CSIT 840 Spring 2015 Object-Oriented Programming in C++	Devvie Schneider Kent Assignment #4 (Due Date per Website)

Preface

This assignment is associated with ***Module 10*****,** and the ***textbook material referenced in that module.***

El Banco Loco has hired you to write an C++ OOP project to manage its customer accounts.

This project uses four classes: ***Customer***, ***Account***, ***Savings***, and ***Checking***. The *Savings* and *Checking* classes inherit from the *Account* class. Inheritance is covered in Chapter 11. A variable of the *Customer* class is a member of the *Account* class. This is called composition, which is covered in Sections 11.9 and 15.4. The assignment also involves polymorphism, which is covered in Chapter 15, especially section 15.2 and 15.3.

The code for the test driver, *test.cpp*, and sample runs are given to you at the end of this document.

Help

While you may feel you're beyond help, help is available, specifically ***Module 10*****, *"Composition, Inheritance, and Polymorphism", and the sections of the textbook referenced in that module.*** Also, when you do run into problems or questions, please make use of the Assignment #4 discussion forum. Remember, you can cash in discussion participation for credit toward your course grade; this includes any attempts to contribute replies to others. Also, you can all learn from the questions and responses posted by others, as well as those posted by you.

Customer Class

The *Customer* class has two *private* member variables:

1. ***accountID***, which is a c-string or a string (your choice) for the customer's account number which you may assume consists of 6 digits.
2. ***name***, which is a c-string or a string (your choice) for the customer's name which you may assume has no embedded spaces and a maximum length of 30 characters.

You may NOT create any other member variables for the *Customer* class.

The *Customer* class has two constructors, a no-argument constructor and a two-argument constructor, and possibly, a destructor.

1. a no-argument constructor which initializes each of the member variables to the null string (an empty string)
2. a two-argument constructor which sets the *name* and *accountID* to the user supplied values

You may write other constructors or a destructor as you decide is necessary. But, if you do so, please justify them in a code comment just before the added constructor or the destructor or in a readme.txt file that you will include in your zip file.

The *Customer* class also has two member functions:

1. ***view()*** displays the name of the customer and the account number
2. an overloaded assignment operator to copy the *Customer* object instantiated by the test driver to the *Customer* member variable of the *Account* instance which is also created in the test driver as shown in the following excerpts of the test driver:

Customer c (custName, acctID);

acctsPtr[aNum] = new Checking (c, startBal, overdraftOk);

//OR

Customer c (custName, acctID);

acctsPtr[aNum] = new Savings (c, startBal, intRate);

If you write additional member functions, please justify your decision in a code comment just before the added member function or in a *readme.txt* file or that you will include in your zip file.

Account Class

The *Account* **base** class represents a generic account which will never be instantiated as an *Account* object. **Subclasses (aka derived classes)** of the *Account* **base** class, namely, the *Savings* account **subclass** and the *Checking* account **subclass** described below, can be instantiated objects.

The *Account* base class has two member variables:

1. ***balance*** which is a floating point type and represents the amount in the account.
2. ***cust*** which is a *Customer* object.

These member variables cannot be *public*. You cannot add any other member variables.

The *Account* base class has two constructors:

1. a no-argument constructor which sets *balance* to zero and calls the corresponding no-argument *Customer* constructor for default initialization of the *Customer* member variables to null strings.

2. a two-argument constructor which sets *balance* to the value of the first parameter, a floating point value entered by the user in the test driver, and sets *cust* to the value of the second parameter, the *Customer* variable *c* created and initialized in the test driver, using the overloaded assignment operator of the *Customer* class.

You may write other constructors or a destructor as you decide is necessary. But, if you do so, please justify it in a code comment just before the added constructor or the destructor or in a readme.txt file that you will include in your zip file.

In addition to the constructors, and destructor if any, the *Account* base class has the following member functions:

1. ***makeDeposit*** – updates the *balance* by adding its parameter (a floating point value which represents the amount of a deposit) to the existing balance. The subclasses may access this function but will also have their own *makeDeposit* member function. This function has no return value.
2. ***makeWithdrawal*** -- updates the *balance* by subtracting its parameter (a floating point value which represents the amount of a withdrawal) from the existing balance. This function returns a boolean value, which is always *true* for this base class. The subclasses may access this function but will also have their own *makeWithdrawal* member function, which may return true or false depending on whether the withdrawal is permitted as described below.
3. ***getBalance*** – returns the current account *balance*.
4. ***view*** – calls the Customer's *view* function to view the customer information, and then displays the current account *balance.* The subclasses may access this function but will also have their own *view* member function. This function has no parameters and no return value.
5. ***adjustBalance*** – does absolutely nothing in this class; will be overridden in the subclasses.

