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, Design Pattern, and Iterators

Due date: November 17, 2023, 12:00

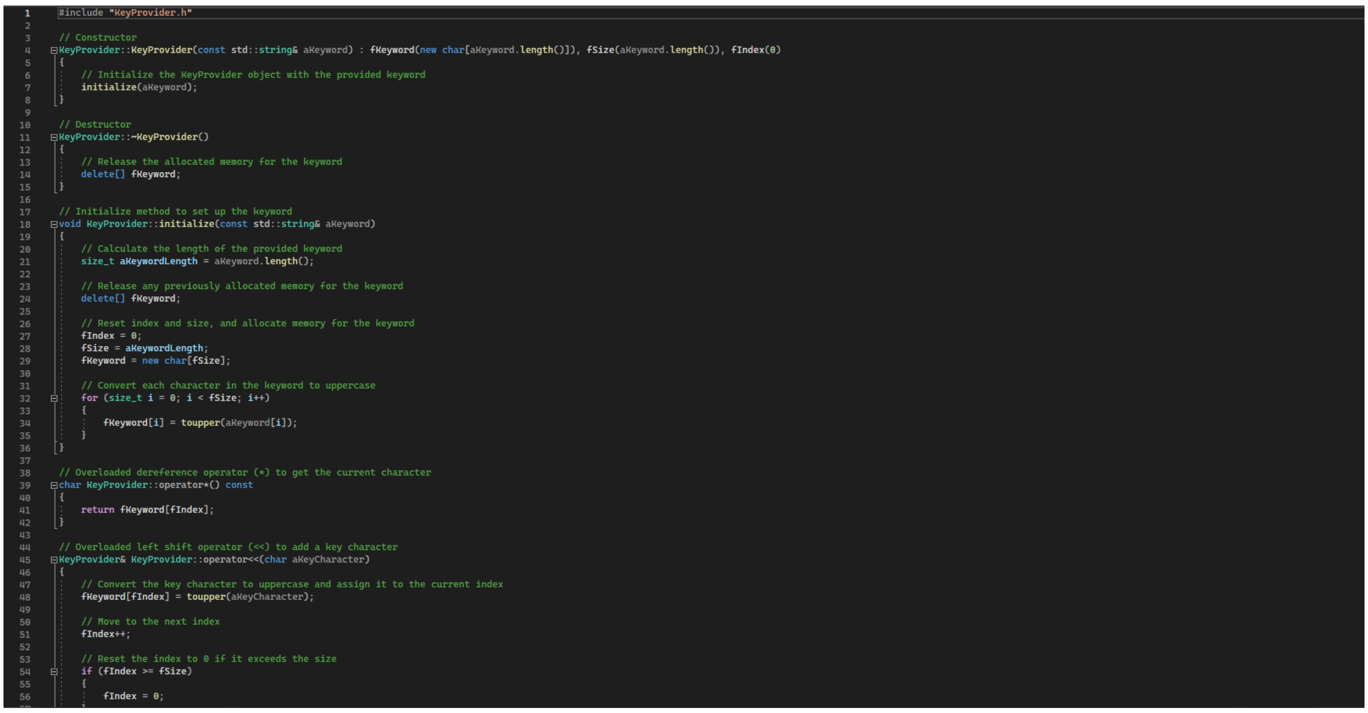
Lecturer: Dr. James Jackson

Your name: Tran Hoang Hai Anh Your student ID:_ 104177513

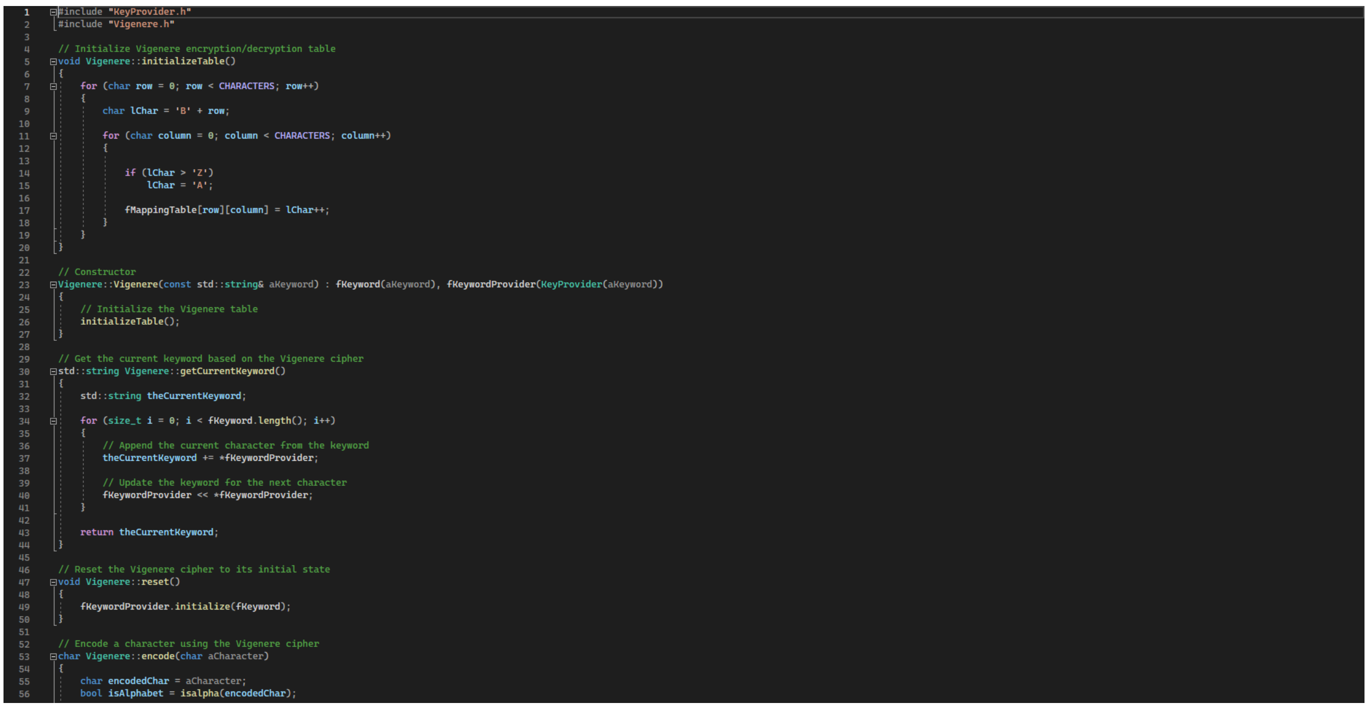
Check Tutorial	Mon 10:30	Mon 14:30	Tues 08:30	Tues 10:30	Tues 12:30	Tues 14:30	Tues 16:30	Wed 08:30	Wed 10:30	Wed 12:30	Thu 08:00
											✅

Marker's comments:

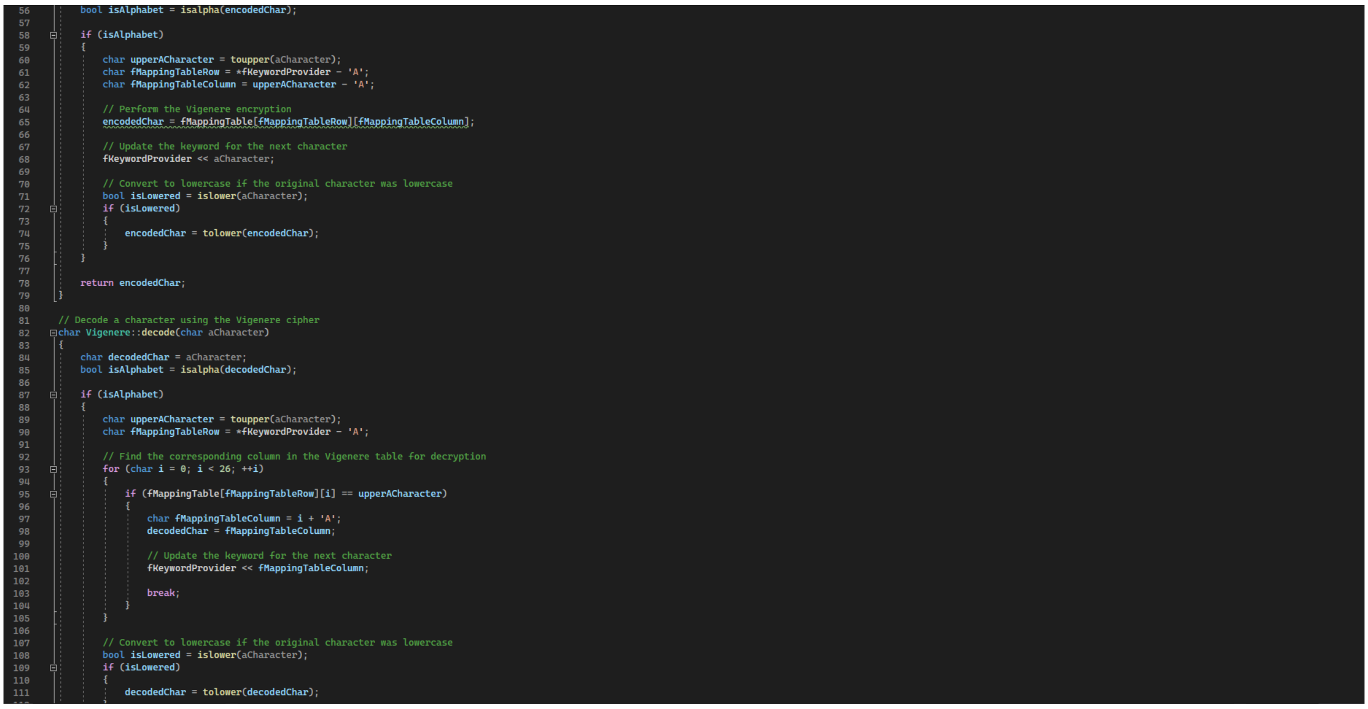
Problem	Marks	Obtained
1	68	
2	120	
3	56	
4	70	
Total	314	



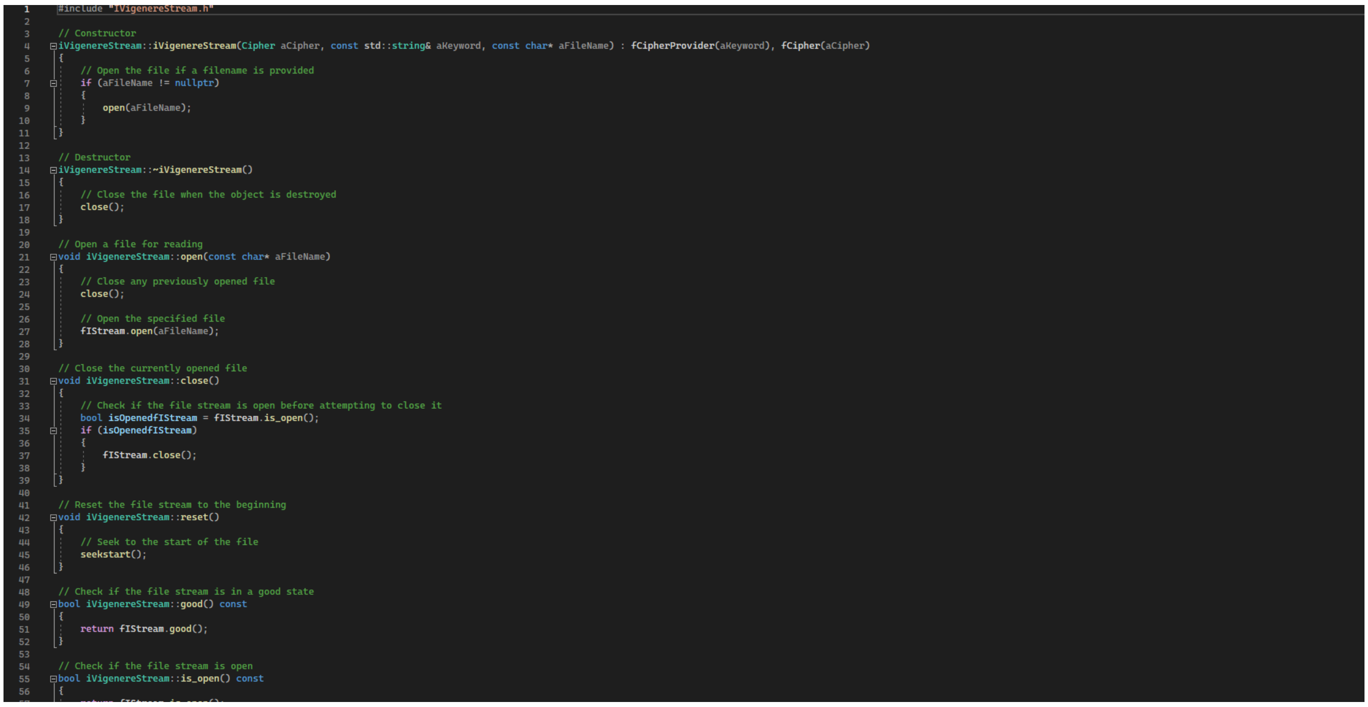
```cpp
#include "KeyProvider.h"

// Constructor
KeyProvider::KeyProvider(const std::string& aKeyword) : fKeyword(new char[aKeyword.length()]), fSize(aKeyword.length()), fIndex(0)
{
    // Initialize the KeyProvider object with the provided keyword
    initialize(aKeyword);
}

// Destructor
KeyProvider::~KeyProvider()
{
    // Release the allocated memory for the keyword
    delete[] fKeyword;
}

// Initialize method to set up the keyword
void KeyProvider::initialize(const std::string& aKeyword)
{
    // Calculate the length of the provided keyword
    size_t aKeywordLength = aKeyword.length();

    // Release any previously allocated memory for the keyword
    delete[] fKeyword;

    // Reset index and size, and allocate memory for the keyword
