**Project**

**Documentation**

**For**

**C++**

**Final**

Author: Gabriel Ponce – Samuel Cook

Date: December 2024

Table of Contents

[1. Project Overview 1](#_Toc184465977)

[2. UML Diagrams 1](#_Toc184465978)

[3. Code Explanation 2](#_Toc184465979)

[Appendix A. References 3](#_Toc184465980)

[Appendix B. Code 3](#_Toc184465981)

List of Figures

[1 Class Diagram 1](#_Toc184465986)

[2 Use Case Diagram 2](#_Toc184465987)

# Project Overview

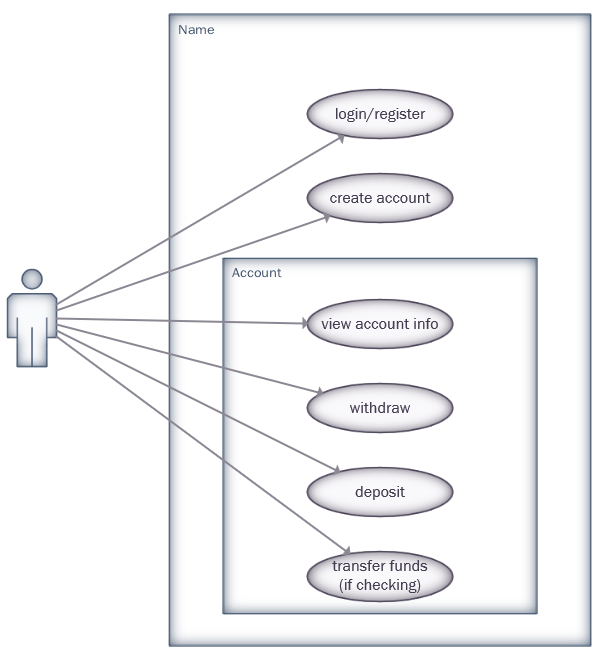
This program is meant to emulate a bank system. It allows the user to login, create a new account or view which accounts they have, and work of those bank accounts. When creating a new account the user is asked what type of account they want to make and from there it will be added to the accounts under their name. Within every account the user is allowed to do basic operations such as deposit and withdraw as well as a specific function tied to each account. The three account types are Checking, Savings, and Fixed Deposit. The Checking account can transfer funds between itself and a different account. The Savings account displays the amount of interest one will gain. The Fixed Deposit account has a penalty that will be applied to the balance if money is attempted to be withdrawn.

# UML Diagrams

A screenshot of a computer flow chart

Description automatically generated

1 Class Diagram



2 Use Case Diagram

# Code Explanation

This Banking system is broken down into 5 files and two SQLite library files.

The SQL library files are provided by SQLite and allow for easy interaction between the database and our application.

Account.h:

This is the base class that will be used by the specific types of bank accounts derived classes. Two important member variables are here, and they are for the balance and account number. These two variables will hold balance and account number accordingly. Under public there is a constructor and three pure virtual functions (deposit, withdraw, display) that need to be included by derived classes. The constructor holds a solid amount of logic as it is meant to assign balance based on a query to the database to find the correct account through account number. The constructor takes in a database Handler pointer and an account number then the database is opened and a stmt pointer is opened. The stmt pointer comes from the sqlite3 library and allows for easier retrieval of information from sql queries. Then an error message variable is initialized to nullptr and that would simply be used to display a description of an error should one come up.Then the SQL query is prepared and it’s a select query grabbing balance where account numbers match. Then the stmt pointer is stepped through which means it grabs the first row and in this case it simply means its grabbing the only row as no two account can have the same numbers and afterwards the stmnt is finalized to free up database for other operations and the constructor ends. This class is also set to simply use the default destructor.

DerivedAccounts.h and .cpp (header and implementation file):

This file holds all the derived classes, and its corresponding implementation file holds the implementation of the functions described in the header file. A lot of the functions are very similar with only slight differences so there will be a full description of one of each type of function and then an explanation of the differences in the other implementations of the same function. There are a total of three sub classes: Savings Account, Checking Account, and Fixed Deposit Account. Each of these sub classes has one special quirk or function and then implements the three functions from the base class.

**Savings Account’s** implementation of the functions is probably the simplest. Its special function is calculating interest and informing the user of how much interest they would gain. This calculated interest is based on the account’s balance (with multiple if else if statements) amount and gives the user a higher “interest” the higher their balance is with the maximum savings interest rate of 7% being assigned to accounts that have over $10000. Then the implementation of the three functions from the base class are as follows (these will be the detailed descriptions).

Withdraw function starts off by declaring a double variable for the amount to be withdrawn. Then a do while loop that checks that amount put in is not bigger than balance and that it is not -1 as inputting -1 is the exit option. It them has a do while loop that is taking in user input for amount and after checking if the input failed (say if the user put in letters) the cache for the input stream is cleared and the user is prompted to input again. If the input is successful the amount is checked if it’s bigger than the balance or if it’s negative if it is then they are prompted to input a workable value. If the amount is all correct, then a SQL query is used to decrease the balance by the amount specified. If balance updates successfully then a message is displayed and the balance variable is updated in a similar way that its originally assigned in the constructor. It then would prompt the user to withdraw more money or type -1 to exit in the case they type -1 they would go back to managing their account after account details are displayed.

The Deposit functions are all the same. It starts similar to the withdraw with a do while loop but the conditions it’s checking for is just -1 as there is no need to check if the amount is bigger than the balance. It then has the same do while loop inside to error check the user input for amount that withdraw has. After that it simply checks if the amount to input by the user and if it is positive the balance updates happen. The logic works similar to the withdraw function but instead of removing the amount from the balance, it’s increased by the amount. Then the balance is updated exactly the same as the withdraw and the constructor.

The display function simply has a formatted print statements that display the account number, the balance, and the account type. Specifically for the Savings account it also displays the interest earnings and interest rate here, it calculates the earnings with the calculate interest function described in the following paragraph.

The Savings Account specific function is calculate interest which simply assigns a value to the interest rate variable and returns balance multiplied by the interest rate. The interest rate is decided in an if else if structure based on the balance variable.

**Checking Account** has probably the most complex implementation. The base class functions don’t really have a difference to savings account apart from the fact that the large do while statement checks that the amount entered by the user doesn’t cause the account to fall under the “overdraft limit”, which is ten percent, rather than amount being bigger than the total balance. The check looks like this - balance-amount < balance\*0.1 -

and it is also used in the if statement that would give the user feedback if they’re amount was too high to withdraw. The deposit function has exactly the same implementation as the Savings account. The display function for checking simply displays account number, balance, and account type.

The specific function for Checking account is transferring funds between the checking account to any other account. It starts by prompting the user to put the account number of the account they wish to transfer the funds to. This block of code is wrapped in the do while for checking user input that is used everywhere else and mentioned in the description of the savings account withdraw function. It then prompts the user to determine the amount they wish to transfer, if this amount is within the overdraft limit and the amount is not negative then the function continues otherwise a message is displayed and the user is sent back to the menu. So if everything is passed in correctly the function then checks to make sure that the passed in account number isn’t the same as the current account number and then checks that the passed in account number exists. If all of that passes then the two updates statements are ran updating the balances of the two accounts accordingly. Then the balance of the current account is updated the same way they’re updated in the other functions.