You will need to determine which of these functions will be ***virtual***, which will be ***pure virtual,*** and which will be ***neither virtual nor pure virtual.*** You may be better able to determine these after examining the specifications for the *Savings subclass* which follows.

If you write additional member functions, please justify your decision in a code comment just before the added member function or in a *readme.txt* file or that you will include in your zip file.

Savings Class

The Savings subclass (or derived class) has one member variable:

1. ***interestRate*** which is a floating point type and represents the monthly interest rate.

This member variable cannot be *public*. You cannot add any other member variables.

The *Savings subclass* has two constructors:

1. a no-argument constructor which invokes the *Account* base class no-argument constructor and then sets *interestRate* to zero.
2. a three-argument constructor which invokes the *Account* base class two-argument constructor and then sets *interestRate* to the user-supplied interest rate (which the user supplies via the test driver).

You may write other constructors or a destructor as you decide is necessary. But, if you do so, please justify in a code comment just before the added constructor or the destructor or in a *readme.txt* file that you will include in your zip file.

The *Savings* subclass has the following member functions:

1. ***makeDeposit*** – calls the base *Account* class' *makeDeposit* function after determining whether or not a bank "reward" should be added to the deposit. As a promotional deal, the bank adds a constant $100 *REWARD* to all deposits of $10,000 or more. Thus, either the user-supplied deposit alone **or** the user-supplied deposit plus reward is passed as a parameter to the Account class' *makeDeposit* function. Function *makeDeposit* in the *Savings* subclass has no return value. (*** See ***Note*** Below ***)
2. ***makeWithdrawal*** – determines whether a withdrawal can be made, and if so, calls the base *Account* class' *makeWithdrawal* function, passing the user supplied withdrawal amount as a parameter. A withdrawal can only be made if doing so will not result in a negative balance. Returns a boolean value which is true if the withdrawal can be and is made; false otherwise.
3. ***adjustBalance*** – updates the balance by applying the user-supplied monthly interest rate. This function has no parameters and no return value.
4. ***view*** – displays the heading "Savings Account" and then calls the base *Account* class' *view* function to display general customer / account information. This function has no parameters and no return value.

If you write additional member functions, please justify your decision in a code comment just before the added member function or in a *readme.txt* file or that you will include in your zip file.

*** ***Note*** *** *El Banco Loco* is betting that its customers are not sharp enough to figure out that they can simply deposit a total of $10,000 or more into their savings account and then withdrawal that entire amount, and still receive the $100 reward.

Checking Class

The *Checking subclass* (or *derived* class) has one member variable:

1. ***overdraftProtection****,* a boolean representing whether or not the account has overdraft protection such that its balance can become negative.

This member variable cannot be *public*. You cannot add any other member variables.

The *Checking subclass* has two constructors:

1. a no-argument constructor which invokes the *Account* base class no-argument constructor and then sets *overdraftProtection* to false.
2. a three-argument constructor which invokes the *Account* base class two-argument constructor and then sets the *overdraftProtection* to the user-supplied value (which the user supplies via the given test driver).

You may write other constructors or a destructor as you decide is necessary. But, if you do so, please justify in a code comment just before the added constructor or the destructor or in a *readme.txt* file that you will include in your zip file.

The *Checking* subclass has the following member functions:

1. ***makeDeposit*** – calls the base class' *makeDeposit* function*,* passing the user supplied deposit amount as a parameter. This function has no return value.
2. ***makeWithdrawal*** – determines whether a withdrawal can be made, and if so, calls the base *Account* class' *makeWithdrawal* function, passing the user-supplied withdrawal amount as a parameter. A withdrawal can only be made if (1) making the withdrawal will not result in a negative balance, or (2) the customer has overdraft protection. Returns a boolean true if the withdrawal can be and is made; false otherwise.
3. ***adjustBalance*** – updates the balance by applying a constant service charge, if any. The service charge (a constant $10) is applied to the balance if the balance falls below the specified minimum (a constant $1,000) after the total deposit and allowed withdrawal amounts are applied. This function has no parameters and no return value.
4. ***view*** – displays the heading "Checking Account" and then calls the base *Account* class' *view* function to display general customer / account information. This function has no parameters and no return value.

If you write additional member functions, please justify your decision in a code comment just before the added member function or in a *readme.txt* file or that you will include in your zip file.

