Banking System with Dynamic Memory Using Structs

Assignment Overview:

In this assignment, you will create a small banking system using only **structs** and dynamic memory management (**new** and **delete**). The goal is to implement a program that allows users to manage multiple bank accounts through operations such as creating accounts, depositing and withdrawing money, checking balances, and closing accounts. Each bank account will be represented by a **struct**, and the accounts will be stored dynamically in memory using a **std::vector** of pointers to **Account** structs.

Requirements:

1. **Define the Account Struct:**
   - Create a **struct** named **Account** with the following fields:
     - **std::string accountHolder**: The name of the account holder.
     - **int accountNumber**: A unique account number.
     - **double balance**: The current balance in the account.
2. **Implement Banking Functions:**
   - Use a **std::vector** to store **pointers** to dynamically allocated **Account** structs.
   - Implement the following functions:
     - **Create New Account:** Dynamically allocate memory for a new **Account** struct, prompt the user for the account holder's name and initial balance, assign a unique account number, and add it to the vector.
     - **Deposit Money:** Prompt the user for the account number and amount to deposit, locate the account, and update its balance. Ensure the amount to deposit isn't negative.
     - **Withdraw Money:** Prompt the user for the account number and amount to withdraw, ensure the withdrawal amount does not exceed the current balance, and update the account if the withdrawal is valid.
     - **Check Balance:** Prompt the user for the account number and display the account's current balance.
     - **Close Account:** Prompt the user for the account number, locate the account, delete it from memory, and remove the pointer from the vector.
     - **List All Accounts:** Display all accounts with their details, including account number, holder name, and balance.
3. **User Interaction:**
   - Create a menu-driven interface that allows the user to select the desired operation (7 operations in total, including "Exit"). The program should continue running until the user chooses to exit.
   - Should check for valid input with numbers only (don't need to check against non-numeric cases).
4. **Memory Management:**
   - Ensure proper memory management by using **new** for dynamic allocation and **delete** to free memory when closing accounts or exiting the program.

Submission Guidelines:

- Submit your code files in a zip archive containing **main.cpp**, **bank.cpp**, and **bank.h** by the due date to canvas. Other than the source code files, no specific naming of folders are necessary. No Makefile.
- Include comments in your code to explain the functionality of different sections.
- Test your program to ensure all functionalities work correctly.

Rubric (Total: 50 Points)

Criteria	Points	Description
Struct Definition	10	**Account** struct is correctly defined with all required fields.
Functionality Implementation	20/20	**Full functionality implemented**: All required functions (create account, deposit, withdraw, check balance, close account, list accounts) are correctly implemented and work as intended. Vector contains pointers to Account structs.
	10/20	**Partial functionality implemented**: Most functions are implemented, but one or two functions have minor issues or do not work as intended. The user experience may be affected by minor bugs, but the core functionalities are present.
	5/20	**Minimal functionality implemented**: Only a couple of functions are implemented, and they may not work as intended. The program does not provide a complete banking experience, or some user interactions may lead to confusion or errors.
	0/20	**No functionality implemented**: The program does not implement any of the required functions, or it fails to compile. There is no evidence of attempting to meet the assignment requirements.
User Interface	5	The menu-driven interface is user-friendly, allowing easy navigation through options.
Memory Management	5	Proper use of dynamic memory management (using **new** and **delete** appropriately).
Code Quality	5	Code is well-organized, properly indented, and includes comments explaining the logic. Comments above each function prototype in the .h file are detailed and clearly explain exactly what each function does. Implementation adheres to 3-file format
Testing	5	Code has been tested for correctness; edge cases and error handling are demonstrated.