**Fixed Deposit Account’s** implementation is nothing crazy the only big change to the functions is the check for withdrawing and its calculate penalty function which is the unique function to this derived class. The calculate penalty function takes the 7.5% penalty and returns the product of balance times 0.075 or the 7.5%. This return value is used in the display function to inform the user of how much of withdraw penalty they would receive and is used in the withdraw function to not allow the user to go into a negative balance.

The logic for how much money can be withdrawn in the fixed deposit account incorporates the penalty. It adds the condition of checking that the sum of the amount and the penalty is not greater than the balance. This means the user has to keep in mind the 7.5% penalty when removing any funds from the fixed deposit account.

***main.cpp***

This file consists of one menu with several submenus. The menus are all displayed with do-while loops that will loop indefinitely unless the user selects the specific exit or back option that is available to them. Since this is a command line program, the menu is traditional, offering all selections as 1, 2, 3, etc. The selections are all retrieved using the getline function, and all basic navigational selections are string input, which saved a lot of error checking.

Before the menu is displayed to the user, a few SQL interactions are executed. First, a database file called bank\_system.db is connected to. If the file does not already exist, a new database is created. Then a few queries are executed in this database. They create the tables needed for this program to run, and they consist of account\_types, users, and active\_accounts. One handles all available types of accounts that can be opened (Chequings, Savings, Fixed-Deposit), one handles all users that have accounts at the bank, and the other handles all active bank accounts, pulling foreign keys from both previously mentioned tables to create rows that have a primary key (account number), a user ID tied to it, the type of account (type\_id), and the balance that account has. After the tables are made, I populate the database with the three types of account types. I also create dummy data by adding some users and some active accounts tied to those users. The dummy data could be removed, I just kept it in for ease of testing.

Once the database connection has been established and the populate queries have run, the menu is finally shown to the user. The starting point offers Login, Register, and Exit. Exit will end the program if selected, Login allows the user to enter data that already exists in the DB, and register allows the user to sign up for a new account, which adds their information to the database. I will explain register first.

***Register***

When the user selects register, they are brought to a new submenu. Variables are made for what the user will be asked to enter. They are asked to enter a name, address, and phone number. The email address and the password they enter will be used to sign them in after their account has been created. When they enter these pieces of information, a simple insert query will be executed, which creates a new entry in the users table consisting of the information that was entered. In the finished version of this project, there exist no data verification methods to ensure the user is entering things correctly. The user can currently enter anything they want, and it will be inserted into the database. With more time, I would have loved to add security and a way to make sure users were entering data correctly. After the account information is added to the DB, the user is informed and returned to the main menu where they can now login.

***Login***

When the user selects login, they are asked to enter their email and password to authenticate them. As soon as a combination is entered, a query runs to check to see how many instances of the provided email and password are in the database. If the number returned is anything other than one, the login will fail and inform the user of the failed attempt, returning them to the main menu. If the login was successful, the bulk of this program begins.

***My Account***

At the start of the main function, a heap instance of the AccountHolder class is created. It has nothing in it until any user logs in. Upon login, a query is run that pulls all information tied to the login credentials down and stores them in the class object. This makes it easier to reference the user’s information, like their unique ID and name, without having to query every time. Every time a new user logs in, this object will be updated with their information.

After this, the user is brought to My Account, which is the submenu where the user will spend most of their time banking. Their options are to logout, which takes them back to the main menu, view their accounts, which will query for all of the active accounts under their userID and display them, open a new account, which will ask them which type of account they want to open and, upon selection, insert a new account into active\_accounts of the correct type, with their userID, and a balance of 0, and finally there’s manage account, which will display all of their active accounts and allow them to enter the account number of the account they would like to work with.

***View My Accounts***

This is a very simple interaction. As I previously mentioned, this option simply queries for all rows in the database in active\_accounts where the user’s ID is found and displays all returned results to the user in a nice format, featuring the account number, type, and balance. The user can press any key in this screen to return to My Account.

***Open New Account***

This option opens a simple menu with a few options. There’s a back option, which returns to the previous menu, and then an option to make one of each account type (chequing, savings, fixed deposit). When the user enters one of these options to make an account, it simply inserts a new active account entry into active\_accounts of the corresponding type, tied to their user\_id, with a balance of 0. There are no questions, they simply press a button to open the account. These fresh accounts can be seen in View My Accounts. The user is returned to My Account after an account is opened.

***Manage Account***

This option displays all of the user’s active accounts with the account numbers, not unlike View My Accounts. The user is asked to enter the account number they wish to work with, or 0 to return to My Account. When they enter anything other than 0, a query is run to select the account number and the type\_id for an account with the entered number that is also in the user’s name (using their ID). If no value is returned, indicating that the account does not exist or it is not in their name, nothing happens, and the user can enter another account number. If a result is returned, the account type that was returned is evaluated. Depending on the type of account, a menu will appear offering options and information unique to that account. This is done by creating stack objects for the account based on which type it is. For chequing accounts, a stack object for CheckingAccount is made, and passed the queried account number. A menu is displayed featuring all the options the user can choose from relevant to chequing accounts. They can withdraw, deposit, or transfer funds to any existing account that isn’t the one transferring funds. These functionalities come from the functions specific to the CheckingAccount class. Each account type has its own unique menu that calls the display function for each account class and offers ways for the user to engage the different functions available to each class.

At the end of the main function (reached when the user exits the menu), the heap user object is freed, and the database handler is closed.

# References

GeeksforGeeks. (2018, August 22). *SQL using C/C++ and SQLite*. GeeksforGeeks. <https://www.geeksforgeeks.org/sql-using-c-c-and-sqlite/>

SQLite. (n.d.). *An introduction to the SQLite C/C++ Interface*. https://www.sqlite.org/cintro.html

Used for query verification, and kind of like a search engine

OpenAI. (2024). *ChatGPT* [Large language model]. <https://chatgpt.com>

# Code

**File (Account.h):**

// Gabriel Ponce, Samuel Cook

// PROG2100

// 2024-11-14

// Account.h

// Base class for Bank Account Types to derive from.

