**Program Three / CSC 1310**

hashing passwords

# important dates

**Assignment Date**: Tuesday, October 17, 2023 in class

**Due Date**: Thursday, Nov 2, 2023

# Description of program – what does this program do?

This program is a hash table, which is used to store a list of users and their login credentials. As we know, plain-text passwords should **never** be sent across a network, so in this program we see how password hashes are handled when communicating with a server. For this purpose, we are treating the hash table header file as though it were on a server and the driver as being a local client.

## Files you will submit in your zip file

This program contains multiple files as described below:

* **driver.cpp** – **partially given** – the driver will read a list of sample users, and allower the user to attempt to login, create a new account, or delete their account.
* **hashTable.h** – **partially given** – the header file contains two classes: a hash table class, and a private nested class for an entry in the has table.
* **hashTable.cpp** – **partially given** – the source file for the hash table class.
* **sha256.h** – **given to you** – algorithm used for encrypting a password before sending it across a network, one of the most commonly used for hashing passwords. I have no idea what’s going on in this file, so please don’t ask me.
* **sha256.cpp** – **given to you** – same as previous.
* **testList.txt** – **given to you** – a sample list of users and their passwords.
* **Reflection.pdf** – write a short reflection on your work for this assignment.

## driver.cpp

All the input from the sample list of users is done for you, as well as the options menu. The sample list has the number of users in the file on the top line, usernames follow the same pattern of first initial, last name, and passwords are four random words stuck together. Password managers are the best way to go for generating and keeping track of your passwords, but if you have one that needs to be committed to memory, one of the next best approaches is to put [random words together](https://xkcd.com/936/)[https://xkcd.com/936/] (a dictionary attack on one word is very easy, a dictionary attack on four is not so easy).

The parts that you need to implement yourself are dynamically allocating a hash table, adding new users as the driver reads input from the sample file, and call the corresponding hash table function for each menu option. For the purposes of this program, we’re treating the driver like a client and the hash table object like the server. This means that a password should *never* reach the object without being hashed first.

Every password has a “salt” attached to it, which is just a random string to add complexity to the password (when we create a new user, that user has a salt that gets stored with it. This salt does not change). When we have a password that needs to be sent, we first grab the salt (or generate one if we’re creating a new user), then append that salt to the end of the password, and send our newly appended string through a has function. To call the hash function on a password, you just need to call **sha256(string)**. The server (object, in this case) does not save a copy of your password, it only stores the result of the hash function, so when it validates your password it’s comparing the results of two hash functions, not the password itself. Again, *never send a password without hashing it first! The server should never touch a password that hasn’t been hashed!*

I tried to keep the instructions simple in this section, but if more depth on what’s going on with salts and hashes will help you think through this problem (or if you’re just curious), I recommend reading through these two posts:

This one on hashing passwords <https://auth0.com/blog/hashing-passwords-one-way-road-to-security/>

And this one on salting your hashes <https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/>

## hashtable class

This hash table will use chaining for its collision handling. The username is being used as the unique key to map an entry to a bucket in the array.

Your hash table class has a nested class that describes an entry. This works exactly the same way as having a struct inside a class (like we did for list nodes) it’s just a class instead of a struct.

### Attributes

* **class entry** – this line is written for you, it does the same thing as a function prototype, but for a class instead of a function.
* **tableSize** – the space allocated for the hash table’s array.
* **hashArray** – this is the main array for the object, which we use to reference our buckets. It’s an array of entry pointers.

### public Functions

Function name: **hashTable** constructor

* Parameters: an int, the amount of space to allocate for the array
* Purpose: dynamically allocates a new array of entry pointers and sets every array element to null. It also seeds the random number generator that we’ll use to generate the users’ salts.

Function name: **hashTable** destructor

* Parameters: none
* Purpose: deletes every user from the has table by looping through the array and deleting the linked list in every bucket.

Function name: **generateSalt**

* Parameters: none
* Returns: a string, to append at the end of a password before hashing it
* Purpose: assigns each attribute to the given parameter, and sets the next pointer to null.
* Specifications: this is provided for you, do not touch it. There are better ways to generate a salt, but for the sake of simplicity for this program we’re just generating a random string of characters. The length of this string is the size of the hash table because why not.

Function name: **getSalt**

* Parameters: a string for the user we’re grabbing a salt from
* Returns: a string, which is the salt we’re grabbing
* Purpose: to be able to append the salt to a password before we hash it
* Specifications: this call the getEntry function to find the user associated with the key we got as a parameter. Once we have a user, we can call the getSalt function for that user. Think about what needs to happen if that user isn’t in the table.

Function name: **addEntry**

* Parameters: three strings, which will be sent to the entry constructor
* Returns: void
* Purpose: to add a new user to the table
* Specifications: dynamically allocate a new entry, call the hash function to determine which bucket the new user will be stored in, and add them to the list in that bucket. Make sure you aren’t adding a username that already exists in the table.

Function name: **validateLogin**

* Parameters: two strings, a username and password (which has already been hashed)
* Returns: a bool, which indicates whether or not the login was successful.
* Purpose: to check if the user has entered the correct login credentials
* Specifications: this calls the getEntry function to find the data that goes with the username they entered, then checks if the password hash that was sent matches the password hash that was saved under that username.

Function name: **removeUser**

* Parameters: two strings, a username and password (which has already been hashed)
* Returns: a bool, which indicates whether the user was removed successfully
* Purpose: to remove a user from the hash table
* Specifications: First we need to find the user who’s trying to remove themself. Once we’ve found that user, we need to verify their login credentials (don’t let someone delete a user without the correct password, then anyone could delete any other user in the system). Once we’ve verified that user, then we can remove their entry from the table.

### private Functions

Function name: **hash**

* Parameters: a string, the key we need to map to a bucket index
* Returns: an int, the bucket index our key is mapped to
* Purpose: map a key onto a bucket to figure out where it belongs in the hash table
* Specifications: this function is written for you, so don’t touch it! It maps a string onto an integer by adding together the ascii values of each character in the string. This is probably the worst possible way to map a string in a hash table because it has a high chance of collisions. We’re doing it this way because I’ve designed it for you to have a lot of collisions.

Function name: **getEntry**

* Parameters: a string, the key we’re looking up in the table
* Returns: an entry pointer, all the data associated with the key the we passed
* Purpose: retrieve a user for use in other functions
* Specifications: apply the hash function to the key to figure out which bucket we need to look in, traverse the list in that bucket until we find the node with a matching key, then return that node once we’ve found it. Think about what you need to do if that key doesn’t correspond with an entry in the table.

## entry class

This class is nested inside the hash table class, which means it’s ony accessible from inside the hash table. It does exactly the same the struct nodes we were using inside a linked list, but let’s us stop ourselves from altering user data.

### Attributes

* **username** – a username, saved as a string.
* **salt** – a string for the user’s password salt.
* **hashedpwd** – the user’s hashed password. Remember, this isn’t plain text, it has been hashed already.
* **next** – a pointer to the next entry in the bucket. This is a public attribute (but is only accessible to the hash table, since it’s in a private class).

### public Functions

Function name: **entry** constructor

* Parameters: three strings, one for each attribute
* Purpose: assigns each attribute to the given parameter, and sets the next pointer to null.

Function name: **getUsername**

* Parameters: no parameters
* Returns: string

Purpose: returns the username

Function name: **getSalt**

* Parameters: no parameters
* Returns: string

Purpose: returns the salt for hashing the password

Function name: **getHashedpwd**

* Parameters: no parameters
* Returns: string
* Purpose: returns the hashed password