Project Report

Title: Encryption/Decryption Application
Group Members: Aawaiz Ali, Raahim Hussain, Hashir Ahmed Khan
Date: 14th May 2023

1. Introduction
   1.1 Project Background
   - In today's digital age, the need for secure data transmission and storage has become paramount.
   - The Encryption/Decryption Application project was initiated to address the growing concerns regarding data security.

   1.2 Objectives
   - Develop an encryption/decryption application that is easy to use and provides strong security features.
   - Implement industry-standard encryption algorithms to ensure data confidentiality and integrity.
   - Adhere to object-oriented programming (OOP) principles to create a modular and maintainable codebase.

2. Project Background
   - The team's immense interest in Cyber Security encouraged the idea of the project.
   - Existing encryption tools were too complex for the team's understanding.
   - The team felt the need to create a simple but complex algorithm that encrypts daily use strings.

3. Methodology
   3.1 Object-Oriented Programming Principles
   - Inheritance:
     - The classes `AtbashCipher`, `CaesarCipher`, and `VigenereCipher` inherit from the base class `EncryptionAlgorithm`.
     - Inheritance allows these classes to inherit the common functionality defined in the base class, promoting code reuse and modularity.
   - Abstraction:
     - The base class `EncryptionAlgorithm` provides an abstraction of the common functionalities required for encryption and decryption.
     - The derived classes encapsulate the implementation details specific to each encryption algorithm, allowing the user to interact with them through the common interface provided by the base class.
     - Abstraction simplifies the usage of different encryption algorithms by providing a high-level interface without exposing unnecessary details.
   - Polymorphism:
     - The base class `EncryptionAlgorithm` declares two pure virtual functions: `encrypt` and `decrypt`.
     - The derived classes `AtbashCipher`, `CaesarCipher`, and `VigenereCipher` override these virtual functions with their specific implementations.

    - Polymorphism allows these derived classes to be treated as instances of the base class, enabling flexibility and extensibility in the code.
- Encapsulation:
    - The member functions and variables of each class are encapsulated within their respective classes.
    - Data members such as `key` in the `CaesarCipher` and `VigenereCipher` classes are encapsulated to ensure data integrity and control access.
    - Encapsulation helps in hiding the internal implementation details and providing a clear interface to interact with the classes.
- Destructor:
    - The base class `EncryptionAlgorithm` declares a virtual destructor, `~EncryptionAlgorithm()`.
    - This ensures that when objects of derived classes are destroyed, their appropriate destructors are called, preventing memory leaks and ensuring proper resource management.
- Overall, the implementation of OOP principles enhances code organization, reusability, maintainability, and extensibility. It promotes modularity by separating concerns and providing a clear hierarchy between classes.

4. Results

4.1 Encryption/Decryption Functionality

1. Atbash Cipher

- Results:
    - The Atbash Cipher reverses the alphabet, substituting each letter with its corresponding opposite.
    - It provides a simple form of encryption by swapping each letter with its mirror image.
    - The algorithm produces encrypted text that can be decrypted by applying the same process in reverse
- Benefits:
    - The Atbash Cipher is easy to implement and understand.
    - It can be used for simple encryption tasks where strong security is not a primary concern.
    - The algorithm helps users protect sensitive information from casual viewers or maintain privacy in basic communication.

2. Caesar Cipher:

- Results:
    - The Caesar Cipher is a substitution cipher that shifts each letter by a fixed number of positions in the alphabet.
    - It replaces each letter with a letter that is a fixed number of positions down the alphabet.
    - The algorithm provides encryption and decryption functionality based on the selected shift key.
- Benefits:
    - The Caesar Cipher is a widely known and historically significant encryption technique.

- It offers a simple implementation with the ability to choose different shift keys.
- The algorithm can be used for basic encryption purposes or to introduce students to the concept of cryptography.

