Simulated Banking Transaction System

Name: 劉家均

ID: 411221426

Video Presentation:

(https://youtu.be/rzJUNK_BBnI)

Introduction

This C++ program simulates basic banking operations, allowing users to manage accounts through a menu-driven interface. The program supports viewing balances, depositing and withdrawing funds, creating new accounts, and transferring money between accounts. Data persistence is achieved using text files. **The complete source code is pasted at the end of the report.**

Account Class

The `Account` class encapsulates the details of a bank account and provides methods for saving and loading account data from files.

Members and Constructors

The class has three members: `Name`, `Customer_number`, and `Balance`. It also includes a parameterized constructor for initialization and a default constructor.

```
class Account {
public:
  string Name;
  int Customer_number;
  int Balance;
  Account(string name, int customer_number, int balance)
     : Name(name), Customer_number(customer_number), Balance(balance) {}
  Account() : Name(""), Customer_number(0), Balance(0) {}
};
```

Saving to a File function

The `save_to_file` method saves the account details to a text file named after the `Customer_number.` Here is the breakdown of the `save_to_file` method:

Filename Generation

- The filename is generated by converting the `Customer_number` to a string using `to_string(Customer_number)` and appending the `.txt` extension.

Opening the File

- An `ofstream` object named `outfile` is created to handle file output operations. The file is opened with the name specified by `filename`.

Checking if the File is Open

- Before attempting to write to the file, the method checks if the file has been successfully opened using the `is_open()` method of the `ofstream` object (`outfile`).

Writing to the File

- If the file is open, the method uses the insertion operator (`<<`) to write the account details (`Name`, `Customer_number`, `Balance`) to the file. Each detail is written on a new line.

Closing the File

- After writing the details, the file is closed using the `close()` method of the `ofstream` object. This ensures that all data is properly saved and resources are released.

Error Handling

- If the file cannot be opened (i.e., `outfile.is_open()` returns `false`), an error message is printed to the console indicating the failure to create the file with the specified `filename`.
-

Here's the complete method code for reference:

```
void save_to_file() {
  string filename = to_string(Customer_number) + ".txt";
  ofstream outfile(filename);
  if (outfile.is_open()) {
    outfile << "Name: " << Name << "\n";
    outfile << "Customer Number: " << Customer_number << "\n";
    outfile << "Balance: " << Balance << "\n";
    outfile.close();
```

```
  } else {
    cout << "Failed to create file " << filename << endl;
  }
}
```

Loading from a File

The `load_from_file` method loads account details from a text file based on the `customer_number`. It returns `true` if the file is successfully loaded, `false` otherwise.

Here is the breakdown of the `load_from_file` method:

1. **Filename Generation**
    - The filename is generated by converting the `customer_number` to a string using `to_string(customer_number)` and appending the `.txt` extension.
2. **Opening the File**
    - An `ifstream` object named `infile` is created to handle file input operations. The file specified by `filename` is opened for reading.
3. **Checking if the File is Open**
    - The method checks if the file has been successfully opened using the `is_open()` method of the `ifstream`object (`infile`).
4. **Reading and Parsing Data**
    - If the file is open, the method reads each line from the file using `getline(infile, line)`.
    - The account details (`Name`, `Customer_number`, `Balance`) are parsed from each line:
        - The `Name` is extracted from the first line after the substring ": ".
        - The `Customer_number` is extracted and converted to an integer from the second line after the substring ": ".
        - The `Balance` is similarly extracted and converted to an integer from the third line after the substring ": ".
5. **Closing the File**
    - After reading and parsing the data, the file is closed using the `close()` method of the `ifstream` object (`infile`). This ensures that all resources associated with the file are released.
6. **Returning Success or Failure**

- If the file was successfully opened and data was read, the method returns `true`, indicating successful loading of account details.
- If the file could not be opened (i.e., `infile.is_open()` returns `false`), the method returns `false`, indicating failure to load account details.

