```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
* @return void
*/
void VigenereForwardIterator::encodeCurrentChar() noexcept {
    //Check if the current character is alphabetic
    if (std::isalpha(fCurrentChar)) {
        //Get the index value of the keyword letter
        size_t row = *fKeys - 'A';
        //Get the index value of the source letter
        size_t col = std::toupper(fCurrentChar) - 'A';
        //Get the encrypted keyword based on the index of source and keyword letter
        char temp = fMappingTable[row][col];
        //Check that if the current char is upper-case
        if (std::isupper(fCurrentChar))
            //If then assign the encrypted keyword
            fCurrentChar = temp;
        else
            //Else then assign the encrypted keyword in lower case
            fCurrentChar = std::tolower(temp);
        //Increment the keyword letter to one index
        fKeys++;
    }
}

/*
* @brief Constructor of VigenereForwardIterator class, accepts the keyword, the phrase that wanted to be encrypted/decrypted and which mode to run
* @param aKeyword (const std::string&): The keyword
* @param aSource (const std::string&): The phrase that needs to encode/decode
* @param aMode (EVigenereMode): Vigenere Mode (Encode/Decode)
* @return None
*/
VigenereForwardIterator::VigenereForwardIterator(const std::string& aKeyword, const std::string& aSource, EVigenereMode aMode) noexcept
    : fMode(aMode),fKeys(aKeyword, aSource), fSource(aSource), fIndex(0)
{
    //Initialize the table
    initializeTable();
    //Check if the encode/decode string is empty and the first character is empty
    if (!fSource.empty() && std::isalpha(fSource.at(0)))
    {
        //If not then set the current char to be the first char of the phrase
        fCurrentChar = fSource[fIndex];
        if (fMode == EVigenereMode::Decode)
            //If EVigenereMode is decode then call decodeCurrentChar()
            decodeCurrentChar();
        else
            //Else call encodeCurrentChar()
            encodeCurrentChar();
    }
}

/*
* Get the keyword character where the iterator is pointed on
* @param None
* @return char: The keyword character where the iterator is pointed on
*/
char VigenereForwardIterator::operator*() const noexcept {
    return fCurrentChar;
}

/*
* Advance the iterator to one index and return the updated iterator
* @param None
* @return VigenereForwardIterator&: the updated iterator
*/
VigenereForwardIterator& VigenereForwardIterator::operator++() noexcept {
    //Ensurt that fIndex must not exceed the source phrase length
    assert(fIndex++ < fSource.size());
    //set the current char to be the character at the advanced index
    fCurrentChar = fSource[fIndex];
    //Must ensure that current char is alphabetic
    if (std::isalpha(fCurrentChar))
    {
        if (fMode == EVigenereMode::Decode)
            //If EVigenereMode is decode then call decodeCurrentChar()
            decodeCurrentChar();
        else
            //Else call encodeCurrentChar()
            encodeCurrentChar();
    }
    //Return the VigenereForwardIterator instance
    return *this;
}

/*
* Advance the iterator to one index and return the old iterator
* @param None
* @return VigenereForwardIterator&: the old iterator
*/
VigenereForwardIterator VigenereForwardIterator::operator++(int) noexcept {
    VigenereForwardIterator fTemp = *this;
    ++(*this);
    return fTemp;
}

/*
* Return the equality of the underlying collection and the position of this
* object and other VigenereForwardIterator object
* @param aOther (const VigenereForwardIterator&): Other VigenereForwardIterator object
* @return bool: The equality between objects
*/
bool VigenereForwardIterator::operator==(const VigenereForwardIterator& aOther) const noexcept {
    return fIndex == aOther.fIndex && fSource == aOther.fSource;
}

/*
* Return the in-equality of the underlying collection and the position of this
* object and other VigenereForwardIterator object
* @param aOther (const VigenereForwardIterator&): Other VigenereForwardIterator object
* @return bool: The in-equality between objects
*/
bool VigenereForwardIterator::operator!=(const VigenereForwardIterator& aOther) const noexcept {
    return !(*this == aOther);
}

/*
* Return a copy of 'this' iterator object positioned in the first index of the string
* @param None
* @return VigenereForwardIterator: A copy of 'this' iterator at the first index
*/
VigenereForwardIterator VigenereForwardIterator::begin() const noexcept {
    VigenereForwardIterator fTemp = *this;
    fTemp.fIndex = 0;
    return fTemp;
}

/*
* Return a copy of 'this' iterator object positioned in the last index of the string
* @param None
* @return VigenereForwardIterator: A copy of 'this' iterator at the last index
*/
```

```cpp
VigenereForwardIterator VigenereForwardIterator::end() const noexcept {
    VigenereForwardIterator fTemp = *this;
    fTemp.fIndex = fSource.size();
    return fTemp;
}


```