    fIndex = 0;
    fSize = aKeywordLength;
    fKeyword = new char[fSize];

    // Convert each character in the keyword to uppercase
    for (size_t i = 0; i < fSize; i++)
    {
        fKeyword[i] = toupper(aKeyword[i]);
    }
}

// Overloaded dereference operator (*) to get the current character
char KeyProvider::operator*() const
{
    return fKeyword[fIndex];
}

// Overloaded left shift operator (<<) to add a key character
KeyProvider& KeyProvider::operator<<(char aKeyCharacter)
{
    // Convert the key character to uppercase and assign it to the current index
    fKeyword[fIndex] = toupper(aKeyCharacter);

    // Move to the next index
    fIndex++;

    // Reset the index to 0 if it exceeds the size
    if (fIndex >= fSize)
    {
        fIndex = 0;
```



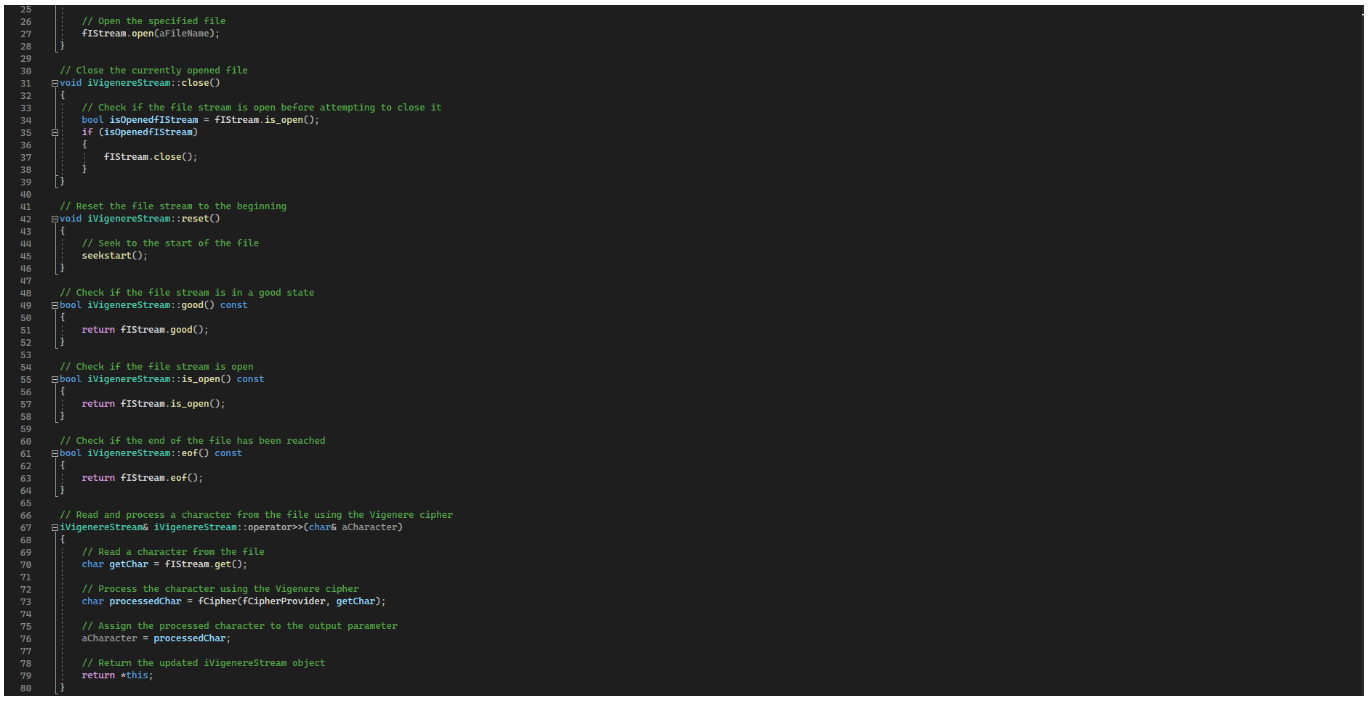
```cpp
#include "KeyProvider.h"
#include "Vigenere.h"

// Initialize Vigenere encryption/decryption table
void Vigenere::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++;
        }
    }
}

// Constructor
Vigenere::Vigenere(const std::string& aKeyword) : fKeyword(aKeyword), fKeywordProvider(KeyProvider(aKeyword))
{
    // Initialize the Vigenere table
    initializeTable();
}

// Get the current keyword based on the Vigenere cipher
std::string Vigenere::getCurrentKeyword()
{
    std::string theCurrentKeyword;