Here's the complete method code for reference:

```
bool load_from_file(int customer_number) {
  string filename = to_string(customer_number) + ".txt";
  ifstream infile(filename);
  if (infile.is_open()) {
    string line;
    getline(infile, line);
    Name = line.substr(line.find(": ") + 2);
    getline(infile, line);
    Customer_number = stoi(line.substr(line.find(": ") + 2));
    getline(infile, line);
    Balance = stoi(line.substr(line.find(": ") + 2));
    infile.close();
    return true;
  } else {
    return false;
  }
}
```

Bank Class

The `Bank` class provides various functionalities for managing bank accounts, including a menu interface.

Menu Display

The `menu` method displays the available options to the user.

```
void menu() {
  cout << "\n**** menu ****\n"
       << "----1: balance\n"
       << "----2: deposit\n"
       << "----3: withdraw\n"
       << "----4: create account\n"
       << "----5: transfer\n"
       << "----6: cancel\n";
}
```

Checking Balance

The `balance` method displays the current balance of a specified account.

Here's the breakdown of the `balance` method:

1. **Account Initialization**
    - An instance of the `Account` class named `account` is created. This object will be used to load account details from a file.
2. **Loading Account Details**
    - The `load_from_file` method of the `Account` class is called with the `customer_number` parameter. This method attempts to load the account details from a text file associated with the specified `customer_number`.
    - If `load_from_file` returns `true`, it means the account details were successfully loaded from the file into the `account` object.
3. **Displaying Account Balance**
    - If the account details were successfully loaded (`load_from_file` returned `true`), the method prints the customer's name (`account.Name`) and their current balance (`account.Balance`) formatted as a monetary value.
4. **Error Handling**
    - If `load_from_file` returns `false`, it indicates that the account details could not be loaded, typically because the file corresponding to the `customer_number` does not exist or could not be opened.
    - In such cases, the method outputs an error message indicating that the account for the given customer number could not be found.
    -

Here is the complete method code for reference:

```
void balance(int customer_number) {
  Account account;
  if (account.load_from_file(customer_number)) {
    cout << "Customer " << account.Name << "'s current balance is: $"
        << account.Balance << endl;
  } else {
    cout << "Cannot find account for the customer number.\n";
  }
}
```

Depositing Money

The `deposit` method allows the user to deposit money into a specified account.

Here's the breakdown of the `deposit` method:

1. **Input Prompt**
    - The method begins by prompting the user to enter a deposit amount using `cout << "Enter deposit amount: $";`.
2. **Amount Input**
    - The user's input for the deposit amount is read from the standard input (`cin >> amount;`).
3. **Account Initialization**
    - An instance of the `Account` class named `account` is created. This object will be used to load and update account details.
4. **Loading Account Details**
    - The `load_from_file` method of the `Account` class is called with the `customer_number` parameter. This method attempts to load the account details from a text file associated with the specified `customer_number`.
    - If `load_from_file` returns `true`, it means the account details were successfully loaded from the file into the `account` object.
5. **Deposit Validation and Execution**
    - If the account details were successfully loaded (`load_from_file` returned `true`), the method checks if the `amount` entered by the user is greater than zero (`amount > 0`).

- If `amount` is valid (greater than zero), the deposit amount (`amount`) is added to the current balance (`account.Balance`).
- The updated account details are then saved back to the file using the `save_to_file` method of the `Account`class (`account.save_to_file()`).
- A success message, "Deposit successful!", is displayed to indicate that the deposit operation was completed.

6. **Error Handling**

- If `load_from_file` returns `false`, it indicates that the account details could not be loaded, typically because the file corresponding to the `customer_number` does not exist or could not be opened.
- In such cases, the method outputs an error message indicating that the account for the given customer number could not be found (`cout << "Cannot find account for the customer number.\n";`).

