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: 3 – Design Patterns and 12 Bit I/O
Due date: Wednesday, 30th October 2024, 23:59
Lecturer: Ms. Siti Hawa

Your name:______________________________ **Your student id:**_________________

Marker's comments:

Problem	Marks	Obtained
1	138	
Total	138	

Extension certification:

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

Signature of Convener:___________________________

Problem Set 3: Design Pattern and 12-bit I/O

Problem 1:

Start with the 12-bit I/O class `ofstream12` discussed in tutorial 9. Implement the matching input class `ifstream12` satisfying the following specification:

```
#pragma once

#include <cstddef>              // std::byte
#include <fstream>
#include <optional>

class ifstream12
{
private:
  std::ifstream fIStream;

  std::byte* fBuffer;                     // input buffer
  size_t fBufferSize;                     // input buffer size

  size_t fByteCount;                      // available input bytes
  size_t fByteIndex;                      // current byte index
  int fBitIndex;                          // current bit index (can be negative)

  void reset();                           // reset buffer, [6 marks]
  void fetch_data();                      // read data, [12 marks]
  std::optional<size_t> readBit();        // read next bit, [42 marks]

public:
  // using C++11's nullptr
  ifstream12( const char* aFileName = nullptr, size_t aBufferSize = 128 ); // [18 marks]
  ~ifstream12();       // [4 marks]

  void open( const char* aFileName ); // [12 marks]
  void close();        // [2 marks]

  bool isOpen() const; // [4 marks]
  bool good() const;  // [4 marks]
  bool eof() const;   // [4 marks]

  ifstream12& operator>>( size_t& aValue );  // [28 marks]
};
```

The class `ifstream12` defines an object adapter for `std::ifstream`. Clearly, the corresponding file input stream has to be **binary** and the data would constitute strings of `0`s and `1`s. What makes this task somewhat difficult is the requirement that we need to use a physical 8-bit stream to read 12-bit values. We cannot do this directly. Instead, we have to employ a buffering mechanism to first "collect" the bits from the underlying 8-bit input stream and then construct 12-bit values from the bits in the buffer.

Class `ifstream12` requires a constructor and a destructor. The constructor has to initialize the object, acquire the necessary buffer memory, and open the input file. The destructor has to close the underlying file and free the buffer memory.

The methods `open`, `close`, `isOpen`, `good`, and `eof` correspond to their respective `std::ifstream` methods. You should study the features of `std::ifstream` carefully.

The member function `eof` returns **`true`**, if there are no bytes left in the input stream (i.e., `fByteCount == 0`). Please note that `fByteCount` should be 0, if you have never read anything from `fIStream`. There is a subtle handshake between `std::ifstream`, `ifstream12`, and clients of `ifstream12` when it comes to the detection of end-of-file. The object adapter `ifstream12` has to return **`true`** for EOF, if and only if there are no further bits available. A boundary scenario allows the underlying `std::ifstream` object to be in state EOF while the object adapter `ifstream12` is not.

The function `readBit` implements the mapping process. It returns an optional value 0 or 1. The return type `std::optional<size_t>` signifies that `readBit` can reach EOF while reading the next bit.

The base type of `fBuffer` is `std::byte`. Unfortunately, type `std::byte` offers only a limit set of operations that focus on bit manipulations and make it somewhat difficult to interpret `std::byte` values as plain integers. For the conversion of bit patterns to a single value, we can use the following declaration:

```
std::byte lByte = fBuffer[fByteIndex] & (std::byte{1} << fBitIndex);
```

The value of `lByte` is a bit-mask for the bit at index `fBitIndex`. It is either 1 or 0. The value `lByte` is still of type `std::byte`. You need to apply a type conversion to `size_t` to interpret the value as integer using the following expression:

```
std::to_integer<size_t>(lByte).
```

If the resulting value is greater than zero, then it denotes the bit 1. Otherwise, it means the bit 0.

The function `readBit` also triggers `fetch_data`, if necessary. More precisely, at the start, you must check if `fByteCount` is 0. In this case, `fetch_data` must be called (the buffer does not contain any data). You may reach EOF here. In this case, `readBit` has no value to return.

When fetching the next bit (using the expressions shown above), store the result temporarily, and then advance the indices `fByteIndex` and `fBitIndex` to the next position (some additional logic is required that you must devise to make it work). If `fBitIndex` (which runs from highest to lowest) becomes negative, then you need to switch to the next byte in the buffer. This also means that you have processed a byte. Hence, you need to decrement `fByteCount`. Once all indices have been properly adjusted, you return the result.

The `operator>>` implements the read12Bits algorithm as shown in the tutorial. You need to adjust it to incorporate `std::optional` values. That is, `readBit` may return no value (when EOF has been reached). In this case, you need to break from the for-loop. In addition. You may need to use a static cast to `size_t` to set the corresponding bit in the 12-bit value.

The file `Main.cpp` contains a test function to check your implementation. Your program should produce the following output:

```
Writing data.
Write 4096 codes
Reading data.
Read 4096 codes
Done
```