qualify the receiver object by the prefix `this->` to call base class methods. Otherwise, the compiler reports errors.

In non-template classes, a base class method named `f` can be called using the expression `f()` in a subclass method. The compiler resolves name `f` to a base class method automatically as the name `f` occurs within the scope of the subclass.

The situation is different for templates. C++ distinguishes between dependent and non-dependent names in templates. The meaning of a name may depend on types of type template parameters and values of non-type template parameters. The lookup of those names is postponed until the template parameters are known. A non-dependent name, on the other hand, is resolved at the point of template definition. Referring to a base class method `f` solely by expression `f()` makes `f` a non-dependent name. The compiler will only check the current template class. The name `f` is not defined in the current template class, as it refers to a base class feature. Qualifying the expression `f()` with `this->` makes the expression `f()` dependent on the template arguments. The compiler now defers the lookup of the method named `f` until the template is instantiated and reports an error only if expression `f()` is ill-formed under template specialization.

Problem 1

We start with the template class `ArraySorter`. Technically, this data type does not implement any sorting algorithm. It is a container type that maintains a copy of the array to be sorted and provides methods to access array elements and swap two array elements. In addition, it declares a suitable output operator as a friend.

The template class `ArraySorter` is defined as follows:

```
#pragma once

#include <cstdint>
#include <ostream>
#include <cassert>

template<typename T>
class ArraySorter
{
public:

  ArraySorter( const T aArray[] = nullptr, size_t aSize = 0 ) noexcept;

  virtual ~ArraySorter() noexcept;

  size_t size() const noexcept;
  size_t swapCount() const noexcept;

  void swap( T& aLeft, T& aRight ) noexcept;

  T& operator[]( size_t aIndex ) const noexcept;

  virtual void sort( bool aPrintIntermitted = false,
                     std::ostream& aOStream = std::cout ) noexcept;

  friend std::ostream& operator<<( std::ostream& aOStream,
                                   const ArraySorter& aSorter );

private:

  T* fElements;
  size_t fSize;
  size_t fSwapCount;
};
```

Objects of template class `ArraySorter` maintain a private copy of the array to be sorted. The constructor allocates heap memory and copies the argument array. The destructor has to free the heap memory.

The getter `size()` and `swapCount()` return the number of elements in the underlying array and the number of swap operations performed during sorting, respectively.

Access to the elements is only via the public indexer or exchange elements via method `swap()`. The latter must increment `fSwapCount`.

The `sort()` method does not implement any sorting algorithm. However, it sets `fSwapCount` to zero.

The output operator must assign a textual representation to the underlying array. Use the approach that we developed in Tutorials 1 and 4.

To test your implementation of `ArraySorter`, uncomment **`#define`** `P1` and compile your solution. The test driver should produce the following output:

```
Testing ArraySorter.
The array before sorting:
[34,2,890,40,16,218,20,49,10,29]
Number of swaps: 0
The array after sorting:
[34,2,890,40,16,218,20,49,10,29]
```

Problem 2

We now wish to implement *Bubble Sort*, that is, we define the template class `BubbleSorter` as a subclass of the template class `ArraySorter` as shown below:

```
#pragma once

#include "ArraySorter.h"

template<typename T, typename Order = std::greater<T>>
class BubbleSorter : public ArraySorter<T>
{
private:

  Order fOrderFtn;

public:

  BubbleSorter( const T aArray[] = nullptr, size_t aSize = 0 ) noexcept;

  void sort( bool aDoLog = false,
             std::ostream& aOStream = std::cout ) noexcept override;
};
```

The template class `BubbleSorter` is a subclass of `ArraySorter`. It has an extra type template parameter that captures the desired sorting criterion. By default, we sort the array in decreasing order.

The constructor must forward the parameters to the base class constructor and initialize the member variable `fOrderFtn`.

The method `sort()` implements *Bubble Sort*. It must call its corresponding base class method to reset the swap count. In addition, the method `sort()` must intermittently send the underlying array to the output stream `aOStream` if `aDoLog` is **`true`**.

