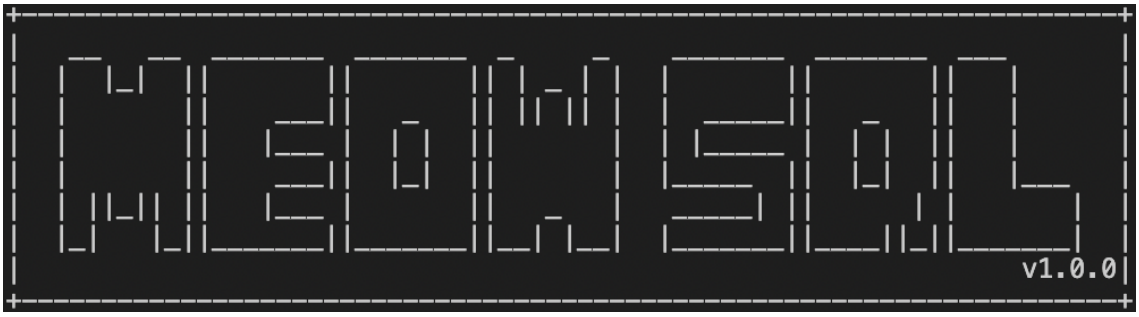
CS 135
Computer Science I

MeowSQL – A Database

Part I



For use by:
School of Computer Science
Howard Hughes College of Engineering
University of Nevada, Las Vegas

Created and Authored by:
Alex St. Aubin © 2022

Contents

Predefined Global Constants

This assignemnt is accompanied by a `main.cpp` that has many constants already defined that will be necessary for this program. Below is a list of these predefined constants. They will all need to be used in this assignment. There will be a username and password for the database. The credentials have already been defined in `main.cpp`:

```
const string USER = "meowmin";
const string PASS = "1w@ntch1cken";
```

Figure 1: Login credentials for the database

As the database takes commands it will prompt the user for new commands with a `>` and the arguments of the commands will be delimited by a space. These values have already been defined in `main.cpp`:

```
const string COMMAND_PROMPT = ">";
const string COMMAND_DELIMITER = " ";
```

Figure 2: Command options for the database

There will be four commands that will be implemented in this assignment. The names of these commands have already been defined in `main.cpp`:

```
const string QUIT_CMD = "quit";
const string SHOW_CMD = "show";
const string CREATE_CMD = "create";
const string DELETE_CMD = "delete";
```

Figure 3: Commands to be implemented in the database

The database will also have work with `.csv` file types which will be located in the directory `data/`. The database will also work with a file to keep track of the names of all the tables in the database. These values have already been defined in `main.cpp`:

```
const string TABLE_FILETYPE = ".csv";
const string TABLE_FILE_DIRECTORY = "data/";
const string TABLES_TABLE = "data/tables.csv";
```

Figure 4: Useful directories/files/file types for the database

When the `SHOW_CMD` is implemented it will have options it can take. For this assignment there will only be one option `SHOW_CMD` can handle. This value has already been defined in `main.cpp`:

```
const string SHOW_ARG_1 = "tables";
```

Figure 5: Options for available for SHOW

Lastly, there are also be some usage/welcome/error messages that need to be output by the database. These messages have already been defined in `main.cpp`:

```
const string USAGE_MSG = "Usage: ./a.out <username> <password>";
const string WELCOME_MSG = "Welcome ";

const string VALID_ARG_MSG = "";
const string QUIT_ARG_CNT_MSG = "QUIT Error: invalid argument count";
const string SHOW_ARG_CNT_MSG = "SHOW Error: invalid argument count";
const string CREATE_ARG_CNT_MSG = "CREATE Error: invalid argument count";
const string DELETE_ARG_CNT_MSG = "DELETE Error: invalid argument count";

const string TABLE_CREATE_SUCCESS_MSG = " table created successfully";
const string TABLE_DELETE_SUCCESS_MSG = " table removed successfully";
const string INVALID_CREDENTIALS_MSG = "Error: invalid credentials";
const string INV_CMD_MSG = "Error: invalid command name";
const string SHOW_INV_OPT_MSG = "SHOW Error: invalid option";
const string CREATE_INV_TABLE_NAME_MSG = "CREATE Error: table name should only contain
    alpha numeric characters, '-', or '_'";
const string CREATE_EXISTS_MSG = "CREATE Error: table already exists";
const string CREATE_INV_HEADERS_MSG = "CREATE Error: column names should be separated
    by a ',' and should only contain alpha numeric characters, '-', or '_'";
const string DELETE_UNDELETABLE_MSG = "DELETE Error: table cannot be deleted";
const string DELETE_INV_TABLE_NAME_MSG = "DELETE Error: invalid table name";
```

Figure 6: Messages the database will output

Introduction

Data makes the world go around, at least the computational world. Everything that is done in a computer revolves around data from the most basic instructions up to complex applications like video games and social media platforms. As data is such a vital part of computing it is equally as vital to learn to use and store data. In this two part assignment you will learn how to build a database that will store and retrieve rows of data in comma separated value (`.csv`) files.

