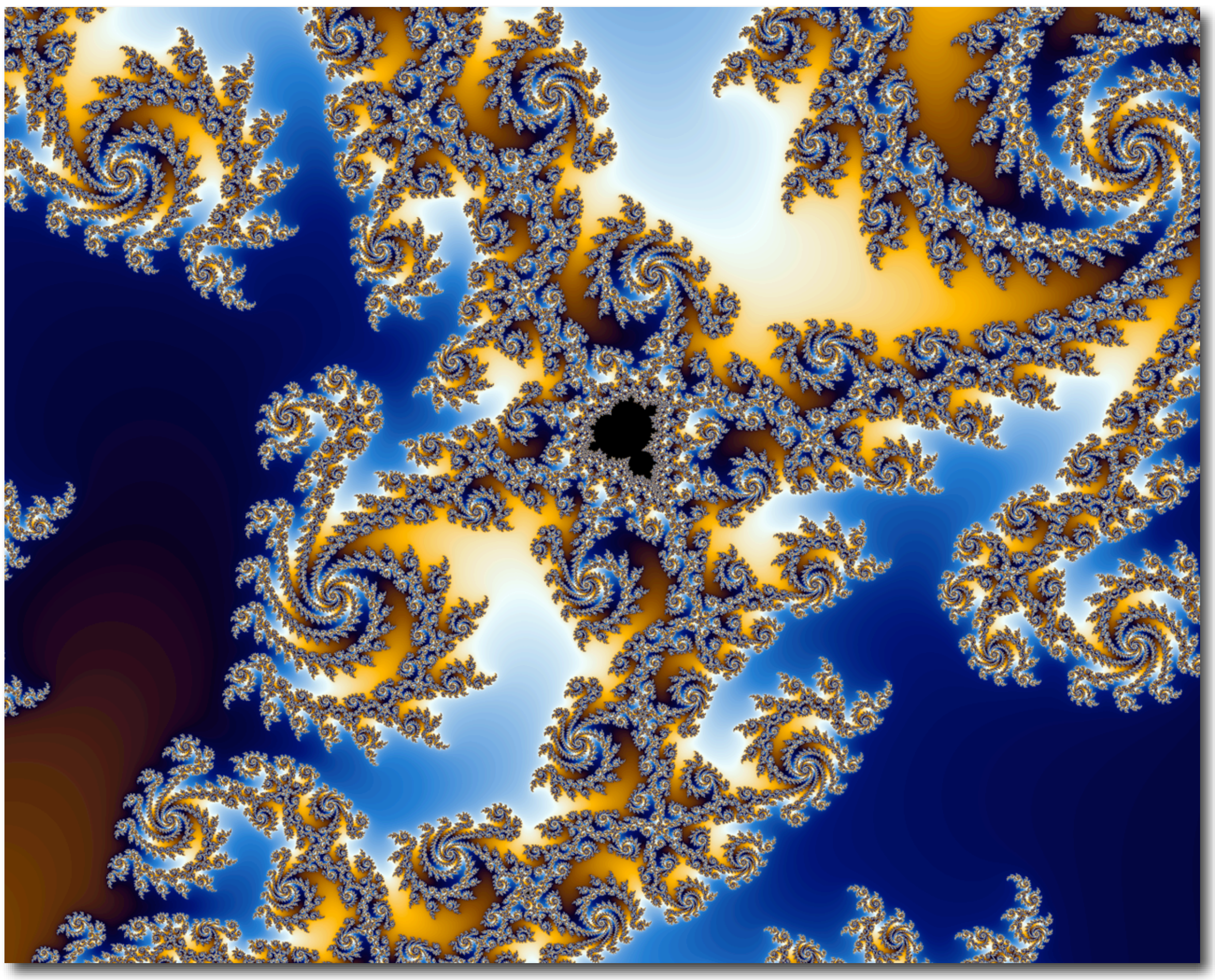
Swinburne University Of Technology

Faculty of Information and Communication Technologies

LABORATORY COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Lab number and title:	4, Iterators
Lecturer:	Dr. Markus Lumpe



Problem 1

Consider the program `FileCharacterCounter` that we developed in tutorial 3. It analyzed a given text file and recorded the frequencies of the individual characters occurring in that file and saved the statistics in an output text file. We used an array and standard array access (within a for loop) to write the individual frequencies to an output stream. Moreover, in order to remove clutter (irrelevant information) we had to define a guard to filter zeros from the result.