Additional Requirements:

If you finish all parts of this assignment, you should have a total of 9 files, no more, no less, including the 1, test.cpp, that is given to you below:

Given to you:

test.cpp

Written by you:

customer.h

customer.cpp

account.h

account.cpp

savings.h

savings.cpp

checking.h

checking.cpp

Please ***zip*** all 9 completed files together. Your project must have these 9 files, no more and no less; a class declaration file and implementation file for each of the four classes, and *test.cpp*, the "driver" file which tests the class. Also, please ***include a comment at the top of each of these files that gives your name.***

NOTE: You cannot put all your code in one file, or create additional files. **You also cannot change the driver file code I have given you below, with the exception of commenting out code that your were unable to implement.**

```
//test.cpp

#include <iostream>

using namespace std;
```

```
#include "account.h"
#include "checking.h"
#include "savings.h"
const int MAX = 6;
void doTransactions (Account*);

int main()
{
        Account* acctsPtr [MAX];
        char acctType;
        bool validType = true;
         char custName[NAME_SIZE];  //Alternative:  string custName;
         char acctID [ID_SIZE];              //Alternative:  string acctID;
        double startBal;

        int aNum;
        for (aNum = 0; aNum < MAX && validType; aNum++)
        {
                cout << "Enter c for checking; s for savings; any other character to quit: ";
                cin >> acctType;
                acctType = tolower (acctType);

                if (acctType == 'c' || acctType == 's')
```

```
{
        cout << "Enter customer name: ";
        cin >> custName;
        cout << "Enter account number: ";
        cin >> acctID;
        cout << "Enter account beginning balance: ";
        cin >> startBal;

}

switch (acctType)
{
    case 'c':
    {
        char response;
        bool overdraftOk;
        cout << "Does this account have overdraft protection? ";
        cin >> response;
        overdraftOk = (tolower(response) == 'y')? true : false;
        Customer c (custName, acctID);
        acctsPtr[aNum] = new Checking (c, startBal, overdraftOk);
        doTransactions (acctsPtr[aNum]);
        acctsPtr[aNum]->adjustBalance();
```

```
            cout << "Balance after service charge (if any): " <<
                        acctsPtr[aNum]->getBalance() << endl;
            break;
    }

      case 's':
    {
            double intRate;
            cout << "Enter current monthly interest rate: ";
            cin >> intRate;
            Customer c (custName, acctID);
            acctsPtr[aNum] = new Savings (c, startBal, intRate);
            doTransactions (acctsPtr[aNum]);
            acctsPtr[aNum]->adjustBalance();
            cout << "Balance after interest: " <<
                        acctsPtr[aNum]->getBalance() << endl;
               break;
    }

    default:
            validType = false;
            aNum--;
            break;
```

```
            }
        }

        int totalAccts = aNum;
        cout << "\nAccounts:\n";
        for (int i = 0; i < totalAccts; i++)
        {
            acctsPtr[i]->view();
            cout << "\n";
        }

        for (int i = 0; i < totalAccts; i++)
            delete acctsPtr[i];
}

void doTransactions (Account * aPtr)
{
        double depAmt, withdAmt;

        cout << "Enter total deposits: ";
        cin >> depAmt;
        aPtr->makeDeposit (depAmt);
        cout << "Balance after deposits: "
```

```
            << aPtr->getBalance() << endl;


        cout << "Enter total withdrawals: ";

        cin >> withdAmt;

        if (aPtr->makeWithdrawal (withdAmt))

                cout << "Balance after withdrawals: "

                        << aPtr->getBalance() << endl;

        else

                cout << "Withdrawal not made -- balance too low\n"

                        << " and no overdraft protection\n";

}
```

Summary of Requirements and Restrictions:

- Each of the 4 required classes must declare the member variables given above for that class, but do **not** include any additional member variables. Also, these **member variables *cannot* be *public*.**
- Each of the 4 classes must have the constructors given above for that class. You may write other constructors or a destructor as you decide is necessary. But, if you do so, please justify in a code comment just before the added constructor or the destructor or in a *readme.txt* file that you will include in your zip file.
- Each of the 4 classes must have the member functions given above for that class. If you write additional member functions, please justify your decision in a code comment just before the added member function or in a *readme.txt* file that you will include in your zip file.
- The Account class is the ***base*** class (or ***superclass***). The Savings and Checking classes are ***derived*** from the Account class, or are ***subclasses*** of the Account class. This is an ***inheritance*** relationship. The Account class ***contains*** the Customer class. This is a ***composition*** relationship.
- Use the test driver code that is given above. You may not change it, except to comment out code as you build your project in pieces or to comment out code that calls a member function which you are not able to implement.
- ***If you complete the program***, you should have the 9 files listed above which should be submitted as a zip file. You cannot put all your code in one file, or create additional files. Of course, you can still obtain partial credit if you do not complete the entire program; in that case you will have less files. For example, if your submission includes completed Account and Savings classes, but no Checking class, it would not include the

checking.h and the checking.cpp files. If you submit only a partial solution, be sure to comment out any necessary code in the test driver file.

