Part E - Polymorphism

Virtual Functions

Workshop 8 (out of 10 marks - 3% of your final grade)

In this workshop, you are to implement specific behavior based on an abstract definition of that behavior.

LEARNING OUTCOMES

Upon successful completion of this workshop, you will have demonstrated the abilities to

- to define a pure virtual function
- to code an abstract base class
- to implement behavior specified in a pure virtual function
- to explain the difference between an abstract base class and a concrete class
- to describe to your instructor what you have learned in completing this workshop

SUBMISSION POLICY

The "in-lab" section is to be completed during your assigned lab section. It is to be completed and submitted by the end of the workshop period. If you attend the lab period and cannot complete the in-lab portion of the workshop during that period, ask your instructor for permission to complete the in-lab portion after the period. If you do not attend the workshop, you can submit the “in-lab” section along with your “at-home” section (with a penalty; see below). The “at-home” portion of the lab is due on the day of your next scheduled workshop (23:59).

All your work (all the files you create or modify) must contain your name, Seneca email and student number.

You are responsible to regularly back up your work.

Late submission penalties:

- In-lab submitted late, with at-home: Maximum of 20/50 for in-lab and Maximum of 50/50 for at home
- Workshop late for one week: in-lab, at-home and reflection must all be submitted for maximum of 50 / 100
- Workshop late for more than one week: in-lab, at-home and reflection must all be submitted for maximum of 30 / 100
- If any of in-lab, at-home or reflection is missing the mark will be zero.

BANKING ACCOUNTS:

In this workshop, we create an inheritance hierarchy that a bank might use to represent customers' bank accounts. All customers at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. More specific types of accounts also exist. *Savings accounts*, for instance, earn interest on the money they hold. *Checking accounts*, on the other hand, charge a fee per transaction (i.e., credit or debit).

Design Overview:

First, we design a base class **Account**, and then derive classes **SavingsAccount** and **CheckingAccount** that inherit from class **Account**. The base class **Account** should include one data member of type double to represent the *account balance*. The class should provide a constructor that receives an initial balance and uses it to initialize the data member. The account class provides member function credit (add an amount to the current balance), member function debit (withdraws money from the Account), member function `getBalance` and `setBalance` (set and get the balance value).

We derive the **SavingsAccount** and **CheckingAccount** from the abstract **Account** class. The **SavingsAccount** class inherits the basic functionality of **Account** class and just overrides the `display` function and defines its own `calculateInterest` function. The **checkingAccount** overrides the `credit` and `debit` functionality in the base class and overrides the `display` as well.

IN-LAB INSTRUCTION:

Download or clone workshop 8 from https://github.com/Seneca-244200/BTP-Workshop8

Open in_lab directory and view the code in **w8_in_lab.cpp**, **Account.h**, **Account.cpp. SavingsAccount.h,** and **SavingsAccount.cpp.**

Your main task is to complete the code of the **Account** as the base class and the **SavingAccount** as the derived class from **Account** class and complete the code in the appropriate files.

Account Class:

The **Account** class is designed to represent customers' bank accounts (base class). It includes a data member (of type double) to represent the account `balance`. This class

provides a constructor that receives an initial balance and uses it to initialize the data member.

Write the constructor that validates the initial balance to ensure that it's greater than or equal to 0. If not, set the `balance` to the safe empty state (`balance` equals to -1.0).

Write the virtual member function `credit` that adds an amount to the current balance.

Write the virtual member function `debit` that withdraws money from the account and ensure that the debit amount does not exceed the Account's balance. If the balance is less the amount, the balance should be left unchanged and the function should return false (otherwise it should return true).

Add the prototype of *pure* virtual function `display` that receives a reference to *ostream*. This function needs to be overridden by the classes derived from the **Account** class.

NOTE: The getBalance and setBalance have been already defined as the protected function in **Account** class so they can be used in any member function of the derived class.

SavingsAccount:

Derived class **SavingsAccount** that inherits the functionality of an **Account**, but also include a data member of type double indicating the interest rate (for example 0.12) assigned to the Account (`interestRate`).

Write the SavingsAccount's constructor that receives the initial balance, as well as an initial value for the SavingsAccount's *interest rate*, and then initializes the object. If *interest rate* is less than zero, the `interstRate` will be set to zero.