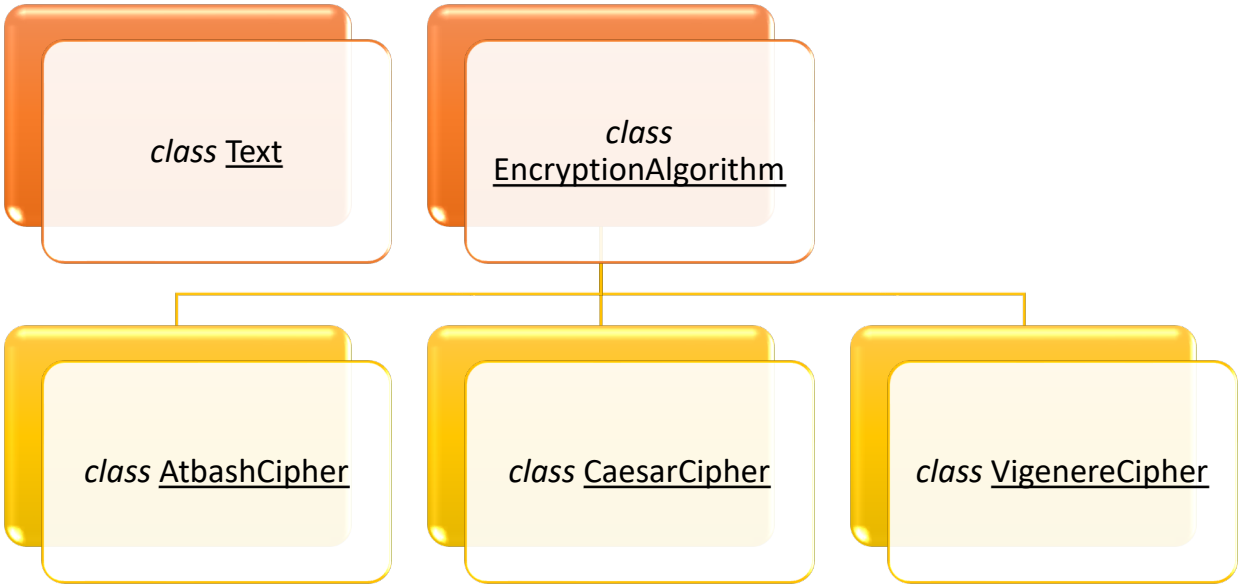
3. Vigenere Cipher:

- Results:
  - The Vigenere Cipher is a polyalphabetic substitution cipher that uses a keyword to determine the shift for each letter.
  - It uses a series of Caesar Ciphers based on different letters of a keyword.
  - The algorithm provides encryption and decryption functionality based on the chosen keyword.
- Benefits:
  - The Vigenere Cipher improves upon the Caesar Cipher by introducing a keyword for greater encryption strength.
  - It offers a more complex encryption scheme that is resistant to frequency analysis attacks.
  - The algorithm provides users with a stronger encryption method for protecting sensitive information.

5. Future Scope

- Addition of a Graphical User Interface.
- Addition of more security to bypass the login phase, such as biometric recognition.
- Records can be added to a well-maintained database instead of a .txt file.

UML Diagram



Classes

```
class EncryptionAlgorithm {
public:
    virtual string encrypt(string plaintext, string username) = 0;
    virtual string decrypt(string ciphertext, string username) = 0;

    virtual ~EncryptionAlgorithm() {}

};
```

The `EncryptionAlgorithm` class is an abstract base class that serves as a blueprint for different encryption algorithms. It declares two pure virtual functions, `encrypt` and `decrypt`, which are responsible for performing encryption and decryption operations, respectively. The class also provides a virtual destructor to ensure proper cleanup when derived classes are destroyed.

The class has the following components:

- Access Specifier: The `public` access specifier indicates that the member functions and variables are accessible from outside the class.

- Pure Virtual Functions: The class declares two pure virtual functions:
  - `encrypt`: This function takes a plaintext string and a username as input parameters and returns the encrypted ciphertext as a string. The specific implementation of encryption is left to the derived classes that inherit from `EncryptionAlgorithm`.
  - `decrypt`: This function takes a ciphertext string and a username as input parameters and returns the decrypted plaintext as a string. The specific implementation of decryption is left to the derived classes.

- Virtual Destructor: The class declares a virtual destructor using the `virtual ~EncryptionAlgorithm() {}` syntax. This allows proper destruction of derived class objects through a base class pointer. The destructor is empty (no implementation) since there are no specific cleanup requirements in the base class.

