Account Manager Using Polymorphism

50 points - Due Friday, December 8, 11:59 PM

Assignment

The goal of this project is to build a simple account manager, similar to what you might see in an online banking website. The account manager should be able to handle an unlimited number of accounts, with three different kinds of accounts: a checking account, a savings account, and a credit card account.

The program should repeatedly print out the different accounts (their names and balances), and then offer the user a choice of four different actions:

- **Create** a new account of a given name and type. The initial balance should always be zero.
- **Deposit** money into an account.
- **Withdraw** money from an account.
- **Quit** the program.

A Few Requirements:

- **Saving/Loading:** Each time you exit the program, it should save all the account names and balances in a text file. Each time you run the program, it should (if possible) load this information and recreate all the account objects with their proper balances. If it's not possible to load this info, the program should still *work*, but without any accounts. The file should be saved in the current directory (don't specify a path).
- **Counting the number of runs:** The program should keep track of the number of times it has been run (you can save this information into the file). This should be printed each time the program is run.
- **Computing Interest:** Every **4th** time the program is run, it should update the account balances with the appropriate amount of interest. Each account object should have a virtual method to compute/handle its own interest (as described below); every 4th time, this method should be called on each object.

Implementation:

This program must be implemented using polymorphism. Create an Account base class, which could (but doesn't have to) look something like this:

```
Account
{
   public:
      Account();
      virtual bool deposit( float howMuch ) = 0;
      virtual bool withdraw( float howMuch ) = 0;
      virtual void computeInterest();
   protected:
      float balance;
};
```

The other account classes derive from this one. You can refine this as appropriate, but like the last project, the main loop should deal with **Account*** pointers and leave as many details as possible to the derived classes.

Account Types/Classes:

Here's how the different account types work. **None of these balances should be allowed to go negative.**

Checking Account:

Each checking account has a maximum daily withdrawal limit of **$200**. (Pretend that each run of the program is a "day"; so whenever you restart the program, this limit is reset.)

- Withdrawal: If there is enough money in the account to cover the withdrawal, and the withdrawal doesn't go over the daily withdrawal limit, the withdrawal is allowed and the amount is subtracted from the balance.
- Deposit: adds the deposit amount to the balance.

Savings Account:

Whenever interest is computed, **1.5%** of the balance is added.

- Withdrawal: If there is enough money in the account to cover the withdrawal, the withdrawal is allowed and the amount is subtracted from the balance.
- Deposit: adds the deposit amount to the balance.

Credit Card Account:

The credit card works differently than the other two. A withdrawal is a *charge*, which *raises* the balance. A deposit is a *payment*, which *lowers* the balance. The credit card has a maximum balance of **$5000.** Whenever interest is computed, **18.99%** (ouch!) of the current balance is added (and this can push the balance over the $5000 limit).

- Withdrawal (charge): as long as the charge doesn't push the balance over $5000, it is allowed. Otherwise it's denied.
- Deposit (payment): subtracts the deposit amount from the balance. Note that the balance can't go negative.

Specific Notes:

- When printing out currency values, we normally use 2 digits after the decimal point. You can get this sort of output by doing an `#include <iomanip>` and using `fixed` and `setprecision` iostream manipulators:

```
cout << setprecision(2) << fixed << account->getBalance() << endl;
```

- When doing math with floating point values, there's always a possibility that you'll get answers slightly different than what you expected. (The computer might calculate 0.9999999 when you expect 1, for example.) This is normal. If you find that these tiny errors are causing your numbers to be off by a cent or two, don't worry about it.

- Since you need to allow an unlimited number of accounts, a `vector<Account*>` might be a good choice for storing them. (Not a `vector<Account>` though... why would this be?)

- Once again, there's a considerable amount of flexibility in how you implement this project. Do your best to make good use of classes, etc.

- The part about saving/loading your program's saved state doesn't need to be complicated. Use `ifstream` and `ofstream`, and it'll be just as simple as using `cin` and `cout`.

What to turn in: a directory containing...

- Your code, of course!
- The text file with the saved state of your program, so when I run the program it will automatically load its last saved state.
- A README file containing any relevant info, such as your name and compilation instructions (include the exact command you used to compile your code). Again, please include in the README the approximate amount of time you spent on this assignment. Also, the better you document your program/design in the README, the easier it will be for me to see what you were doing. This is always a good thing. :-)

Extra Credit

None this time. Sorry.

Non-specific Notes

- Start early; ask questions.
- Refer back to the syllabus for hints about writing good C++ programs (or programs in any language): only work on small pieces of code at a time, and test them well before moving on. Compile lots, and fix the warnings and errors. Write lots of comments.
- **Be sure your code compiles on the CS department's Linux machines before turning it in**. Even if you write it elsewhere (on a different OS, a different IDE, etc), give it a quick compile and test on a lab machine before you turn it in. Remember, if your code does not compile, it will get (at most) 50% credit.
- You must submit your code using the department's electronic submit procedure. A link to instructions for doing this is on the class website. Our course number (for the purposes of the submit program) is **c109sb**.
- Just a reminder: the CLAS academic (dis)honesty policies will be enforced. You're allowed to discuss the assignment with others, but not to collaborate or share code. Nor are you allowed to find or use code off the Internet.