Here is the complete method code for reference:

```
void deposit(int customer_number) {
  int amount;
  cout << "Enter deposit amount: $";
  cin >> amount;
  Account account;
  if (account.load_from_file(customer_number)) {
    if (amount > 0) {
      account.Balance += amount;
      account.save_to_file();
      cout << "Deposit successful!\n";
    } else {
      cout << "Invalid amount. Please enter a positive value.\n";
    }
  } else {
    cout << "Cannot find account for the customer number.\n";
  }
}
```

Withdrawing Money

The `withdraw` method allows the user to withdraw money from a specified account.

Here's the breakdown of the `withdraw` method:

1. **Input Prompt**
    - The method starts by prompting the user to enter a withdrawal amount using `cout << "Enter withdrawal amount: $";`.
2. **Amount Input**
    - The user's input for the withdrawal amount is read from the standard input (`cin >> amount;`).
3. **Account Initialization**
    - An instance of the `Account` class named `account` is created. This object will be used to load and update account details.
4. **Loading Account Details**
    - The `load_from_file` method of the `Account` class is called with the `customer_number` parameter. This method attempts to load the account details from a text file associated with the specified `customer_number`.
    - If `load_from_file` returns `true`, it means the account details were successfully loaded from the file into the `account` object.
5. **Withdrawal Validation and Execution**
    - If the account details were successfully loaded (`load_from_file` returned `true`), the method checks the validity of the withdrawal amount:
        - It first verifies if `amount` is greater than zero (`amount > 0`).
        - Then, it checks if `amount` is less than or equal to the current balance (`amount <= account.Balance`).
    - If both conditions are met, the withdrawal amount (`amount`) is subtracted from the current balance (`account.Balance`).
    - The updated account details are then saved back to the file using the `save_to_file` method of the `Account`class (`account.save_to_file()`).
    - A success message, "Withdrawal successful!", is displayed to indicate that the withdrawal operation was completed.
6. **Error Handling**
    - If `load_from_file` returns `false`, it indicates that the account details could not be loaded, typically because the file corresponding to the `customer_number` does not exist or could not be opened.
    - In such cases, the method outputs an error message indicating that the account for the given customer number could not be found (`cout << "Cannot find account for the customer number.\n";`).

- If the withdrawal amount is greater than the current balance (`amount > account.Balance`), the method outputs a message indicating insufficient balance (`cout << "Insufficient balance.\n";`).
- If the withdrawal amount is not greater than zero (`amount <= 0`), the method outputs a message indicating an invalid amount (`cout << "Invalid amount. Please enter a positive value.\n";`).

Here is the complete method code for reference:

```
void withdraw(int customer_number) {
  int amount;
  cout << "Enter withdrawal amount: $";
  cin >> amount;
  Account account;
  if (account.load_from_file(customer_number)) {
    if (amount > 0 && amount <= account.Balance) {
      account.Balance -= amount;
      account.save_to_file();
      cout << "Withdrawal successful!\n";
    } else if (amount > account.Balance) {
      cout << "Insufficient balance.\n";
    } else {
      cout << "Invalid amount. Please enter a positive value.\n";
    }
  } else {
    cout << "Cannot find account for the customer number.\n";
  }
}
```

Creating a New Account

The `create_account` method prompts the user for details and creates a new account.

Here's the breakdown of the `create_account` method:

1. **Input Prompt**
    - The method begins by prompting the user to enter their name using `cout << "Enter your name: ";`.
    - The user's input for the name is read from the standard input (`cin >> name;`).
2. **Customer Number Input**
    - The method prompts the user to enter their customer number using `cout << "Enter your customer number: ";`.
    - The user's input for the customer number is read from the standard input (`cin >> customer_number;`).
3. **Account Creation**
    - An instance of the `Account` class named `new_account` is created using the provided `name`, `customer_number`, and an initial balance of `0`.
    - This initializes the account object with the user-provided name, customer number, and a starting balance.
4. **Saving Account Details**
    - The `save_to_file` method of the `Account` class is called on the `new_account` object. This method saves the account details (name, customer number, balance) to a text file named after the customer number (`customer_number.txt`).
5. **Success Message**
    - After successfully saving the account details, a confirmation message is displayed to the user: `cout << "Account created successfully!\n";`.