Overall, the `EncryptionAlgorithm` class provides an interface for encryption algorithms, specifying the functions that derived classes must implement. It defines the common behavior expected from any encryption algorithm, allowing for polymorphic usage and providing a consistent interface for encryption and decryption operations.

```
class AtbashCipher : public EncryptionAlgorithm {
public:
    string encrypt(string plaintext, string username){

        string ciphertext = "";
        for (int i = 0; i < plaintext.length(); i++) {
            char c = plaintext[i];
            if (isalpha(c)) {
                if (isupper(c)) {
                    ciphertext += static_cast<char>('Z' - (c - 'A'));
                } else {
                    ciphertext += static_cast<char>('z' - (c - 'a'));
                }
            } else {
                ciphertext += c;
            }
        }

        ofstream file("encrypted.txt", ios::app);
        if (file.is_open()) {
            file << "Username: " << username << "\n";
            file << "Encrypted Text: " << ciphertext << "\n";
            file << "Plain Text: " << plaintext << "\n";
            file.close();
        }

        return ciphertext;
    }
    string decrypt(string ciphertext, string username) {

        string plaintext = "";
        int n = ciphertext.length();
        for (int i = 0; i < n; i++) {
            char c = ciphertext[i];
            if (isalpha(c)) {
                if (isupper(c)) {
                    plaintext += static_cast<char>('Z' - (c - 'A'));
                } else {
                    plaintext += static_cast<char>('z' - (c - 'a'));
```

```
                }
            } else {
                plaintext += c;
            }
        }

        ofstream file("decrypted.txt", ios::app);
        if (file.is_open()) {
            file << "Username: " << username << "\n";
            file << "Encrypted Text: " << ciphertext << "\n";
            file << "Plain Text: " << plaintext << "\n";
            file.close();
        }

        return plaintext;
    }
};
```

The `AtbashCipher` class is a derived class of the `EncryptionAlgorithm` base class, implementing the Atbash cipher encryption and decryption algorithms. It overrides the pure virtual functions `encrypt` and `decrypt` defined in the base class.

The class has the following components:

- Class Inheritance: The `AtbashCipher` class is derived from the `EncryptionAlgorithm` base class using the `public` access specifier. This means that `AtbashCipher` inherits the public members (functions and variables) from `EncryptionAlgorithm`.

- Function Overrides: The class overrides the `encrypt` and `decrypt` functions inherited from the base class. These functions implement the specific logic for encrypting and decrypting using the Atbash cipher.

- Encryption Function (`encrypt`): This function takes a plaintext string and a username as input parameters. It iterates over each character of the plaintext string and checks if it is an alphabetic character. If it is, the character is converted to its Atbash cipher equivalent by subtracting its ASCII value from either 'Z' or 'z' depending on its case. The resulting cipher character is appended to the `ciphertext` string. Non-alphabetic characters are appended as is. The encrypted ciphertext is then stored in a file along with the username and plaintext for record-keeping.

- Decryption Function (`decrypt`): This function takes a ciphertext string and a username as input parameters. It iterates over each character of the ciphertext string and checks if it is an alphabetic character. If it is, the character is converted to its original plaintext character by subtracting its ASCII value from either 'Z' or 'z' depending on its case. The resulting plaintext character is appended to the `plaintext` string. Non-alphabetic characters are appended as is. The decrypted plaintext is then stored in a file along with the username and encrypted ciphertext for record-keeping.

- File Output: Both the `encrypt` and `decrypt` functions open a file ("encrypted.txt" for encryption and "decrypted.txt" for decryption) in append mode using an `ofstream`. If the file is successfully opened, the function writes the username, encrypted or decrypted text, and plaintext (in the case of encryption) to the file. The file is then closed.

Overall, the `AtbashCipher` class provides the implementation for encrypting and decrypting text using the Atbash cipher algorithm. It inherits the interface from the `EncryptionAlgorithm` base class and overrides the required functions. The class also includes functionality to store the encrypted and decrypted text in separate files for record-keeping purposes.

```
class CaesarCipher : public EncryptionAlgorithm {
private:
    int key;
public:
    CaesarCipher(int k) {
        key = k;
    }
    string encrypt(string plaintext, string username) {
        string ciphertext = "";
        for (int i = 0; i < plaintext.length(); i++) {
            char ch = plaintext[i];
            if (isalpha(ch)) {
                ch = toupper(ch);
                ch = ((ch - 'A' + key) % 26) + 'A';
            }
            ciphertext += ch;
        }
        ofstream file("encrypted.txt", ios::app);
        if (file.is_open()) {
            file << "Username: " << username << "\n";
            file << "Encrypted Text: " << ciphertext << "\n";
            file << "Plain Text: " << plaintext << "\n";
            file.close();
        }
        return ciphertext;
```

```
    }
    string decrypt(string ciphertext, string username) {
        string plaintext = "";
        for (int i = 0; i < ciphertext.length(); i++) {
            char ch = ciphertext[i];
            if (isalpha(ch)) {
                ch = toupper(ch);
                ch = ((ch - 'A' - key + 26) % 26) + 'A';
            }
            plaintext += ch;
        }
        ofstream file("decrypted.txt", ios::app);
        if (file.is_open()) {
            file << "Username: " << username << "\n";
            file << "Encrypted Text: " << ciphertext << "\n";
            file << "Plain Text: " << plaintext << "\n";
            file.close();
        }
        return plaintext;
    }
};
```

The `CaesarCipher` class is a derived class of the `EncryptionAlgorithm` base class, implementing the Caesar cipher encryption and decryption algorithms. It has an additional private member variable `key` to store the encryption key.

