Lab 2: Swat ATM (Machine (Machine))

Due: February 19th at 11:59pm

Overview

The goal of this lab is to continue your familiarization with the C++ programming with Classes, as well as preview some data structures. Your program will make use of classes, inheritance, and array maintenance. **This is an individual lab**, and you will have two weeks complete it.

Deliverables

Your submission should minimally include the following code and supplemental files:

- `Makefile`
- `README`
- `ATM.cpp`
- `bank.[cpp|h]`
- `account.[cpp|h]`
- `checking_account.[cpp|h]`

You may submit additional files, but only those files listed above will be considered during grading.

Submission

You will submit using `handin35`. Be sure to place all relevant files in `cs35/labs/02`.

The Swat ATM

The Swat ATM consists of three main components: A Bank, a base bank Account, and a Checking Account. Finally, there is the ATM program, which I have provided for you. Your task is to program the remaining parts of the code to have a functional ATM (machine (machine)).

Note on terminology: I have capitalized terms that refer to an object/class in the ATM. For example, Account refers to the Account class, and Bank refers to the Bank class, and etc.

Bank

The Bank class, whose definition is found in `bank.h`, represents a basic data-structure to maintain Accounts. I refer you to the header file for more details of the specific components. Below, I describe the process of maintaining Accounts. Consider the class definition for the private member variables:

```
class Bank{

 private:
  int numAccounts; // Current number of accounts
  int maxAccounts; //size of the Accounts * array

  //An array of pointer to Accounts. E.g. The type of accounts[i] is
  // an Account * which can reference an instance of Account
  Account ** accounts;

//...

};
```

The comments should give you an indication for how these variables are used. Focusing on the `accounts` variable; the type definition and its use may be confusing at first. Here is a double pointer, but this is better viewed as an array of pointers to Account pointers; that is, in each cell of the `accounts` array there is a reference to an Account objects. This is, perhaps, best understood through an example of the `accounts` array usage:

```
  Accounts * acc = new Account("adam", 100) //name and account number
  accounts[i] = acc;
```

Recall the relationship between arrays and pointers, the first part of the double pointer we can think of referencing an array, which means we can assign a reference to an Account to a cell in the array. Once you start programming with the `accounts` array, it should become more clear.

The two other private member variables, `numAccunts` and `maxAccounts`, provide information about the current state of the `accounts` array. The first, `numAccounts`, indicates how many accounts are currently being stored by the Bank, and the latter, `maxAccounts`, indicates how many Accounts the bank can store maximally, i.e., the size of the `accounts` array.

The tricky, but not necessarily difficult, part is managing the `accounts` array as Accounts are added and removed. There are two main functions for managing accounts: `addAccount(Account * acc)` and `rmAccount(int idx)`. The `addAccount()` member function will add an account into the first open slot in the `accounts` array, and `rmAccount()` will remove the account at a given index. By far the more complicated of the two routines is `addAccount()` because it must find an open slot in the `accounts` array to place the Account. Initially, the `accounts` array will be empty; that is, all slots in the array will reference `NULL` or 0. So the first slot, index 0, is available. However, consider what happens

when there was once a full array, but now some Accounts have been removed. The `accounts` array may look like below, where `ACC` indicates an account in the slot of the array and `NULL` indicates an empty slot:

```
                0     1      2      3      4
              +-----+-----+------+------+-----+
accounts  ->  | Acc | ACC | NULL | NULL | ACC |
              +-----+-----+------+------+-----+
```

There are two open slots, one at index 2 and one at index 3. So upon inserting a new Account, you must iterate through the array to find the first open slot, at index 2, and place the account there. To do so, you will use a standard for loop. You can `break` the for loop once the insertion into a slot occurs.

Finally, there are numerous error conditions possible. For example, what happens if a user tries to remove an Account from a slot where there is no account? What if the Bank is full? And etc. **You must consider these error conditions in your code and report useful feedback to the user.**

For descriptions of other member functions of the Bank, you should refer to the header file, `bank.h`. You will place your definitions of the member functions in the source file, `bank.cpp`. I have programmed some of the definitions, while you must define the rest.

Account and CheckingAccount

You are also tasked with implementing two Account classes. The first is the base class, Account, and the second is derived from Account, a CheckingAccount. The details of the member functions and their roles are well defined in the header files, `account.h` and `checking_account.h`, and I refer you to the documentation and comments for more details.

As a high level overview, an Account has three main functions: `deposit()`, deposit some amount of money into the account; `withdraw()`, remove some amount of money from the account; and `transfer()`, move some amount of money from one account to another. Both Account and CheckingAccount will support these operations, but a CheckingAccount has a limitation on the number of withdraws and transfers. This is where you must consider inheritance and other principals in ensuring that an account holder doesn't exceed his/her check limits.