    for (size_t i = 0; i < fKeyword.length(); i++)
    {
        // Append the current character from the keyword
        theCurrentKeyword += *fKeywordProvider;

        // Update the keyword for the next character
        fKeywordProvider << *fKeywordProvider;
    }

    return theCurrentKeyword;
}

// Reset the Vigenere cipher to its initial state
void Vigenere::reset()
{
    fKeywordProvider.initialize(fKeyword);
}

// Encode a character using the Vigenere cipher
char Vigenere::encode(char aCharacter)
{
    char encodedChar = aCharacter;
    bool isAlphabet = isalpha(encodedChar);
```



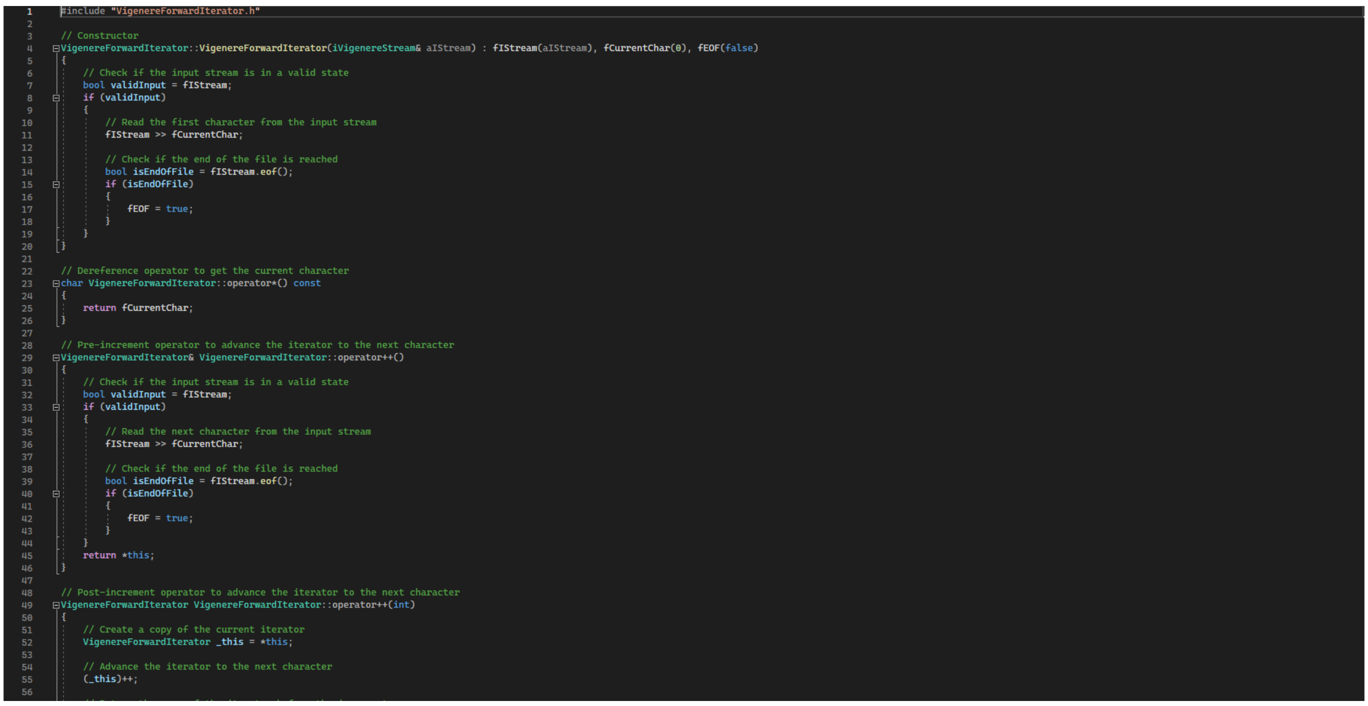
```cpp
    bool isAlphabet = isalpha(encodedChar);

    if (isAlphabet)
    {
        char upperACharacter = toupper(aCharacter);
        char fMappingTableRow = *fKeywordProvider - 'A';
        char fMappingTableColumn = upperACharacter - 'A';

        // Perform the Vigenere encryption
        encodedChar = fMappingTable[fMappingTableRow][fMappingTableColumn];

        // Update the keyword for the next character
        fKeywordProvider << aCharacter;

        // Convert to lowercase if the original character was lowercase
        bool isLowered = islower(aCharacter);
        if (isLowered)
        {
            encodedChar = tolower(encodedChar);
        }
    }