Write the public member function `calculateInterest` for **SavingsAccount** class that returns the amount of interest earned by an account. Member function `calculateInterest` should determine this amount by multiplying the interest rate by the account balance.

Note: SavingsAccount should inherit member functions `credit` and `debit` as are without redefining them.

Override the `display` function in the **Account** class that print a **SavingsAccount** in the following format (this is an example):

Account type: Saving
Balance: $ 400.00
Interest Rate (%): 12.00

The following code (w8_in_lab.cpp) will test your code:

```
// BTP200 Workshop 8: Virtual Functions
// File: 244_w8_home.cpp
// Version: 1.0
// Date: 2017/03/15
// Author: Heidar Davoudi
// Description:
// This file tests in_lab section of your workshop
/////////////////////////////////////////////////

#include <iostream>
#include "Account.h"
#include "SavingsAccount.h"

using namespace ict;
using namespace std;

int main()
{

     // Create Account for Angelina

     Account * Angelina_Account[2];

     // initialize Angelina Accounts (Both Saving)

     Angelina_Account[ 0 ] = new SavingsAccount( 400.0, 0.12 );

     Angelina_Account[ 1 ] = new SavingsAccount( 600.0, 0.15 );

     cout << "**************************" << endl;

     cout << "DISPLAY Angelina Accounts:" << endl;

     cout << "**************************" << endl;

     Angelina_Account[0]->display(cout);

     cout << "-----------------------" << endl;

     Angelina_Account[1]->display(cout);

     cout << "**************************" << endl ;
```

```
    cout << "DEPOSIT $ 2000 $ into Angelina Accounts ..." << endl ;

    for(int i=0 ; i < 2 ; i++){
        Angelina_Account[i]->credit(2000);
    }

    cout << "WITHDRAW $ 1000 and $ 500 from Angelina Accounts ..." << endl ;

    Angelina_Account[0]->debit(1000);

    Angelina_Account[1]->debit(500);


    cout << "*************************" << endl;

    cout << "DISPLAY Angelina Accounts:" << endl;

    cout << "*************************" << endl;

    Angelina_Account[0]->display(cout);

    cout << "Interest is: " <<
          ((SavingsAccount *) Angelina_Account[0])->calculateInterest() << endl;

    cout << "----------------------" << endl;

    Angelina_Account[1]->display(cout);

    cout << "Interest is: " <<
         ((SavingsAccount *) Angelina_Account[1])->calculateInterest() << endl;

    cout << "----------------------" << endl;

    return 0;
}
```

The following is the exact output of the tester program:

```
Account type: Saving
Balance: $ 400.00
Interest Rate (%): 12.00
----------------------
Account type: Saving
Balance: $ 600.00
Interest Rate (%): 15.00
*************************
DEPOSIT $ 2000 $ into Angelina Accounts ...
WITHDRAW $ 1000 and $ 500 from Angelina Accounts ...
*************************
DISPLAY Angelina Accounts:
*************************
```

```
Account type: Saving
Balance: $ 1400.00
Interest Rate (%): 12.00
Interest is: 168.00
----------------------
Account type: Saving
Balance: $ 2100.00
Interest Rate (%): 15.00
Interest is: 315.00
----------------------
```

IN-LAB SUBMISSION

If not on matrix already, upload **Account.h**, **Account.cpp. SavingsAccount.h,** and **SavingsAccount.cpp** to your matrix account. Compile and run your code and make sure everything works properly.

Then run the following script from your account: (replace profname.proflastname with your professors Seneca userid)

```
~profname.proflastname/submit 200_w8_lab <ENTER>
```

and follow the instructions.

Please note that a successful submission does not guarantee full credit for this workshop.

If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

AT-HOME INSTRUCTION (40%):

Open at_home directory and view the code in **w8_at_home.cpp**, **CheckingAccount.h** and **CheckingAccount.cpp.**

Copy **Account.h**, **Account.cpp, SavingsAccount.h,** and **SavingsAccount.cpp** form your in_lab part solution.

CheckingAccount

Derived class **CheckingAccount** that inherits from base class **Account** and include an additional data member of type double that represents the fee charged per transaction (`transactionFee`).

Write Checking- Account's constructor that receives the initial balance, as well as a parameter indicating a *transaction fee* amount. If *transaction fee* is less than zero, the `transactionFee` will be set to zero.