There are a number of situations where errors can occur with Accounts and CheckingAccounts that must be reported to the user by priting an error message. For example, what if the user tries to overdraw the account? Or what if the user is out of checks? And etc. **You must conisder these error conditions in your code and report useful feedback to the user**

In addition to the standard account functionality, there are some other functions you will need to define that may differ between the two accounts, such as `printInfo()`, which prints out the information of the account, and `toString()` which coverts the information to a string and returns that string. Focusing on `toString()`, you will accomplish this task using a feature of C++ not presented in class called string streams. Like `cout`, which is a iostream, a string stream functions in much the same way, but instead of printing to the terminal, it stores the information in a string. An example is provided on the following page.

```
#include <sstream>

 stringstream ss;
 string str;

 ss << "Hello" << " " << 5; //use string stream like cout
 ss << "World" << endl;
 str = ss.str(); // convert to string

 cout << str; //prints "Helo 5World"
```

For more details, I refer you to the header and source files for both Account and CheckingAccount. I have provided the entire class definition for Account, but you must fill it the remaining class definition for CheckingAccount. This means that your program will not compile without, at least, first defining the class and basic member functions. They need not be complete, just defined so that C++ will recognize them and type-check.

ATM

Finally, I have provided you with the `main()` portion of the program in a file `ATM.cpp`. When executed, the ATM will prompt the user for a set of commands. You can type "help" to list the commands or type "quit" to quit the ATM. The usage of the ATM is best described via some sample output. Additionally, as part of the code provided during `update35`, a working executable of the ATM program (with the accounts defined) so that you may get a sense of the goal of the assignment. Below is some sample output of ATM:

```
bash>./atm
Welcome to the CS-Swat ATM!
Type 'help' for help and 'quit' to exit
ATM> help
ATM Commands
         help : print this help message
         quit : quit the ATM
         print : print the accounts
         new : create a new bank account
         transfer : transfer money between accounts
         withdraw : withdraw money from an account
         deposit : deposit amount into acc1
         remove : remove an account from the bank

ATM> add
Error, unkown command
ATM> new
New Acccount, ok!

Select the type of account:
(1) Bank Account  (2) Checking Account
Selection: 1
Account Holder Name: adam
Account Created!
idx: 0  Name: adam       Acc#: 100       Bal: $0
ATM> deposit
Deposit, ok!

Account idx: 0
Depositting into Acc# 100
ammount: 1000.10
ATM> print
Current BankAccounts
idx: 0  Name: adam       Acc#: 100       Bal: $1000.1
```

Additionaly, as an example of same trace where the account is CheckingAccount rather than a standard Account, consider the output of print:

```
ATM> print
Current BankAccounts
idx: 0  Name: adam       Acc#: 100       Bal: $1000.1   Checks: 0/5
```

Testing

In addition to the ATM source files, I have provide three test files to help your development: `account_test.cpp`, `checking_account_test.cpp`, and `bank_test.cpp`. Each of these files test the core functionality of each of the classes. To compile the test files you will issue the following make comand

```
bash> make test
```

If you were to just type, `make`, you would compile the core ATM program, and not the test files. **Note, although test files are provided, you are responsible to fully testing the functionality of your program.**

Hints, Tips, and Tricks

- **Start Early and Seek Help Early**: This is a much more time consuming lab assignment as compared to the two previous. There are new challenges and programming paradigms that you may not be familiar with yet, so it's important to identify those issues so you can get the requisite assistance from the teaching staff.
- **Test as you Develop**: Do not program and program and program without testing along the way. It's really easy to make a mistake early in the development that can propagate in strange ways. Such errors can cause you to rewrite large swaths of code.
- **Incremental Progress is Progress**: Focus on solving small problems than solving the big problem all at once. For example, get the adding Accounts to the bank working first before worying about removing. Solving many small problems adds up to solving a lot of big problems.
- **Use Valgrind to test for Memory Violations and Leaks**: Try running your program through `valgrind`, both the ATM program and the test program to see if you have any memory violations or leaks. Memory violation errors can cause problems in strange ways, and fixing them can be challenging. Using `valgrind` will greatly assist in the process and help identify issues before they become big problems. Check out Prof. Tia Newhall's guide: `http://web.cs.swarthmore.edu/~newhall/unixhelp/purify.html`.
- **Use GDB to test logic**: GDB, the GNU debugger, will iterate through your program line by line during execution, and is one of the best ways to test for logic errors. This why, in the `Makefile`, we compile your code with the `-g`, which produces the debug symbols enabling GDB. Check out Prof. Tia Newhall's guide:`http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_gdb.html`.
- **Ask Questions**: If there is anything that is confusing or that you don't quite understand, do not be afraid to ask questions. Feel free to email me directly, use Piazza, and attend the Ninja sessions.

Extra Credit and Challenge Exercises

- **(Challenge)**

  Create an additional Account class, a MoneyMarket account which has the property that there is a limit on both withdraws and deposits.

- **(Extra Credit) 0.5**

  Take your MoneMarket account and hook it into the ATM machine.

- **(Extra Credit) .75**

  Add a member function to the Bank, that once the bank is full, you create a new array larger than the current max number of accounts. You should copy over info from the old array and be able to use the new `accounts` array going forward. Be careful of memory leaks!

  Note that this is a challenging extra credit. Consult with the professor or TA before attempting.