MyBankSys

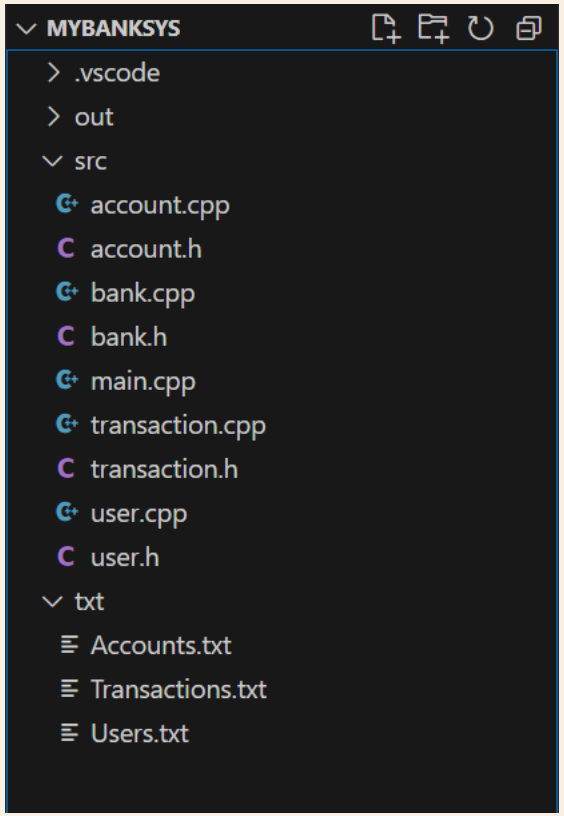
By: Ahmad AlKafri

What is MyBankSys

MyBankSys is a simple banking system, that:

- Designed based on OOP foundation,
- Excels in executing CRUD operations.
- Ensure data persistence by using file I/O.
- Heap and stack allocations optimize memory usage.

FILES HIERARCHY



CRUD Operations with IO (input/output)

MyBankSys ensures data reliability through essential CRUD operations. Functions like **loadUsers, saveUsers, loadAccounts, saveAccounts, loadTransactions, and saveTransactions,** read from and write to files.

EXAMPLES:

```
// Sample for: CRUD Operations with IO (input/output)

// User class
void loadUsers();
void saveUsers() const;

// Bank class
void loadAccounts(std::ifstream& inputFile);
void saveAccounts(std::ofstream& outputFile) const;

// Transaction class
void loadTransactions(std::ifstream& inputFile);
void saveTransactions(std::ofstream& outputFile) const;
```

CODE SAMPLE (FROM BANK CLASS)

```
123  void Bank::withdraw(int accountNumber, double amount) {
124      auto it = std::find_if(accounts.begin(), accounts.end(),
125                             [accountNumber](const Account* account) {
126                                 return account->getAccountNumber() == accountNumber;
127                             });
128
129      if (it != accounts.end()) {
130          (*it)->withdraw(amount);
131          std::cout << "Withdrawal successful. New balance: " << (*it)->getBalance() << std::endl;
132          saveAccounts(); // Save accounts after withdrawal
133          updateAccountBalances(transactionManager); // Update account balances after each transaction
134      } else {
135          std::cout << "Account not found." << std::endl;
136      }
137  }
```

HEAP AND STACK ALLOCATION

- In MyBankSys, both heap and stack allocations employed.
- The Account class showcases heap allocation with the account holder variable, allowing dynamic memory management.
- Stack allocation is employed for local variables within functions, ensuring efficient memory usage for short-lived data

STACK SAMPLE (From Bank class)

```
139  void Bank::updateAccountBalances(const Transaction& transactionManager) {
140      const auto& transactions = transactionManager.getTransactions();
141
142      // Keep track of the last processed transaction index
143      static size_t lastProcessedIndex = 0;
144
145      // Process only new transactions
146      for (size_t i = lastProcessedIndex; i < transactions.size(); ++i) {
147          const auto& transaction = transactions[i];
```

HEAP SAMPLE (From Account class)

```
 6  class Account {
 7  private:
 8      int accountNumber;
 9      std::string* accountHolder; // Using heap allocation for dynamic string data
10      double balance;
11
```

OBJECT-ORIENTED PROGRAMMING (OOP)

- In MyBankSys, Object-Oriented Programming (OOP) is the cornerstone of the design.
- In my implementation, classes such as User, Bank, Transaction, and Account encapsulate related data and functionality, mirroring real-world entities in a banking system.
- For example, the Account class:
  - Holds account data.
  - Manages deposit and withdrawal operations.
- This modular approach enhances code organization and readability.

SAMPLE FROM THE CODE

```
// User class encapsulating user-related data and functionality
class User {
    // ...
};

// Bank class representing a banking system, encapsulating related data and operations
class Bank {
    // ...
};

// Transaction class managing transaction-related functionality
class Transaction {
    // ...
};

// Account class encapsulating account-specific data and operations
class Account {
    // ...
};
```

SAMPLE FROM THE CODE – ACCOUNT CLASS HEADER FILE

```cpp
#ifndef ACCOUNT_H
#define ACCOUNT_H

#include <string>

class Account {
private:
    int accountNumber;
    std::string* accountHolder; // Using heap allocation for dynamic string data
    double balance;

public:
    Account(int number, const std::string& holder, double initialBalance);
    ~Account(); // Destructor to release heap-allocated memory

    int getAccountNumber() const;
    std::string getAccountHolder() const;
    double getBalance() const;

    void setAccountHolder(const std::string& newHolder);
    void deposit(double amount);
    void withdraw(double amount);
    void displayDetails() const;
};

#endif
```

TRANSACTION DEMONSTRATION

In this brief demonstration, we'll focus on a deposit operation, showcasing the coordination between the main function and relevant classes.