Here is the complete method code for reference:

```
void create_account() {
  string name;
  int customer_number;
  cout << "Enter your name: ";
  cin >> name;
  cout << "Enter your customer number: ";
  cin >> customer_number;
  Account new_account(name, customer_number, 0);
  new_account.save_to_file();
  cout << "Account created successfully!\n";
```

```
}
```

Transferring Money

The `transfer` method allows the user to transfer money from one account to another.

Here's the breakdown of the `transfer` method:

1. **Input Prompt**
    - The method starts by prompting the user to enter the transfer amount using `cout << "Enter transfer amount: $";`.
    - The user's input for the transfer amount is read from the standard input (`cin >> amount;`).
2. **Account Initialization**
    - Two instances of the `Account` class named `from_account` and `to_account` are created. These objects will be used to load and update account details for the sender and receiver, respectively.
3. **Loading Account Details**
    - The `load_from_file` method of the `Account` class is called on both `from_account` and `to_account`objects with `from_customer_number` and `to_customer_number` parameters, respectively. This method attempts to load the account details from text files associated with the specified customer numbers.
    - If `load_from_file` returns `true` for both accounts, it means the account details were successfully loaded from the files into the respective `from_account` and `to_account` objects.
4. **Transfer Validation and Execution**
    - If both account details were successfully loaded (`load_from_file` returned `true` for both accounts), the method checks the validity of the transfer amount:
        - It verifies if `amount` is greater than zero (`amount > 0`).
        - It checks if `amount` is less than or equal to the current balance of the sender (`amount <= from_account.Balance`).
    - If both conditions are met, the transfer amount (`amount`) is subtracted from the sender's balance (`from_account.Balance`) and added to the receiver's balance (`to_account.Balance`).
    - The updated account details for both accounts are then saved back to their respective files using the `save_to_file` method of the `Account` class (`from_account.save_to_file()` and `to_account.save_to_file()`).
    - A success message is displayed, indicating the successful transfer, along with details such as the transferred amount and the names of the sender and receiver.
5. **Error Handling**

- If either `load_from_file` returns `false`, it indicates that the account details could not be loaded, typically because the file corresponding to the customer number does not exist or could not be opened.
- In such cases, specific error messages are displayed:
  - If the sender's account details could not be loaded (`!from_account.load_from_file(from_customer_number)`), a message indicates that the account for the sender could not be found.
  - If the receiver's account details could not be loaded (`!to_account.load_from_file(to_customer_number)`), a message indicates that the account for the receiver could not be found.

6. **Invalid Transfer Amount**

- If the transfer amount is not greater than zero (`amount <= 0`), the method outputs a message indicating an invalid amount and prompts the user to enter a positive value.

Here is the complete method code for reference:

```
void transfer(int from_customer_number, int to_customer_number) {
  int amount;
  cout << "Enter transfer amount: $";
  cin >> amount;
  Account from_account, to_account;
  if (from_account.load_from_file(from_customer_number) &&
     to_account.load_from_file(to_customer_number)) {
   if (amount > 0 && amount <= from_account.Balance) {
     from_account.Balance -= amount;
     to_account.Balance += amount;
     from_account.save_to_file();
     to_account.save_to_file();
     cout << "Transfer successful! $" << amount
          << " transferred from customer " << from_account.Name
          << " to customer " << to_account.Name << endl;
   } else if (amount > from_account.Balance) {
     cout << "Insufficient balance.\n";
```

```
    } else {
      cout << "Invalid amount. Please enter a positive value.\n";
    }
  } else {
    if (!from_account.load_from_file(from_customer_number)) {
      cout << "Cannot find account for the transfer out customer number.\n";
    }
    if (!to_account.load_from_file(to_customer_number)) {
      cout << "Cannot find account for the transfer in customer number.\n";
    }
  }
}
```

Ending the Session

The `cancel` method ends the session with a goodbye message.

```
void cancel() {
  cout << "Session ended! Goodbye~\n";
}
```

Main Function

The main function handles user authentication and menu navigation.

