Swinburne University of Technology

School of Science, Computing and Emerging Technologies

MIDTERM COVER SHEET

Subject Code:	COS30008
Subject Title:	Data Structures and Patterns
Assignment number and title:	Midterm Project: Solution Design & Iterators
Due date:	April 16, 2025, 18:59
Lecturer:	Dr. Markus Lumpe

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

Marker's comments:

Problem	Marks	Obtained
1	64	
3	196	
Total	260	

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. This method, known as *chiffre quarré* or *chiffre indéchiffrable*, was considered secure for 300 years. In 1863, F. W. Kasiski, a retired Major in the Prussian Army, published a book detailing an algorithm for breaking the *chiffre quarré*.

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 and 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 `G'day mate`. First, the coding table provides only mappings for upper-case letters. But this is not a problem. The mapping is identical for upper and lower case letters. We rewrite the keyword to consist of upper-case letters only. When encoding a message, we convert each occurring letter 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 of the letter. All characters not covered in the Vigenère cipher remain unchanged in the output. No keyword character is consumed in this case!

The Vigenère cipher is an *autokey cipher*. Both, the keyword and the message are used for coding. To derive the encoded text using the mapping table, one must find the intersection

of the row defined by the corresponding keyword letter and the column designated by the message letter to select the encoded letter (message encoded completely):

AutoKey:	**GD AY MA TE***T OB EO RNOT TO BET HATISTHE QUESTION*
Clear Message:	`To be, or not to be: that is the question:`
Scrambled Message:	`As cd, bs htn iq gt: lvpn ch vmy yvybmcws:`

Decoding an encrypted message works similarly, but requires a linear search. This time one uses the keyword letter to select a row in the table and then traces the row to the column containing the encoded letter. The index of that column is the decoded letter (the letter above that column). Note, that the autokey is extended with the decoded letter as we process the encrypted message.

The values at the start of the process:

Auto Key:	**GD AY MA TE**
Scrambled Message:	`As cd, bs htn iq gt: lvpn ch vmy yvybmcws:`
Clear Message:	

The values after processing the first six letters in the encoded message:

Auto Key:	**GD AY MA TE***T OB EO R*
Scrambled Message:	`As cd, bs htn iq gt: lvpn ch vmy yvybmcws:`
Clear Message:	*To be, or*

The values after processing the complete encoded message:

Auto Key:	**GD AY MA TE***T OB EO RNOT TO BET HATISTHE QUESTION*
Scrambled Message:	`As cd, bs htn iq gt: lvpn ch vmy yvybmcws:`
Clear Message:	*To be, or not to be: that is the question:*

Perform a hand execution of the process. Test both, the encoding and the decoding of a text string.

There are many ways to implement the Vigenère cipher. In this task, we employ a forward iterator to implement the Vigenère cipher for plain English text.

The cipher manipulates letters. In C++, letters can be used like integers whose values are defined in the 7-bit ASCII table. For instance, the letter `'A'` is `65` or `0x41`. We can use letters to perform integer arithmetic. We write `'R' - 'A'` to mean integer value `17`, or the eighteenth row in the Vigenère table. Similarly, we write `19 + 'A'` to mean the letter `'T'`, or the twentieth column in the Vigenère table.

Problem 1 (64 marks)

Start with the class `AutoKey`. The purpose of this data type is to simplify the design of the Vigenère iterator you implement in the following problem.

A suggested specification of the class `AutoKey` is shown below:

```
#pragma once

#include <string>
#include <ostream>
#include <cctype>
#include <concepts>

class AutoKey
{
private:

  std::string fValue;
  size_t fKeyLength;
  size_t fIndex;

public:

  using difference_type = std::ptrdiff_t;
  using value_type = char;

  AutoKey( const std::string& aKeyword = "" ) noexcept;      // [16 marks]

  size_t size() const noexcept;                              // [4 marks]

  char operator*() const noexcept;                           // [4 marks]
  AutoKey& operator++() noexcept;                            // [4 marks]
  AutoKey operator++(int) noexcept;                          // [10 marks]

  AutoKey& operator+=( char aChar ) noexcept;                // [16 marks]

  void reset() noexcept;                                     // [8 marks]
};

static_assert(std::input_iterator<AutoKey>);
```

Class `AutoKey` defines a wrapper for a `std::string` object and satisfies the `std::input_iterator` constraint. In other words, class `AutoKey` defines a pointer-like data type to read the characters from the underlying string in an incremental forward fashion.