Here's a description of the class:

- Class Inheritance: The `CaesarCipher` class inherits publicly from the `EncryptionAlgorithm` base class, which means it inherits the public members of the base class.

- Constructor: The class has a constructor that takes an integer `k` as a parameter. This constructor initializes the private member variable `key` with the value of `k`.

- Encryption Function (`encrypt`): This function takes a plaintext string and a username as input parameters. It iterates over each character of the plaintext string. If the character is an alphabetic character, it is converted to uppercase using `toupper()`. The character is then shifted by adding the encryption key (`key`) to its ASCII value. The result is wrapped around within the range of uppercase alphabets ('A' to 'Z') using modulo arithmetic. The shifted character is appended to the `ciphertext` string. Non-alphabetic characters are appended as is. The encrypted ciphertext is then stored in a file along with the username and plaintext for record-keeping.

- Decryption Function (`decrypt`): This function takes a ciphertext string and a username as input parameters. It iterates over each character of the ciphertext string. If the character is an alphabetic character, it is converted to uppercase using `toupper()`. The character is then shifted back to its original position by subtracting the encryption key (`key`) from its ASCII value. The result is wrapped around within the range of uppercase alphabets ('A' to 'Z') using modulo arithmetic. The shifted character is appended to the `plaintext` string. Non-

alphabetic characters are appended as is. The decrypted plaintext is then stored in a file along with the username and encrypted ciphertext for record-keeping.

- File Output: Both the `encrypt` and `decrypt` functions open a file ("encrypted.txt" for encryption and "decrypted.txt" for decryption) in append mode using an `ofstream`. If the file is successfully opened, the function writes the username, encrypted or decrypted text, and plaintext (in the case of encryption) to the file. The file is then closed.

Overall, the `CaesarCipher` class provides the implementation for encrypting and decrypting text using the Caesar cipher algorithm. It inherits the interface from the `EncryptionAlgorithm` base class and overrides the required functions. The class also includes functionality to store the encrypted and decrypted text in separate files for record-keeping purposes.

```
class VigenereCipher : public EncryptionAlgorithm {
private:
    string key;
public:
    VigenereCipher(string k) {
        key = k;
    }
    string encrypt(string plaintext, string username) {
        string ciphertext = "";
        int j = 0;
        for (int i = 0; i < plaintext.length(); i++) {
            char ch = plaintext[i];
            if (isalpha(ch)) {
                ch = toupper(ch);
                ch = ((ch - 'A' + (key[j] - 'A')) % 26) + 'A';
                j = (j + 1) % key.length();
            }
            ciphertext += ch;
        }
        ofstream file("encrypted.txt", ios::app);
        if (file.is_open()) {
            file << "Username: " << username << "\n";
            file << "Encrypted Text: " << ciphertext << "\n";
            file << "Plain Text: " << plaintext << "\n";
            file.close();
        } else {
            cout << "Error writing into file";
        }
        return ciphertext;
    }
    string decrypt(string ciphertext, string username) {
        string plaintext = "";
        int j = 0;
        for (int i = 0; i < ciphertext.length(); i++) {
            char ch = ciphertext[i];
            if (isalpha(ch)) {
```

```
                ch = toupper(ch);
                ch = ((ch - 'A' - (key[j] - 'A') + 26) % 26) + 'A';
                j = (j + 1) % key.length();
            }
            plaintext += ch;
        }
        ofstream file("decrypted.txt", ios::app);
        if (file.is_open()) {
            file << "Username: " << username << "\n";
            file << "Encrypted Text: " << ciphertext << "\n";
            file << "Plain Text: " << plaintext << "\n";
            file.close();
        }
        return plaintext;
    }
};
```

The `VigenereCipher` class is a derived class of the `EncryptionAlgorithm` base class, implementing the Vigenere cipher encryption and decryption algorithms. It has an additional private member variable `key` to store the encryption key.