Authentication

The user is prompted to enter a four-digit PIN. If the entered PIN matches the predefined password (`1234`), access is granted.

Here's the breakdown of the `main` function, which serves as the entry point for the banking system simulation:

1. **Bank Object Initialization**

- An instance of the `Bank` class named `bank` is created. This object provides the interface and functionality to interact with accounts and perform banking operations.

2. **Pin Validation Loop**

   - The function starts by displaying a welcome message and prompting the user to enter their four-digit PIN using `cout << "Enter your four digit pin: ";`.
   - Inside a `while` loop, the program continuously reads the user's input for the PIN (`cin >> pin;`).
   - If the entered `pin` matches the predefined `password` (`password = 1234`), the loop breaks, allowing the user to proceed to the main menu of banking operations.
   - If the entered `pin` does not match the `password`, an error message is displayed (`cout << "Access fail!\n\n";`) and the user is prompted again to enter the PIN.

3. **Banking Operations Menu**

   - After successful PIN validation, a `while` loop is used to continuously display the banking operations menu and handle user input for selecting operations.
   - The menu is displayed using the `menu` method of the `Bank` class, which prints options such as balance inquiry, deposit, withdrawal, account creation, transfer, and session cancellation.

4. **Handling User Input**

   - Inside the `while` loop, the program reads the user's input (`cin >> instruct;`) to determine which banking operation to perform based on the selected option (`instruct`).
   - A `switch` statement is used to execute the corresponding method from the `Bank` class based on the user's input:
     - `case 1`: Balance inquiry (`bank.balance(customer_number);`).
     - `case 2`: Deposit operation (`bank.deposit(customer_number);`).
     - `case 3`: Withdrawal operation (`bank.withdraw(customer_number);`).
     - `case 4`: Account creation (`bank.create_account();`).
     - `case 5`: Transfer operation (`bank.transfer(from_customer_number, to_customer_number);`).
     - `case 6`: Session cancellation (`bank.cancel(); return 0;`).

5. **Error Handling**

   - The default case of the `switch` statement handles invalid options by displaying an error message (`cout << "Invalid option. Please try again. \n";`) and prompting the user to select a valid option.

6. **End of Program**

- The program ends when the user chooses to cancel the session (`case 6`). The `return 0;` statement terminates the `main` function and indicates successful completion of the program.

Here is the complete `main` function code for reference:

```
int main() {
  Bank bank;
  int pin, password = 1234;
  cout << "Welcome to your bank account ~\n"
       << "Enter your four digit pin: ";
  while (cin >> pin) {
    if (pin == password)
      break;
    else
      cout << "Access fail!\n\n";
    cout << "Enter your four digit pin: ";
  }
```

Menu Loop

The program displays a menu of options and processes the user's input in a loop until the session is ended. Depending on the user's choice, the corresponding `Bank` method is called.

Here's the breakdown of the `Menu loop` function

1. **Infinite Loop (`while (true) { ... }`):**
   - This loop ensures that the program keeps running until explicitly terminated. Inside this loop, the user can perform multiple banking operations repeatedly.
2. **Displaying the Menu (`bank.menu();`):**
   - The `menu()` method of the `Bank` class is called to display a list of available banking operations. This typically includes options like checking balance, depositing money, withdrawing money, creating a new account, transferring money between accounts, and canceling the session.
3. **Reading User Input (`cin >> instruct;`):**

- The program waits for the user to input their choice of operation (`instruct`). This input is read from the standard input stream (`cin`).