In this tutorial, we wish to experiment with iterators. In particular, we shall create an iterator for `CharacterCounter`, called `CharacterCounterIterator`, that implements the standard interface for a C++ forward iterator. As an added feature, this iterator automatically filters all zero occurrences, hence the iterator deference operator only returns non-zero elements.

Unfortunately, this task is more complex than it appears. First, when using an iterator to print the character frequencies we need to map a given frequency to its corresponding character. This was easy, when we used an array and a for-loop. The index variable could be converted into a `char` value (see solution for tutorials 2 and 3). To recover this feature, we would have to maintain an extra variable. The result would become rather clumsy and counterintuitive to the elegance offered by iterators. We need to find a better solution. We need a new data type – `FrequencyMap` – which establishes the required association between a character and its number of occurrences. The following class specification suggests a possible solution:

```
class FrequencyMap
{
private:
  char fChar;
  int fFrequency;

public:
  FrequencyMap();   // default constructor needed when used as base type of arrays
  FrequencyMap( char aChar, int aFrequency ); // initialize with concrete values

  // read-only getters
  char getCharacter() const;      // retrieve character
  int getFrequency() const;       // retrieve frequency
};
```

We need to add a new method to class `CharacterCounter` in order to populate the map:

```
#pragma once

#include <iostream>

class CharacterCounter
{
private:
  int fTotalNumberOfCharacters;
  int fCharacterCounts[256];

public:
  CharacterCounter();

  void count( unsigned char aCharacter );      // We count all 256 byte values

  friend std::ostream& operator<<( std::ostream& aOStream,
                                   const CharacterCounter& aCharacterCounter );

  // new tutorial 4: frequency indexer method
  int operator[]( unsigned char aCharacter ) const;
};
```

We can now define a `CharacterCounterIterator`:

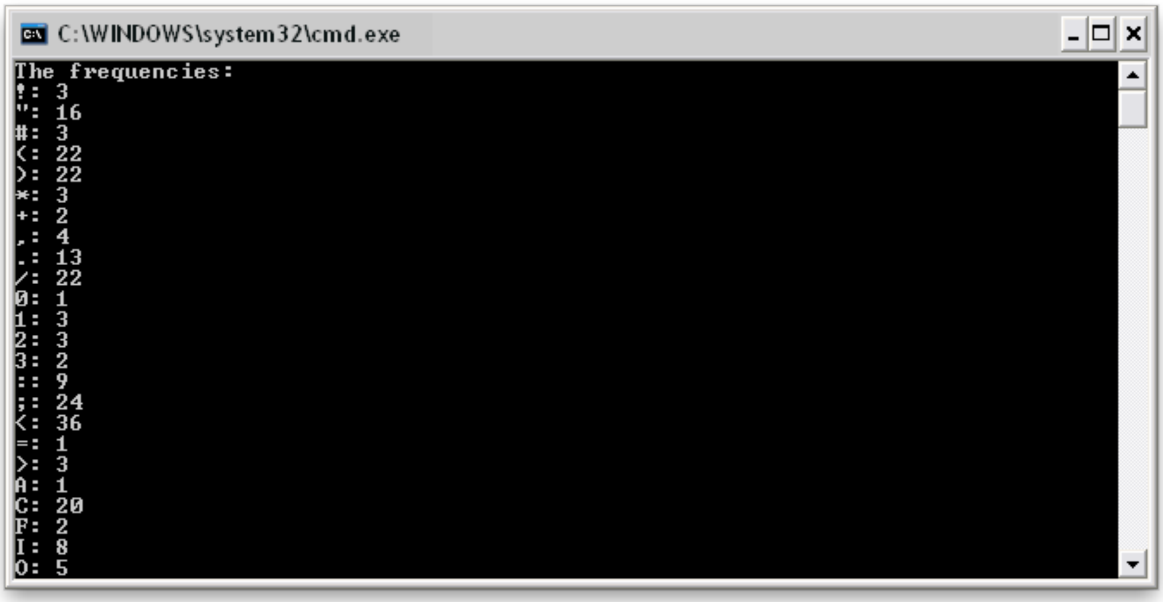
```
#include "CharacterCounter.h"
#include "FrequencyMap.h"

class CharacterCounterIterator
{
private:
  FrequencyMap fMaps[256];
  int fIndex;

public:
  CharacterCounterIterator( CharacterCounter& aCounter );

  const FrequencyMap& operator*() const;       // return current frequency map
  CharacterCounterIterator& operator++();      // prefix
  CharacterCounterIterator operator++( int );  // postfix (extra unused argument)
  bool operator==( const CharacterCounterIterator& aOther ) const;
  bool operator!=( const CharacterCounterIterator& aOther) const;

  CharacterCounterIterator begin() const;
  CharacterCounterIterator end() const;
};
```