Here's a description of the class:

- Class Inheritance: The `VigenereCipher` class inherits publicly from the `EncryptionAlgorithm` base class, which means it inherits the public members of the base class.

- Constructor: The class has a constructor that takes a string `k` as a parameter. This constructor initializes the private member variable `key` with the value of `k`.

- Encryption Function (`encrypt`): This function takes a plaintext string and a username as input parameters. It iterates over each character of the plaintext string. If the character is an alphabetic character, it is converted to uppercase using `toupper()`. The character is then shifted by adding the corresponding key character's value to its ASCII value. The key character is obtained using the `j` index, which cycles through the key string using modulo arithmetic. The result is wrapped around within the range of uppercase alphabets ('A' to 'Z') using modulo arithmetic. The shifted character is appended to the `ciphertext` string. Non-alphabetic characters are appended as is. The encrypted ciphertext is then stored in a file along with the username and plaintext for record-keeping.

- Decryption Function (`decrypt`): This function takes a ciphertext string and a username as input parameters. It iterates over each character of the ciphertext string. If the character is an alphabetic character, it is converted to uppercase using `toupper()`. The character is then shifted back to its original position by subtracting the corresponding key character's value from its ASCII value. The key character is obtained using the `j` index, which cycles through the key string using modulo arithmetic. The result is wrapped around within the range of uppercase alphabets ('A' to 'Z') using modulo arithmetic. The shifted character is appended to the `plaintext` string. Non-alphabetic characters are appended as is. The decrypted

plaintext is then stored in a file along with the username and encrypted ciphertext for record-keeping.

- File Output: Both the `encrypt` and `decrypt` functions open a file ("encrypted.txt" for encryption and "decrypted.txt" for decryption) in append mode using an `ofstream`. If the file is successfully opened, the function writes the username, encrypted or decrypted text, and plaintext (in the case of encryption) to the file. The file is then closed.

Overall, the `VigenereCipher` class provides the implementation for encrypting and decrypting text using the Vigenere cipher algorithm. It inherits the interface from the `EncryptionAlgorithm` base class and overrides the required functions. The class also includes functionality to store the encrypted and decrypted text in separate files for record-keeping purposes.

```
class Text
{
    private:
        string content;
    public:
        Text(string t)
        {
            content = t;
        }

    string getContent()
    {
        return content;
    }

    void setContent(string t)
    {
        content = t;
    }
};
```

The `Text` class represents a piece of text and provides methods to get and set its content.

Here's a breakdown of the class:

- Private Member Variable: The class has a private member variable `content` of type string. It stores the content of the text.

- Constructor: The class has a constructor that takes a string `t` as a parameter. When an object of the `Text` class is created, the constructor initializes the `content` member variable with the value of `t`.

- `getContent` Method: This method returns the content of the text. It retrieves the value of the `content` member variable and returns it as a string.

- `setContent` Method: This method updates the content of the text. It takes a string `t` as a parameter and assigns the value of `t` to the `content` member variable.

The `Text` class provides a simple encapsulation of text content, allowing you to create a `Text` object with an initial content and then retrieve or update that content using the `getContent` and `setContent` methods, respectively.

Functions

```
void DecorativeLines(){
    cout << endl;
    cout <<
"*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=*=" <<
endl << endl;
}
```

The given function `DecorativeLines` is a void function that does not take any arguments. It is responsible for printing a decorative line pattern on the console.

The function starts by outputting a newline character (`endl`), which moves the cursor to a new line. This is done to create some space before the decorative line pattern.

The next line of code prints a long string of characters using the `cout` statement. The string consists of alternating asterisks (`*`) and equal signs (`=`), forming a repeated pattern. The pattern is meant to create a visually appealing decorative line.

The pattern is quite long, indicated by the repetition of the `*=` sequence. This can be customized by adding or removing characters as desired.

After printing the decorative line pattern, another newline character is outputted using `endl`, creating some space after the pattern.