#ifndef ACCOUNT\_H

#define ACCOUNT\_H

#include <sqlite3.h>

#include <iostream>

#include <cmath>

class *Account*

{

    //protected so that the derived classes can access the members

protected:

    // will hold the balance of the account and the account number

    double balance;

    int accountNumber;

public:

    // Constructor for the Account class that grabs the balance from the database

    Account(int *aN*,*sqlite3* \**dbHelper*): accountNumber(*aN*){

        int dbStatus; // variable that holds the status of the database

*sqlite3\_stmt*\* stmt; // variable that holds the sql statement results

        const char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

        //grabbing balance from the database based on the passed in account number

        std::*string* sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(*aN*);

        //preparing the query

        dbStatus = sqlite3\_prepare\_v2(*dbHelper*, sql.c\_str(), -1, &stmt, &errorMessage);

        //checking if the query was prepared correctly

        if (dbStatus != SQLITE\_OK) {

            std::cout << "There was an error preparing the statement: " << errorMessage << std::endl;

        }

        //executing the query and grabbing the balance which would be in the first column

        if (sqlite3\_step(stmt) == SQLITE\_ROW) {

            balance = sqlite3\_column\_double(stmt, 0);

        }else{

            std::cout << "Account not found." << std::endl;

            //closes the program if it fails to find the account

            //this shouldn't happen because the accounts have

            //already been found in main.cpp when looking for all

            //the accounts associated with the user

            exit(1);

        }

        sqlite3\_finalize(stmt); // finalizing the statement

    };

    ~Account() = default;

    // virtual functions that will be overridden by the derived classes but will do very similar things

    virtual void withdraw(*sqlite3* \**dbHandler*) = 0; // subtracts from the balance and updates the database and then updates the variable

    virtual void deposit(*sqlite3* \**dbHandler*) = 0;  // adds to the balance and updates the database and then updates the variable

    virtual void display() = 0; //displays the account number and the balance

};

#endif

**File(DerivedAccounts.h):**

// Gabriel Ponce, Samuel Cook

// PROG2100

// 2024-11-14

// Account.h

// Base class for Bank Account Types to derive from.

#ifndef ACCOUNT\_H

#define ACCOUNT\_H

#include <sqlite3.h>

#include <iostream>

#include <cmath>

class *Account*

{

    //protected so that the derived classes can access the members

protected:

    // will hold the balance of the account and the account number

    double balance;

    int accountNumber;

public:

    // Constructor for the Account class that grabs the balance from the database

    Account(int *aN*,*sqlite3* \**dbHelper*): accountNumber(*aN*){

        int dbStatus; // variable that holds the status of the database

*sqlite3\_stmt*\* stmt; // variable that holds the sql statement results

        const char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

        //grabbing balance from the database based on the passed in account number

        std::*string* sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(*aN*);

        //preparing the query

        dbStatus = sqlite3\_prepare\_v2(*dbHelper*, sql.c\_str(), -1, &stmt, &errorMessage);

        //checking if the query was prepared correctly

        if (dbStatus != SQLITE\_OK) {

            std::cout << "There was an error preparing the statement: " << errorMessage << std::endl;

        }

        //executing the query and grabbing the balance which would be in the first column

        if (sqlite3\_step(stmt) == SQLITE\_ROW) {

            balance = sqlite3\_column\_double(stmt, 0);

        }else{

            std::cout << "Account not found." << std::endl;

            //closes the program if it fails to find the account

            //this shouldn't happen because the accounts have

            //already been found in main.cpp when looking for all

            //the accounts associated with the user

            exit(1);

        }

        sqlite3\_finalize(stmt); // finalizing the statement

    };

    ~Account() = default;

    // virtual functions that will be overridden by the derived classes but will do very similar things

    virtual void withdraw(*sqlite3* \**dbHandler*) = 0; // subtracts from the balance and updates the database and then updates the variable

    virtual void deposit(*sqlite3* \**dbHandler*) = 0;  // adds to the balance and updates the database and then updates the variable

    virtual void display() = 0; //displays the account number and the balance

};

#endif

**File(DerivedAccounts.cpp):**

// Gabriel Ponce, Samuel Cook

// PROG2100

// 2024-11-14

// DerivedAccounts.cpp

// All logic for classes that derive from Account.h

#include "DerivedAccounts.h"

//Withdraw function implementations

void *SavingsAccount*::withdraw(*sqlite3* \**dbHandler*){

    //amount to be withdrawn from the account

    double amount;

    do{//error checking loop for when the user puts an amount higher that the balance and allows them to exit when a negative number is put in

        do{ //error checking loop

                std::cout << "Enter the amount you would like to withdraw, or -1 to exit: ";

                std::cin >> amount;

                //checks if it failed ie checks for alphabetical/special characters here

                if (std::cin.fail()) {

                    std::cin.clear(); // clears the error flag on cin so that it can be used again

                    std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

                    std::cout << "\n ERROR: Invalid entry! Please enter an integer value. \n" << std::endl; // gives the user an error message and prompts them to enter a valid integer

                }else {

                    break;

                }

        }while(true);

        if (amount > balance || amount < 0){ //error message for when the user tries to withdraw more than they have

            std::cout << "Insufficient funds or invalid amount (negative), Enter a valid amount." << std::endl;

        }else{

            //updates balance value in database

            std::*string* sql = "UPDATE active\_accounts SET balance = balance - " + std::to\_string(amount) + " WHERE account\_number = " + std::to\_string(accountNumber);

            char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

            int dbStatus = sqlite3\_exec(*dbHandler*, sql.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

            //check for succesfull updating

            if (dbStatus != SQLITE\_OK) {

                std::cout << "There was an error udating balance: " << errorMessage << std::endl;

            }

            else {

                std::cout << "Withdrawal successful." << std::endl;

                // updates the balance member variable with the updated balance from the database

                sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber);

*sqlite3\_stmt*\* stmt;

                dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sql.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK){

                    std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

                    return;

                }

                dbStatus = sqlite3\_step(stmt);

                if(dbStatus == SQLITE\_ROW){

                    balance = sqlite3\_column\_double(stmt, 0);

                }else{

                    std::cout << "No Data Found" << std::endl;

                    break;

                }

            }

        }

    }while(amount > balance || amount != -1);

}

//functionally exactly the same as the savings account withdraw function

void *CheckingAccount*::withdraw(*sqlite3* \**dbHandler*){

    double amount;

    do{//error checking for the amount to be withdrawn but checks for overdraft instead of total balance

        do{

                std::cout << "Enter the amount you would like to withdraw, or -1 to exit: ";

                std::cin >> amount;

                if (std::cin.fail()) {

                    std::cin.clear();

                    std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

                    std::cout << "\n ERROR: Invalid entry! Please enter an integer value. \n" << std::endl;

                }else {

                    break;

                }

        }while(true);

        // rather than checking for total balance, we check for overdraft

        if (balance-amount < balance\*0.1 || amount < 0){

            std::cout << "Funds would go into overdraft or Invalid Amount (negative), Enter a valid amount." << std::endl;

        }else{

            std::*string* sql = "UPDATE active\_accounts SET balance = balance - " + std::to\_string(amount) + " WHERE account\_number = " + std::to\_string(accountNumber);

            char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

            int dbStatus = sqlite3\_exec(*dbHandler*, sql.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

            if (dbStatus != SQLITE\_OK) {

                std::cout << "There was an error udating balance: " << errorMessage << std::endl;

            }

            else {

                std::cout << "Withdrawal successful." << std::endl;

                sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber);

*sqlite3\_stmt*\* stmt;

                dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sql.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK){

                    std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

                    return;

                }

                dbStatus = sqlite3\_step(stmt);

                if(dbStatus == SQLITE\_ROW){

                    balance = sqlite3\_column\_double(stmt, 0);

                }else{

                    std::cout << "No Data Found" << std::endl;

                    break;

                }

            }

        }

    }while(balance-amount < balance\*0.1 || amount != -1 );