    return encodedChar;
}

// Decode a character using the Vigenere cipher
char Vigenere::decode(char aCharacter)
{
    char decodedChar = aCharacter;
    bool isAlphabet = isalpha(decodedChar);

    if (isAlphabet)
    {
        char upperACharacter = toupper(aCharacter);
        char fMappingTableRow = *fKeywordProvider - 'A';

        // Find the corresponding column in the Vigenere table for decryption
        for (char i = 0; i < 26; ++i)
        {
            if (fMappingTable[fMappingTableRow][i] == upperACharacter)
            {
                char fMappingTableColumn = i + 'A';
                decodedChar = fMappingTableColumn;

                // Update the keyword for the next character
                fKeywordProvider << fMappingTableColumn;

                break;
            }
        }

        // Convert to lowercase if the original character was lowercase
        bool isLowered = islower(aCharacter);
        if (isLowered)
        {
            decodedChar = tolower(decodedChar);
```



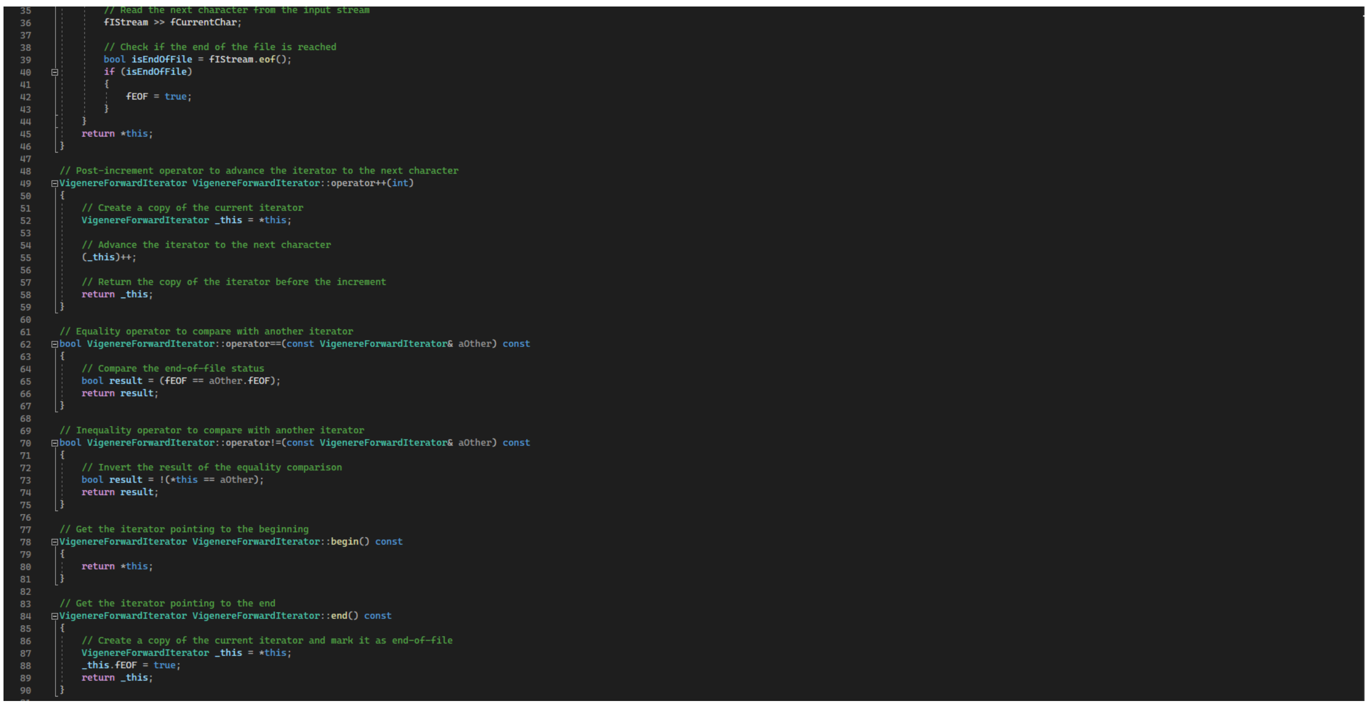
```cpp
#include "iVigenereStream.h"

// Constructor
iVigenereStream::iVigenereStream(Cipher aCipher, const std::string& aKeyword, const char* aFileName) : fCipherProvider(aKeyword), fCipher(aCipher)
{
    // Open the file if a filename is provided
    if (aFileName != nullptr)
    {