Overall, this function is designed to add a decorative line pattern to visually separate or highlight certain sections of output in a console-based program.

```
void EncryptMenu(string name){


    string content;
    int choice;

    beginning:

    DecorativeLines();

    cout << "Enter the string you would like to encrypt: " << endl;
    fflush(stdin);
    getline(cin, content);
```

```
    DecorativeLines();

    cout << "Which encryption algorithm would you like to use? " << endl << "1. Caesar encryption" << endl << "2. Vignere encryption" << endl << "Atbash encryption" << endl;
    fflush(stdin);
    cin >> choice;

    switch (choice)
    {
        case 1:
        {
            int key;

            DecorativeLines();

            cout << "Enter encryption key: " << endl;
            cin >> key;

            Text plaintext(content);

            EncryptionAlgorithm* algo1 = new CaesarCipher(key);
            string ciphertext = algo1->encrypt(plaintext.getContent(), name);

            DecorativeLines();

            cout << "Encrypted text: " << ciphertext << endl;

            break;
        }
        case 2:
        {
            string key;

            DecorativeLines();

            cout << "Enter encryption key: " << endl;
            cin >> key;

            Text plaintext(content);

            EncryptionAlgorithm* algo1 = new VigenereCipher(key);
            string ciphertext = algo1->encrypt(plaintext.getContent(), name);

            DecorativeLines();

            cout << "Encrypted text: " << ciphertext << endl;

            break;
        }
```

```
        case 3:
        {
            Text plaintext(content);

            EncryptionAlgorithm* algo1 = new AtbashCipher();
            string ciphertext = algo1->encrypt(plaintext.getContent(), name);

            DecorativeLines();

            cout << "Encrypted text: " << ciphertext << endl;

            break;

        }
        default:
        {
            cout << "Invalid choice " << endl;
            goto beginning;
            break;
        }
    }
}
```

The `EncryptMenu` function is a menu-driven program that prompts the user to enter a string for encryption and choose an encryption algorithm. It supports three encryption algorithms: Caesar encryption, Vigenere encryption, and Atbash encryption.

The function begins by displaying a decorative line pattern using the `DecorativeLines` function.

The user is then prompted to enter the string they want to encrypt, and the input is stored in the `content` variable.

Another decorative line is displayed to visually separate the sections of the menu.

The user is presented with a menu of encryption algorithm choices: Caesar encryption, Vigenere encryption, and Atbash encryption. The user's choice is stored in the `choice` variable.

Based on the user's choice, the function performs the following steps:
- Displays a decorative line.
- Prompts the user to enter the encryption key or passphrase.
- Reads the input and stores it in the appropriate variable.
- Converts the user-provided string into a `Text` object.
- Creates an instance of the corresponding encryption algorithm using `new`.
- Calls the `encrypt` function of the algorithm, passing the plaintext content and the `name` parameter, and stores the resulting ciphertext in the `ciphertext` variable.
- Displays another decorative line.
- Prints the encrypted text to the console.

If the user enters an invalid choice, an error message is displayed, and the program returns to the beginning of the menu.

Overall, this function provides a concise menu for encrypting a user-provided string using different encryption algorithms, displaying the encrypted text as output.

```
void DecryptMenu(string name){
    string content;
    int choice;

    beginning:

    DecorativeLines();

    cout << "Enter the string you would like to decrypt: " << endl;
    fflush(stdin);
    getline(cin, content);

    DecorativeLines();

    cout << "Which encryption algorithm did you use? " << endl << "1. Caesar
encryption" << endl << "2. Vignere encryption" << endl << "Atbash encryption" <<
endl;
    cin >> choice;

    switch (choice)
    {
        case 1:
        {
            int key;

            DecorativeLines();

            cout << "Enter encryption key: " << endl;
            cin >> key;

            EncryptionAlgorithm* algo1 = new CaesarCipher(key);
            string plaintext = algo1->decrypt(content, name);

            DecorativeLines();

            cout << "Decrypted text: " << plaintext << endl;

            break;
        }
        case 2:
        {
            string key;

            DecorativeLines();
```

```
            cout << "Enter encryption key: " << endl;
            cin >> key;

            EncryptionAlgorithm* algo1 = new VigenereCipher(key);
            string plaintext = algo1->encrypt(content, name);

            DecorativeLines();

            cout << "Encrypted text: " << plaintext << endl;

            break;
        }
        case 3:
        {
            DecorativeLines();

            EncryptionAlgorithm* algo1 = new AtbashCipher();
            string plaintext = algo1->encrypt(content, name);

            DecorativeLines();

            cout << "Encrypted text: " << plaintext << endl;

            break;
        }
        default:
        {
            cout << "Invalid choice " << endl;
            goto beginning;
            break;
        }
    }
}
```

The `DecryptMenu` function is a menu-driven program that prompts the user to enter a string for decryption and choose the corresponding encryption algorithm used for encryption. It supports three encryption algorithms: Caesar encryption, Vigenere encryption, and Atbash encryption.