}

//functionally the same as the savings account withdraw function but with a penalty applied onto the balance when withdrawing

void *FixedDepositAccount*::withdraw(*sqlite3* \**dbHandler*){

    double amount;

    do{

        do{ //inform the user of the fee that will be charged

                std::cout << "Enter the amount you would like to withdraw, a fee of 7.5% of your balance will be charged, or -1 to exit: ";

                std::cin >> amount;

                if (std::cin.fail()) {

                    std::cin.clear();

                    std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

                    std::cout << "\n ERROR: Invalid entry! Please enter an integer value. \n" << std::endl;

                }else {

                    break;

                }

        }while(true);

        if (amount > balance || (amount + calculatePenalty()) > balance || amount < 0 ){

            std::cout << "Insufficient funds or invalid amount (negative), Enter a valid amount." << std::endl;

        }else{                                                                  // the total amount the will be subtracted from the balance with amount and the penalty

            std::*string* sql = "UPDATE active\_accounts SET balance = balance - " + std::to\_string(amount + calculatePenalty()) + " WHERE account\_number = " + std::to\_string(accountNumber);

            char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

            int dbStatus = sqlite3\_exec(*dbHandler*, sql.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

            if (dbStatus != SQLITE\_OK) {

                std::cout << "There was an error udating balance: " << errorMessage << std::endl;

            }

            else {

                std::cout << "Withdrawal successful." << std::endl;

                sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber);

*sqlite3\_stmt*\* stmt;

                dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sql.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK){

                    std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

                    return;

                }

                dbStatus = sqlite3\_step(stmt);

                if(dbStatus == SQLITE\_ROW){

                    balance = sqlite3\_column\_double(stmt, 0);

                }else{

                    std::cout << "No Data Found" << std::endl;

                   break;

                }

            }

        }

    }while(amount > balance || amount != -1);

}

//Deposit Function Implementations

//functionally similar to the withdraw functions but adds to the balance instead of subtracting

//the deposit functions are all the same

void *SavingsAccount*::deposit(*sqlite3* \**dbHandler*){

    double amount;

    do{ //error checking loop for invalid amounts

        do{ //error checking loop same as withdraw functions

                std::cout << "Enter the amount you would like to deposit, or -1 to exit: ";

                std::cin >> amount;

                 if (std::cin.fail()) {

                    std::cin.clear();

                    std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

                    std::cout << "\n ERROR: Invalid entry! Please enter an double value. \n" << std::endl;

                }else {

                    break;

                }

        }while(true);

        if (amount > 0){

            // updating and increasing the balance

            std::*string* sql = "UPDATE active\_accounts SET balance = balance + " + std::to\_string(amount) + " WHERE account\_number = " + std::to\_string(accountNumber);

            char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

            int dbStatus = sqlite3\_exec(*dbHandler*, sql.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

            // updating balance member variable

            if (dbStatus == SQLITE\_OK){

                std::cout << "Deposit successful." << std::endl;

                sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber);

*sqlite3\_stmt*\* stmt;

                dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sql.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK){

                    std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

                    return;

                }

                dbStatus = sqlite3\_step(stmt);

                if(dbStatus == SQLITE\_ROW){

                    balance = sqlite3\_column\_double(stmt, 0);

                }else{

                    std::cout << "No Data Found" << std::endl;

                   break;

                }

            }else {

                std::cout << "There was an error udating balance: " << errorMessage << std::endl;

            }

        }else if (amount == -1){

            break;

        }else{

            std::cout << "Invalid amount." << std::endl;

        }

    }while(amount != -1);

}

void *CheckingAccount*::deposit(*sqlite3* \**dbHandler*){

    double amount;

    do{

        do{

                std::cout << "Enter the amount you would like to deposit, or -1 to exit: ";

                std::cin >> amount;

                 if (std::cin.fail()) {

                    std::cin.clear();

                    std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

                    std::cout << "\n ERROR: Invalid entry! Please enter an double value. \n" << std::endl;

                }else {

                    break;

                }

        }while(true);

        if (amount > 0){

            std::*string* sql = "UPDATE active\_accounts SET balance = balance + " + std::to\_string(amount) + " WHERE account\_number = " + std::to\_string(accountNumber);

            char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

            int dbStatus = sqlite3\_exec(*dbHandler*, sql.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

            if (dbStatus == SQLITE\_OK){

                std::cout << "Deposit successful." << std::endl;

                sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber);

*sqlite3\_stmt*\* stmt;

                dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sql.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK){

                    std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

                    return;

                }

                dbStatus = sqlite3\_step(stmt);

                if(dbStatus == SQLITE\_ROW){

                    balance = sqlite3\_column\_double(stmt, 0);

                }else{

                    std::cout << "No Data Found" << std::endl;

                   break;

                }

            }else {

                std::cout << "There was an error udating balance: " << errorMessage << std::endl;

            }

        }else if (amount == -1){

            break;

        }else{

            std::cout << "Invalid amount." << std::endl;

        }

    }while(amount != -1);

}

void *FixedDepositAccount*::deposit(*sqlite3* \**dbHandler*){

    double amount;

    do{

        do{

                std::cout << "Enter the amount you would like to deposit, or -1 to exit: ";

                std::cin >> amount;

                 if (std::cin.fail()) {

                    std::cin.clear();

                    std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

                    std::cout << "\n ERROR: Invalid entry! Please enter an double value. \n" << std::endl;

                }else {

                    break;

                }

        }while(true);

        if (amount > 0){

            std::*string* sql = "UPDATE active\_accounts SET balance = balance + " + std::to\_string(amount) + " WHERE account\_number = " + std::to\_string(accountNumber);

            char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

            int dbStatus = sqlite3\_exec(*dbHandler*, sql.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

            if (dbStatus == SQLITE\_OK){

                std::cout << "Deposit successful." << std::endl;

                sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber);

*sqlite3\_stmt*\* stmt;

                dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sql.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK){

                    std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

                    return;

                }

                dbStatus = sqlite3\_step(stmt);

                if(dbStatus == SQLITE\_ROW){

                    balance = sqlite3\_column\_double(stmt, 0);

                }else{

                    std::cout << "No Data Found" << std::endl;

                   break;

                }

            }else {

                std::cout << "There was an error udating balance: " << errorMessage << std::endl;

            }

        }else if (amount == -1){

            break;

        }else{

            std::cout << "Invalid amount." << std::endl;

        }

    }while(amount != -1);

}

//Display Function Implementations

// add menus for doing stuff with the accounts

void *SavingsAccount*::display(){ //displays the account number, balance, account type, and the interest that would be earned

    std::cout << "Account Number: " << accountNumber << std::endl;

    std::cout << "Balance: $" << balance << std::endl;

    std::cout << "Account Type: Savings" << std::endl;

    std::cout << "Interest Earnings: $" << calculateInterest() << " with a rate of " << interestRate\*100 << "%" << std::endl;

}

void *CheckingAccount*::display(){ //displays the account number, balance, and account type

    std::cout << "Account Number: " << accountNumber << std::endl;

    std::cout << "Balance: $" << balance << std::endl;

    std::cout << "Account Type: Checking" << std::endl;