        open(aFileName);
    }
}

// Destructor
iVigenereStream::~iVigenereStream()
{
    // Close the file when the object is destroyed
    close();
}

// Open a file for reading
void iVigenereStream::open(const char* aFileName)
{
    // Close any previously opened file
    close();

    // Open the specified file
    fIStream.open(aFileName);
}

// Close the currently opened file
void iVigenereStream::close()
{
    // Check if the file stream is open before attempting to close it
    bool isOpenedfIStream = fIStream.is_open();
    if (isOpenedfIStream)
    {
        fIStream.close();
    }
}

// Reset the file stream to the beginning
void iVigenereStream::reset()
{
    // Seek to the start of the file
    seekstart();
}

// Check if the file stream is in a good state
bool iVigenereStream::good() const
{
    return fIStream.good();
}

// Check if the file stream is open
bool iVigenereStream::is_open() const
{
```



```cpp

    // Open the specified file
    fIStream.open(aFileName);
}

// Close the currently opened file
void iVigenereStream::close()
{
    // Check if the file stream is open before attempting to close it
    bool isOpenedfIStream = fIStream.is_open();
    if (isOpenedfIStream)
    {
        fIStream.close();
    }
}

// Reset the file stream to the beginning
void iVigenereStream::reset()
{
    // Seek to the start of the file
    seekstart();
}

// Check if the file stream is in a good state
bool iVigenereStream::good() const
{
    return fIStream.good();
}

// Check if the file stream is open
bool iVigenereStream::is_open() const
{
    return fIStream.is_open();
}

// Check if the end of the file has been reached
bool iVigenereStream::eof() const
{
    return fIStream.eof();
}

// Read and process a character from the file using the Vigenere cipher
iVigenereStream& iVigenereStream::operator>>(char& aCharacter)
{
    // Read a character from the file
    char getChar = fIStream.get();

    // Process the character using the Vigenere cipher
    char processedChar = fCipher(fCipherProvider, getChar);

    // Assign the processed character to the output parameter
    aCharacter = processedChar;

    // Return the updated iVigenereStream object
    return *this;
}
```



```cpp
#include "VigenereForwardIterator.h"

// Constructor
VigenereForwardIterator::VigenereForwardIterator(iVigenereStream& aIStream) : fIStream(aIStream), fCurrentChar(0), fEOF(false)
{
    // Check if the input stream is in a valid state
    bool validInput = fIStream;
    if (validInput)
    {
        // Read the first character from the input stream
        fIStream >> fCurrentChar;

        // Check if the end of the file is reached