Write the `chargeFee` member function that updates the balance by deducting the `transactionFee` from the balance.

Override member functions `debit` for class **CheckingAccount** so that it subtracts the `transactionFee` from the account balance (call `chargeFee`). If the operation is successful, it will return true otherwise it does nothing and will return false (debit is successful if the amount is not greater than the balance). CheckingAccount's versions of this function should invoke the base-class **Account** version to perform the debit operation.

Hint: Define Account's `debit` function so that it returns a bool indicating whether money was withdrawn. Then use the return value to determine whether a fee should be charged.

Override member functions credit for class **CheckingAccount** so that it subtracts the `transactionFee` from the account balance (call `chargeFee`). CheckingAccount's versions of this function should invoke the base-class **Account** version to perform the credit operation.

Override the display function in the Account class that insert that print a **SavingsAccount** in the following format (example):

Account type: Checking
Balance: $ 400.00
Transaction Fee: 1.00

The following code (w8_at_home.cpp) will test your code:

```
// BTP200 Workshop 8: Virtual Functions
// File: w8_at_home.cpp
// Version: 1.0
// Date: 2017/03/15
// Author: Heidar Davoudi
// Description:
// This file tests at_home section of your workshop
////////////////////////////////////////////////

#include <iostream>
#include "Account.h"
#include "SavingsAccount.h"
```

```
#include "CheckingAccount.h"

using namespace ict;
using namespace std;

int main()
{

     // Create Account for Angelina

     Account * Angelina_Account[2];

     // initialize Angelina Accounts

     Angelina_Account[ 0 ] = new SavingsAccount( 400.0, 0.12 );

     Angelina_Account[ 1 ] = new CheckingAccount( 400.0, 1.0);

     cout << "*************************" << endl;

     cout << "DISPLAY Angelina Accounts:" << endl;

     cout << "*************************" << endl;

     Angelina_Account[0]->display(cout);

     cout << "----------------------" << endl;

     Angelina_Account[1]->display(cout);

     cout << "*************************" << endl ;

     cout << "DEPOSIT $ 2000 $ into Angelina Accounts ..." << endl ;

     for(int i=0 ; i < 2 ; i++){
          Angelina_Account[i]->credit(2000);
     }

     cout << "WITHDRAW $ 1000 and $ 500 from Angelina Accounts ..." << endl;

     Angelina_Account[0]->debit(1000);

     Angelina_Account[1]->debit(500);


     cout << "*************************" << endl;

     cout << "DISPLAY Angelina Accounts:" << endl;

     cout << "*************************" << endl;
```

```
    Angelina_Account[0]->display(cout);

    cout << "----------------------" << endl;

    Angelina_Account[1]->display(cout);

    cout << "----------------------" << endl;

    return 0;
}
```

The following is the exact output of the tester program:

```
DISPLAY Angelina Accounts:
*************************
Account type: Saving
Balance: $ 400.00
Interest Rate (%): 12.00
----------------------
Account type: Checking
Balance: $ 400.00
Transaction Fee: 1.00
*************************
DEPOSIT $ 2000 $ into Angelina Accounts ...
WITHDRAW $ 1000 and $ 500 from Angelina Accounts ...
*************************
DISPLAY Angelina Accounts:
*************************
Account type: Saving
Balance: $ 1400.00
Interest Rate (%): 12.00
----------------------
Account type: Checking
Balance: $ 1898.00
Transaction Fee: 1.00
----------------------
```

REFLECTION (10%)

Please provide brief answers to the following questions in a text file named **`reflect.txt`**.

1. What is the difference between an abstract base class and a concrete class?
2. Why do we need to have a pure virtual function in a base class?
3. Explain what have you learned in this workshop.

AT-HOME SUBMISSION

If not on matrix already, upload **w8_at_home.cpp**, **Account.h**, **Account.cpp, CheckingAccount.cpp, SavingsAccount.h, CheckingAccount.h, SavingsAccount.cpp, and reflect.txt .** to your matrix account. Compile and run your code and make sure everything works properly.

Then run the following script from your account: (replace profname.proflastname with your professors Seneca userid)

```
~profname.proflastname/submit 200_w8_home <ENTER>
```

and follow the instructions.

Please note that a successful submission does not guarantee full credit for this workshop.

If the professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.