Term Project

1.Introduction

The project aims to organize a folder named "Unprocessed-Passwords.txt," which contains random passwords, into subfolders based on the first character of each password. While developing the program, I prioritized finding and selecting the most efficient methods to ensure optimal performance and effectiveness.

Key Components:

1. **Password Processing**:
    - **Function**: **processFolder**
    - **Task**: Reads passwords from unprocessed files, normalizes them, and writes unique entries to processed files.
    - **Normalization**:
        - Converts the first character of each password to lowercase if it's alphabetical.
        - Replaces special characters not allowed in filenames (e.g., ****, **/**, **:**, *****, **?**, **"**, **<**, **>**, **|**) with underscores (_).
2. **Password Indexing**:
    - **Function**: **index**
    - **Task**: Reads processed passwords, computes their MD5, SHA-1, and SHA-256 hashes, and stores them in an indexed folder structure.
    - **Folder Structure**: Based on the first character of the password, creating subfolders and storing passwords in files. Ensures that files do not exceed a certain number of lines (500 in this case).
3. **Password Searching**:
    - **Function**: **Search**
    - **Task**: Searches for a given password in the indexed files. If not found, computes its hash values, stores it in the appropriate file, and updates the file line count.
    - **Output**: Indicates if the password was found and records it if not.
4. **Hash Computation**:
    - **Functions**: **calculate_md5**, **calculate_sha1**, **calculate_sha256**
    - **Task**: Computes and returns MD5, SHA-1, and SHA-256 hash values for given password strings.
5. **Main Execution**:
    - **Function**: **main**
    - **Task**: Orchestrates the processing, indexing, and searching operations. Measures and reports the time taken for each search operation and calculates the average search time over multiple iterations.

Scope of the Project:

1. **Input**:
    - Unprocessed password files located in a specified directory.
2. **Output**:
    - Processed password files stored in a designated directory.
    - Indexed files with hash values stored in a structured directory for efficient searching.
3. **Performance Measurement**:
    - Time taken for each search operation is measured and the average search time is calculated over a set of searches.

2. Tools and Technologies Used:

The project was implemented using the C++ programming language. Visual Studio Code (VS Code) served as the integrated development environment (IDE). The following libraries were utilized to achieve the project's objectives:

Standard Libraries:

- **<iostream>**: For standard input and output operations.
- **<fstream>**: For file input and output operations.
- **<filesystem>**: For directory manipulation and traversal.
- **<unordered_set>**: For efficiently storing unique passwords.
- **<unordered_map>**: For tracking the number of lines in each output file.
- **<iomanip>**, **<sstream>**, **<cctype>**: For formatting, string manipulation, and character type checking.

Cryptographic Libraries:

- **<openssl/md5.h>** and **<openssl/sha.h>**: Header files from the OpenSSL library, which provides a comprehensive suite of cryptographic functions.

Cryptographic Functions

OpenSSL is an open-source implementation of SSL and TLS protocols, designed to secure communication over networks. It offers a wide range of cryptographic functions, including those used in this project:

- **MD5:**
    - Produces a 128-bit hash value.
    - Initially designed for cryptographic use but later found to have vulnerabilities.

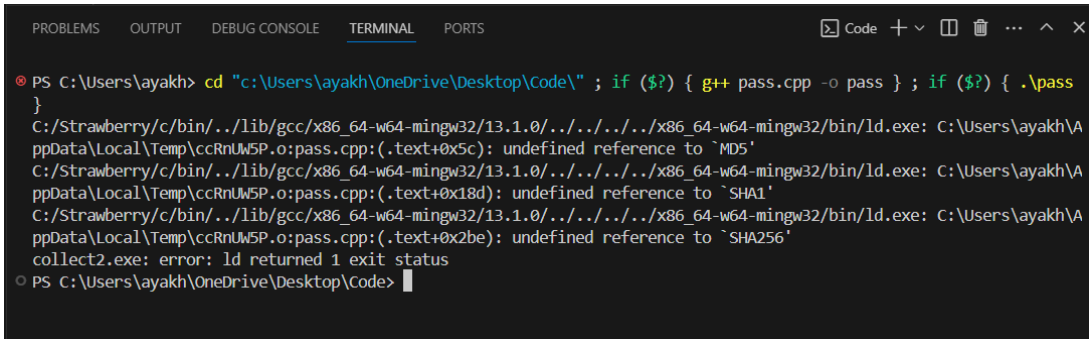
    - Still used for database partitioning and file transfer validation due to its checksum capabilities.

