# LAB10: Polymorphism

Bank Account

Rung-Bin Lin

International Bachelor Program in Informatics
Yuan Ze University

05/06/2021

# Object Access in Inheritance Hierarchies without Polymorphism

- **In the hierarchy of inheritance, an object of derived class is an object of its base class. So we can assign an object of derived class to an object of base class.**

```
class Animal
{
public:
    void move();
}
class Frog: public Animal
{
Public:
    void move();
}
```

```
class Fish: public Animal
{
public:
    void move();
}
class Bird: public Animal
{
Public:
    void move();
}
```

# Access without Polymorphism

- **To invoke a member function of an object without polymorphism, the member function called depends on the type of handles defined, not depending on the type of objects the handles are pointing to.**
  - Three types of handles: pointer, reference, name

```
Animal anAnimal;  // name handle
Frog aFrog;             // name handle
Frog *aFrogPtr = &aFrog;  // pointer handle
Frog &aliasFrog=*aFrogPtr;   // reference handle
anAnimal.move();  // name handle, call member function(MB) move() of Animal
aFrog.move(); // name handle, call MB move() of Frog
aFrogPtr->move();  // pointer handle, call MB move() of Frog
aliasFrog.move();  // reference handle, call MB move() of Frog
anAnimal = aFrog; //OK
anAnimal.move(); // call MB move of Animal
Frog anotherFrog = anAnimal; // NOT Ok
```

## Class Definition for Polymorphism

- **Polymorphism enables us to program in the general rather than in the specific.**
  - **Taking the animal's move as an example, the action of move of one animal differs from the action of move of another animal. Then we can define a virtual "move()" member function in base class generally and give specific implementation of move() in each derived class**

```
class Animal
{
public:
    virtual void move();
}
class Frog: public Animal
{
Public:
    virtual void move();
}
```

```
class Fish: public Animal
{
public:
    virtual void move();
}
class bird: public Animal
{
Public:
    virtual void move();
}
```

# Access with Polymorphism

- **To call the "move()" member function of a particular animal, we can also use one of two types of base class handles: reference and pointer. However, the member function called is the one in the object pointed by the handle, ie., depending on the object pointed by the handle.**

```
Animal anAnimal;  // name handle
Frog aFrog;             // name handle
Frog *aFrogPtr = &aFrog;  // pointer handle
anAnimal.move();  // name handle, call member function(MB) move() of Animal
aFrog.move(); // name handle, call MB move() of Frog
aFrogPtr->move();  // pointer handle, call MB move() of Frog
anAnimal = aFrog; //OK
anAnimal.move(); // call MB move of Animal
Animal &isAnimal=*aFrogPtr;   // reference handle, *aFrogPtr can be replaced by aFrog
isAnimal.move();  // reference handle, call MB move() of Frog
Bird aBird;
Animal *anotherAnimal = &aBird; // pointer handle
anotherAnimal->move(); // call MB move() of Bird
```

## Abstract Class and Pure Virtual Function

- What is Animal move()?
  - No specific actions of move for animal, however, a move for a frog, a bird, or a fish can be defined precisely. So the move() in Animal class can have no implementation. Only the move() of derived classes such as Frog, Fish, and Bird has an implementation, i.e., having a function body.
- The classes in a hierarchy used to describe this sort of concept are called abstract classes.
- Abstract classes are used as base classes for deriving more concrete classes.
- Attempting to instantiate an object of an abstract class causes compilation error.
  - What we can do is to create pointers of abstract class objects.

# Forming Pure Virtual Function

- **virtual void move() = 0 in Animal Class definition indicating that move() is a pure virtual function.**

```
class Animal
{
public:
     virtual void move() = 0;
}
// No function body (implementation)
of move() for Animal class.

class Frog: public Animal
{
Public:
     virtual void move();
}
```

```
class Fish: public Animal
{
public:
     virtual void move();
}
class bird: public Animal
{
Public:
     virtual void move();
}
```

## Virtual Function & Pure Virtual Function

- Virtual function can have or have not an implementation. The derived class can have no implementation of a virtual function if its base class has one.
- Pure virtual function should not have an implementation in base class. The member function of a derived class should provide its own implementation.

