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: Xuan Tuan Minh Nguyen **Your student ID:** 103819212

Marker's comments:

Problem	Marks	Obtained
1	106	
2	194	
Total	300	

```cpp
//Neccessary libraries, where cassert for assert() and cctype for isalpha() and toupper()
#include "KeyProvider.h"
#include <cctype>
#include <cassert>

/*
* The function preprocessString will take in a string and returns a new string
* (Exceptions for Non-alphabetic characters) that contains the word in uppercase
* @param aString (const std::string&): The given string
* @return std::string: New string in uppercase and no non-alphabetic characters
*/
std::string KeyProvider::preprocessString(const std::string& aString) noexcept
{
    //Declare a string for storing characters
    std::string result;
    //Loop through each characters on the given string
    for (const char c : aString)
    {
        if (std::isalpha(c))
            //If that character is alphabetic -> push_back that characters
            //(In uppercase) to the result string
            result.push_back(std::toupper(c));
    }
    //Return the string
    return result;
}

/*
* Constructor of KeyProvider class, which will also be used for comptuing keyword sequnce
* based on the input text. In addition, this function must guarantee that the size of fKeys
* must match with the inputted phrase
* @param aKeyword (const std::string&): The keyword
* @param aSource (const std::string&): The source phrase
* @return void
*/
KeyProvider::KeyProvider(const std::string& aKeyword, const std::string& aSource) noexcept
    : fIndex(0)
{
    //Uppercase the keyword and the source phrase
    std::string processedKeyword = preprocessString(aKeyword);
    std::string processedSource = preprocessString(aSource);
```

```cpp
    //Allocate more spaces in fKeys based on the size of processed source phrase
    fKeys.reserve(processedSource.size());

    //Get the size of processed keyword and source phrase
    size_t keywordLength = processedKeyword.size();
    size_t sourceLength = processedSource.size();

    //Here, 'i' will be the iterator for the source phrase
    //While 'j' will be the iterator for the keyword phrase
    size_t j = 0;
    for (size_t i = 0; i < sourceLength; i++)
    {
        //Loop through sourceLength.size() - 1 times, append the keyword in the length
        //of source phrase. For example, source: Me is Simon, keyword: abc -> fKeys: ABCABCABC
        if (j >= keywordLength)
            j = 0;
        fKeys.push_back(processedKeyword[j]);
        j++;
    }

    // Ensures fKeys is properly sized
    assert(fKeys.size() == sourceLength);
}

/*
* Get the keyword character where the iterator is pointed on
* @param None
* @return char: The keyword character where the iterator is pointed on
*/
char KeyProvider::operator*() const noexcept
{
    assert(fIndex < fKeys.size()); // Prevent out-of-bounds access
    // The character at the fIndex
    return fKeys.at(fIndex);
}

/*
* Advance the iterator to one index and return the updated iterator
* @param None
* @return KeyProvider&: the updated iterator
*/
KeyProvider& KeyProvider::operator++() noexcept
{
    //Advance one index
    ++fIndex;
    //Return the updated iterator
```

```
 88      return *this;
 89  }
 90
 91  /*
 92  * Advance the iterator to one index and return the old iterator
 93  * @param None
 94  * @return KeyProvider&: the updated iterator
 95  */
 96  KeyProvider KeyProvider::operator++(int) noexcept
 97  {
 98      //Make a copy of old iterator
 99      KeyProvider temp = *this;
100      //Using operator++() and return the old iterator
101      ++(*this);
102      return temp;
103  }
104
105  /*
106  * Return the equality of the underlying collection and the position of
       this
107  * object and other KeyProvider object
108  * @param aOther (const KeyProvider&): Other KeyProvider object
109  * @return bool: The equality between objects
110  */
111  bool KeyProvider::operator==(const KeyProvider& aOther) const noexcept
112  {
113      return fIndex == aOther.fIndex && fKeys == aOther.fKeys;
114  }
115
116  /*
117  * Return the in-equality of the underlying collection and the position of
       this
118  * object and other KeyProvider object
119  * @param aOther (const KeyProvider&): Other KeyProvider object