- **SHA (Secure Hash Algorithm):**
    - **SHA-1:**
        - Generates a 160-bit hash (20 bytes).
        - Designed for cryptographic applications but has known vulnerabilities, making it unsuitable for secure applications today.
    - **SHA-2:**
        - Includes several variants, with SHA-256 being the most commonly used.
        - Produces a 256-bit hash value (64 hexadecimal digits).
        - Recommended by the National Institute of Standards and Technology (NIST) for its enhanced security compared to MD5 and SHA-1.
        - SHA-256 is about 20-30% slower to compute than MD5 or SHA-1 but offers significantly better security.

By leveraging these tools and technologies, the project ensures efficient and secure processing and organization of password data.

Installing and compiling OpenSSL:

After installing the OpenSSL library, I encountered some difficulties linking the libraries to the program.



Through research, I resolved this issue by using the **g++** command, specifying the source file (**pass.cpp**), the output executable name (**-o pass**), and the flags needed to link against the OpenSSL library (**-lssl -lcrypto**).

The command used was:

```
g++ pass.cpp -o pass -lssl -lcrypto
```

This command instructs the compiler to link against the OpenSSL libraries (**libssl** and **libcrypto**) when compiling **pass.cpp**.

Once successfully compiled, the program can be run with the following command:

Once the program is successfully compiled, we can run the executable and it will work just fine 😊 :

```
./pass
```

3.Project Design and Configuration:

The folder structure for the project includes the following directories:

Code: Contains the source code files for the project.

Unprocessed-Passwords: Contains the raw password files to be Processed.

To my understanding “Processing the passwords”: is unifeing the passwords and making them suitable to be file or folder name by changing the first character if it is among the characters that are not allowed to be file/folder names. I used the following algorithm to achieve that:

```
                char firstChar = line[0];
                   if (!isalpha(firstChar)) {
                        char c = firstChar;
                       if (c == '\x5C' || c == '/' || c == ':' || c == '*' ||
c == '?' || c == '"' || c == '<' || c == '>' || c == '|') {
                           line[0] = '_';
                       }
                   } else {
                       firstChar = tolower(line[0]);
```

Processed: Upon processing, passwords from the Unprocessed-Passwords folder are stored in this directory. Each processed password is saved in a file with the same name as its corresponding raw file.

Index: This folder contains the organized processed passwords, which are stored into subfolders based on the first character of each password. Each subfolder represents an alphabetic character and contains files storing passwords starting with that character.

4.Indexing Process:

To implement the indexing process I used the index() function:

This function indexes the passwords stored in the processed directory by computing hash values for each password, organizing them into corresponding subfolders based on their first character, and storing them in output files within those subfolders. It iterates through the files in the processed folder using the `directory_iterator` from the `<filesystem>` library.

```
for (const auto &entry : fs::directory_iterator(folderPath)) {
```

- **const auto &entry** : declares a loop variable **entry** of type **auto**, which automatically deduces the type of each element returned by the directory iterator
- **fs::directory_iterator(folderPath) :** initializes a directory iterator object for the directory specified by **folderPath**. It allows us to iterate over the contents of the directory.

```
 if (entry.is_regular_file()) {
      ifstream inputFile(entry.path());
```

- if the file opened is “regular”, it opens an input file using the file’s path.

```
if (!inputFile) {
        cerr << "Error opening file: " << entry.path() << endl;
        continue;
```

- If a file cannot be opened, the function displays an error message and proceeds to the next iteration.

Then the file enters a while loop to read and process each line in it, using *getline()*.

