Swinburne University of Technology

Faculty of Science, Engineering and Technology

MIDTERM COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Assignment number and title:	Midterm: Solution Design & Iterators
Due date:	April 26, 2024, 10:30
Lecturer:	Dr. Markus Lumpe

Your name:_____ **Your student ID:**_____

Marker's comments:

Problem	Marks	Obtained
1	106	
2	194	
Total	300	

Midterm: Vigenère Cipher

Around 1550 Blaise de Vigenère, a French diplomat from the court of Henry III of France, developed a new table-based scrambling technique to cipher written language. The technique, also known as *chiffre quarré* or *chiffre indéchiffrable*, was considered safe until the middle of the 19th century.

The *Vigenère Cipher* is a polyalphabetic substitution technique based on a mapping table like the one shown below:

Key\Letter	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	**L**	M	N	O	P	Q	R
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

The Vigenère cipher uses this table together with a keyword to encode a message.

To illustrate the use of this encryption method, suppose we wish to scramble the following message (Hamlet 3/1):

```
To be, or not to be: that is the question:
```

using the keyword `Relations`. First, we notice that the table provides only mappings for upper case characters. But this is not really a problem. The mapping is identical for upper and lower case characters. We just rewrite the keyword to consist of upper case characters only. When encoding a message, we convert each character to an upper case one, perform the corresponding encryption function, and output the result in either upper case or lower case depending on the original spelling. All characters not covered in the Vigenère cipher remain unchanged in the output. No keyword character is consumed in this case!

We begin by writing the keyword. We repeat the keyword until all characters of the input text are covered. To derive the encoded text using the mapping table, for each letter in the

message, one has to find the intersection of the row given by the corresponding keyword letter and the column given by the message letter itself to pick out the encoded letter.

Keyword:	**RE LA TI ONS** RE LA TION SR ELA TIONSREL
Message:	To be, or not to be: that is the question:
Scrambled Message:	Lt nf, ia ccm lt nf: nqph bk ytf kdtgmatz:

Decoding of an encrypted message is equally straightforward. One writes the keyword plus the decoded message:

Keyword:	RE LA TI ONS RE LA TION SR ELA TIONSREL
Scrambled Message:	Lt nf, ia ccm lt nf: nqph bk ytf kdtgmatz:
Decoded Message:	To be, or not to be: that is the question:

This time one uses the keyword letter to pick a row of the table and then traces the row to the column containing the encoded letter. The index of that column is the decoded letter.

There are many ways to implement the Vigenère cipher. In this midterm, we use two forward iterators to perform the encoding and decoding of plain English text. The first iterator serves as keyword provider, whereas the second one implements the Vigenère cipher algorithm for both encoding and decoding plain English text.

Problem 1 (106 marks)

Define a forward iterator, called `KeyProvider`, that yields a sequence of keyword characters.

A suggested specification of class `KeyProvider` is shown below:

```
#pragma once

#include <string>

class KeyProvider
{
private:
  std::string fKeys;
  size_t fIndex;

  std::string preprocessString(const std::string& aString) noexcept; // 18 marks

public:
  // 32 marks
  KeyProvider(const std::string& aKeyword, const std::string& aSource) noexcept;

  char operator*() const noexcept; // 4 marks

  KeyProvider& operator++() noexcept; // 4 marks
  KeyProvider operator++(int) noexcept; // 10 marks

  bool operator==(const KeyProvider& aOther) const noexcept; // 10 marks
  bool operator!=(const KeyProvider& aOther) const noexcept; // 4 marks

  KeyProvider begin() const noexcept; // 10 marks
  KeyProvider end() const noexcept; // 10 marks
};
```

The underlying collection for iterator `KeyProvider` is the string `fKeys`. The iterator index is `fIndex`.

The iterator `KeyProvider` has to compute the underling collection first. You start with the keyword, say `"Relations"`, and populate the string `fKeys` be repeating the keyword, in upper-case letters, for the length of the message.

Keyword:	"Relations"
Message:	"To be, or not to be: that is the question:"
fKeys:	"RELATIONSRELATIONSRELATIONSREL"

or

Keyword:	"Relations"
Message:	"Be cool"
fKeys:	"RELATI"

Please note, the length of string `fKeys` equals the number of letters in the message. Only letters consume keyword characters.

To facilitate the initialization of `fKeys`, the method `preprocessString()` can be used. This method takes a string and returns a new string that consists only of upper-case letters. For example, `preprocessString()` applied to `"Relations"` returns `"RELATIONS"`. Similarly, if applied to `"To be, or not to be: that is the question:"`, the method `preprocessString()` returns `"TOBEORNOTTOBETHATISTHEQUESTION"`.

The implementation of `preprocessString()` requires the functions `isalpha()` and `toupper()`, both defined in `<cctype>`.

The constructor of `KeyProvider`, initializes `fKeys` and `fIndex`. The iterator has to be positioned onto the first element in the underlying collection.

The constructor has to compute the keyword sequence for the input text.

As a security feature, the constructor has to end with an assert statement that guarantees that the size of `fKeys` matches the size of the preprocessed input string. The preprocessed input string may be shorter than `aSource`.

The dereference `operator*()` returns the keyword character the iterator is positioned on.

The prefix `operator++()` advances the iterator and returns the updated iterator.

The postfix `operator++(int)` advances the iterator and returns the old iterator.

The equivalence `operator==()` returns true if `this` iterator and `aOther` are equal. Equivalence requires the same underlying collection and the same position.