}

void *FixedDepositAccount*::display(){ //displays the account number, balance, account type, and the penalty that would be charged

    std::cout << "Account Number: " << accountNumber << std::endl;

    std::cout << "Balance: $" << balance << std::endl;

    std::cout << "Account Type: Fixed Deposit" << std::endl;

    std::cout << "Penalty: 7.5%" << std::endl;

    std::cout << "Penalty Amount: $" << calculatePenalty() << std::endl;

}

//Specific Function Implementions

double *SavingsAccount*::calculateInterest(){ //calculates the interest-rate based on the balance and returns the interest that would be earned

    if (balance < 1000){

        interestRate = 0.01;

    }else if (balance < 5000){

        interestRate = 0.03;

    }else if (balance < 10000){

        interestRate = 0.05;

    }else{

        interestRate = 0.07;

    }

    return balance \* interestRate;

}

void *CheckingAccount*::transferFunds(*sqlite3* \**dbHandler*){ // allows the user to transfer funds from the checking account to another account

    double amount;

    int accountNumber2;

    do{

        std::cout << "Enter the account number you would like to transfer funds to: ";

        std::cin >> accountNumber2;

        if (std::cin.fail()) {

            std::cin.clear();

            std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

            std::cout << "\n ERROR: Invalid entry! Please enter an double value. \n" << std::endl;

        }else {

            break;

        }

    }while(true);

    do{

        std::cout << "Enter the amount you would like to transfer: ";

        std::cin >> amount;

        if (std::cin.fail()) {

            std::cin.clear();

            std::cin.ignore(std::numeric\_limits<std::*streamsize*>::max(), '\n'); // clear out whatever is in the buffer.

            std::cout << "\n ERROR: Invalid entry! Please enter an double value. \n" << std::endl;

        }else {

            break;

        }

    }while(true);

    if (balance-amount < balance\*0.1 || amount < 0){

        std::cout << "Funds would go into overdraft or Invalid Amount (negative)" << std::endl;

    }else{

        if (accountNumber == accountNumber2){

            std::cout << "You can't transfer to the same account." << std::endl;

            display();

            return;

        }

        //checking if the account to transfer to exists

        std::*string* sqlCount = "SELECT COUNT(\*) FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber2);

*sqlite3\_stmt*\* stmt;

        int dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sqlCount.c\_str(), -1, &stmt, nullptr);

        if (dbStatus != SQLITE\_OK){

            std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

            return;

        }

        dbStatus = sqlite3\_step(stmt);

        //exits the function if the account to transfer to doesn't exist or the query didn't return any results

        if(dbStatus == SQLITE\_ROW){

            if (sqlite3\_column\_int(stmt, 0) == 0){

                std::cout << "Account not found." << std::endl;

                display();

                return;

            }

        }else{

            std::cout << "No Data Found" << std::endl;

            display();

            return;

        }

        sqlite3\_finalize(stmt);

        std::*string* sql = "UPDATE active\_accounts SET balance = balance + " + std::to\_string(amount) + " WHERE account\_number = " + std::to\_string(accountNumber2);

        std::*string* sql2 = "UPDATE active\_accounts SET balance = balance - " + std::to\_string(amount) + " WHERE account\_number = " + std::to\_string(accountNumber);

        char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

        dbStatus = sqlite3\_exec(*dbHandler*, sql.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

        if (dbStatus != SQLITE\_OK) {

            std::cout << "There was an error udating balance: " << errorMessage << std::endl;

        }else{

            dbStatus = sqlite3\_exec(*dbHandler*, sql2.c\_str(), 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

            if (dbStatus != SQLITE\_OK) {

                std::cout << "There was an error udating balance: " << errorMessage << std::endl;

            }

            else {

                sql = "SELECT balance FROM active\_accounts WHERE account\_number = " + std::to\_string(accountNumber);

*sqlite3\_stmt*\* stmt;

                dbStatus = sqlite3\_prepare\_v2(*dbHandler*, sql.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK){

                    std::cout << "There was an error preparing the statement: " << sqlite3\_errmsg(*dbHandler*) << std::endl;

                    return;

                }

                dbStatus = sqlite3\_step(stmt);

                if(dbStatus == SQLITE\_ROW){

                    balance = sqlite3\_column\_double(stmt, 0);

                }else{

                    std::cout << "No Data Found" << std::endl;

                }

            }

        }

    }

    display();

}

//returns the penalty that will be charged

double *FixedDepositAccount*::calculatePenalty(){

    return (balance \* penalty);

}

**File(AccountHolder.h):**

// Gabriel Ponce, Samuel Cook

// PROG2100

// 2024-11-14

// AccountHolder.h

// Class used to store login and personal information for bank customers.

#include <string>

#ifndef ACCOUNT\_HOLDER\_H

#define ACCOUNT\_HOLDER\_H

class *AccountHolder*

{

    // They will be assigned manualy in the main.cpp file.

public:

    // Member variables to hold the user's information.

    int id;

    std::*string* name;

    std::*string* address;

    std::*string* email;

    std::*string* password;

    std::*string* phone;

    ~AccountHolder() = default;

};

#endif

**File(main.cpp):**

// Gabriel Ponce, Samuel Cook

// PROG2100

// 2024-11-14

// main.cpp

// The logic for the menu interface users will interact with.

#include "DerivedAccounts.h"

#include "AccountHolder.h"

#include <iostream>

int main() {

*sqlite3* \*dbHandler; // Pointer variable used to carry out database interactions.

    int dbStatus; // Variable used to track the status of recently executed queries. Successful connections are 0s.

*AccountHolder*\* user = **new** *AccountHolder*(); // I made this at the start so it can reach all corners of the menu.

    // It's an AccountHolder object that will be used to hold logged-in user's data.

    // Every time a new user logs in, this object will have their data.

    // I thought it would be okay to make it here since the user has no way of accessing it before logging in.

    dbStatus = sqlite3\_open("bank\_system.db", &dbHandler); // Attempt to open a database called bank\_system.db. If it doesn't exist, make it.

    if (dbStatus != 0) { // Check to see if the database connection failed.

        std::cout << "An error occurred while trying to access the database.\nExiting..." << std::endl;

        std::cout << " Press any key to continue...";

        system("pause >nul");

        exit(0); // Exit the program.