120  * @return bool: The in-equality between objects
121  */
122  bool KeyProvider::operator!=(const KeyProvider& aOther) const noexcept
123  {
124      return !(*this == aOther);
125  }
126
127  /*
128  * Return a copy of 'this' iterator object positioned in the first index of
       the string
129  * @param None
130  * @return KeyProvider: A copy of 'this' iterator at the first index
131  */
132  KeyProvider KeyProvider::begin() const noexcept
133  {
```

```
134     //Get a copy and set the index as 0
135     KeyProvider temp = *this;
136     temp.fIndex = 0;
137     //Return the copy
138     return temp;
139 }
140 
141 /*
142 * Return a copy of 'this' iterator object positioned in the last index of the string
143 * @param None
144 * @return KeyProvider: A copy of 'this' iterator at the last index
145 */
146 KeyProvider KeyProvider::end() const noexcept
147 {
148     //Get a copy and set the index as the size of the keyword
149     KeyProvider temp = *this;
150     temp.fIndex = fKeys.size();
151     //Return the copy
152     return temp;
153 }
154 
```

```cpp

#include "VigenereForwardIterator.h"
#include <cassert>
/*
* @brief The decodeCurrentChar function will be use for handling decryptions from decoded letter to original letter
* based on the created mapping table given in the fMappingTable. It will choose which original letter would suit based on both
* the encoded letter and the keyword letter. For example, fKeys = 'A', fCurrentChar (encoded) = 'D' -> fCurrentChar (decrypted) = 'C'
* @param None
* @return void
*/
void VigenereForwardIterator::decodeCurrentChar() noexcept {
    //Check if the current character is alphabetic
    if (std::isalpha(fCurrentChar)) {
        //Get the row based on the current keyword letter
        size_t row = *fKeys - 'A';
        for (size_t i = 0; i < CHARACTERS; ++i) {
            //Iterating through the table
            //Check if the character at the row and the column matches with the current character
            if (fMappingTable[row][i] == std::toupper(fCurrentChar)) {
                //Check if the character is uppercase
                if (std::isupper(fCurrentChar))
                    //If then set the founded original character in uppercase
                    fCurrentChar = 'A' + i;
                else
                    //Else then set the founded original character in lowercase
                    fCurrentChar = 'a' + i;
                //Must break otherwise it will continuously running the loop -> cause error
                break;
            }
        }
        //Increment the keyword letter to one index
        fKeys++;
    }
}
/*
* @brief The encodeCurrentChar function will be use for handling encryptions from original letter to encoded letter
* based on the created mapping table given in the fMappingTable. It will choose which letter would suit based on both
* the source letter and the keyword letter. For example, fKeys = 'A', fCurrentChar (decrypted) = 'C' -> fCurrentChar (encoded) = 'D'
* @param None
```

```
40  * @return void
41  */
42  void VigenereForwardIterator::encodeCurrentChar() noexcept {
43      //Check if the current character is alphabetic
44      if (std::isalpha(fCurrentChar)) {
45          //Get the index value of the keyword letter
46          size_t row = *fKeys - 'A';
47          //Get the index value of the source letter
48          size_t col = std::toupper(fCurrentChar) - 'A';
49          //Get the encrypted keyword based on the index of source and
              keyword letter
50          char temp = fMappingTable[row][col];
51          //Check that if the current char is upper-case
52          if (std::isupper(fCurrentChar))
53              //If then assign the encrypted keyword
54              fCurrentChar = temp;
55          else
56              //Else then assign the encrypted keyword in lower case
57              fCurrentChar = std::tolower(temp);
58          //Increment the keyword letter to one index
59          fKeys++;
60      }
61  }
62
63  /*
64  * @brief Constructor of VigenereForwardIterator class, accepts the
      keyword, the phrase that wanted to be encrypted/decrypted and which mode
       to run
65  * @param aKeyword (const std::string&): The keyword
66  * @param aSource (const std::string&): The phrase that needs to encode/