To test your implementation of `BubbleSorter`, uncomment **`#define`** `P2` and compile your solution. The test driver should produce the following output:

```
Testing BubbleSorter.
The array before sorting:
[34,2,890,40,16,218,20,49,10,29]
Number of swaps: 21
The array after sorting:
[890,218,49,40,34,29,20,16,10,2]
```

Problem 3

Consider an array with 10 elements:

```
int lArray[] = { 45, 34, 8, 6, 4, 1, 0, -2, -3, -100};
```

How can we sort this array, so that the elements are arranged in increasing order?

Sorting by Fisher and Yates:

This sorting method uses the Fisher and Yates shuffling. This technique exploits the fact that shuffling can produce a sorted array. For an array with 10 elements the odds of obtaining a sorted array through shuffling are 1 in 3,628,800 or 0.000000275573192. Even though these odds are not very good, they are still better than hitting the jackpot in the lottery. Moreover, the "*randomness of nature*" dictates that we may reach a sorted array much faster. But we also need to be prepared to wait longer than expected. Nevertheless, it is a fundamental property of randomness that a "possible" event, even a very improbable one, occurs eventually. Fisher and Yates shuffling is defined as follows:

let n := N

while n > 1 **do**

  let k := *rand*() % n k is a random index between 0 and n-1

  n := n – 1

  A[n] :=: A[k]

A C++ solution for sorting by Fisher and Yates requires the proper initialization and use of a pseudo-random number generator. We can use:

```
std::srand( static_cast<unsigned int>( std::time( NULL ) ) );
```

to seed the C++ random number generator with the current time since January 1, 1970, in seconds. The function `std::srand()` is defined in `cstdlib`, whereas `std::time()` is defined in `ctime`. To obtain the subsequent random number we call `std::rand()` which yields an integral number between 0 and `RAND_MAX`. We limit this number to a specific range using the modulus operation. For example, `std::rand()%n` yields a random number between 0 and n-1.

We define Fisher and Yates shuffling as a private member function. The Fisher and Yates sorter operates as follows:

while true

  output A if necessary

  if isSorted(A)

    break

  shuffle(A)

We can define sorting by Fisher & Yates in a template class `FisherAndYatesSorter` that is a subclass of the template class `ArraySorter` as shown below:

```
#pragma once

#include "ArraySorter.h"

#include <cstdlib>
#include <ctime>

template<typename T, typename Order = std::greater<T>>
class FisherAndYatesSorter : public ArraySorter<T>
{
private:

  Order fOrderFtn;

  void shuffle() noexcept;

  bool isSorted() noexcept;

public:

  FisherAndYatesSorter( const T aArray[] = nullptr,
                        size_t aSize = 0 ) noexcept;

  void sort( bool aDoLog = false,
             std::ostream& aOStream = std::cout ) noexcept override;
};
```

The template class `FisherAndYatesSorter` is a subclass of `ArraySorter`. It has an extra type template parameter that captures the desired sorting criterion. By default, we sort the array in decreasing order.

The constructor must forward the parameters to the base class constructor and initialize the member variable `fOrderFtn`. In addition, it must initialize the random number generator. Technically, we do not initialize the random number generator but seed it.

The method `sort()` implements *Sorting by Fisher & Yates*. It must call its corresponding base class method to reset the swap count. In addition, the method `sort()` must intermittently send the underlying array to the output stream `aOStream` if `aDoLog` is **`true`**. The implementation of `sort()` relies on methods `isSorted()` and `shuffle()`. The `isSorted()` method defines a simple loop that tests if the underlying array is sorted. We can use `fOrderFtn` for this purpose but negate the result.

To test your implementation of `FisherAndYatesSorter`, uncomment **`#define`** `P3` and compile your solution. The test driver should produce the following output:

```
Testing FisherAndYatesSorter.
The array before sorting:
[34,2,890,40,16,218,20,49,10,29]
Number of swaps: 26181441
The array after sorting:
[890,218,49,40,34,29,20,16,10,2]
```

26,181,441 swap operations have occurred. 10*10! = 36,288,000. The algorithm has found the sorted array faster. It is, however, only one possible run.

Can you sort an array with 11, 12, or 13 elements?

Which sorting algorithm is better? Which one would you use in practice?