Swinburne University of Technology

School of Science, Computing and Engineering Technologies

ASSIGNMENT COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures & Patterns
Assignment number and title:	2 - Iterators
Due date:	Wednesday, 9th October 2024, 23:59
Lecturer:	Ms. Siti Hawa

Your name:__________________________ **Your student id:**_______________

Marker's comments:

Problem	Marks	Obtained
1	40	
2	70	
Total	110	

Extension certification:

This assignment has been given an extension and is now due on ____________

Signature of Convener:________________________

Problem Set 2: Iterators

[1]

In mathematics, the Fibonacci numbers (or the Fibonacci sequence) are an infinite series of positive numbers with the following pattern

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

$$\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2),$$

with seed values

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

Fibonacci numbers appear in numerous places, including computer science and biology. Unfortunately, evaluating the 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 Fibonacci(n) first and then builds larger values from them. This process has $O(2^n)$ complexity (see lecture slides page 179).

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 Fibonacci(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:

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

For $n \geq 1$, this algorithm produces the desired sequence in linear time and constant space.

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

Problem 1:

Using the dynamic programming solution, we can construct a C++ class, called `FibonacciSequenceGenerator`, that produces the correct Fibonacci sequence up to the maximum integer value representable on a given computer architecture (e.g., 2^{64}-1).

```
#pragma once

#include <string>
#include <cstddef>

class FibonacciSequenceGenerator
{
private:
  const std::string fID;  // sequence identifier
  long long fPrevious;    // previous Fibonacci number (initially 0)
  long long fCurrent;     // current Fibonacci number (initially 1)

public:

  // Constructor to set up a Fibonacci sequence
  FibonacciSequenceGenerator(const std::string& aID = "") noexcept;

  // Get sequence ID
  const std::string& id() const noexcept;

  // Get current Fibonacci number
  const long long& operator*() const b;

  // Type conversion to bool
  operator bool() const noexcept;

  // Reset sequence generator to first Fibonacci number
  void reset() noexcept;

  // Tests if there is a next Fibonacci number.
  // Technically, there are infinitely many Fibonacci numbers,
  // but the underlying integer data type limits the sequence.
  bool hasNext() const noexcept;

  // Advance to next Fibonacci number
  // Function performs overflow assertion check.
  void next() noexcept;
};
```

Class `FibonacciSequenceGenerator` defines three member variables. The values `fPrevious` and `fCurrent` serve are the storage cells to compute the Fibonacci sequence. The member variable `fID`, on the other hand, provides a programmatic means to establish nominal equivalence between two objects of class `FibonacciSequenceGenerator`. We need this feature when we define an iterator for `FibonacciSequenceGenerator` objects. The constructor has to properly initialize these variables using a member initializer list.

The method `id()` has to return a constant reference to the id string, whereas the dereference operator, `operator*()`, returns a constant reference to the current Fibonacci number. There is no need to expose the value of the previous Fibonacci number.

Objects of class `FibonacciSequenceGenerator` generate the Fibonacci sequence via repeatedly calling the `next()` method. This method just computes just the next Fibonacci number using the approach shown in pseudo code (loop body). However, method `next()` will eventually reach an overflow condition, that is, the next Fibonacci number is not representable as a positive integer on a 64-Bit computer architecture. In this case the number becomes negative. Method `next()` has to include a precondition assertion to guarantee it never produces a negative value.

Method `hasNext()` returns true if the next Fibonacci number is not negative. This method has to calculate the next number locally. The method `hasNext()` allows us to stop the generation of Fibonacci numbers gracefully before an overflow occurs. On a 64-Bit architecture, `hasNext()` returns false for the 93th Fibonacci number.

The `operator bool()` allows for objects of class `FibonacciSequenceGenerator` to be implicitly converted into a Boolean when needed. It returns true if there is a next Fibonacci number.

Finally, method `reset()` reverts a `FibonacciSequenceGenerator` object to its initial state. We need this feature when we define an iterator for `FibonacciSequenceGenerator` objects.

The file `Main.cpp` contains a test function to check your implementation. Uncomment **`#define`** `P1` and compile your solution with x64 (default on macOS). Your program should produce the following output:

```
Fibonacci sequence P1 for long long:
 1: 1
 2: 1
 3: 2
 4: 3
 5: 5
 6: 8
 7: 13
 8: 21
 9: 34
10: 55
11: 89
12: 144
13: 233
14: 377
15: 610
16: 987
…
82: 61305790721611591
83: 99194853094755497
84: 160500643816367088
85: 259695496911122585
86: 420196140727489673
87: 679891637638612258
88: 1100087778366101931
89: 1779979416004714189
90: 2880067194370816120
91: 4660046610375530309
92: 7540113804746346429
Fibonacci sequence generated successfully.
1 test(s) run.
```

Problem 2:

The class `FibonacciSequenceIterator` implements a standard forward iterator for `FibonacciSequenceGenerator` objects. It maintains to instance variables: an object of class `FibonacciSequenceGenerator` and the iterator position.

Please note that Fibonacci iterators maintain a copy of the underlying collection. Technically, this make iterator comparison difficult if not impossible. However, all objects of class `FibonacciSequenceGenerator` have an id string and we can obtain it via method `id()`. Consequently, two iterators of class `FibonacciSequenceIterator` are positioned on the same element (i.e., the same Fibonacci number) if their sequence objects has the same id string and if their respective indices are equal.

```
#pragma once

#include "FibonacciSequenceGenerator.h"

class FibonacciSequenceIterator
{
private:
  FibonacciSequenceGenerator fSequenceObject;      // sequence object
  long long fIndex;                                // current iterator position

public:

  // iterator constructor
  // FibonacciSequence objects has an id to allow for comparision
  FibonacciSequenceIterator( const FibonacciSequenceGenerator& aSequenceObject,
                             long long aStart = 1 ) noexcept;

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

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

  // iterator auxiliary methods

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

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

The implementation of `FibonacciSequenceIterator` follows standard practice for a forward iterator (see lecture and tutorial notes).

There is, however, a minor challenge when an iterator moves into the end position. We use the prefix increment operator to move the iterator forward. The increment operator has to compute the next Fibonacci number. This fails when the iterator moves into the end position. You need to devise a solution to prevent this. At no time must the iterator trigger an assertion violation.

Finally, the `begin()` method has to return an iterator that is positioned at the first element. Method `begin()` has to return a copy of the iterators whose sequence object has been reset also. For method `end()` determine the proper end index.

The file `Main.cpp` contains a test function to check your implementation. Uncomment **`#define`** `P2` and compile your solution with x64 (default on macOS). Your program should produce the following output:

```
Fibonacci sequence P2 for long long:
 1: 1
 2: 1
 3: 2
 4: 3
 5: 5
 6: 8
 7: 13
 8: 21
 9: 34
10: 55
11: 89
12: 144
13: 233
14: 377
15: 610
16: 987
…
82: 61305790721611591
83: 99194853094755497
84: 160500643816367088
85: 259695496911122585
86: 420196140727489673
87: 679891637638612258
88: 1100087778366101931
89: 1779979416004714189
90: 2880067194370816120
91: 4660046610375530309
92: 7540113804746346429
Fibonacci sequence generated successfully.
1 test(s) run.
```