1 Logging In

In order to keep the database "secure" we will need to make sure that only a user that knows the username and password to the database can access the database. The reason why our database will be "secure" is because the username and password will be stored in plain text in constants in the program. In an actual application the password would need to be encrypted and the values would be stored in some database or file.

1.1 Getting Username and Password

In order to get the username and password from the user we will have the user pass them via command line arguments when they run the program. We will have the first command line argument be the username, and the second be the password. A function call to a function:

```
void getCredentials(int, char const *, string&, string&)
```

has been provided in `main()` in the provided `main.cpp` that will get the username and password from the command line arguments and save them via `reference parameters` into the `user` and `pass` variables in `main()`. Implement this function in the body of the provided function stub for `getCredentials()`. You will need to:

1. Validate that there is 3 command line arguments (`./a.out`, the username, and the password). If there is not 3 command line arguments output the `USAGE_MESSAGE` and quit the program with `exit(0)`. If there is 3 command line arguments continue on to (2) and (3).
2. Get the username from the first command line argument and save it in the `user` reference parameter.
3. Get the password from the second command line argument and save it in the `pass` reference parameter.

1.2 Validating Username and Password

Once the username and password has been read they must be verified. `main.cpp` contains `USER` (username) and `PASS` (password) constants which hold the credentials:

- **Username:** meowmin
- **Password:** 1w@ntch1cken

The username and password the user entered will be contained in the `user` and `pass` variables in main after the call to `getCredentials()` has finished executing. A function call to a function:

```
bool validateCredentials(string, string)
```

has been provided in `main()` which passes the credentials the user entered, validates them, and then returns

`true` if they are valid and `false` if they are invalid. Implement this function in the body of the provided function stub for `validateCredentials()`. You will need to:

1. Verify the passed username is the same as the one stored `USER` constant, and that the passed password is the same as the one stored `PASS` constant.
2. If the passed username or password is incorrect the `INVALID_CREDENTIALS_MSG` should be output and the function should return `false`.
3. If the passed username or password is correct the `WELCOME_MSG` and the `USER` should be output and the function should return `true`.

Once the call to the `validateCredentials()` function has returned it's return value will be tested in the provided `if...` statement in `main()`. If the credentials are valid you will want to print the header for the database. A function has been provided with the body already filled out called `header()`. This function will simply print the header for the database. Make a call to the `header()` function in the provided `if...` statement in `main()`.

If the credentials are however invalid the `if...` statement will not run, causing `main()` to continue on to the `return 0;` which quits the program.

At this point the program will be able to accept only a valid count of command line arguments, store those arguments into 2 string variables, then verify they are a valid username/password combo. Running the program with an invalid count of command line arguments should output:

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out
Usage: ./a.out <username> <password>
alex-imac24@Alexs-Home-iMac as7 % ./a.out tooFew
Usage: ./a.out <username> <password>
alex-imac24@Alexs-Home-iMac as7 % ./a.out too many arguments
Usage: ./a.out <username> <password>
alex-imac24@Alexs-Home-iMac as7 %
```

Figure 7: Output from Section 1.1 if invalid command line argument count

Providing either an invalid username or password should output:

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out invalid 1w@ntch1cken
Error: invalid credentials
alex-imac24@Alexs-Home-iMac as7 % ./a.out meowmin invalid
Error: invalid credentials
alex-imac24@Alexs-Home-iMac as7 % ./a.out invalid invalid
Error: invalid credentials
alex-imac24@Alexs-Home-iMac as7 %
```

Figure 8: Output from Section 1.2 if incorrect username and/or password

And providing the correct username and password should output:

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out meowmin 1w@ntch1cken
Welcome meowmin
+------------------------------------------------------------------------+
|   __   __  _______  _______  _     _    _______  _______  ___        |
|  |  |_|  ||       ||       || | _ | |  |       ||       ||   |       |
|  |       ||    ___||   _   || || || |  |  _____||   _   ||   |       |
|  |       ||   |___ |  | |  ||       |  | |_____ |  | |  ||   |       |
|  |       ||    ___||  |_|  ||       |  |_____  ||  |_|  ||   |___    |
|  | ||_|| ||   |___ |       ||   _   |   _____| ||       | |       |   |
|  |_|   |_||_______||_______||__| |__|  |_______||____||_||_______|   |
|                                                                  v1.0.0|
+------------------------------------------------------------------------+
alex-imac24@Alexs-Home-iMac as7 %
```

Figure 9: Output from Section 1.2 if correct username and password are entered

2 Getting Input (commands) from the user

Now that the user has been verified they can start entering commands. To add this functionality first a function prototype/header/body needs to be added to `main.cpp`:

```
void commandLoop()
```

The `commandLoop()` function will also have to be called in `main()` after the call to the `header()` function inside of the `if...` statement. The `commandLoop()` function will be the heart of the database. This function will get an input from the user, validate that it is a valid command, then run command or output an error if an invalid command is entered. In this function you will first want to create a vector to hold the `arguments` of the input from the user. Then, you will want to start a loop which can run for an indefinite amount of time, which will be terminated by hitting some `break` statement in the loop. This loop is the core of the database program which will:

1. Get input from the user and tokenize it into arguments.
2. Validate the input is a valid command.
3. Execute the command or output an error message if an invalid command is input.
4. Continue from (1) until a QUIT command is input.

To do this a prototype, header, and body must be defined for 3 functions:

1. `vector<string> getInput ()`
2. `string validateArguments(vector<string> args)`
3. `void executeCommand(vector<string> args)`

2.1 Getting User Input

These functions will be called in the loop in `commandLoop()` function to create the core functionality of the database. Firstly, in the `commandLoop()` the `getInput()` function needs to be called and have its return value saved into the `vector` local to `commandLoop()` which was declared to hold the `arguments`. This vector will hold the `arguments` (words in this case) the user entered into the prompt. To get these `arguments` in the function body of `getInput()`:

1. Create a `vector` to add the `arguments` the user entered to and then return at the end of the function.
2. Output the `COMMAND_PROMPT` (global constant declared in `main.cpp` already) to the user.
3. Read in an input from the user into a `string`.
4. Push the `arguments` (words) the user input into the vector created in (1). Before pushing a `string` onto the vector pass it to the `toLower()` function, which will simply lowercase an entire `string` and return it, provided in `main.cpp` and push the `string` returned onto the `vector`.
5. `return` the `vector` of `arguments` to the caller.

2.2 Calling a Function to Validate Arguments

After the call to `getInput()` the `arguments` vector will contain the `arguments` the user entered. These `arguments` must then be passed to the `validateArguments()` function which will validate the `arguments` as a valid command in the database. To verify the `arguments` as a valid command, in the function body of `validateArguments()`:

1. Output the string `inside validateArguments()"` then `return` the empty string (`""`) for now. More code will be added to this function to validate the arguments the user entered in the following sections.

Since `validateArguments()` is a `string` returning function the `string` that is returned from the function must be saved. For now the `string` returned will always be the empty string, but after future sections it will be the empty string if no errors were found in the `arguments`, otherwise it will contain an error message saying why the `arguments` were invalid. Although there is no error messages being returned right now we will still add code to handle them since it is quite simple. If there is an error message you will simply want to output the error message, then continue on to getting the next input from the user.

2.3 Calling a Function to Execute Commands

If there is no error message returned from the call to `validateArguments()`, then the input from the user was a valid command. The command must now be run. To do this the `arguments` need to be passed to the `executeCommand()` function so the command can be executed. To execute the command, in the function body of `executeCommand()`:

1. Output the string `inside executeCommand()"`. More code will be added to this function to allow for executing commands in the following sections.

The last thing in the `commandLoop()` function should be a call to the `executeCommand()` function passing the valid `arguments` to it. In future sections, this will cause the command to be executed and then the loop to restart (getting a new command from the user). Running the program with valid credentials and giving it any input should output the following (on the next page):

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out meowmin 1w@ntch1cken
Welcome meowmin
+--------------------------------------------------------------------------+
|   __   __  _______  _______  _     _    _______  _______  ___       |
|  |  |_|  ||       ||       || | _ | |  |       ||       ||   |      |
|  |       ||    ___||   _   || || || |  |  _____||   _   ||   |      |
|  |       ||   |___ |  | |  ||       |  | |_____ |  | |  ||   |      |
|  |       ||    ___||  |_|  ||       |  |_____  ||  |_|  ||   |___   |
|  | ||_|| ||   |___ |       ||   _   |   _____| ||      | |       |  |
|  |_|   |_||_______||_______||__| |__|  |_______||____||_||_______|  |
|                                                              v1.0.0|
+--------------------------------------------------------------------------+
>test
inside validateArguments()
inside executeCommand()
>SHOW tables
inside validateArguments()
inside executeCommand()
>anything input should show the same thing for now
inside validateArguments()
inside executeCommand()
>QUIT
inside validateArguments()
inside executeCommand()
>
```