        bool isEndOfFile = fIStream.eof();
        if (isEndOfFile)
        {
            fEOF = true;
        }
    }
}

// Dereference operator to get the current character
char VigenereForwardIterator::operator*() const
{
    return fCurrentChar;
}

// Pre-increment operator to advance the iterator to the next character
VigenereForwardIterator& VigenereForwardIterator::operator++()
{
    // Check if the input stream is in a valid state
    bool validInput = fIStream;
    if (validInput)
    {
        // Read the next character from the input stream
        fIStream >> fCurrentChar;

        // Check if the end of the file is reached
        bool isEndOfFile = fIStream.eof();
        if (isEndOfFile)
        {
            fEOF = true;
        }
    }
    return *this;
}

// Post-increment operator to advance the iterator to the next character
VigenereForwardIterator VigenereForwardIterator::operator++(int)
{
    // Create a copy of the current iterator
    VigenereForwardIterator _this = *this;

    // Advance the iterator to the next character
    (_this)++;

```



```cpp
            // Read the next character from the input stream
            fIStream >> fCurrentChar;

            // Check if the end of the file is reached
            bool isEndOfFile = fIStream.eof();
            if (isEndOfFile)
            {
                fEOF = true;
            }
        }
        return *this;
}

// Post-increment operator to advance the iterator to the next character
VigenereForwardIterator VigenereForwardIterator::operator++(int)
{
    // Create a copy of the current iterator
    VigenereForwardIterator _this = *this;

    // Advance the iterator to the next character
    (_this)++;

    // Return the copy of the iterator before the increment
    return _this;
}

// Equality operator to compare with another iterator
bool VigenereForwardIterator::operator==(const VigenereForwardIterator& aOther) const
{
    // Compare the end-of-file status
    bool result = (fEOF == aOther.fEOF);
    return result;
}

// Inequality operator to compare with another iterator
bool VigenereForwardIterator::operator!=(const VigenereForwardIterator& aOther) const
{
    // Invert the result of the equality comparison
    bool result = !(*this == aOther);
    return result;
}

// Get the iterator pointing to the beginning
VigenereForwardIterator VigenereForwardIterator::begin() const
{
    return *this;
}

// Get the iterator pointing to the end
VigenereForwardIterator VigenereForwardIterator::end() const
{
    // Create a copy of the current iterator and mark it as end-of-file
    VigenereForwardIterator _this = *this;
    _this.fEOF = true;
    return _this;
}
```