Rec03

Intro to classes

This recitation will follow a “tutorial style”. We will break the overall task down into several parts. Code each one up, and please use comments to indicate which part of your program corresponds to which task number. Do not delete or comment out any code.

Note that this recitation may introduce some syntax or library methods that were not shown in lecture. Hopefully we have explained it clearly here, but of course, please ask questions if anything is unclear.

The topic for this recitation is bank accounts. No, I can’t think why anyone would want to write such code, but I have had students say that would be more “interesting” than warriors or dragons or … Go figure.

1. Define a *struct* that can hold information about accounts
    a. First:
        - Each instance will hold the name for the account and an account number.
        - Create an input file that has at least three accounts
        - Read a file of account information, filling a vector of account objects.
            - define local variables corresponding to each data item
            - Loop through the file, reading into these variables
            - Each time through the loop, define a local instance of the account struct, assign the values that were read to the fields of the struct and push back the object onto your vector.
            - Close the file
            - Display all of the objects.

    b. Clear the vector, reopen the file (using the same stream variable, i.e. call the open method), and <u>repeat the above</u>, except using *curly braced initializers* to initialize the instance. E.g. if we had a struct Thing with two fields, we could define and initialize an instance with:
        - `Thing thingOne{valueForFirstField, valueForSecondField};`
            - A user of Eclipse reported needing to specify c++14 in their build.

2. Define a *class* to hold accounts
    a. Redo the above steps from task 1 that you did with the struct version.
        - Use the *same* stream variable!
        - Obviously, use the class's constructor to initialize the fields
        - Write getters to access the fields in the accounts when printing.

    b. Add an output operator for your class
        - Do not make it a friend!
        - Repeat the print loop using this output operator

    c. *Now* make the output operator a friend, by adding a friend *prototype* to your account class definition.
        - Comment out the line where you are using getters. (Yes, I know we said you shouldn't comment out any code…)
        - Add a line in the output operator that prints the fields "directly", i.e. without using the getters.

    d. Redo your input but pass *temporary* instances to the push_back method.
        - This means that you will *not* define a variable to hold the account, but instead the vector push_back method will be passed a temporary object defined as the argument to push_back.
            - E.g. If I had a class Foo whose constructor took three ints, and a vector of Foo's called vf, I could add an instance of Foo to the vector with: `vf.push_back(Foo(2,4,6));`

    e. Note that we keep making objects and then adding copies of those objects onto the vector (since that's what push_back does). Here we will use the vector's `emplace_back` method. Instead of passing account objects to emplace_back, we can just pass the arguments that we would pass to the account's constructor and emplace_back will initialize an instance in the vector!
        - Repeat the filling of the vector using `emplace_back` and again display the contents.

3. Add *transactions* to your world.
    a. A transaction will be implemented as a class and have
        - a field to indicate whether this is a deposit or a withdrawal.
        - a field to say how much is being deposited or withdrawn.
    b. The account class will keep track of transactions applied to it. It will need a vector to store this history of transactions.
    c. It will also need a “balance” field to indicate how much is in the account.
    d. How will we use this?
        - The account will have methods “deposit” and “withdrawal” which will
            - be passed the amount
            - will add an appropriate transaction object to the history
            - and modify the balance as needed.
        - The output operator for the account will need to change so that it can display (you format it) the history, i.e. the transactions.
    e. Test the above code out by reading in a file that has commands such as
        - Account moe 6
          Deposit 6 10
          Withdraw 6 100
        - Note that the number 6 in the deposit and withdraw commands refers to the account. You will have to locate the account in your collection of accounts.
        - Withdrawals should not be allowed to put the account in the red, so if there are insufficient funds in the account then generate an error message and go on.

4. Now make the transaction class private within the account class. What else do you have to change to make this work?
    a. One small hint. You will want to put the whole definition of the Transaction class’s output operator inside the Transaction class.

5. If you have time in lab, create a client class.
    a. Every account is “tied to” a client, i.e. the account should have a field that is a client. This would replace the field that just held the account’s name.
    b. The client *class* will not be defined inside the account class, but when an account is created, it’s client field will be initialized from the information about who the client is. (I.e. the account constructor will take in the fields that must be initialized within the client.)
    c. What fields and methods should go in the client class? We will leave that up to you. Obviously the client needs a name. Perhaps also a SSN (social security number)?
    d. Update the output operator for your account class to display the client information and test all operations.