Sample Runs:

Sample Run #1 (Assumes you have implemented all but the Checking class):

```
Enter c for checking; s for savings; any other character to quit: s
Enter customer name: aaa
Enter account number: 111111
Enter account beginning balance: 5000
Enter current monthly interest rate: .1
Enter total deposits: 2500
Balance after deposits: 7500
Enter total withdrawals: 400
Balance after withdrawals: 7100
Balance after interest: 7810
Enter c for checking; s for savings; any other character to quit: S
Enter customer name: bbb
Enter account number: 222222
Enter account beginning balance: 3000
Enter current monthly interest rate: .05
Enter total deposits: 10000
Balance after deposits: 13100
Enter total withdrawals: 5000
Balance after withdrawals: 8100
Balance after interest: 8505
Enter c for checking; s for savings; any other character to quit: s
Enter customer name: ccc
Enter account number: 333333
Enter account beginning balance: 5000
Enter current monthly interest rate: .1
Enter total deposits: 2500
Balance after deposits: 7500
Enter total withdrawals: 7501
Withdrawal not made -- balance too low
and no overdraft protection
Balance after interest: 8250
Enter c for checking; s for savings; any other character to quit: z
```

Accounts:
Savings Account:
Name: aaa
Account #: 111111
Balance: $ 7810

Savings Account:
Name: bbb
Account #: 222222
Balance: $ 8505

Savings Account:
Name: ccc
Account #: 333333
Balance: $ 8250

Sample Run #2 (Assumes you have implemented everything, including the Checking class):

Enter c for checking; s for savings; any other character to quit: c
Enter customer name: ddd
Enter account number: 444444
Enter account beginning balance: 1000
Does this account have overdraft protection? Y
Enter total deposits: 500
Balance after deposits: 1500
Enter total withdrawals: 1550
Balance after withdrawals: -50
Balance after service charge (if any): -60
Enter c for checking; s for savings; any other character to quit: C
Enter customer name: eee
Enter account number: 555555
Enter account beginning balance: 1000
Does this account have overdraft protection? n
Enter total deposits: 500
Balance after deposits: 1500
Enter total withdrawals: 1550
Withdrawal not made -- balance too low
and no overdraft protection
Balance after service charge (if any): 1500
Enter c for checking; s for savings; any other character to quit: c
Enter customer name: fff

Enter account number: 666666
Enter account beginning balance: 1200
Does this account have overdraft protection? N
Enter total deposits: 500
Balance after deposits: 1700
Enter total withdrawals: 800
Balance after withdrawals: 900
Balance after service charge (if any): 890
Enter c for checking; s for savings; any other character to quit: S
Enter customer name: ggg
Enter account number: 777777
Enter account beginning balance: 5000
Enter current monthly interest rate: .05
Enter total deposits: 12000
Balance after deposits: 17100
Enter total withdrawals: 12000
Balance after withdrawals: 5100
Balance after interest: 5355
Enter c for checking; s for savings; any other character to quit: S
Enter customer name: hhh
Enter account number: 888888
Enter account beginning balance: 1000
Enter current monthly interest rate: .1
Enter total deposits: 800
Balance after deposits: 1800
Enter total withdrawals: 400
Balance after withdrawals: 1400
Balance after interest: 1540
Enter c for checking; s for savings; any other character to quit: s
Enter customer name: iii
Enter account number: 999999
Enter account beginning balance: 7500
Enter current monthly interest rate: .025
Enter total deposits: 2000
Balance after deposits: 9500
Enter total withdrawals: 10000
Withdrawal not made -- balance too low
and no overdraft protection
Balance after interest: 9737.5

Accounts:
Checking Account:
Name: ddd
Account #: 444444

Balance: $ -60

Checking Account:
Name: eee
Account #: 555555
Balance: $ 1500

Checking Account:
Name: fff
Account #: 666666
Balance: $ 890

Savings Account:
Name: ggg
Account #: 777777
Balance: $ 5355

Savings Account:
Name: hhh
Account #: 888888
Balance: $ 1540

Savings Account:
Name: iii
Account #: 999999
Balance: $ 9737.5

***NOTE*:**

For each of your four header files, you should use the *#**ifndef**, #define* and ***#endif*** preprocessor directives for the reasons given in ***in Section 7.11 of your textbook, "...Separating Class Specification from Implementation",*** . For example, your *account.h file* should be coded as follows:

```
#ifndef ACCOUNT_H

#define ACCOUNT_H

// Body of account.h goes here; that is, everything you need to declare in account.h

#endif
```

Analogous preprocessor directives should be used for *checking.h*, *savings.h*, and *customer.h*