The constructor takes a constant reference to a `std::string` object and initializes all members with sensible values. Some variables must be updated in the constructor's body.

The constructor must inspect all characters in the parameter `aKeyword`, ideally via a for loop, and copy only letters in upper case to `fValue`. You should use the functions `std::isalpha()` and `std::toupper()` to filter out letters and convert them to upper case. Research class `std::string` to identify the proper method to append characters to `fValue`. There are two approaches to achieving the desired outcome. Inspect the class `AutoKey` thoroughly. You might find clues for reusing features effectively when populating `fValue`. Crucially, at the end of the constructor, you must set `fKeyLength` to the correct value.

The member function `size()` returns the length of the auto key. This value changes when we process messages.

The **`operator*`**, **`operator++()`**, and **`operator++(int)`** work as expected. These operators implement the features of an input iterator.

The **operator+=** allows us to extend the auto key value, `fValue`, by appending a new character, `aChar`. We cannot simply append `aChar` to `fValue` though. First, we must test whether `aChar` is a letter. If `aChar` is a letter, we add it in upper case to `fValue` using a suitable `std::string` operation. Otherwise, `aChar` is discarded. The **operator+=** must return the updated `AutoKey` object.

Finally, the method `reset()` reverts the current `AutoKey` object to its initial value. The auto key `fValue` must be trimmed to the keyword. The keyword is a prefix string of `fValue`. Remember, the constructor determines the length of the keyword.

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

```
Test AutoKey:
Initial lAutoKey: "GDAYMATE", Length = 8
Test phrase: "To be, or not to be: that is the question:"
Character 'T' must be added.
Character 'o' must be added.
Character ' ' must be skipped.
Character 'b' must be added.
Character 'e' must be added.
Character ',' must be skipped.
Character ' ' must be skipped.
Character 'o' must be added.
Character 'r' must be added.
Character ' ' must be skipped.
Character 'n' must be added.
Character 'o' must be added.
Character 't' must be added.
Character ' ' must be skipped.
Character 't' must be added.
Character 'o' must be added.
Character ' ' must be skipped.
Character 'b' must be added.
Character 'e' must be added.
Character ':' must be skipped.
Character ' ' must be skipped.
Character 't' must be added.
Character 'h' must be added.
Character 'a' must be added.
Character 't' must be added.
Character ' ' must be skipped.
Character 'i' must be added.
Character 's' must be added.
Character ' ' must be skipped.
Character 't' must be added.
Character 'h' must be added.
Character 'e' must be added.
Character ' ' must be skipped.
Character 'q' must be added.
Character 'u' must be added.
Character 'e' must be added.
Character 's' must be added.
Character 't' must be added.
Character 'i' must be added.
Character 'o' must be added.
Character 'n' must be added.
Character ':' must be skipped.
Number of characters added to lKeyword: 30
lAutoKey after the lPhrase has been added:
"GDAYMATETOBEORNOTTOBETHATISTHEQUESTION", Length = 38
lAutoKey after reset: "GDAYMATE", Length = 8
Test AutoKey complete.
```

Problem 2 (196 marks)

You have thoroughly tested the `AutoKey` class. Use it to implement the class `VigenereIterator`, which defines a forward iterator for encoding and decoding characters according to the Vigenère cipher.