    }

    // Create a variable that will hold a query used to create tables for the database.

    const char \*createTables = R"(

        CREATE TABLE IF NOT EXISTS account\_types (

            type\_id INTEGER UNIQUE PRIMARY KEY AUTOINCREMENT NOT NULL,

            type VARCHAR(50) UNIQUE NOT NULL

        );

        CREATE TABLE IF NOT EXISTS users (

            user\_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,

            email VARCHAR(100) UNIQUE NOT NULL,

            password VARCHAR(32) NOT NULL,

            name VARCHAR(50) NOT NULL,

            address VARCHAR(100) NOT NULL,

            phone VARCHAR(14) NOT NULL

        );

        CREATE TABLE IF NOT EXISTS active\_accounts (

            account\_number INTEGER UNIQUE PRIMARY KEY AUTOINCREMENT NOT NULL,

            user\_id INTEGER NOT NULL,

            type\_id INTEGER NOT NULL,

            balance DOUBLE NOT NULL

        );

    )";

    char \*errorMessage = nullptr; // This is used to display the resulting error message (if there is an error).

    dbStatus = sqlite3\_exec(dbHandler, createTables, 0, 0, &errorMessage); // this will return a 0 if the query executes successfully.

    if (dbStatus != 0) { // Check for any issues making the database.

        std::cout << "An error occured while creating tables: " << errorMessage << std::endl;

        exit(0);

    }

    // Adding sample users to the DB

    const char \*populate1 = "INSERT INTO users (email, password, name, address, phone) VALUES "

                                "('sam@nscc.ca', 'sac00i2', 'Samuel Cook', '1 Infinite Loop, CA', '(902)688-1229'),"

                                "('albowlly@nscc.ca', 'albowlly1897', 'Al Bowlly', 'Overlook Hotel, USA', '(244)531-5673'),"

                                "('mikemyers@nscc.ca', 'halloweenFan', 'Michael Myers', '45 Lampkin Lane, Haddonfield', '(196)719-7831')";

    // Add the account types to the DB. This is a must!

    const char \*populate2 = "INSERT INTO account\_types (type) VALUES "

                            "('Chequings'),"

                            "('Savings'),"

                            "('Fixed-Deposit')";

    // Add some active accounts to the DB

    const char \*populate3 = "INSERT INTO active\_accounts (account\_number, user\_id, type\_id, balance) VALUES "

                            "('1', '1', '1', '10436.46'),"

                            "('2', '1', '3', '2934.52'),"

                            "('3', '2', '1', '635672.47'),"

                            "('4', '2', '2', '7357.84'),"

                            "('5', '3', '1', '167.34'),"

                            "('6', '3', '2', '34.75'),"

                            "('7', '3', '3', '1978.31')";

    // Execute the populate queries

    sqlite3\_exec(dbHandler, populate1, 0, 0, &errorMessage);

    sqlite3\_exec(dbHandler, populate2, 0, 0, &errorMessage);

    sqlite3\_exec(dbHandler, populate3, 0, 0, &errorMessage);

    // populate1 and populate3 are not necessary, they could be removed entirely and there would be no accounts.

    // Main Menu starts here.

    system("cls"); // Clear the terminal, this happens a lot in this program.

    do { // Print the main menu for the user.

        std::*string* selection; // Used to hold the user's choice.

        std::cout << " ============ THE BANK ============" << std::endl;

        std::cout << " 1 - Login" << std::endl;

        std::cout << " 2 - Register" << std::endl;

        std::cout << " 3 - Exit\n Please enter your selection: ";

        getline(std::cin, selection);

        // Evaluate the user's selection.

        if (selection == "1") {

            // variables to hold login credentials.

            std::*string* email;

            std::*string* password;

            system("cls"); // Print the login screen.

            std::cout << " ============= LOGIN ==============" << std::endl;

            std::cout << " Please enter your email: ";

            getline(std::cin, email);

            std::cout << " Please enter your password: ";

            getline(std::cin, password);

            // Create a query to check for matches of the email and password.

            std::*string* sqlSelect = "SELECT COUNT(\*) FROM users WHERE email = '" + email + "' AND password = '" + password + "';";

*sqlite3\_stmt*\* stmt;

            // Prepare the query

            dbStatus = sqlite3\_prepare\_v2(dbHandler, sqlSelect.c\_str(), -1, &stmt, nullptr);

            if (dbStatus != SQLITE\_OK) { // Check if the query failed.

                std::cout << " There was an error signing you in." << std::endl;

            }

            int count = 0; // Variable used to count how many rows were returned.

            dbStatus = sqlite3\_step(stmt); // Step through every row.

            if (dbStatus == SQLITE\_ROW) {

                count = sqlite3\_column\_int(stmt, 0); // Set the count to the number of returned rows.

            }

            if (count == 0 || count > 1) { // If no records are found or, for some reason, duplicate records are found, inform the user of the failed login.

                std::cout << " Login Failed.\n We have no record of you at this bank.\n Please consider registering an account with us.\n" << std::endl;

                std::cout << " Press any key to continue...";

                system("pause >nul");

            }

            else {

                // Populate the user object upon login with the account information from the DB.

                // It will be used to open accounts and display information without working with a specific account.

                std::*string* sqlSelect = "SELECT user\_id, email, password, name, address, phone FROM users WHERE email = '" + email + "' AND password = '" + password + "';";

*sqlite3\_stmt*\* stmt;

                // Check if the query failed for some reason.

                dbStatus = sqlite3\_prepare\_v2(dbHandler, sqlSelect.c\_str(), -1, &stmt, nullptr);

                if (dbStatus != SQLITE\_OK) {

                    std::cout << " An error occurred while fetching your information." << std::endl;

                }

                dbStatus = sqlite3\_step(stmt); // Step through every row.

                if (dbStatus == SQLITE\_ROW) {

                    // Retrieve all of the queried values.

                    int user\_id = sqlite3\_column\_int(stmt, 0);

                    const unsigned char\* emailResult = sqlite3\_column\_text(stmt, 1);

                    const unsigned char\* passwordResult = sqlite3\_column\_text(stmt, 2);

                    const unsigned char\* name = sqlite3\_column\_text(stmt, 3);

                    const unsigned char\* address = sqlite3\_column\_text(stmt, 4);

                    const unsigned char\* phone = sqlite3\_column\_text(stmt, 5);

                    // Cast the query results into strings for the constructor to use.

                    std::*string* accEmail(reinterpret\_cast<const char\*>(emailResult));

                    std::*string* accPass(reinterpret\_cast<const char\*>(passwordResult));

                    std::*string* accName(reinterpret\_cast<const char\*>(name));

                    std::*string* accAddr(reinterpret\_cast<const char\*>(address));

                    std::*string* accPhone(reinterpret\_cast<const char\*>(phone));

                    // I couldn't figure out how to avoid making a second set of variables.

                    // Set the variables of the user object to the retrieved values of the most

                    // recently logged-in user. This will happen every time a new user logs in.

                    // We just made these variables public for simplicity's sake, but we could

                    // have done the same thing with private variables by using setters.

                    user->id = user\_id;

                    user->email = accEmail;

                    user->password = accPass;

                    user->name = accName;

                    user->address = accAddr;

                    user->phone = accPhone;

                }

                sqlite3\_finalize(stmt); // Finalize the statement

                system("cls");

                do { // Print the "Homepage" for logged in users.

                    selection = "";

                    std::cout << " ============ MY ACCOUNT ==========" << std::endl;

                    std::cout << " Hello, " << user->name << "." << std::endl;

                    std::cout << " 1 - View My Accounts" << std::endl;

                    std::cout << " 2 - Open New Account" << std::endl;

                    std::cout << " 3 - Manage Account" << std::endl;

                    std::cout << " 4 - Logout\n Please enter your selection: ";

                    getline(std::cin, selection);

                    if (selection == "1") { // Displaying all accounts tied to the logged-in user.

                        system("cls");

                        std::cout << " Showing Accounts for " << user->name << "..." << std::endl;

                        // Select all accounts based on the user's unique ID.