4. **Switch Statement (`switch (instruct) { ... }`):**
   - Based on the value of `instruct`, which corresponds to the user's chosen operation, the program executes different blocks of code:
     - **Case 1:** Balance Inquiry
       - Prompts the user to enter their customer number (`cin >> customer_number;`) and then calls the `balance(customer_number)` method of the `Bank` class to display the account balance.
     - **Case 2:** Deposit
       - Prompts the user to enter their customer number and then calls the `deposit(customer_number)` method of the `Bank` class to deposit money into the specified account.
     - **Case 3:** Withdrawal
       - Prompts the user to enter their customer number and then calls the `withdraw(customer_number)` method of the `Bank` class to withdraw money from the specified account.
     - **Case 4:** Create Account
       - Calls the `create_account()` method of the `Bank` class to create a new bank account by prompting the user to enter their name and customer number.
     - **Case 5:** Transfer
       - Prompts the user to enter the customer numbers of the accounts involved in the transfer (`customer_number` for the sender and `to_customer_number` for the recipient) and then calls the `transfer(customer_number, to_customer_number)` method of the `Bank` class to transfer money between accounts.
     - **Case 6:** Cancel Session
       - Calls the `cancel()` method of the `Bank` class to terminate the session and exits the program with `return 0;`.
5. **Default Case: Invalid Option**
   - If the user inputs a value that doesn't match any of the defined cases (1 through 6), the `default` case of the switch statement executes. It displays an error message (`cout << "Invalid option. Please try again.\n";`) and prompts the user to input a valid option.
6. **End of Program (`return 0;`):**
   - Once the user chooses to cancel the session (`case 6`), the `return 0;` statement terminates the main function and indicates successful completion of the program.

Here is the complete `Menu loop` function code for reference:

```
while (true) {

 bank.menu();
```

```
int instruct, customer_number, to_customer_number;
cin >> instruct;
switch (instruct) {
case 1:
  cout << "Enter your customer number: ";
  cin >> customer_number;
  bank.balance(customer_number);
  break;
case 2:
  cout << "Enter your customer number: ";
  cin >> customer_number;
  bank.deposit(customer_number);
  break;
case 3:
  cout << "Enter your customer number: ";
  cin >> customer_number;
  bank.withdraw(customer_number);
  break;
case 4:
  bank.create_account();
  break;
case 5:
  cout << "Enter the customer number to transfer from: ";
  cin >> customer_number;
  cout << "Enter the customer number to transfer to: ";
  cin >> to_customer_number;
  bank.transfer(customer_number, to_customer_number);
  break;
```

```
    case 6:
      bank.cancel();
      return 0;
    default:
      cout << "Invalid option. Please try again.\n";
    }
  }
  return 0;
}
```

Conclusion

This C++ program demonstrates a basic banking system that handles account management through file-based persistence. Users can interact with the system to perform various banking operations, making it a practical example of file I/O, class design, and user interaction in C++.