A suggested specification of the class `VigenereIterator` is shown below:

```
#pragma once

#include <concepts>

#include "AutoKey.h"

constexpr size_t CHARACTERS = 26;

enum class EVigenereMode
{
  Encode,
  Decode
};

class VigenereIterator
{
public:

  using iterator = VigenereIterator;
  using difference_type = std::ptrdiff_t;
  using value_type = char;

  VigenereIterator( const std::string& aKeyword = "",       // [26 marks]
                    const std::string& aSource = "",
                    EVigenereMode aMode = EVigenereMode::Encode ) noexcept;

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

  VigenereIterator& operator++() noexcept;                  // [14 marks]
  VigenereIterator operator++(int) noexcept;                // [10 marks]

  bool operator==( const VigenereIterator& aOther ) const noexcept;
                                                            // [10 marks]
  VigenereIterator begin() const noexcept;                  // [30 marks]
  VigenereIterator end() const noexcept;                    // [10 marks]

private:

  EVigenereMode fMode;
  char fMappingTable[CHARACTERS][CHARACTERS];
  AutoKey fKeys;
  std::string fSource;
  size_t fIndex;
  char fCurrentChar;

  void initializeTable() noexcept;
  void encodeCurrentChar() noexcept;                        // [42 marks]
  void decodeCurrentChar() noexcept;                        // [52 marks]
};

template<typename T>
concept BasicForwardIterator =
std::forward_iterator<T> &&     // a basic forward iterator is a forward iterator
requires( const T i, const T j )
{
    { i.begin() } -> std::same_as<T>;   // basic forward iterator provides begin()
    { i.end() } -> std::same_as<T>;     // basic forward iterator provides end()
};

static_assert(BasicForwardIterator<VigenereIterator>);
```

The iterator `VigenereIterator` maintains the cipher mode, the mapping table, the auto key, the source text, the current position, and the current character the iterator has to return.

The constructor must initialize all member variables with sensible variables. All member variables, except `fMappingTable`, must be initialized using member initializers. Within the constructor body, call `initializeTable()` as the first statement to set up the mapping table. Next, the constructor must advance the iterator to the first character. Depending on the mode of operation, the iterator has to encode and decode the first source character, respectively. This is possible when the iterator is not at the end. Remember, the expression **`*this`** refers to the current object (the current iterator). Call the private methods `encodeCurrentChar()` and `decodeCurrentChar()` to set the first character.

The methods `encodeCurrentChar()` and `decodeCurrentChar()` implement the encoding and decoding processes, respectively, as illustrated above. These methods only affect letters. Character manipulations are defined for upper-case letters only. Nevertheless, both methods preserve the spelling of a letter (i.e., lower-case letters must be restored to lower-case letters after processing). An autokey character must be consumed when a letter is altered through the cipher process. Hence, after reading the current auto key character, the auto key must be advanced to the next position and extended with the processed letter (source letter for encoding, decoded letter for decoding). At the end, both methods update the iterator variable `fCurrentChar`. Do some hand execution for both methods to better understand the required steps.

The dereference **`operator*`**`()` returns the character at which the iterator is positioned. 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. Advancing to the next character means the increment operator must call the methods `encodeCurrentChar()` and `decodeCurrentChar()`.

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

The equivalence **`operator==`**`()` returns **`true`** if **`this`** iterator and `aOther` are equal. Determine, which attributes establish equivalence between iterators. Avoid unnecessary comparisons when equivalence is implied or can never be achieved. Align equivalence testing with the positioning approach defined in the method `end()`. Two iterators are equal if positioned on the same element and underlying collection.

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 the iterator copy. Remember, `fKeys` is also an iterator. It must be reset to the start. The result of `begin()` must be consistent with the iterator state after construction.

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

The file `Main.cpp` contains a test function to check your implementation. Uncomment **`#define`** `P2`. Somebody else has already started with the implementation. Finish it and compile your solution.

Your program should produce the following output:

```
Test VigenereIterator:
First phrase: "To be, or not to be: that is the question:"
Encoded text: "As cd, bs htn iq gt: lvpn ch vmy yvybmcws:"
Decoded text: "To be, or not to be: that is the question:"
Second phrase: "Be cool."
Encoded text: "Ii dnbm."
```

```
Decoded text: "Be cool."
Third phrase: "12345"
Encoded text: "12345"
Decoded text: "12345"
Test VigenereIterator complete.
```

Submission deadline: Wednesday, April 16, 2025, 18:59.

Submission procedure:

Follow the instructions on Canvas. Submit the PDF of the two printed source files `AutoKey.cpp` and `VigenereIterator.cpp` electronically. Upload the source files `AutoKey.cpp` and `VigenereIterator.cpp` to Canvas.

The sources will be assessed and compiled in the presence of the solution artifacts provided on Canvas.