                        std::*string* sqlSelect = "SELECT account\_number, type\_id, balance FROM active\_accounts WHERE user\_id = '" + std::to\_string(user->id) + "';";

*sqlite3\_stmt*\* stmt;

                        dbStatus = sqlite3\_prepare\_v2(dbHandler, sqlSelect.c\_str(), -1, &stmt, nullptr);

                        if (dbStatus != SQLITE\_OK) {

                            std::cout << " There was an error processing your request." << std::endl;

                        }

                            std::*string* accType; // Used to hold a user-readable display of each account's type.

                            // Loop through the results.

                            while ((dbStatus = sqlite3\_step(stmt)) == SQLITE\_ROW) {

                                // Retrieve all queried values.

                                int account\_number = sqlite3\_column\_int(stmt, 0);

                                int type\_id = sqlite3\_column\_int(stmt, 1);

                                double balance = sqlite3\_column\_double(stmt, 2);

                                // Evaluate the type\_id to display the corresponding account type.

                                switch (type\_id) {

                                    case 1:

                                        accType = "Chequings";

                                        break;

                                    case 2:

                                        accType = "Savings";

                                        break;

                                    case 3:

                                        accType = "Fixed-Deposit";

                                        break;

                                }

                                // For each returned result, display the account type, the number, and the balance.

                                std::cout << "\n =========== ACCOUNT #" << account\_number << " ===========" << std::endl;

                                std::cout << " " << accType << ", Balance: $" << balance << std::endl;

                            }

                            if (dbStatus != SQLITE\_DONE) {

                                std::cout << " An error occurred while fetching your accounts." << std::endl;

                            }

                            sqlite3\_finalize(stmt);

                        std::cout << "\n Press any key to return to the previous screen. ";

                        system("pause >nul");

                        system("cls");

                    }

                    else if (selection == "2") { // Create a new account.

                        system("cls");

                        do { // Display the menu.

                            selection = "";

                            std::cout << " ========= CREATE ACCOUNT =========" << std::endl;

                            std::cout << " 1 - New Chequing Account" << std::endl;

                            std::cout << " 2 - New Savings Account" << std::endl;

                            std::cout << " 3 - New Fixed-Deposit Account" << std::endl;

                            std::cout << " 4 - Back\n Please enter your selection: ";

                            getline(std::cin, selection);

                            if (selection == "1") { // New Chequing Account

                                // Query to make a new active\_account entry of chequing acount type, tied to the user's ID, with an empty balance.

                                std::*string* checkingQuery = "INSERT INTO active\_accounts (user\_id, type\_id, balance) VALUES ('" + std::to\_string(user->id) + "', 1, 0.00)";

                                dbStatus = sqlite3\_exec(dbHandler, checkingQuery.c\_str(), 0, 0, &errorMessage);

                                if (dbStatus != SQLITE\_OK) {

                                    std::cout << " There was an error creating your account.\n Please try again." << errorMessage << std::endl;

                                }

                                else {

                                    std::cout << " New Chequing Account Opened!" << std::endl;

                                }

                                std::cout << " Press any key to continue...";

                                system("pause >nul");

                                break;

                            }

                            else if (selection == "2") { // Make the same kind of entry in active\_accounts, but for a savings account.

                                std::*string* savingsQuery = "INSERT INTO active\_accounts (user\_id, type\_id, balance) VALUES ('" + std::to\_string(user->id) + "', 2, 0.00)";

                                dbStatus = sqlite3\_exec(dbHandler, savingsQuery.c\_str(), 0, 0, &errorMessage);

                                if (dbStatus != SQLITE\_OK) {

                                    std::cout << " There was an error creating your account.\n Please try again." << errorMessage << std::endl;

                                }

                                else {

                                    std::cout << " New Savings Account Opened!" << std::endl;

                                }

                                std::cout << " Press any key to continue...";

                                system("pause >nul");

                                break;

                            }

                            else if (selection == "3") { // ... and again, but for the Fixed-Deposit Account.

                                std::*string* fixedQuery = "INSERT INTO active\_accounts (user\_id, type\_id, balance) VALUES ('" + std::to\_string(user->id) + "', 3, 0.00)";

                                dbStatus = sqlite3\_exec(dbHandler, fixedQuery.c\_str(), 0, 0, &errorMessage);

                                if (dbStatus != SQLITE\_OK) {

                                    std::cout << " There was an error creating your account.\n Please try again." << errorMessage << std::endl;

                                }

                                else {

                                    std::cout << " New Fixed-Deposit Account Opened!" << std::endl;

                                }

                                std::cout << " Press any key to continue...";

                                system("pause >nul");

                                break;

                            }

                            else if (selection == "4") { // Return to My Account with a simple break.

                                break;

                            }

                            else { // Invalid entry just clears the screen.

                                system("cls");

                            }

                        } while (true); // Loop forever.

                        system("cls"); // Clear the screen upon exiting the loop.

                    }

                    else if (selection == "3") { // Manage an existing account.

                        do { // Display all active accounts.

                            system("cls");

                            selection = "";

                            std::cout << " ========= MANAGE ACCOUNT =========" << std::endl;

                            std::cout << " Which account would you like to manage?" << std::endl;

                            // This is the same query as the "View My Accounts" feature. It retrieves all active accounts tied to the user.

                            std::*string* sqlSelect = "SELECT account\_number, type\_id, balance FROM active\_accounts WHERE user\_id = '" + std::to\_string(user->id) + "';";

*sqlite3\_stmt*\* stmt;

                            dbStatus = sqlite3\_prepare\_v2(dbHandler, sqlSelect.c\_str(), -1, &stmt, nullptr);

                            if (dbStatus != SQLITE\_OK) {

                                std::cout << " There was an error processing your request." << std::endl;

                            }

                            // This is doing the same thing that happens for View All Accounts, making the type meaningful.

                            std::*string* accType;

                            while ((dbStatus = sqlite3\_step(stmt)) == SQLITE\_ROW) {

                                int account\_number = sqlite3\_column\_int(stmt, 0);

                                int type\_id = sqlite3\_column\_int(stmt, 1);

                                double balance = sqlite3\_column\_double(stmt, 2);

                                switch (type\_id) {

                                    case 1:

                                        accType = "Chequings";

                                        break;

                                    case 2:

                                        accType = "Savings";

                                        break;

                                    case 3:

                                        accType = "Fixed-Deposit";

                                        break;

                                }

                                // Display each account with the number, type, and balance.

                                std::cout << "\n =========== ACCOUNT #" << account\_number << " ===========" << std::endl;

                                std::cout << " " << accType << ", Balance: $" << balance << std::endl;

                            }

                            if (dbStatus != SQLITE\_DONE) {

                                std::cout << " An error occurred while fetching your accounts." << std::endl;

                            }

                            sqlite3\_finalize(stmt);

                            // Ask for input.

                            std::cout << "\n Please enter the account number, or\n enter 0 to return to My Account: ";

                            getline(std::cin, selection);

                            if (selection == "0") { // Check if the user wants to leave right away. 0 cannot be an account number, which is why it was selected.