The equivalence `operator!=()` returns true if `this` iterator and `aOther` are not equal.

The method `begin()` returns a copy of `this` iterator positioned at the first keyword character.

The method `end()` returns a copy of `this` iterator positioned after the last keyword character.

You may test your implementation of `KeyProvider` using the following test driver (enable **`#define`** `P1` in main.cpp).

Running the test driver should produce the following output:

```
Test KeyProvider
First key sequence: "RELATIONSRELATIONSRELATIONSREL"
Second key sequence: "RELATI"
Third key sequence: ""
done
```

Problem 2 (194 marks)

Assume the implementation of `KeyProvider` is correct.

Using `KeyProvider`, define the forward iterator `VigenerForewardIterator` that satisfies the suggested specification as shown below:

```
#pragma once

#include <string>

#include "KeyProvider.h"

constexpr size_t CHARACTERS = 26;

enum class EVigenereMode
{
  Encode,
  Decode
};

class VigenereForwardIterator
{
private:
  EVigenereMode fMode;
  char fMappingTable[CHARACTERS][CHARACTERS];
  KeyProvider fKeys;
  std::string fSource;
  size_t fIndex;
  char fCurrentChar;

  // Initialize the mapping table
  // Row 1:   B - A
  // Row 26:  A - Z
  void initializeTable();

  void encodeCurrentChar() noexcept; // 38 marks
  void decodeCurrentChar() noexcept; // 48 marks

public:

  VigenereForwardIterator(
            const std::string& aKeyword,
            const std::string& aSource,
            EVigenereMode aMode = EVigenereMode::Encode ) noexcept; // 26 marks

  char operator*() const noexcept; // 2 marks

  VigenereForwardIterator& operator++() noexcept; // 14 marks
  VigenereForwardIterator operator++(int) noexcept; // 10 marks

  bool operator==(const VigenereForwardIterator& aOther) const noexcept; // 10 marks
  bool operator!=(const VigenereForwardIterator& aOther) const noexcept; // 4 marks

  VigenereForwardIterator begin() const noexcept; // 30 marks
  VigenereForwardIterator end() const noexcept; // 10 marks
};
```

The iterator `VigenereForwardIterator` maintains a cipher mode, a mapping table, a keyword provider (another iterator), the source text, the current position, and the current character the iterator has to return.

The constructor has to initialize all member variables with sensible variables. Ideally, all member variables except `fMappingTable` should be initialized using member initializers. Use `initializeTable()` as the first statement in the constructor to set up the mapping table.

Somebody has already implemented this method:

```
void VigenereForwardIterator::initializeTable()
{
  for ( char row = 0; row < CHARACTERS; row++ )
  {
    char lChar = 'B' + row;

    for ( char column = 0; column < CHARACTERS; column++ )
    {
      if ( lChar > 'Z' )
        lChar = 'A';

      fMappingTable[row][column] = lChar++;
    }
  }
}
```

The constructor for `VigenereForwardIterator` has to advance to the first character. That is, depending on the mode, the iterator has to encode and decode the first source character, respectively. So, if the iterator `fKeys` is not at the end, then call `encodeCurrentChar()` or `decodeCurrentChar()` to set the first character.

The methods `encodeCurrentChar()` and `decodeCurrentChar()` provide the scrambling operations. They implement the encoding and decoding process, respectively, as described above. When processing a character, these methods record whether the character is upper case or lower case, and each time a letter is being processed the current keyword character is updated as part of the autokey cipher process. Both methods update the iterator variable `fCurrentChar`.

The dereference `operator*()` returns the character the iterator is positioned on. This may be an encoded or a decoded character value.

The prefix `operator++()` advances the iterator and returns the updated iterator. When advancing, the iterator has to update the next character. This should always be possible.

The postfix `operator++(int)` advances the iterator and returns the old iterator.

The equivalence `operator==()` returns true if `this` iterator and `aOther` are equal. Equivalence requires the same underlying collection and the same position. The underlying collection is the input text being processed by the iterator.

The equivalence `operator!=()` returns true if `this` iterator and `aOther` are not equal.

The method `begin()` returns a copy of `this` iterator positioned at the first character. You cannot use the iterator constructor here. Find a way to set up an iterator copy that is positioned on the first character. Remember, `fKeys` is also an iterator.

The method `end()` returns a copy of `this` iterator positioned after the last character. The underlying collection is the input text being processed by the iterator.

You may test your implementation of `VigenereForwardIterator` using the following test driver (enable **`#define`** `P2` in main.cpp).

Running the test driver should produce the following output:

```
Test VigenereForwardIterator
First phrase: "To be, or not to be: that is the question:"
Encoded text: "Lt nf, ia ccm lt nf: nqph bk ytf kdtgmatz:"
Decoded text: "To be, or not to be: that is the question:"
Second phrase: "Be cool"
Encoded text: "Tj opiu"
Decoded text: "Be cool"
```

```
Third phrase: ""
Encoded text: ""
Decoded text: ""
done
```

Submission deadline: Friday, April 26, 2024, 10:30.

Submission procedure: Follow the instruction on Canvas. Submit electronically the PDF of the printed code for classes `KeyProvider` and `VigenereForwardIterator`, combined with the cover page of the midterm test. In addition, upload the source files to Canvas (i.e., `KeyProvider.cpp` and `VigenereForwardIterator.cpp`).

The sources need to compile in the presence of the solution artifacts provided on Canvas.