# Downcasting

- Down casting enables a program to determine the type of object at execution time and act on that object accordingly.
  - For example, what if we would like to do something extra when a fish moves?

```
vector <Animal *> animalArray(3);
Frog aFrog;
Bird aBird;
Fish aFish;
animalArray[0] = &aFrog;
animalArray[1] = &aBird;
animalArray[2] = &aFish;
for(int i=0; i<3; i++)
animalArray[i].move();
```

```
vector <Animal *> anArray(3);
Frog aFrog;
Bird aBird;
Fish aFish;
anArray[0] = &aFrog;
anArray[1] = &aBird;
anArray[2] = &aFish;
for(int i=0; i<3; i++){
anArray[i].move();
Fish *fishPtr = dynamic_cast<Fish *> (anArray[i]);
If( fishPtr != 0 ) { …..}
}
```

Do when anArray[i] points to a Fish object.

## Lab10 (Part I): Bank Account with Polymorphism (70%)

- Based on the base class Account, and two derived class, saving account and checking account, created in Lab 9, create two virtual functions in the base class Account.
  - virtual void credit(double = 0.0);
  - virtual bool debit(double = 0.0);
  - virtual void print();
- You can only add a statement at the place marked with ** in the beginning of the line in the main() function. The rest of the main() function should be left intact.
- You have to include <typeinfo> into your program.

## Key Points for TA Grading

- The following three functions are indeed declared as virtual functions
  - ✓ virtual void credit(double = 0.0);
  - ✓ virtual bool debit(double = 0.0);
  - ✓ virtual void print();
- The three places each marked with ** indeed have a statement added there.
- The output must be correct.

## Original Base Class

```
class Account
{
public:
    Account(double = 0.0, double =0.0);
    void credit(double =0.0);  // Deposit money >0
    bool debit(double = 0.0);  // Withdraw money>0
    double getBalance(); // Get balance
    double calculateInterest(); // Return interest and add the
interest to the balance
    void print(); // print balance and interest rate
private:
    double balance;  // Account balance >=0
     double interestRate; // Interest rate >=0
};
```

# Member Functions of Account

```
Account::Account(double bal,
double iRate )
{
   if (bal >0)
      balance = bal;
   else
      balance = 0;
   if(iRate >0)
      interestRate = iRate;
   else interestRate = 0;
}
void Account::credit(double
depos)
{
   if(depos > 0)
      balance = balance + depos;
}

double Account::getBalance()
{
   return balance;
}
```

```
bool Account::debit(double withdw)
{
   if(withdw >0 && withdw <= balance)
   {
      balance = balance - withdw;
      return true;
   }
   else if(withdw > balance)
     {
        cout << “   Debit amount exceeded
account balance." << endl;
        return false;
     }
   return false;
}
void Account::print()
{
   cout << "   Balance: " << balance <<
endl;
   cout << "   Interest rate: " <<
interestRate << endl;
}
```

# Original SavingAccount Class

- **For your convenience, a summary of SavingAcount class is given below.**

```
class SavingAccount {
 public:
    SavingAccount(double = 0.0, double = 0.0,  double = 3.0);
    // parameters: balance, interest rate, transaction fee.
    bool debit(double =0.0);
    void print();
 private:
    double transactFee;  // transaction fee for withdrawing
 };
```

  - **transactFee is a**n amount of money paid to the bank by a saving account if a withdraw transaction is made on a saving account. No transaction fee is charged for deposition.
  - debit() can only be done if balance remains positive after withdrawing. There is no transaction fee if a transaction fails.
- You should implement the member functions. No other member functions and data should be added.

# Original CheckingAccount Class

```
class CheckingAccount {
    public:
        CheckingAccount(double = 0.0, double =0.0, double =3.0, double =
    2.0); // Parameters: balance, interest rate, transaction fee for withdraw, transaction fee for deposition
        bool debit(double =0.0); // return true if it can be done successfully.
        void credit(double =0.0);
        void print();
    private:
        double transactFeeW; // withdraw
        double transactFeeD; // Deposit
    }
```

- **There is a transaction fee respectively for withdrawing and depositing if a transaction succeeds. Otherwise, no transaction fee is applied.**
- **debit() and credit()can only be done if their balance remains positive after transaction**
- **Implement the member functions. No other member functions and data should be added.**

