**REPORT**

1. **Object-Oriented Approach**:
   * The application uses classes to encapsulate related functionality. The *UserAccount* structure defines the properties of a user account, such as the account number, username, password, and balance. The *BankingSystem* class manages the collection of user accounts and the operations that can be performed on them.
   * Encapsulation helps in maintaining the integrity of user data by restricting direct access to the account information, which is essential for security in a banking system.
2. **Data Structure Selection**:
   * An *unordered\_map* is chosen to store user accounts, where the username serves as the key and the corresponding *UserAccount* struct serves as the value. This allows for efficient lookups (average O(1) time complexity) when checking for existing usernames during account creation and login processes.
3. **Functionality Implementation**:
   * The core functionalities of the banking system include creating accounts, logging in, depositing and withdrawing funds, and checking account balances. Each of these functionalities is encapsulated within methods of the *BankingSystem* class, promoting modularity and code reusability.
   * Methods are designed to validate user inputs (e.g., checking if a username already exists, ensuring deposit and withdrawal amounts are positive) to enhance robustness.
4. **Menu-Driven Interface**:
   * A simple console-based menu interface allows users to interact with the system. This design choice makes the application user-friendly and straightforward, catering to users who may not be familiar with graphical user interfaces.
   * The menu is implemented using a do-while loop that continuously presents options to the user until they choose to exit the application.
5. **User Feedback**:
   * The application provides immediate feedback to users regarding their actions (e.g., account creation success, invalid login attempts, successful transactions). This feedback is crucial for a good user experience, helping users understand the state of their interactions with the system.
6. **Error Handling**:
   * Basic error handling is incorporated to manage invalid inputs and operations. For example, if a user tries to withdraw more money than available in their account, the system responds with an appropriate message, preventing negative balances.

**Implementation Overview**

* The code begins by including necessary headers for input-output operations and data structures.
* The *UserAccount* struct is defined to hold the necessary attributes for each user account.
* The *BankingSystem* class encapsulates all functionalities related to account management. It contains methods for creating accounts, logging in, depositing and withdrawing money, and checking balances.
* The *isUsernameExists* method checks for the existence of a username in the account map.
* The login method verifies user credentials and returns a pointer to the logged-in user's *UserAccount*.
* The menu method orchestrates the user interface, allowing users to select options and perform corresponding actions.
* The main function instantiates the *BankingSystem* class and invokes the menu to start the application.