The specification captures a standard C++ forward iterator. It takes a `CharacterCounter` object and provides access to non-zero frequency `FrequencyMap` objects. The iterator preserves the original array order, that is, the iterator returns `FrequencyMap` objects in the sequence determined by the ASCII encoding of characters.

In order to test the iterator, use the following approach

```
CharacterCounter lCounter;

unsigned char lChar;

while ( lInput >> lChar )
{
    lCounter.count( lChar );
}

lOutput << lCounter;

// test iterator
cout << "The frequencies: " << endl;
for ( CharacterCounterIterator iter( lCounter ); iter != iter.end(); iter++ )
{
    cout << (*iter).getCharacter() << ": " << (*iter).getFrequency() << endl;
}
```

Applied to the `Main.cpp` file this code should produce to following console output:



```
The frequencies:
!: 3
": 16
#: 3
(: 22
): 22
*: 3
+: 2
,: 4
.: 13
/: 22
0: 1
1: 3
2: 3
3: 2
:: 9
;: 24
<: 36
=: 1
>: 3
A: 1
C: 20
F: 2
I: 8
O: 5
```

Implement the approach. But wait, before proceed develop a plan, that is, analyze the problem in depth, identify the unknowns, check the C++ reference and DSP lecture notes for suitable solution scenarios. You must not write a single line of code prior finishing the problem analysis.

Take 15-20 minutes to sketch out a plan/solution on **paper**. There are a few hidden issues. The existing code does not work immediately. We have to adjust it in order to achieve fitness.

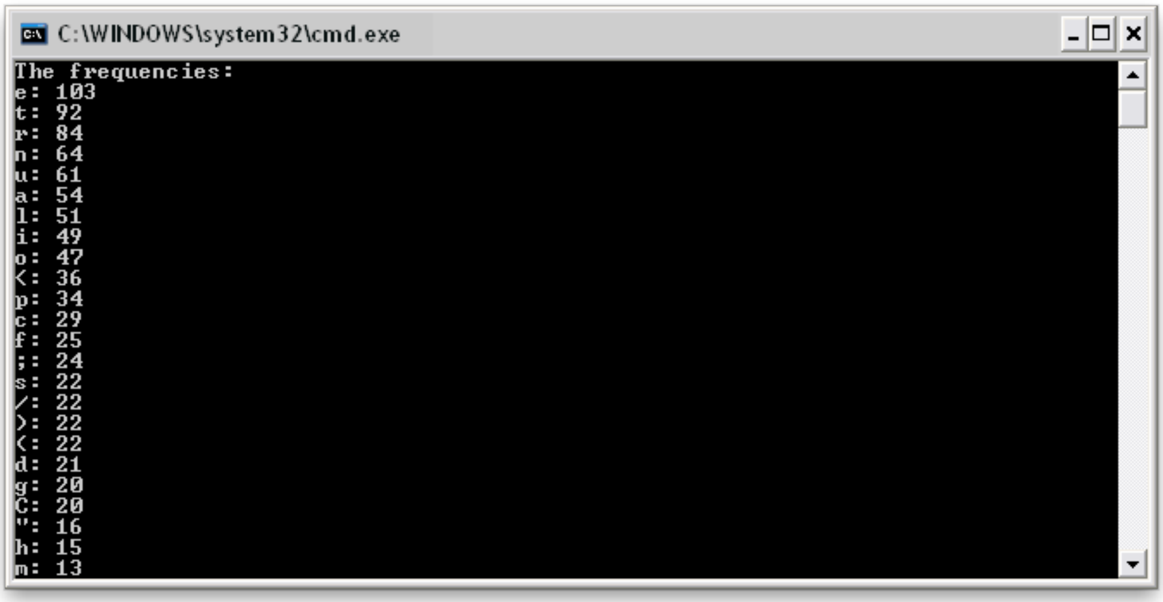
Once we understand all the requirements and possible issues of the project, we can start building the solution.