## Original Extra Global Functions

**bool CheckingToSaving(CheckingAccount&, SavingAccount&, const double);**

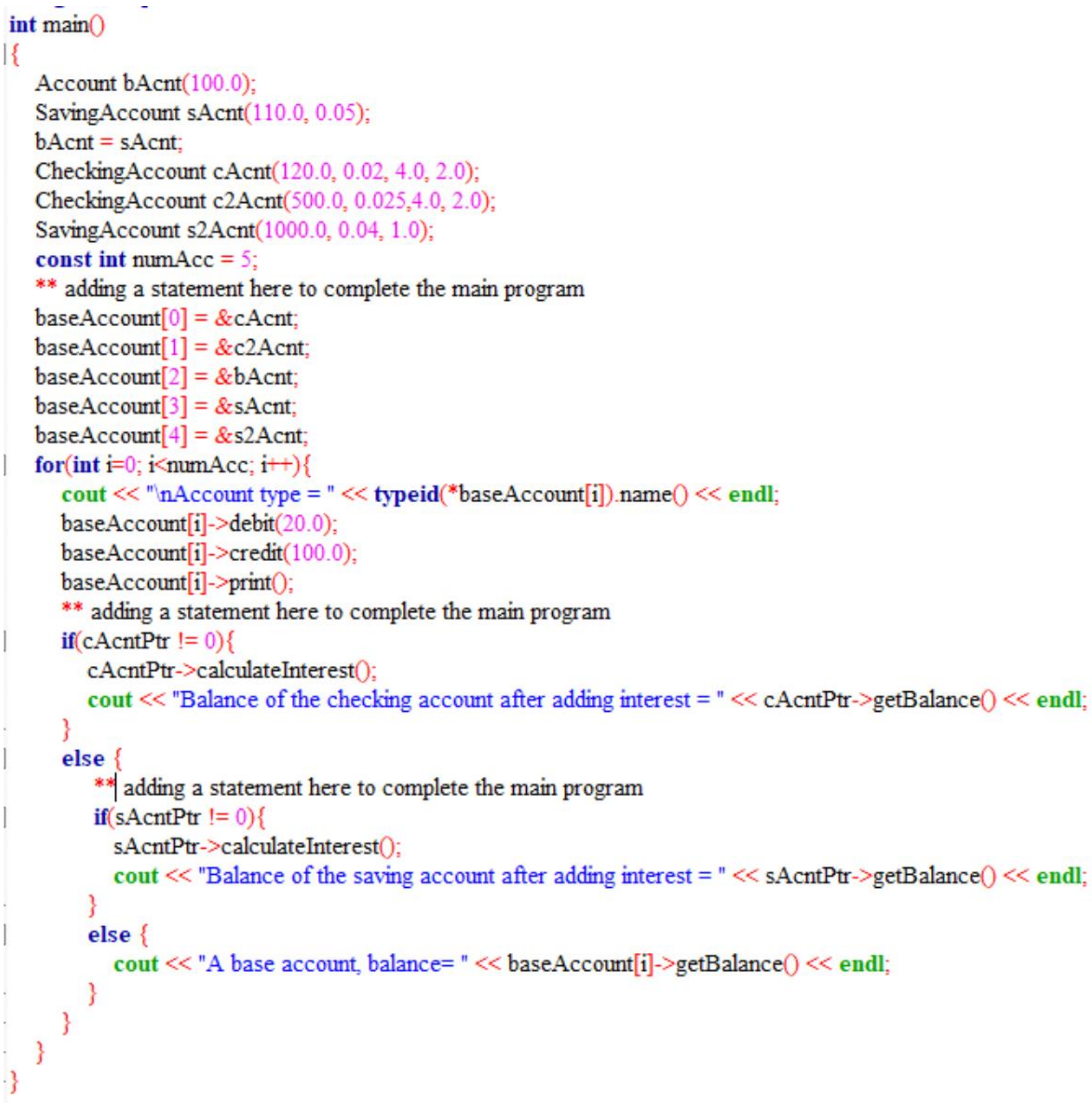
- This function should transfer an amount of money from a checking account to a saving account. The checking account should pay a transaction fee for withdrawing. Return true when the transaction is successful.