      decode
67  * @param aMode (EVigenereMode): Vigenere Mode (Encode/Decode)
68  * @return None
69  */
70  VigenereForwardIterator::VigenereForwardIterator(const std::string&
      aKeyword, const std::string& aSource, EVigenereMode aMode) noexcept
71      : fMode(aMode),fKeys(aKeyword, aSource), fSource(aSource), fIndex(0)
72  {
73      //Initialize the table
74      initializeTable();
75      //Check if the encode/decode string is empty and the first character
          is empty
76      if (!fSource.empty() && std::isalpha(fSource.at(0)))
77      {
78          //If not then set the current char to be the first char of the
              phrase
79          fCurrentChar = fSource[fIndex];
80          if (fMode == EVigenereMode::Decode)
81              //If EVigenereMode is decode then call decodeCurrentChar()
```

```
82              decodeCurrentChar();
83          else
84              //Else call encodeCurrentChar()
85              encodeCurrentChar();
86      }
87  }
88
89  /*
90  * Get the keyword character where the iterator is pointed on
91  * @param None
92  * @return char: The keyword character where the iterator is pointed on
93  */
94  char VigenereForwardIterator::operator*() const noexcept {
95      return fCurrentChar;
96  }
97
98  /*
99  * Advance the iterator to one index and return the updated iterator
100 * @param None
101 * @return VigenereForwardIterator&: the updated iterator
102 */
103 VigenereForwardIterator& VigenereForwardIterator::operator++() noexcept {
104     //Ensurt that fIndex must not exceed the source phrase length
105     assert(fIndex++ < fSource.size());
106     //set the current char to be the character at the advanced index
107     fCurrentChar = fSource[fIndex];
108     //Must ensure that current char is alphabetic
109     if (std::isalpha(fCurrentChar))
110     {
111         if (fMode == EVigenereMode::Decode)
112             //If EVigenereMode is decode then call decodeCurrentChar()
113             decodeCurrentChar();
114         else
115             //Else call encodeCurrentChar()
116             encodeCurrentChar();
117     }
118     //Return the VigenereForwardIterator instance
119     return *this;
120 }
121
122 /*
123 * Advance the iterator to one index and return the old iterator
124 * @param None
125 * @return VigenereForwardIterator&: the old iterator
126 */
127 VigenereForwardIterator VigenereForwardIterator::operator++(int) noexcept ↪
      {
128     VigenereForwardIterator fTemp = *this;
129     ++(*this);
```

```
130     return fTemp;
131 }
132
133 /*
134 * Return the equality of the underlying collection and the position of this
135 * object and other VigenereForwardIterator object
136 * @param aOther (const VigenereForwardIterator&): Other VigenereForwardIterator object
137 * @return bool: The equality between objects
138 */
139 bool VigenereForwardIterator::operator==(const VigenereForwardIterator& aOther) const noexcept {
140     return fIndex == aOther.fIndex && fSource == aOther.fSource;
141 }
142
143 /*
144 * Return the in-equality of the underlying collection and the position of this
145 * object and other VigenereForwardIterator object
146 * @param aOther (const VigenereForwardIterator&): Other VigenereForwardIterator object
147 * @return bool: The in-equality between objects
148 */
149 bool VigenereForwardIterator::operator!=(const VigenereForwardIterator& aOther) const noexcept {
150     return !(*this == aOther);
151 }
152
153 /*
154 * Return a copy of 'this' iterator object positioned in the first index of the string
155 * @param None
156 * @return VigenereForwardIterator: A copy of 'this' iterator at the first index
157 */
158 VigenereForwardIterator VigenereForwardIterator::begin() const noexcept {
159     VigenereForwardIterator fTemp = *this;
160     fTemp.fIndex = 0;
161     return fTemp;
162 }
163
164 /*
165 * Return a copy of 'this' iterator object positioned in the last index of the string
166 * @param None
167 * @return VigenereForwardIterator: A copy of 'this' iterator at the last index
168 */
```

```
169 VigenereForwardIterator VigenereForwardIterator::end() const noexcept {
170     VigenereForwardIterator fTemp = *this;
171     fTemp.fIndex = fSource.size();
172     return fTemp;
173 }
174
175
```