Complete Source Code

```
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class Account {
public:
  string Name;
  int Customer_number;
  int Balance;
```

```
Account(string name, int customer_number, int balance)
    : Name(name), Customer_number(customer_number), Balance(balance) {}

Account() : Name(""), Customer_numbr(0), Balance(0) {}

void save_to_file() {
  string filename = to_string(Customer_number) + ".txt";
  ofstream outfile(filename);
  if (outfile.is_open()) {
    outfile << "Name: " << Name << "\n";
    outfile << "Customer Number: " << Customer_number << "\n";
    outfile << "Balance: " << Balance << "\n";
    outfile.close();
  } else {
    cout << "Failed to create file " << filename << endl;
  }
}

bool load_from_file(int customer_number) {
  string filename = to_string(customer_number) + ".txt";
  ifstream infile(filename);
  if (infile.is_open()) {
    string line;
    getline(infile, line);
```

```
      Name = line.substr(line.find(": ") + 2);
      getline(infile, line);
      Customer_number = stoi(line.substr(line.find(": ") + 2));
      getline(infile, line);
      Balance = stoi(line.substr(line.find(": ") + 2));
      infile.close();
      return true;
    } else {
      return false;
    }
  }
};

class Bank {
public:
  Bank() {}

  void menu() {
    cout << "\n**** menu ****\n"
         << "----1: balance\n"
         << "----2: deposit\n"
         << "----3: withdraw\n"
         << "----4: create account\n"
         << "----5: transfer\n"
```

```
        << "----6: cancel\n";
}

void balance(int customer_number) {
  Account account;
  if (account.load_from_file(customer_number)) {
    cout << "Customer " << account.Name << "'s current balance is: $"
         << account.Balance << endl;
  } else {
    cout << "Cannot find account for the customer number.\n";
  }
}

void deposit(int customer_number) {
  int amount;
  cout << "Enter deposit amount: $";
  cin >> amount;
  Account account;
  if (account.load_from_file(customer_number)) {
    if (amount > 0) {
      account.Balance += amount;
      account.save_to_file();
      cout << "Deposit successful!\n";
    } else {
```

```
      cout << "Invalid amount. Please enter a positive value.\n";
    }
  } else {
    cout << "Cannot find account for the customer number.\n";
  }
}

void withdraw(int customer_number) {
  int amount;
  cout << "Enter withdrawal amount: $";
  cin >> amount;
  Account account;
  if (account.load_from_file(customer_number)) {
    if (amount > 0 && amount <= account.Balance) {
      account.Balance -= amount;
      account.save_to_file();
      cout << "Withdrawal successful!\n";
    } else if (amount > account.Balance) {
      cout << "Insufficient balance.\n";
    } else {
      cout << "Invalid amount. Please enter a positive value.\n";
    }
  } else {
    cout << "Cannot find account for the customer number.\n";
```

```
  }
}

void create_account() {
  string name;
  int customer_number;
  cout << "Enter your name: ";
  cin >> name;
  cout << "Enter your customer number: ";
  cin >> customer_number;
  Account new_account(name, customer_number, 0);
  new_account.save_to_file();
  cout << "Account created successfully!\n";
}

void transfer(int from_customer_number, int to_customer_number) {
  int amount;
  cout << "Enter transfer amount: $";
  cin >> amount;
  Account from_account, to_account;
  if (from_account.load_from_file(from_customer_number) &&
      to_account.load_from_file(to_customer_number)) {
    if (amount > 0 && amount <= from_account.Balance) {
      from_account.Balance -= amount;
```

```
        to_account.Balance += amount;
        from_account.save_to_file();
        to_account.save_to_file();
        cout << "Transfer successful! $" << amount
             << " transferred from customer " << from_account.Name
             << " to customer " << to_account.Name << endl;
      } else if (amount > from_account.Balance) {
        cout << "Insufficient balance.\n";
      } else {
        cout << "Invalid amount. Please enter a positive value.\n";
      }
    } else {
      if (!from_account.load_from_file(from_customer_number)) {
        cout << "Cannot find account for the transfer out customer number.\n";
      }
      if (!to_account.load_from_file(to_customer_number)) {
        cout << "Cannot find account for the transfer in customer number.\n";
      }
    }
  }

  void cancel() { cout << "Session ended! Goodbye~\n"; }
};
```

```
int main() {
  Bank bank;
  int pin, password = 1234;
  cout << "Welcome to your bank account ~\n"
       << "Enter your four digit pin: ";
  while (cin >> pin) {
    if (pin == password)
      break;
    else
      cout << "Access fail!\n\n";
    cout << "Enter your four digit pin: ";
  }

  while (true) {
    bank.menu();
    int instruct, customer_number, to_customer_number;
    cin >> instruct;
    switch (instruct) {
    case 1:
      cout << "Enter your customer number: ";
      cin >> customer_number;
      bank.balance(customer_number);
      break;
    case 2:
```

```
  cout << "Enter your customer number: ";
  cin >> customer_number;
  bank.deposit(customer_number);
  break;
case 3:
  cout << "Enter your customer number: ";
  cin >> customer_number;
  bank.withdraw(customer_number);
  break;
case 4:
  bank.create_account();
  break;
case 5:
  cout << "Enter the customer number to transfer from: ";
  cin >> customer_number;
  cout << "Enter the customer number to transfer to: ";
  cin >> to_customer_number;
  bank.transfer(customer_number, to_customer_number);
  break;
case 6:
  bank.cancel();
  return 0;
default:
  cout << "Invalid option. Please try again.\n";
```

```
    }
  }

  return 0;
}
```