**bool SavingToChecking(SavingAccount&, CheckingAccount&, const double);**

  - This function should transfer an amount of money from a saving account to a checking account. The saving account should pay a transaction fee for withdrawing and the checking account should pay a transaction fee for deposition. Return true when the transaction is successful.

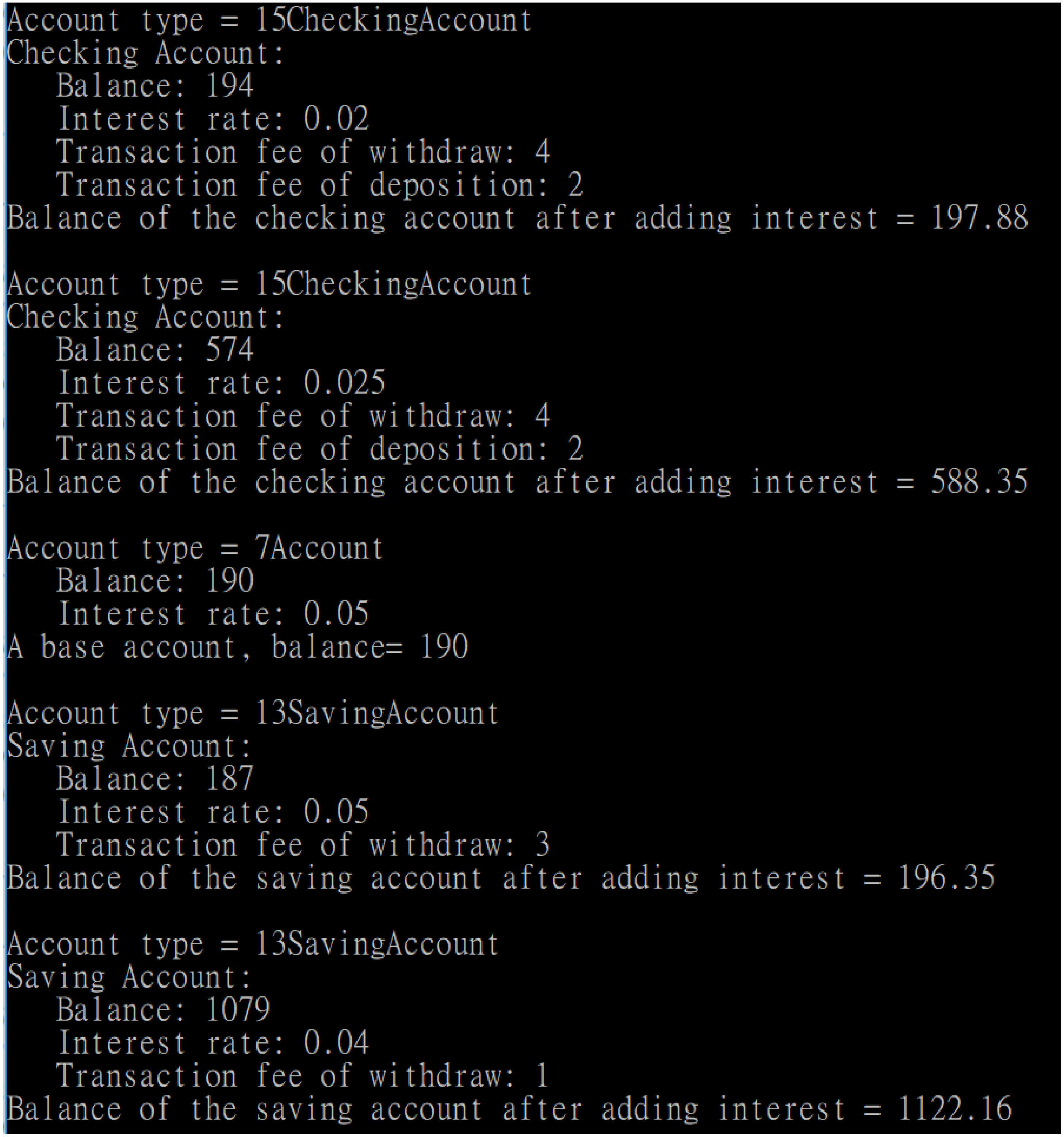
- **These two functions should make *friend* to CheckingAccount class and SavingAccount class.**

