**Lab Exercises**

Class: 软件1801 Name: 田豪举 StuID:\_\_201816040116\_\_\_

Date: Score:

**Lab Exercise 1 — Account Hierarchy**

**I Lab Objectives**

In this lab, you will practice:

1. Using inheritance to create an account hierarchy that includes an Account class, a SavingsAccount class and a CheckingAccount class.
2. Using private data members to limit access to data members.
3. Redefining base-class member functions in a derived class.

**II Description of the Problem(译文见教材P387 11.10)**

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).

Create an inheritance hierarchy containing base class Account and derived classes SavingsAccount and CheckingAccount that inherit from class Account. 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 constructor should validate the initial balance to ensure that it is greater than or equal to 0.0. If not, the balance should be set to 0.0 and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function credit should add an amount to the current balance. Member function debit should withdraw money from the Account and ensure that the debit amount does not exceed the Account’s balance. If it does, the balance should be left unchanged and the function should print the message "Debit amount exceeded account balance." Member function getBalance should return the current balance.

Derived class SavingsAccount should inherit the functionality of an Account, but also include a data member of type double indicating the interest rate (percentage) assigned to the Account. SavingsAccount’s constructor should receive the initial balance, as well as an initial value for the SavingsAccount’s interest rate. SavingsAccount should provide a public member function calculateInterest that returns a double indicating 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 is without redefining them.]

Derived class CheckingAccount should inherit from base class Account and include an additional data member of type double that represents the fee charged per transaction. CheckingAccount’s constructor should receive the initial balance, as well as a parameter indicating a fee amount. Class CheckingAccount should redefine member functions credit and debit so that they subtract the fee from the account balance whenever either transaction is performed successfully. CheckingAccount’s versions of these functions should invoke the base-class Account version to perform the updates to an account balance. CheckingAccount’s debit function should charge a fee only if money is actually withdrawn (i.e., the debit amount does not exceed the account balance). [*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.]

After defining the classes in this hierarchy, write a program that creates objects of each class and tests their member functions. Add interest to the SavingsAccount object by first invoking its calculateInterest function, then passing the returned interest amount to the object’s credit function.

**III Sample Output**



**IV Problem-Solving Tips**

1. Each derived class constructor, SavingsAccount and CheckingAccount, should call the Account constructor explicitly.
2. Do not use the debit member function inside the chargeFee member function, because the debit member function would then call the chargeFee member function, leading to infinite recursion. Instead use the inherited *get* and *set* functions for the account balance.

**V Your Solution**

**SavingsAccount::SavingsAccount(double initialBalance,double Rate)**

**:Account(initialBalance)**

**{**

**interestRate=Rate;**

**}**

**// return the amount of interest earned**

**double SavingsAccount::calculateInterest()**

**{**

**return balance\*interestRate;**

**}**

**void CheckingAccount::credit( double amount)**

**{**

**balance=balance+amount;**

**}**

**// debit (subtract) an amount from the account balance and charge fee**

**bool CheckingAccount::debit(double amount)**

**{**

**if ( amount > balance ) // debit amount exceeds balance**

**{**

**cout << "Debit amount exceeded account balance." << endl;**

**return false;**

**} // end if**

**else // debit amount does not exceed balance**

**{**

**balance = balance - amount;**

**chargeFee();**

**return true;**

**}**

**}**

**// subtract transaction fee**

**void CheckingAccount::chargeFee()**

**{**

**balance=balance-transactionFee;**

**cout<<"The account has been debit and the transaction fee has been deducted";**

**}**

**double interestEarned;**

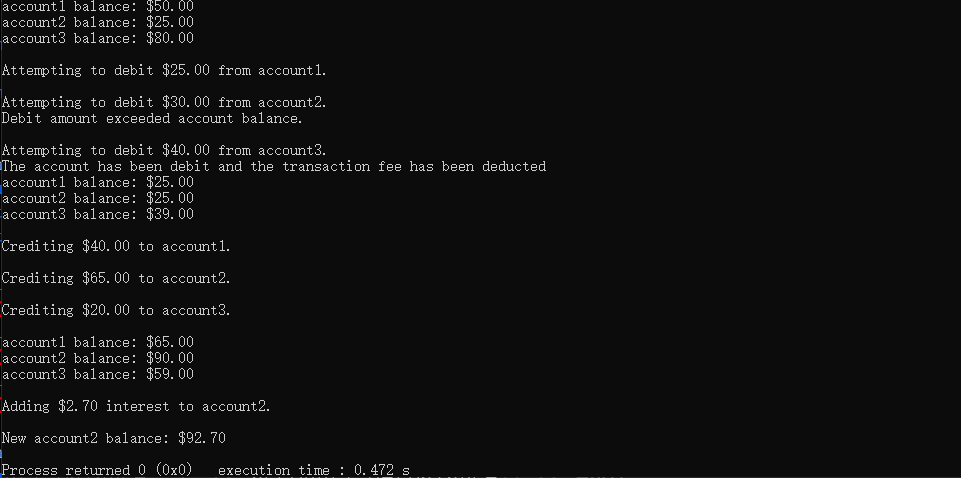
**interestEarned=account2.calculateInterest();**