Figure 10: Output from Section 2 when the user enters input

3 Adding Commands

At this point the user can log into the database and enter commands. However, there is a major issue, the commands do nothing! In this assignment 4 commands will be added: `QUIT`, `CREATE`, `SHOW`, and `DELETE`. These commands will allow the user to quit the system, create a table (`.csv`), delete a table (`.csv`), or show all the table names (`.csv` file names) in the database. For each command you will verify the command is valid in `validateArguments()` then execute the command in `executeCommand()`.

Note: You can add many helper functions to make this section easier, but they are not required.

3.1 QUIT the Database

The first command that will be added is the `QUIT` command, which will allow the user to quit the database system. First we need to check if the `QUIT` command was entered. In the database system the first argument will tell us what command the user wants to execute. So, in `validateArguments()` a selection statement needs to be added to check if the user entered the `QUIT_CMD` predefined constant. If it was entered then the command can be verified as a valid `QUIT_CMD`, otherwise we will want to return the `INV_CMD_MSG` as the `QUIT_CMD` is the only valid comand (for now).

To validate the `QUIT_CMD` simply check that the user only entered 1 argument (the `QUIT_CMD`). If they entered more than 1 argument return the `QUIT_ARG_CNT_MSG`. Otherwise the command is valid, return the `VALID_ARG_MSG`.

Now that `validateArguments()` has finished executing `commandLoop()` will be able to see if there was any errors with `validateArguments()`, and if there are it will output them, then continue on to the next command (because this functionality was already added in previous sections). If there was no errors though, `commandLoop()` will continue on to calling `executeCommand()` which is where the `QUIT_CMD` will be executed.

To execute the `QUIT_CMD` inside of the `executeCommand()` add a selection statement to check if the current command is the `QUIT_CMD`. If it is just terminate the program. Executing the program at this point should output:

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out meowmin 1w@ntch1cken
Welcome meowmin
+------------------------------------------------------------------------+
|   __   __  _______  _______  _     _    _______  _______  ___        |
|  |  |_|  ||       ||       || | _ | |  |       ||       ||   |       |
|  |       ||    ___||   _   || || || |  |  _____||   _   ||   |       |
|  |       ||   |___ |  | |  ||       |  | |_____ |  | |  ||   |       |
|  |       ||    ___||  |_|  ||       |  |_____  ||  |_|  ||   |___    |
|  | ||_|| ||   |___ |       ||   _   |   _____| ||       | |       |   |
|  |_|   |_||_______||_______||__| |__|  |_______||____||_||_______|   |
|                                                                  v1.0.0|
+------------------------------------------------------------------------+
>invalid command
Error: invalid command name
>quit with too many arguments
QUIT Error: invalid argument count
>QUIT
alex-imac24@Alexs-Home-iMac as7 %
```

Figure 11: Output from Section 3.1 when the user wants to quit the database

3.2 CREATE a Table

The next command to be added will be the `CREATE_CMD`, which will allow the user to create a new table (`.csv` file) with with a set of attributes (headers, columns, etc.). A valid `CREATE_CMD` looks like:

`CREATE <table_name> <attribute_list>`, where the `<table_name>` is the name of the new table and the `<attribute_list>` is the list of attributes each row will have to have.

The `CREATE_CMD` has a few rules that must be validated in `validateArguments()` to be a valid command:

1. The command must have 3 arguments. If it doesn't return the `CREATE_ARG_CNT_MSG`.
2. The `<table_name>` may only contain valid characters. Valid characters are `'-'`, `'_'`, `'a' - 'z'`, `'A' - 'Z'`, and `'0' - '9'`. If the `<table_name>` contains any letters that are invalid return the `CREATE_INV_TABLE_NAME_MSG`.

3. The <table_name> can't have the same name as a current table in the system. There is a predefined constant, TABLE_FILE_DIRECTORY, which is the path to the directory the tables (.csv files) in the database are stored in. There is also a predefined constant, TABLE_FILETYPE, which is the file type the tables are stored in (.csv). If you concatenate TABLE_FILE_DIRECTORY with the name of the table the user is trying to create and the TABLE_FILETYPE you will have a path to the table the user is trying to create. If you try opening this path and it successfully opens that means the table already exists, return the CREATE_EXISTS_MSG.

4. Lastly, the <attribute_list> needs to be validated. First, the list can not start or end with a ',' because if it did then the table would have a blank attribute name for the first or last attribute. Next, each attribute name has to be valid and be separated by a ','. To do this, iterate through the characters of list of headers and check that each character is a valid character: ',', '-', '_', 'a' - 'z', 'A' - 'Z', and '0' - '9'. If while you are iterating through the characters you run into a character that is a ',' you will want to check if the previous character is a comma, if it is that means there are 2 commas in a row meaning there is a blank attribute name, which is an error. If there is anything wrong with the list of headers, return the CREATE_INV_HEADERS_MSG.

If the command is valid return the VALID_ARG_MSG. As with any command, once validateArguments() has finished executing commandLoop() will be able to see if there was any errors with validateArguments(), and if there are it will output them, then continue on to the next command. Otherwise executeCommand() will be called to execute the CREATE_CMD. To execute the CREATE_CMD inside of executeCommand():

1. Concatenate the TABLE_FILE_DIRECTORY with the name of the table the user is trying to create and the TABLE_FILETYPE and open this file path as an output file.

2. Write the row of headers to the file opened in (1).

3. There is a .csv that holds the names of all the tables in the database. The path/name of this .csv is held in the predefined constant TABLES_TABLE. Open this file as an appendable file (ex: ofstream.open("file.csv", ios_base::app)), and then write the new table name out to the file.

4. Now that the new table is successfully created, output to the terminal the table name followed by the TABLE_CREATE_SUCCESS_MSG.

Executing the program using the CREATE_CMD should look like the following (on the following page):

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out meowmin 1w@ntch1cken
Welcome meowmin
+---------------------------------------------------------------------+
|   __   __  _______  _______  _     _    _______  _______  ___      |
|  |  |_|  ||       ||       || | _ | |  |       ||       ||   |     |
|  |       ||    ___||   _   || || || |  |  _____||   _   ||   |     |
|  |       ||   |___ |  | |  ||       |  | |_____ |  | |  ||   |     |
|  |       ||    ___||  |_|  ||       |  |_____  ||  |_|  ||   |___  |
|  | ||_|| ||   |___ |       ||   _   |   _____| ||      | |       | |
|  |_|   |_||_______||_______||__| |__|  |_______||____||_||_______| |
|                                                              v1.0.0|
+---------------------------------------------------------------------+
>invalid command
Error: invalid command name
>create
CREATE Error: invalid argument count
>CREATE table
CREATE Error: invalid argument count
>CREATE table col1,col2 too many args
CREATE Error: invalid argument count
>CREATE invalid.table.name col1,col2
CREATE Error: table name should only contain alpha numeric characters, '-', or '_'
>CREATE tables col1,col2
CREATE Error: table already exists
>CREATE table col.1,col.2
CREATE Error: column names should be separated by delimiter and should only contain
alpha numeric characters, '-', or '_'
>CREATE table col1,,col2
CREATE Error: column names should be separated by delimiter and should only contain
alpha numeric characters, '-', or '_'
>CREATE table col1,col2
table table created successfully
>QUIT
alex-imac24@Alexs-Home-iMac as7 % ls data/
table.csv               tables.csv
alex-imac24@Alexs-Home-iMac as7 % cat data/tables.csv
name
tables
table
alex-imac24@Alexs-Home-iMac as7 %
```

Figure 12: Output from Section 3.2 when the user wants to create a new table

3.3 SHOW all Tables

The `SHOW_CMD` will show the names of all the tables in the database and will be in the form:

```
SHOW tables
```

To validate the `SHOW_CMD`, inside of `validateArguments()`:

1. Check if there were 2 `arguments` entered, if there were not return the `SHOW_ARG_CNT_MSG`.

2. The second `argument` must be equal to `SHOW_ARG_1`, if it is not return the `SHOW_INV_OPT_MSG`. In a normal database there would be more than 1 value that the second `argument` of `SHOW_CMD` could handle, but for our database this 1 value is good enough.

If the `arguments` are valid return the `VALID_ARG_MSG`. In `executeCommand()` if the user entered the `SHOW_CMD` simply pass the `TABLES_TABLE` predefined constant to the `bool printTable(string)` file that was provided in `main.cpp` this function will print any `.csv` file in a pretty tabular format.