# Main() Function for Lab 10 (Part I)



```
int main()
{
   Account bAcnt(100.0);
   SavingAccount sAcnt(110.0, 0.05);
   bAcnt = sAcnt;
   CheckingAccount cAcnt(120.0, 0.02, 4.0, 2.0);
   CheckingAccount c2Acnt(500.0, 0.025,4.0, 2.0);
   SavingAccount s2Acnt(1000.0, 0.04, 1.0);
   const int numAcc = 5;
   ** adding a statement here to complete the main program
   baseAccount[0] = &cAcnt;
   baseAccount[1] = &c2Acnt;
   baseAccount[2] = &bAcnt;
   baseAccount[3] = &sAcnt;
   baseAccount[4] = &s2Acnt;
   for(int i=0; i<numAcc; i++){
      cout << "\nAccount type = " << typeid(*baseAccount[i]).name() << endl;
      baseAccount[i]->debit(20.0);
      baseAccount[i]->credit(100.0);
      baseAccount[i]->print();
      ** adding a statement here to complete the main program
      if(cAcntPtr != 0){
         cAcntPtr->calculateInterest();
         cout << "Balance of the checking account after adding interest = " << cAcntPtr->getBalance() << endl;
      }
      else {
          ** adding a statement here to complete the main program
          if(sAcntPtr != 0){
            sAcntPtr->calculateInterest();
            cout << "Balance of the saving account after adding interest = " << sAcntPtr->getBalance() << endl;
         }
         else {
            cout << "A base account, balance= " << baseAccount[i]->getBalance() << endl;
         }
      }
   }
}
```

## Output



```
Account type = 15CheckingAccount
Checking Account:
   Balance: 194
   Interest rate: 0.02
   Transaction fee of withdraw: 4
   Transaction fee of deposition: 2
Balance of the checking account after adding interest = 197.88

Account type = 15CheckingAccount
Checking Account:
   Balance: 574
   Interest rate: 0.025
   Transaction fee of withdraw: 4
   Transaction fee of deposition: 2
Balance of the checking account after adding interest = 588.35

Account type = 7Account
   Balance: 190
   Interest rate: 0.05
A base account, balance= 190

Account type = 13SavingAccount
Saving Account:
   Balance: 187
   Interest rate: 0.05
   Transaction fee of withdraw: 3
Balance of the saving account after adding interest = 196.35

Account type = 13SavingAccount
Saving Account:
   Balance: 1079
   Interest rate: 0.04
   Transaction fee of withdraw: 1
Balance of the saving account after adding interest = 1122.16
```