**cout << "\nAdding $" << interestEarned << " interest to account2."**

**<< endl;**

**account2.credit(interestEarned);**

**cout << "\nNew account2 balance: $" << account2.getBalance() << endl;**



**Lab Exercise 2 — Composition**

**I Lab Objectives**

In this lab, you will practice:

1. Using composition to incorporate one class’s members into another class.

The follow-up question and activity also will give you practice:

1. Comparing inheritance and composition.

**II Description of the Problem (译文见P386 11.3)**

Many programs written with inheritance could be written with composition instead, and vice versa. Rewrite class BasePlusCommissionEmployee of the CommissionEmploy ee–BasePlusCommissionEmployee hierarchy to use composition rather than inheritance.

**III Sample Output**



**IV Problem-Solving Tips**

1. To implement BasePlusCommissionEmployee using composition, include a ComissionEmployee object as a data member in the BasePlusCommission Employee class.
2. To access a member of CommissionEmployee inside a member function of BasePlusCommissionEmployee, it must be preceded by the name of the CommissionEmployee object and the dot operator.
3. Most of BasePlusCommissionEmployee’s member functions will be implemented by simply calling the same member function from the CommissionEmployee object; this is known as “delegation.”

**V Your Solution**

private:

double baseSalary; // base salary

CommissionEmployee employee;

BasePlusCommissionEmployee::BasePlusCommissionEmployee(

const string &first, const string &last, const string &ssn,

double sales, double rate, double salary ):employee(first,last,ssn,sales,rate)

// initialize composed object

{

setBaseSalary( salary ); // validate and store base salary

} // end BasePlusCommissionEmployee constructor

void BasePlusCommissionEmployee::setSocialSecurityNumber(

const string &ssn )

{

employee.setSocialSecurityNumber(ssn);

} // end function setSocialSecurityNumber

// return commission employee's social security number

string BasePlusCommissionEmployee::getSocialSecurityNumber() const

{

return employee.getSocialSecurityNumber();

} // end function getSocialSecurityNumber

// set commission employee's gross sales amount

void BasePlusCommissionEmployee::setGrossSales( double sales )

{

employee.setGrossSales(sales);

} // end function setGrossSales

// return commission employee's gross sales amount

double BasePlusCommissionEmployee::getGrossSales() const

{

return employee.getGrossSales();

} // end function getGrossSales

// set commission employee's commission rate

void BasePlusCommissionEmployee::setCommissionRate( double rate )

{

employee.setCommissionRate(rate);

} // end function setCommissionRate

// return commission employee's commission rate

double BasePlusCommissionEmployee::getCommissionRate() const

{

return employee.getCommissionRate();

} // end function getCommissionRate

// set base salary

void BasePlusCommissionEmployee::setBaseSalary( double salary )

{

baseSalary = ( salary < 0.0 ) ? 0.0 : salary;

} // end function setBaseSalary

// return base salary

double BasePlusCommissionEmployee::getBaseSalary() const

{

return baseSalary;

} // end function getBaseSalary

// calculate earnings

double BasePlusCommissionEmployee::earnings() const

{

return getBaseSalary() +employee.earnings();

} // end function earnings

// print BasePlusCommissionEmployee object

void BasePlusCommissionEmployee::print() const

{

cout << "base-salaried ";

// invoke composed CommissionEmployee object's print function

employee.print();

cout << "\nbase salary: " << getBaseSalary();

} // end function print

employee.setBaseSalary( 1000 ); // set base salary

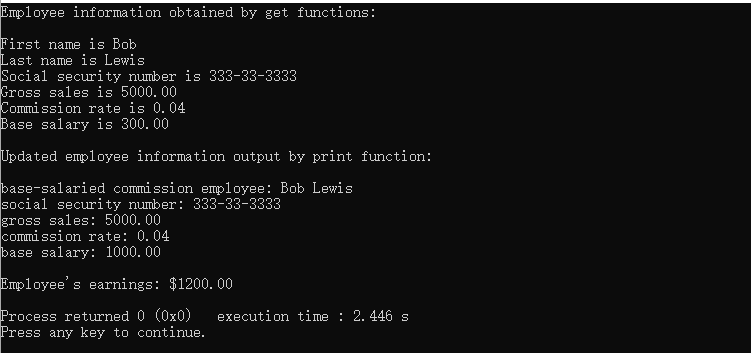
cout << "\nUpdated employee information output by print function: \n"

<< endl;

employee.print(); // display the new employee information

// display the employee's earnings

cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;



**VI Follow-Up Questions and Activities**

1. Assess the relative merits of the two approaches for designing classes Commission-Employee and BasePlusCommissionEmployee, as well as for object-oriented programs in general. Which approach is more natural? Why?
2. 继承：Lab1使用继承，代码比较简短。在派生类上重定义部分基类函数即可。
3. 组合：Lab2使用组合。代码更加简单易读，主函数代码会更清晰

我觉得两种方法各有各的好处，组合需要加大代码量，继承在使用中会比组合更容易出现错误，在不同情况中，两者合适度不同。