The function begins by displaying a decorative line pattern using the `DecorativeLines` function.

The user is then prompted to enter the string they want to decrypt, and the input is stored in the `content` variable.

Another decorative line is displayed to visually separate the sections of the menu.

The user is presented with a menu of encryption algorithm choices: Caesar encryption, Vigenere encryption, and Atbash encryption. The user's choice is stored in the `choice` variable.

Based on the user's choice, the function performs the following steps:
- Displays a decorative line.
- Prompts the user to enter the encryption key or passphrase.
- Reads the input and stores it in the appropriate variable.
- Creates an instance of the corresponding encryption algorithm using `new`.
- Calls the `decrypt` function of the algorithm, passing the ciphertext and the `name` parameter, and stores the resulting plaintext in the `plaintext` variable.
- Displays another decorative line.
- Prints the decrypted text to the console.

If the user enters an invalid choice, an error message is displayed, and the program returns to the beginning of the menu.

Overall, this function provides a concise menu for decrypting a user-provided string using different encryption algorithms, displaying the decrypted text as output.

```
void CheckHistory(string name){
    ifstream encryptedFile("encrypted.txt");
    ifstream decryptedFile("decrypted.txt");
    string line;
    bool foundUser = false;

    while (getline(encryptedFile, line)) {
        if (line.find("Username: " + name) != string::npos) {
            foundUser = true;
            cout << "Encrypted messages for " << name << ":\n";
            getline(encryptedFile, line);
            cout << line << endl;
            getline(encryptedFile, line);
            cout << line << endl << endl;
        }
    }
    if (!foundUser) {
        cout << "No encrypted messages found for " << name << endl << endl;
    }

    foundUser = false;

    while (getline(decryptedFile, line)) {
        if (line.find("Username: " + name) != string::npos) {
            foundUser = true;
            cout << "Decrypted messages for " << name << ":\n";
            getline(decryptedFile, line);
            cout << line << endl;
            getline(decryptedFile, line);
            cout << line << endl << endl;
```

```
        }
    }
    if (!foundUser) {
        cout << "No decrypted messages found for " << name << endl << endl;
    }
}
```

The `CheckHistory` function is responsible for checking the history of encrypted and decrypted messages for a given username (`name`). It reads the contents of two files, `encrypted.txt` and `decrypted.txt`, to search for messages associated with the specified username.

The function begins by opening two file streams, `encryptedFile` and `decryptedFile`, to read the contents of the respective files.

A string variable `line` is declared to store each line read from the files.

A boolean variable `foundUser` is initialized as `false` to keep track of whether any messages are found for the specified username.

The function then enters a `while` loop, reading each line from the `encryptedFile` stream. It checks if the line contains the substring "Username: " followed by the specified `name`. If a match is found, `foundUser` is set to `true`, and the encrypted messages associated with the username are printed to the console.

The function follows a similar process for the `decryptedFile` stream, checking for the presence of the username and printing the corresponding decrypted messages if found.

If no messages are found for the username in either file, appropriate messages are displayed indicating the absence of encrypted or decrypted messages.

Overall, this function allows for checking and displaying the history of encrypted and decrypted messages for a given username, providing information about the messages stored in the `encrypted.txt` and `decrypted.txt` files.

```
void signup()
{
    DecorativeLines();
    srand(time(NULL));
    const int PATTERN_LENGTH = 8;
    const float SLEEP_TIME = 0.75;
    char pattern[PATTERN_LENGTH];
    string username;

    start:

        cout << "Enter username: ";
    cin >> username;
    ifstream infile("passwords.txt");
    string line;
```

```
    bool usernameExists = false;
    while (getline(infile, line))
    {
        if (line.find("Username: " + username) == 0)
        {
            usernameExists = true;
            break;
        }
    }

    infile.close();
    if (usernameExists)
    {
        cout << "Username already exists. Please choose a different username." <<
endl;
        goto start;
    }

    ofstream outfile("passwords.txt", ios_base::app);
    outfile << "Username: " << username << ", Password: ";
    cout << "Your password is: ";
    for (int i = 0; i < PATTERN_LENGTH; i++)
    {
        pattern[i] = rand() % (126 - 33 + 1) + 33;
        outfile << pattern[i];
        cout << pattern[i];
        fflush(stdout);
        usleep(SLEEP_TIME *1000000);
    }

    outfile << endl;
    outfile.close();
    cout << endl << "You have been successfully signed up" << endl;
}
```

The `signup` function is responsible for creating a new user account by prompting the user to enter a username and generating a random password. It also checks if the username already exists in the `passwords.txt` file and handles accordingly.