                                break;

                            }

                            else {

                                // Retrieve the account number from the active account that the user selected. Guarantees the account exists.

                                std::*string* sqlSelect = "SELECT account\_number, type\_id FROM active\_accounts WHERE user\_id = '" + std::to\_string(user->id) + "' AND account\_number = '" + selection + "';";

*sqlite3\_stmt*\* stmt;

                                dbStatus = sqlite3\_prepare\_v2(dbHandler, sqlSelect.c\_str(), -1, &stmt, nullptr);

                                if (dbStatus != SQLITE\_OK) {

                                    std::cout << " An error occurred while selecting your account.\n Please try again later." << std::endl;

                                }

                                // Make sure a value was returned.

                                if ((dbStatus = sqlite3\_step(stmt)) == SQLITE\_ROW) {

                                    int account\_number = sqlite3\_column\_int(stmt, 0);

                                    int type\_id = sqlite3\_column\_int(stmt, 1);

                                    sqlite3\_finalize(stmt);

                                    system("cls");

                                    if (type\_id == 1) { // If the account the user picked was a chequing account...

*CheckingAccount* chequings(account\_number, dbHandler); // Make a stack object for the checking account using the account\_number.

                                        do { // Display the menu.

                                            selection = "";

                                            chequings.display();

                                            std::cout << "\n What would you like to do?" << std::endl;

                                            std::cout << " 1 - Deposit" << std::endl;

                                            std::cout << " 2 - Withdraw" << std::endl;

                                            std::cout << " 3 - Transfer Funds" << std::endl;

                                            std::cout << " 4 - Back\n Please enter your selection: ";

                                            getline(std::cin, selection);

                                            // Evaluate the user's input.

                                            if (selection == "1") {

                                                chequings.deposit(dbHandler); // Calls the deposit function.

                                            }

                                            else if (selection == "2") {

                                                chequings.withdraw(dbHandler); // Calls the withdraw function.

                                            }

                                            else if (selection == "3") {

                                                chequings.transferFunds(dbHandler); // Calls the transfer funds function.

                                                system("pause"); // Let the user see the result of the transfer before clearing.

                                            }

                                            else if (selection == "4") { // Go to the previous menu.

                                                break;

                                            }

                                            else { // Invalid input clears the screen.

                                                system("cls");

                                            }

                                        } while (true);

                                    }

                                    else if (type\_id == 2) { // If the account the user picked was a savings account...

*SavingsAccount* savings(account\_number, dbHandler); // Make a stack object for SavingsAccount.

                                        do { // Display the menu.

                                            selection = "";

                                            savings.display();

                                            std::cout << "\n What would you like to do?" << std::endl;

                                            std::cout << " 1 - Deposit" << std::endl;

                                            std::cout << " 2 - Withdraw" << std::endl;

                                            std::cout << " 3 - Back\n Please enter your selection: ";

                                            getline(std::cin, selection);

                                            if (selection == "1") { // Deposit

                                                savings.deposit(dbHandler);

                                            }

                                            else if (selection == "2") { // Withdraw.

                                                savings.withdraw(dbHandler);

                                            }

                                            else if (selection == "3") { // Go to the previous menu.

                                                break;

                                            }

                                            else { // Clear screen on invalid input.

                                                system("cls");

                                            }

                                        } while (true);

                                    }

                                    else if (type\_id == 3) { // If the account the user picked was a chequing account...

*FixedDepositAccount* fixed(account\_number, dbHandler); // Make a new stack object for FixedDepositAccount.

                                        do { // Display the menu.

                                            selection = "";

                                            fixed.display();

                                            std::cout << "\n What would you like to do?" << std::endl;

                                            std::cout << " 1 - Deposit" << std::endl;

                                            std::cout << " 2 - Withdraw" << std::endl;

                                            std::cout << " 3 - Back\n Please enter your selection: ";

                                            getline(std::cin, selection);

                                            if (selection == "1") { // Deposit.

                                                fixed.deposit(dbHandler);

                                            }

                                            else if (selection == "2") { // Withdraw.

                                                fixed.withdraw(dbHandler);

                                            }

                                            else if (selection == "3") { // Go back.

                                                break;

                                            }

                                            else { // Invalid input, clear screen.

                                                system("cls");

                                            }

                                        } while (true);

                                    }

                                }

                            }

                        } while (true);

                        system("cls");

                    }

                    else if (selection == "4") { // Logout, return to main menu.

                        break;

                    }

                    else { // Invalid input, clear screen.

                        system("cls");

                    }

                } while (true);

            }

            system("cls");

        }

        else if (selection == "2") { // Register a new user-account for the bank. Does not make a Bank Account, just a login for a profile at the bank.

            // Variables to hold the user's entries.

            std::*string* name;

            std::*string* address;

            std::*string* number;

            std::*string* email;

            std::*string* password;

            system("cls");

            std::cout << " ============ REGISTER ============" << std::endl;

            std::cout << " Please enter your full name: ";

            getline(std::cin, name);

            std::cout << " Please enter your full address: ";

            getline(std::cin, address);

            std::cout << " Please enter your phone number: ";

            getline(std::cin, number);

            std::cout << " Please enter your email: ";

            getline(std::cin, email);

            std::cout << " Please enter your password: ";

            getline(std::cin, password);

            // I did not implement any data authentication since security was not one of

            // the main focuses of this project.

            // Query to insert new user into the database using their entries.

            std::*string* registerQuery = "INSERT INTO users (name, address, phone, email, password) VALUES ('" + name +"', '" + address + "', '" + number + "', '" + email + "', '" + password + "')";

            // Execute insert query.

            dbStatus = sqlite3\_exec(dbHandler, registerQuery.c\_str(), 0, 0, &errorMessage);

            if (dbStatus != SQLITE\_OK) {

                std::cout << " There was an error creating your account.\n This email may already be in use." << std::endl;

            }

            else {

                std::cout << " Account created!" << std::endl;

            }

            std::cout << " Press any key to continue...";

            system("pause >nul");

            system("cls");

        }

        else if (selection == "3") { // Exit the program.

            system("cls");

            std::cout << " Exiting...\n" << std::endl;

            break;

        }

        else {

            system("cls");

        }

    } while (true);

    sqlite3\_close(dbHandler); // Close the dbHandler finally.

**delete** user; // Delete the heap user object.

    return 0;

}

# Database Schema (SQL Create)

CREATE TABLE IF NOT EXISTS account\_types (type\_id INTEGER UNIQUE PRIMARY KEY AUTOINCREMENT NOT NULL, type VARCHAR(50) UNIQUE NOT NULL);

CREATE TABLE IF NOT EXISTS users (user\_id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, email VARCHAR(100) UNIQUE NOT NULL, password VARCHAR(32) NOT NULL, name VARCHAR(50) NOT NULL, address VARCHAR(100) NOT NULL, phone VARCHAR(14) NOT NULL);

CREATE TABLE IF NOT EXISTS active\_accounts (account\_number INTEGER UNIQUE PRIMARY KEY AUTOINCREMENT NOT NULL, user\_id INTEGER NOT NULL, type\_id INTEGER NOT NULL, balance DOUBLE NOT NULL);