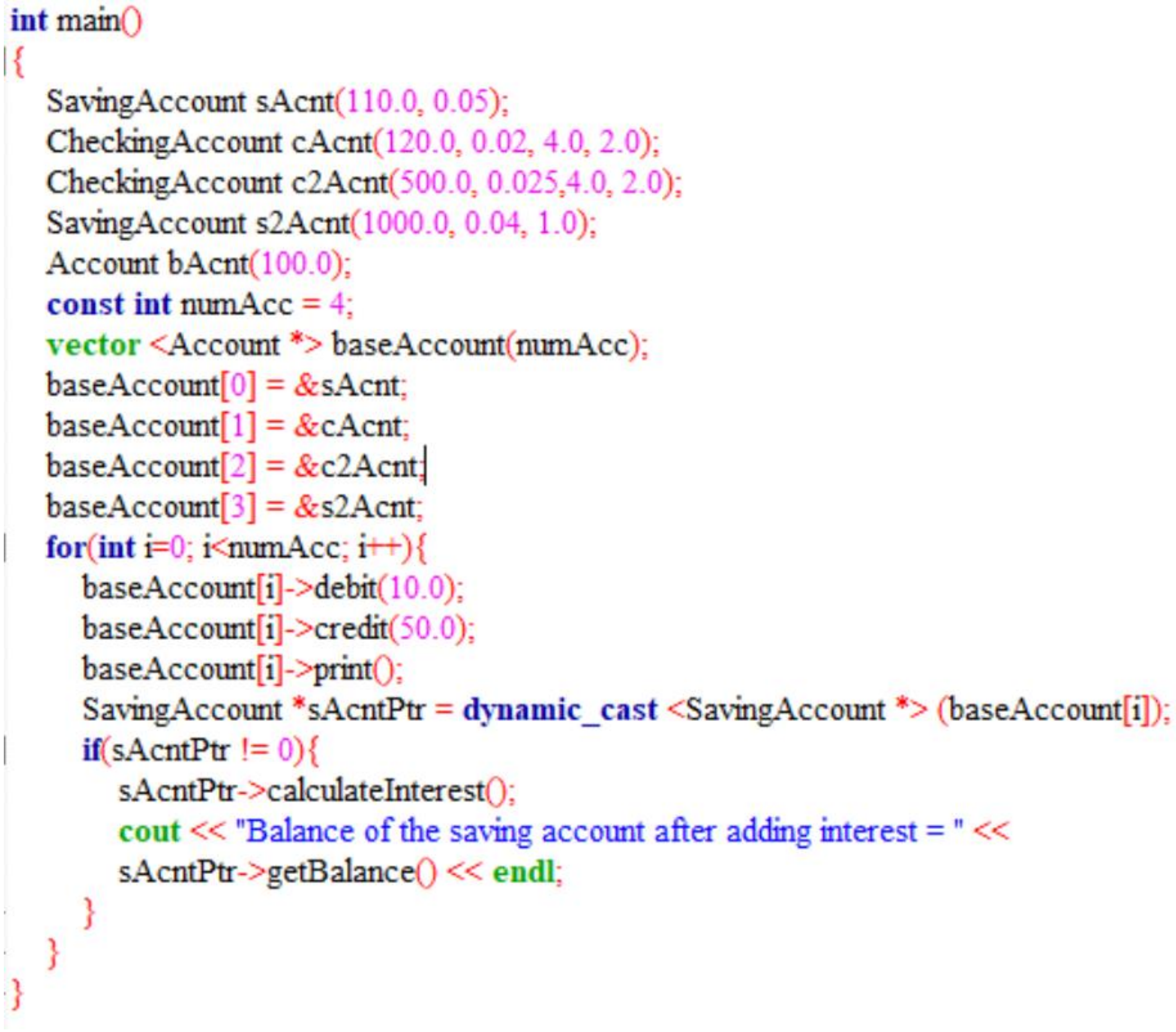
## Part II: Abstract Bank Account Classes (30%)

- Modify the Account base class to declare that credit(double), debit(double), and print() are pure virtual functions although this is not suitable in this example. Remember that there is no implementation for each pure virtual function in base class.
- Besides, you may have to change the private data members in base class account into protected data members. If you keep the two data members as private, you are allowed to add two member functions to get access to or set the values of these two data members.
- You must not modify the main() function except removing a line in order to compile the main() function successfully.