The function begins by displaying a decorative line pattern using the `DecorativeLines` function. It also initializes the random number generator by calling `srand` with the current time as the seed.

Next, the function defines constants `PATTERN_LENGTH` (set to 8) and `SLEEP_TIME` (set to 0.75) to control the length of the generated password and the delay between displaying each character.

A character array `pattern` is declared to store the generated password, and a string variable `username` is declared to store the user-provided username.

The function uses a label `start` for a loop that allows the user to re-enter a username if it already exists.

Inside the loop, the user is prompted to enter a username, and the input is stored in the `username` variable. The function opens the `passwords.txt` file for reading and checks if the entered username already exists by searching for a line that starts with "Username: " followed by the entered username. If a match is found, `usernameExists` is set to `true`, and the loop repeats.

Once a unique username is entered or if the file has been fully searched, the file stream is closed.

If `usernameExists` is `true`, indicating that the entered username already exists, an appropriate message is displayed, and the program jumps back to the `start` label, allowing the user to enter a different username.

If the username is unique, the function opens the `passwords.txt` file in append mode and writes the username along with the password to the file. The password is generated by iterating over `PATTERN_LENGTH` and assigning a random character within the ASCII range of printable characters to each element of the `pattern` array. The generated password is both displayed to the user and written to the file.

Once the password is generated and written, the file stream is closed, and a success message is displayed to indicate that the signup process is complete.

Overall, this function provides a mechanism for creating user accounts by generating random passwords and storing the account details in the `passwords.txt` file.

```
bool login(string &name)
{
    DecorativeLines();
    string username, password;
    char c;
    cout << "Enter username: ";
    cin >> username;
    cout << "Enter password: ";
    cin >> password;
    const int PATTERN_LENGTH = 8;
    const float SLEEP_TIME = 0.75;
    char pattern[PATTERN_LENGTH];
    ifstream infile("passwords.txt");   // Open the input file
    string line;
    bool usernameExists = false;
    bool passwordMatches = false;
    while (getline(infile, line))
    {
        if (line.find("Username: " + username) == 0)
        {
            usernameExists = true;
```

```
            size_t password_start = line.find("Password: ") + strlen("Password: ");
            string stored_password = line.substr(password_start);
            if (password == stored_password)
            {
                passwordMatches = true;
            }

            break;
        }
    }

    infile.close();
    if (!usernameExists)
    {
        cout << "Username not found." << endl;
        return false;
    }

    if (!passwordMatches)
    {
        cout << "Incorrect password." << endl;
        return false;
    }

    name = username;
    return true;
}
```

The `login` function is responsible for handling the login process. It prompts the user to enter a username and password, compares the entered credentials with the stored username-password pairs in the `passwords.txt` file, and returns a boolean value indicating the success or failure of the login attempt. The `name` parameter is passed by reference to store the logged-in username.

The function begins by displaying a decorative line pattern using the `DecorativeLines` function. It then declares string variables `username` and `password` to store the user-provided credentials, and a character variable `c` (which is unused in this code).

The user is prompted to enter a username and password, and the input is stored in the respective variables.

Next, the function defines constants `PATTERN_LENGTH` (set to 8) and `SLEEP_TIME` (set to 0.75) to control the length of the password pattern and the delay between displaying each character.

A character array `pattern` is declared to store the generated password pattern.

The function opens the `passwords.txt` file for reading and starts a loop that reads each line of the file. It checks if the line starts with "Username: " followed by the entered username. If

a match is found, `usernameExists` is set to `true`. It then retrieves the stored password from the line and compares it with the entered password. If they match, `passwordMatches` is set to `true`. The loop is then exited.

Once the file has been fully searched or a match is found, the file stream is closed.

If `usernameExists` is `false`, indicating that the entered username does not exist in the file, an appropriate message is displayed, and the function returns `false` to indicate the login failure.

If `passwordMatches` is `false`, indicating that the entered password does not match the stored password, an appropriate message is displayed, and the function returns `false` to indicate the login failure.

If both the username and password are valid, the `name` parameter is updated with the logged-in username, and the function returns `true` to indicate the login success.

Overall, this function provides a mechanism for verifying the user's credentials by comparing the entered username and password with the stored values in the `passwords.txt` file. It returns a boolean value to indicate the success or failure of the login attempt, and it updates the `name` parameter with the logged-in username in case of a successful login.