Problem 2

An added bonus of using iterators lies in their ability to provide us with different facets of data traversal in a uniform way. The following specification illustrates this vividly.

```
#include "CharacterCounter.h"
#include "FrequencyMap.h"

class SortedCharacterCounterIterator
{
private:
  FrequencyMap fMaps[256];
  int fIndex;

public:
  SortedCharacterCounterIterator( CharacterCounter& aCounter );

  const FrequencyMap& operator*() const;     // return current frequency map
  SortedCharacterCounterIterator& operator++();      // prefix
  SortedCharacterCounterIterator operator++( int );  // postfix (extra unused argument)
  bool operator==( const SortedCharacterCounterIterator& aOther ) const;
  bool operator!=( const SortedCharacterCounterIterator& aOther) const;

  SortedCharacterCounterIterator begin() const;
  SortedCharacterCounterIterator end() const;
};
```

The class SortedCharacterCounterIterator is a forward iterator variant, which provides access to ordered `FrequencyMap` objects. These objects are sorted from the highest to the lowest non-zero frequency.



```
The frequencies:
e: 103
t: 92
r: 84
n: 64
u: 61
a: 54
l: 51
i: 49
o: 47
<: 36
p: 34
c: 29
f: 25
;: 24
s: 22
/: 22
>: 22
<: 22
d: 21
g: 20
C: 20
": 16
h: 15
m: 13
```

As a result, we can now see immediately, which character occurs the most in a given input text file. We do not have to change the main function at all (except using the new iterator).

Nevertheless, there are some challenges to tackle. First, how do we sort the character frequencies and second, how can we implement the iterator most effectively.

As it turns out, we can also play with different techniques to achieve the desired effects. One way is to use "bubble sort" to arrange the frequency map in increasing order (i.e., from the smallest to the largest). This means, the `FrequencyMap` object with highest frequency will be stored as the rightmost element (index 255) in the `fMaps` array. To return the proper sequence, the increment operators have to run from right to left (aka, use decrement internally). This is to demonstrate and important data type concept: abstraction.

The same interface can have different implementations. There is no single prescribed solution. Often reformulating a given problem can make it easier (or more elegant) to yield the desired outcome.

Implement `SortedCharacterCounterIterator` and test its fitness.

Bubble sort works as follows:

```
for i = n - 1 down to 1
  do for j = 0 to i - 1
       do if array[j] > array[j+1] then
            temp = array[j];
            array[j] = array[j+1];
            array[j+1] = temp;
          end
     end
end
```

In other words, the highest value "bubbles" from left to right. Once the highest element has been placed in the rightmost position, the second highest element will be place left of it and so on. Generally speaking, bubble sort has poor performance, but it is straightforward to implement. Moreover, we only need to sort the frequency maps once (maximal 256 elements). Hence there is no noticeable disadvantage here.

This is a rather complex tutorial. The solutions require approx. 600 lines of low density C++ code. Completing this tutorial is crucial for succeeding in the upcoming assignments and tests.

Use this tutorial to practice the idioms of C++ and the proper coding of them.