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:	6, Iterators
Lecturer:	Dr. Markus Lumpe

Iterators allow for a systematic traversal of sets in a data type agnostic way. Moreover, the underlying set does not need to be static. That is, we can construct the underlying set *on-the-fly* at runtime. As a result, iterators in C++ can be used as means to model "*list comprehension*," available for example in Python as syntactic construct, to dynamically create and enumerate a possibly infinite set of values.

Fibonacci Sequence

 [1]

In mathematics, the *Fibonacci numbers* (or *Fibonacci sequence*) are positive numbers in the following sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, …

For $n \geq 3$, we can define this sequence recursively by

$$\textbf{fib}(n) = \textbf{fib}(n - 1) + \textbf{fib}(n - 2),$$

with seed values

$$\textbf{fib}(1) = 1 \text{ and } \textbf{fib}(2) = 1.$$

Fibonacci numbers appear in numerous places, including computer science and biology. Unfortunately, evaluating a Fibonacci sequence for a given n in a recursive and bottom-up fashion is computationally expensive and may exceed available resources (in terms of both space and time). The recursive definition calculates the smaller values of **fib**(n) first and then builds larger values from them.

[1] Source: http://en.wikipedia.org/wiki/File:Fibonacci.png

An alternative mathematical formulation of the Fibonacci sequence is due to *dynamic programming*, a technique developed by Richard E. Bellmann in the 1940s while working for the RAND Corporation. Dynamic programming uses memorization to save values that have already been calculated. This yields a top-down approach that allows **fib**(n) to be split into sub-problems and then calculate and store values. This method produces a very efficient iterative algorithm to generating the Fibonacci sequence.

The iterative formulation of the Fibonacci sequence uses two storage cells, `previous` and `current`, to keep track of the values computed so far:

```
fibIter( n ) =
  previous := 0;
  current := 1;
  for i := 1 to n do
   next := current + previous;
   previous := current;
   current := next;
   end;
```

For n ≥ 1, this algorithm produces the desired sequence in linear time, and only requiring constant space.

Problem 1: Class FibonacciSequence

Using the dynamic programming solution, we can construct a C++ class, called `FibonacciSequence`, that produces the Fibonacci sequence up to a given n. Objects of this class *generate* the Fibonacci sequence via repeatedly calling the `advance` method. The Fibonacci sequence is infinite. We can model the same behavior. However, for practical purposes, we shall limit the sequence to the first 20 numbers. This limit allows the corresponding iterator to detect '*the end*', that is, we can construct an iterator for `FibonacciSequence` objects that is positioned after the desired limit (see Problem 2 for more details). The following class specification suggests a possible solution:

```
#pragma once

// forward declaration to break mutual recursion of
// FibonacciSequence and FibonacciSequence
class FibonacciSequenceIterator;

class FibonacciSequence
{
private:
  uint64_t fPrevious;    // previous Fibonacci number (initially 0)
  uint64_t fCurrent;     // current Fibonacci number (initially 1)
  uint64_t fPosition;    // position in the sequence (starts with 1)
  uint64_t fLimit;       // set limit for sequence (0 for no limit)

public:

  // Default constructor to set up a Fibonacci sequence
  // (aLimit = 0 means infinite)
  FibonacciSequence( uint64_t aLimit = 20 );

  // get current Fibonacci number
  const uint64_t& current() const;

  // advance to next Fibonacci number, may throw out_of_range exception on
  // exceeding limit
  void advance();

  // extract sequence limit
  const uint64_t& getLimit() const;

  // restart sequence
  void reset();

  // return new iterator positioned at start
  FibonacciSequenceIterator begin() const;

  // return new iterator positioned after the desired limit
  FibonacciSequenceIterator end() const;
};
```

A `FibonacciSequence` object requires four member variables. The values `fPrevious` and `fCurrent` serve as the storage cells to compute the Fibonacci sequence. The values `fPosition` and `fLimit` denote the current position in the sequence and the maximum position in the sequence, respectively. The constructor has to properly set up these variables.

The methods `current()`, `advance()`, `getLimit()`, and `reset()` are the service functions of class `FibonacciSequence`. The function `current()` returns the current Fibonacci number and the function `getLimit()` the maximum position in the sequence. The method `advance()` computes the next Fibonacci number according to the dynamic programming scheme shown above. It must test, if the maximum position has not been exceeded. In case it has, an `out_of_range` exception must be thrown. Please note that a

sequence can be infinite. In this case, the limit is 0 and we will never run out of numbers to generate. Design your tests accordingly. Finally, `reset()` restarts the sequence.