- For each line, the function calculates three hash values: MD5, SHA1, and SHA256:

```
         string md5_hash = calculate_md5(line);
         string sha1_hash = calculate_sha1(line);
         string sha256_hash = calculate_sha256(line);
```

Subsequently, the function determines the subfolder path based on the first character of the password:

```
string subfolderPath = indexFolderPath + "/" + firstChar;
```

If the subfolder does not exist, it creates one:

```
fs::create_directory(subfolderPath);
```

Next, the function constructs the file path for storing the indexed passwords:

```
string filePath = subfolderPath + "/output_" + firstChar + ".txt";
```

If the file containing passwords surpasses 10,000 lines, the function creates additional files:

```
if (fileLineCounts[filePath] > 10000 - 1) {
                        int count = fileLineCounts[filePath] / 10000;
                        do {
                          count++;
                          filePath = subfolderPath + "/output_" + firstChar +
"_" + to_string(count - 1) + ".txt";
                          } while (fileLineCounts[filePath] > 10000 - 1);
```

During this phase, an integer variable, **count**, is initialized to count the number of files needed after the first file exceeds 10,000 lines. Within the do-while loop, the program continuously increments the count and updates the file path until it finds or creates a filepath with less than 10,000 lines. This ensures that the passwords are distributed across multiple files, each containing a maximum of 10,000 lines.

Finally, the function appends the password along with its hash values and filename to the designated output file.

The process repeats for each line in each file in the processed folder, ensuring efficient organization and storage of indexed passwords.

5.Search Function and Performance Test:

To implement the searching process I used the search() function:

The `Search` function is designed to locate a specific password within the indexed folder. It takes the password to be searched as input and searches through the indexed folder with the same first character to find a match.

- The function begins by determining the subfolder path based on the first character of the password:

```
      string subfolderPath = indexFolderPath + "/" + pass[0];
```

- It then opens that subfolder and iterates through its files:

```
        for (const auto &entry : fs::directory_iterator(subfolderPath))
         while (getline(inputFile, line))
```

- For each line, the function extracts the first word (password) and compares it with the password being searched for:

```
if (!line.empty() && line.substr(0, line.find('|')) == password) {
                        cout << "Found" << entry.path() << endl;
```

If a match is found, the function displays the file path where the password was found along with the corresponding line, and then exits.

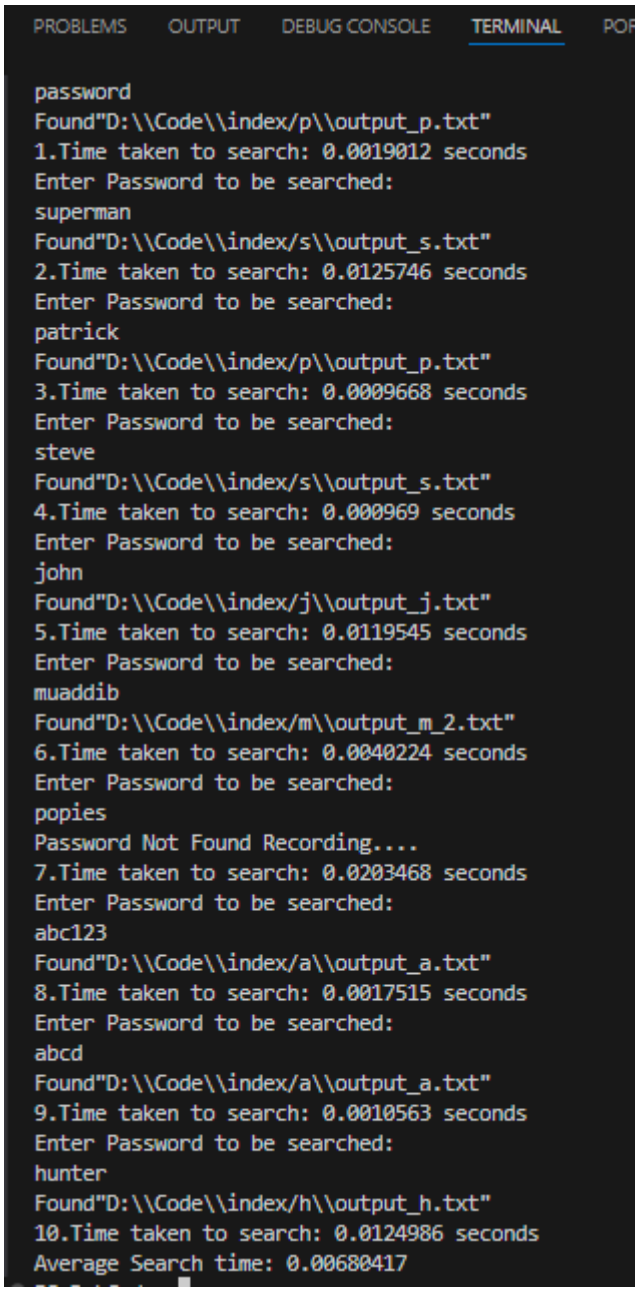
If no match is detected in the current file during the search process, the function proceeds to the next file until all files in the subfolder have been examined exhaustively. However, if the password is not located in any of the files, the function takes additional steps to save the searched password so that it will be found in the next search.