Executing the program using the `SHOW_CMD` should look like the following:

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out meowmin 1w@ntch1cken
Welcome meowmin
+---------------------------------------------------------------------------+
|   __   __  _______  _______  _     _    _______  _______  ___       |
|  |  |_|  ||       ||       || | _ | |  |       ||       ||   |      |
|  |       ||    ___||   _   || || || |  |  _____||   _   ||   |      |
|  |       ||   |___ |  | |  ||       |  | |_____ |  | |  ||   |      |
|  |       ||    ___||  |_|  ||       |  |_____  ||  |_|  ||   |___   |
|  | ||_|| ||   |___ |       ||   _   |   _____| ||      | |       |  |
|  |_|   |_||_______||_______||__| |__|  |_______||____||_||_______|  |
|                                                                 v1.0.0|
+---------------------------------------------------------------------------+
>invalid command
Error: invalid command name
>SHOW too many args
SHOW Error: invalid argument count
>SHOW error
SHOW Error: invalid option
>SHOW tables
+--------------+
| name         |
+--------------+
| tables       |
| my_new_table |
| table        |
+--------------+
3 row(s) in set.
>QUIT
alex-imac24@Alexs-Home-iMac as7 %
```

Figure 13: Output from Section 3.3 when the user wants to show all the tables in the database

3.4 DELETE a Table

The `DELETE_CMD` will delete a table from the database and will be in the form:

`DELETE <table_name>`, where the `<table_name>` is the name of the table the user wants to delete.

To validate the `DELETE_CMD`, inside of `validateArguments()`:

1. Check if there were 2 `arguments` entered, if there were not return the `DELETE_ARG_CNT_MSG`.

2. There is a special table in the database called `tables` located at the path in `TABLES_TABLE`. This table holds all of the names of the table in the database thus it is a table that can not be deleted. Check that the user is not trying to delete the `<table_name>` called `tables`, if they are return the `DELETE_UNDELETABLE_MSG`.

3. The `<table_name>` the user wants to delete must be a valid table in the database. Either check if it is a valid table name by seeing if you can open the file for the table, or by looking for the file name in the `TABLES_TABLE` (I'll leave the choice up to you). If the table name is not a valid table name already in the database return the `DELETE_INV_TABLE_NAME_MSG`.

If the `arguments` are valid return the `VALID_ARG_MSG`. In `executeCommand()` if the user entered the `DELETE_CMD` you will want to delete to entire table from the database by:

1. Open the `TABLES_TABLE` file and iterate through the rows of the file, deleting the row with the `<table_name>` in it.

2. Delete the table's `.csv` file from the `TABLE_FILE_DIRECTORY` using the `remove()` function (the `<stdio.h>` library for this function call has already been included in `main.cpp`). To use the `remove()` function use the following code snippet (changing the filepath with the filepath you want to delete):

   ```
   string filepath = "directory/file.csv";
   remove(filepath.c_str());
   ```

3. Lastly, once the table has been successfully deleted and it's `.csv` file has been removed output the table name that was just deleted and the `TABLE_DELETE_SUCCESS_MSG`.

Executing the program using the `DELETE_CMD` should look like the following:

```
alex-imac24@Alexs-Home-iMac as7 % ./a.out meowmin 1w@ntch1cken
Welcome meowmin
+---------------------------------------------------------------------+
|   __   __  _______  _______  _     _    _______  _______  ___      |
|  |  |_|  ||       ||       || | _ | |  |       ||       ||   |     |
|  |       ||    ___||   _   || || || |  |  _____||   _   ||   |     |
|  |       ||   |___ |  | |  ||       |  | |_____ |  | |  ||   |     |
|  |       ||    ___||  |_|  ||       |  |_____  ||  |_|  ||   |___  |
|  | ||_|| ||   |___ |       ||   _   |   _____| ||      | |       | |
|  |_|   |_||_______||_______||__| |__|  |_______||____||_||_______| |
|                                                             v1.0.0|
+---------------------------------------------------------------------+
>invalid command
Error: invalid command name
>DELETE too many args
DELETE Error: invalid argument count
>DELETE tables
DELETE Error: table cannot be deleted
>DELETE doesnt_exist
DELETE Error: invalid table name
>DELETE table
table table removed successfully
>QUIT
alex-imac24@Alexs-Home-iMac as7 %
```

Figure 14: Output from Section 3.4 when the user wants delete a table