// Main function demonstrating a deposit transaction (main.cpp):

```
176             case 8:
177                 clearScreen();
178                 // Make a deposit transaction into an account
179 
180                 // User chooses to make a deposit
181                 int depositAccountNumber;
182                 double depositAmount;
183                 std::cout << "Enter the account number for the deposit: ";
184                 std::cin >> depositAccountNumber;
185                 std::cout << "Enter the deposit amount: ";
186                 std::cin >> depositAmount;
187 
188                 // Initiating the deposit through the Transaction class
189                 transactionManager.deposit(depositAccountNumber, depositAmount);
190 
191                 // Updating account balances in the Bank class
192                 bank.updateAccountBalances(transactionManager);  // Update account balances
193 
194                 // Saving the transaction data
195                 saveTransactionData(transactionManager);  // Save transaction data after a deposit
196                 std::cout << "****************************************" << std::endl << std::endl;
197                 break;
```

// Transaction class handling deposit operation (Transaction class “transaction.cpp”)

```cpp
void Transaction::deposit(int accountNumber, double amount) {
    TransactionInfo depositTransaction{accountNumber, "DEPOSIT", amount};
    transactions.push_back(depositTransaction);
}
```

// Bank class updating account balances after a transaction (Bank class “bank.cpp”)

```cpp
void Bank::updateAccountBalances(const Transaction& transactionManager) {
    const auto& transactions = transactionManager.getTransactions();

    // Keep track of the last processed transaction index
    static size_t lastProcessedIndex = 0;

    // Process only new transactions
    for (size_t i = lastProcessedIndex; i < transactions.size(); ++i) {
        const auto& transaction = transactions[i];

        auto accountIt = std::find_if(accounts.begin(), accounts.end(),
                                      [&transaction](const Account* account) {
                                          return account->getAccountNumber() == transaction.accountNumber;
                                      });

        if (accountIt != accounts.end()) {
            if (transaction.type == "DEPOSIT") {
                (*accountIt)->deposit(transaction.amount);
            } else if (transaction.type == "WITHDRAW") {
                (*accountIt)->withdraw(transaction.amount);
            }
        }
    }
```

HIGHLIGHTS

1. ADVANTAGES/DISADVANTAGES OF C++ OVER JAVA

ADVANTAGES:

Performance: C++ is often considered more efficient in terms of performance compared to Java, as it allows for more control over memory management and lower-level optimizations.

Direct Memory Access: C++ provides direct access to memory, allowing for more efficient data manipulation.

DISADVANTAGES:

Memory Management: C++ requires manual memory management, which can lead to potential issues such as memory leaks or segmentation faults if not handled carefully.

Learning Curve: C++ has a steeper learning curve compared to Java due to its lower-level features and concepts like pointers.

2. ADVANTAGES/DISADVANTAGES OF USING STANDARD LIBRARY ONLY

ADVANTAGES:

Portability: Code written using standard libraries is more likely to be portable across different platforms.

Consistency: Standard libraries provide a consistent set of functionalities, reducing the need for external dependencies and ensuring a common programming interface.

DISADVANTAGES:

Limited Specialized Features: Standard libraries may lack certain specialized features that are available in external libraries, limiting the capabilities of the program.

Customization Constraints: Using only standard libraries may limit the level of customization and control over certain aspects of the code.

3. OBJECT-ORIENTED PROGRAMMING (OOP) IMPROVEMENTS

EXAMPLE FROM THE CODE:

The use of OOP principles in the code is evident through the creation of classes like:

User, Transaction, and Bank.

These classes encapsulate related functionalities within these classes, promoting better organization and modularity.

4. DIFFERENCE BETWEEN STACK AND HEAP DATA STORAGE:

STACK:

PROS:

- Faster Allocation/Deallocation: Stack memory is managed automatically, and allocation/deallocation is faster compared to heap.
- Memory Locality: Accessing data on the stack is generally faster due to its memory locality.

CONS:

- Limited Size: Stack has a limited size, and exceeding it can lead to a stack overflow.

HEAP:

PROS:

- Dynamic Memory: Heap allows dynamic allocation of memory, suitable for situations where the size of data is not known at compile time.
- Larger Memory Space: Heap provides a larger memory space compared to the stack.

CONS:

- Slower Allocation/Deallocation: Heap memory requires manual management and is generally slower for allocation/deallocation.
- Fragmentation: Over time, heap memory can become fragmented, impacting performance.

5. MOST CHALLENGING ASPECT IN THE PROJECT

The most challenging aspect in the project might be handling memory management, especially with the use of dynamic memory allocation for objects like Account in the Bank class. Ensuring proper allocation, deallocation, and avoiding memory leaks can be challenging in a C++ project.

6. MOST VALUABLE LESSON LEARNED FROM THE COURSE

One of the most valuable lessons learned from the course is the importance of careful memory management, especially in a language like C++. Understanding concepts such as:

dynamic memory allocation, constructors, destructors, and smart pointers

is crucial for writing robust and efficient C++ code.

SUMMERY

- **CRUD operations & File I/O:** In the program I implemented CRUD operations, also the program ensures data persistence through file input/output operations.

- **Memory Management:** By incorporating both heap and stack allocations, MyBankSys optimizes resource usage.

- **OOP Principles:** MyBankSys designed based on Object-Oriented Programming principles, enhancing modularity and code organization.

THANK YOU