Before storing the unfound password, the function processes it. It then determines the appropriate file to store the password based on its first character. Importantly, the function avoids storing the password in a file that has reached its maximum capacity, ensuring efficient management of file sizes and preventing potential data loss or overflow issues.

Performance Test: To measure search performance, the program records the time taken to search for 10 randomly selected passwords. The average search time is calculated based on these measurements. To do this I used <chrono> library.

```
for(int i = 1; i <= 10; i++){
        cout << "Enter Password to be searched: " << endl;
        cin >> pass;
        if (pass == "x") break;
        auto start = chrono::high_resolution_clock::now(); // Start timing
        Search(pass);
        auto end = chrono::high_resolution_clock::now(); // End timing
        chrono::duration<double> duration = end - start;
        cout << i << ".Time taken to search: " << duration.count() << "
seconds" << endl;
        total = duration.count() + total;
        }
       float avg = total/10;
       cout << "Average Search time: "<< avg << endl;
```

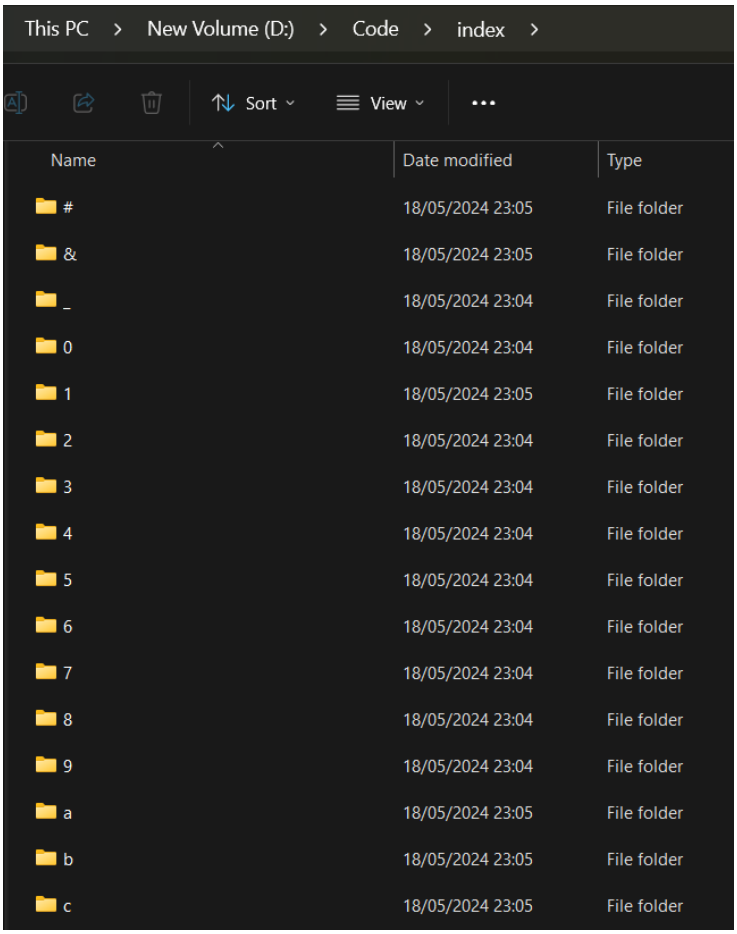
- screenshot of the results of 10 searches:



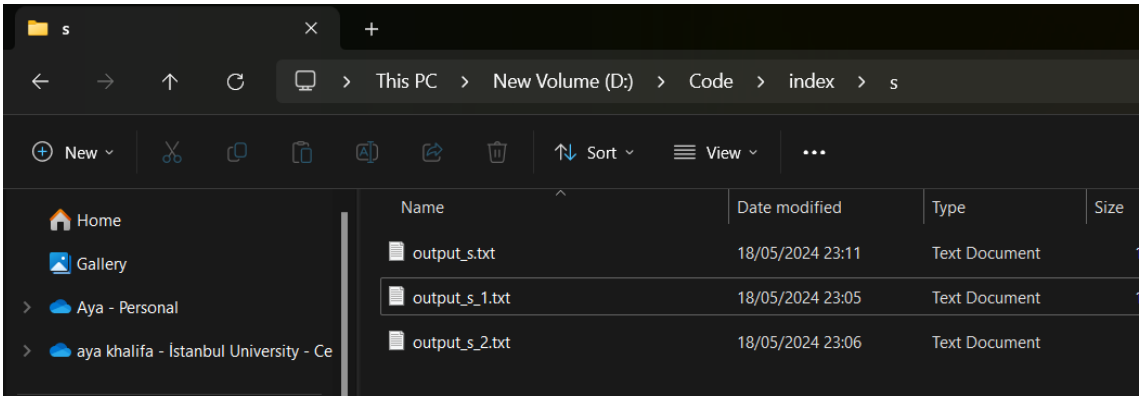
6.Results and Evaluation:

The project effectively sorts and organizes passwords into subfolders based on their initial characters, offering efficient retrieval through the search function. Overall, the project achieves its goals of providing a functional password management system. However, there are areas for improvement, notably in performance optimization and code simplification. The program's execution speed could be enhanced(it took about 6 minutes).

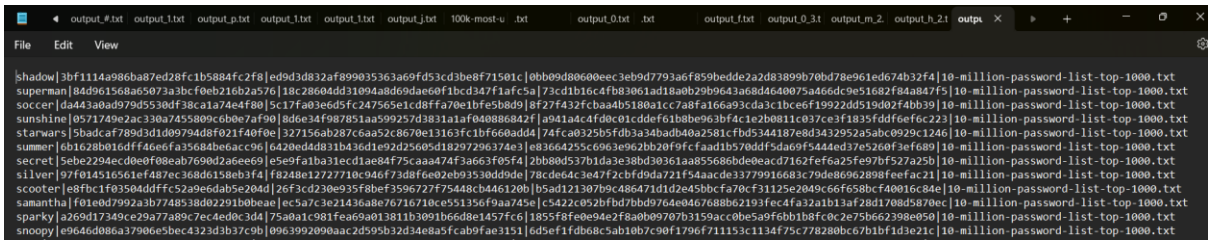
- The index folder after runing the program:



- Inside the ‘s’ folder:



- Inside the ‘s’ file:



References:

- https://www.freecodecamp.org/news/md5-vs-sha-1-vs-sha-2-which-is-the-most-secure-encryption-hash-and-how-to-check-them/
- https://www.openssl.org/
- https://cplusplus.com/reference/unordered_set/unordered_set/
- https://www.geeksforgeeks.org/chrono-in-c/
- https://www.youtube.com/watch?v=aEgG4pidcKU

It is also important to mention that I used ChatGPT and other AI tools to refine and format this document.