## Key Points for TA Grading

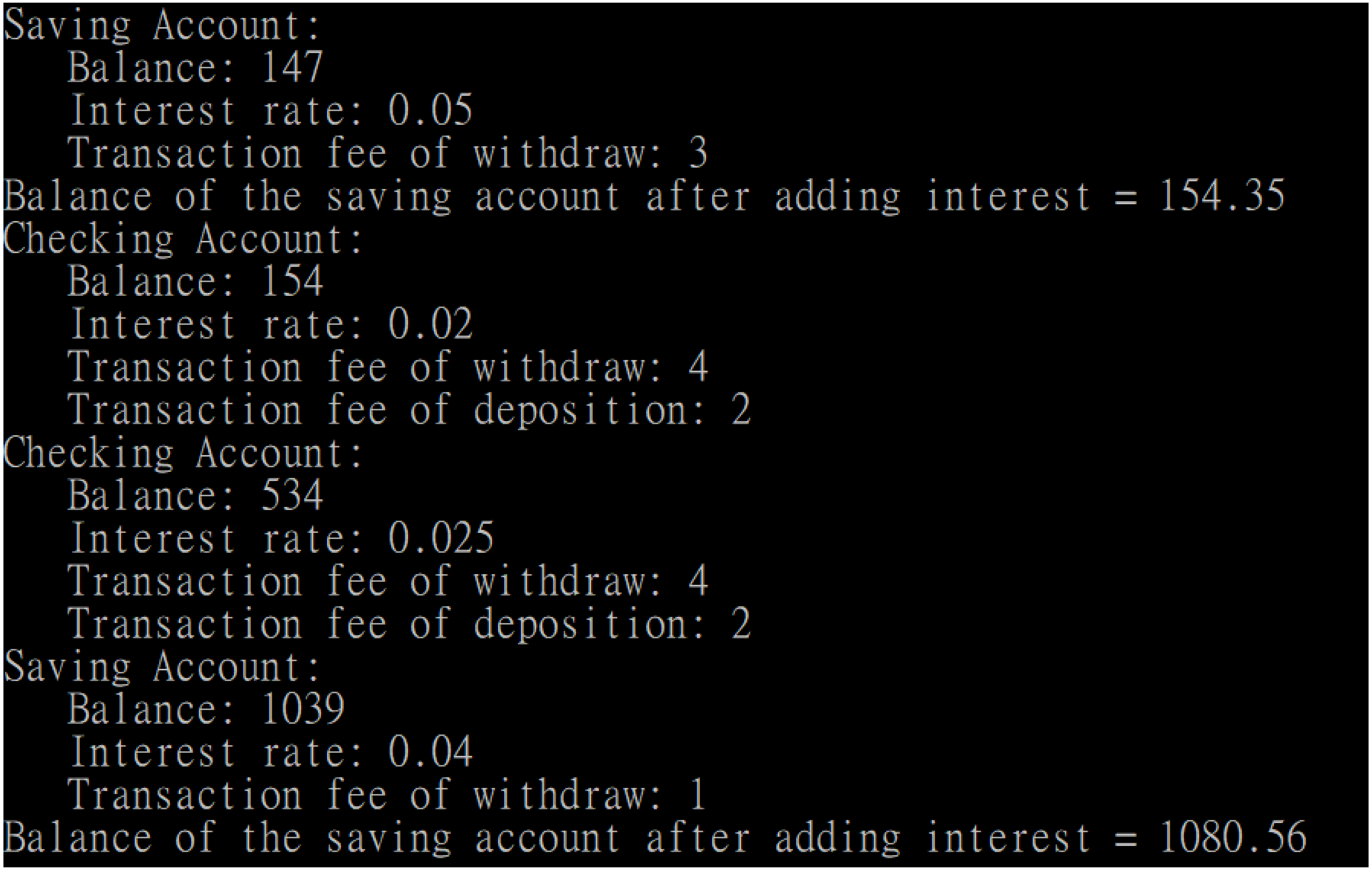
- TA should pay attention to whether the three functions are indeed declared as pure virtual functions.
- The main() function should not be modified except removing a line in order to compile the program successfully.
- The output must be correct.

# Main() Function for Part II



```
int main()
{
    SavingAccount sAcnt(110.0, 0.05);
    CheckingAccount cAcnt(120.0, 0.02, 4.0, 2.0);
    CheckingAccount c2Acnt(500.0, 0.025,4.0, 2.0);
    SavingAccount s2Acnt(1000.0, 0.04, 1.0);
    Account bAcnt(100.0);
    const int numAcc = 4;
    vector <Account *> baseAccount(numAcc);
    baseAccount[0] = &sAcnt;
    baseAccount[1] = &cAcnt;
    baseAccount[2] = &c2Acnt;
    baseAccount[3] = &s2Acnt;
    for(int i=0; i<numAcc; i++){
        baseAccount[i]->debit(10.0);
        baseAccount[i]->credit(50.0);
        baseAccount[i]->print();
        SavingAccount *sAcntPtr = dynamic_cast <SavingAccount *> (baseAccount[i]);
        if(sAcntPtr != 0){
            sAcntPtr->calculateInterest();
            cout << "Balance of the saving account after adding interest = " <<
            sAcntPtr->getBalance() << endl;
        }
    }
}
```

## Output



```
Saving Account:
   Balance: 147
   Interest rate: 0.05
   Transaction fee of withdraw: 3
Balance of the saving account after adding interest = 154.35
Checking Account:
   Balance: 154
   Interest rate: 0.02
   Transaction fee of withdraw: 4
   Transaction fee of deposition: 2
Checking Account:
   Balance: 534
   Interest rate: 0.025
   Transaction fee of withdraw: 4
   Transaction fee of deposition: 2
Saving Account:
   Balance: 1039
   Interest rate: 0.04
   Transaction fee of withdraw: 1
Balance of the saving account after adding interest = 1080.56
```