The methods `begin()` and `end()` return corresponding iterators for `FibonacciSequence` objects. The method `begin()` has to create and return an iterator that is positioned at the start of a Fibonacci sequence. The method `end()` has to create and return an iterator that is positioned at the end of the sequence. Note, an iterator may need to run forever. The methods `begin()` and `end()` rely on the proper implementation of `FibonacciSequenceIterator` in Problem 2.

Use the following program to test your implementation (enable `#define P1`):

```
cout << "Fibonacci sequence up to " << argv[1] << ":" << endl;

FibonacciSequence lSequence( atoi( argv[1] ) );

// test problem 1

for ( uint64_t i = 1; i <= lSequence.getLimit(); i++ )
{
    cout << i << ":\t" << lSequence.current() << endl;
    lSequence.advance();
}

cout << "Start again:" << endl;

lSequence.reset();

for ( uint64_t i = 1; i <= lSequence.getLimit(); i++ )
{
    cout << i << ":\t" << lSequence.current() << endl;
    lSequence.advance();

}
```

It should print the Fibonacci sequence for the first 20 numbers twice.

Problem 2: Class FibonacciSequenceIterator

The class `FibonacciSequenceIterator` implements a standard forward iterator for `FibonacciSequence` objects. It maintains two instance variables: a `FibonacciSequence` object and the iterator position `fIndex`.

The following class specification suggests a possible solution:

```
#pragma once

#include "FibonacciSequence.h"

class FibonacciSequenceIterator
{
private:
  FibonacciSequence fSequenceObject;              // Sequence object
  uint64_t fIndex;                                // current iterator position

public:

  // iterator constructor, copies FibonacciSequence object
  // and aStart set by default to 1
  FibonacciSequenceIterator( const FibonacciSequence& aSequenceObject,
                             uint64_t aStart = 1 );

  // iterator constructor, initializes FibonacciSequence object
  // and aStart set by default to 1
  FibonacciSequenceIterator( uint64_t aLimit = 20, uint64_t aStart = 1 );

  // iterator methods

  const uint64_t& operator*() const;           // return current Fibonacci number
  FibonacciSequenceIterator& operator++();     // prefix, next Fibonacci number
  FibonacciSequenceIterator operator++( int ); // postfix (extra unused argument)

  bool operator==( const FibonacciSequenceIterator& aOther ) const;
  bool operator!=( const FibonacciSequenceIterator& aOther) const;

  // iterator methods

  // return new iterator positioned at start
  FibonacciSequenceIterator begin() const;

  // return new iterator positioned at limit
  FibonacciSequenceIterator end() const;

};
```

The implementation of `FibonacciSequenceIterator` follows standard practice and is similar to the `IntArrayIterator` studied in class. The methods `begin()` and `end()` have to return copies of the current iterator with the position adjusted to the first sequence number and past the last sequence number (or set to 0, if infinite), respectively.

The iterator maintains its own copy of a sequence object. That is, an iterator for a sequence object copies it and continues the sequence. We can also create an iterator for a fresh Fibonacci sequence by specifying just the limit and the start. Either technique yields an iterator that models list comprehension for Fibonacci sequences.

Use the following program to test your implementation (enable `#define P2`):

```
cout << "Fibonacci sequence 1..20:" << endl;

uint64_t i = 1;
for ( const uint64_t& n : FibonacciSequence( 20 ) )
{
    cout << i++ << ":\t" << n << endl;
}

// test for infinite sequence

unsigned long lUserSpecifiedLength;

cout << "Please specify the desired length of the Fibonacci sequence: ";
cin >> lUserSpecifiedLength;

cout << "Running the Fibonacci sequence up to " << lUserSpecifiedLength << endl;

uint64_t l = 1;
for ( const uint64_t& n : FibonacciSequence( 0 ) )
{
    cout << l << ":\t" << n << endl;

    if ( l++ >= lUserSpecifiedLength )
    {
        break;
    }
}
```

It should print the Fibonacci sequence for the first 20 numbers and next as many numbers as you specify. Please not `uint64_t` has 64 Bits. The 94th Fibonacci number will exceed 64 Bits.

The solution for this tutorial requires approx. 150 lines of code.