In class Practice 11/8/2019

Group three or four people team.

Part I

1. If a class contains at least one pure virtual function, it's a(n) ___abstract___class.
2. Classes from which objects can be instantiated are called_concrete_ classes.
3. The ability to associate multiple meanings to one function name using dynamic binding is called polymorphism.
4. C++ implements polymorphism by waiting until run-time to determine which version of a function to use. This is also known as ___dynamic binding.
5. Operator _dynamic_cast can be used to downcast base-class pointers safely.
6. Operator typeid returns a reference to a(n)type_info object.
7. Overridable functions are declared using keyword__virtual.
8. Casting a base-class pointer to a derived-class pointer is called downcasting.
9. What is another name for a child class?
   a. derived class
   b. sub class
   c. Descendent (or child) class
   [✓ d. all of the above]
   e. none of the above
10. Which is the correct way to tell the compiler that the class being declared (ChildClass) is derived from the base class (BaseClass)?
    a. class ChildClass::public BaseClass
    [✓ b. class ChildClass:public BaseClass]
    c. class ChildClass childOf public BaseClass
    d. class ChildClass derived BaseClass

Part II

Answer the following question

1. Why the following caused compilation errors? Give reason of these two errors

```
#include <iostream>
using namespace std;

class A
{
public:
   virtual void f1() {cout << "In A f1()"<<endl;}
   void f2();
};

class B : public A
{
public:
   void f1() override {cout << "In B f1(). It is fine to overridea virtual function"<<endl;}
   void f1() const override {cout << "In B f1() const "<<endl;};
                      // Error above
   void f2() override {cout << "In B f2()"<<endl;}
                      // Error above
};
```

void f1() const override (cout << “in B f(1) const” << endl;)
/Error: A::f1 doesn’t have const, signature is different
void f2() const (cout << “in B f(2) const” << endl;)
/Error: A::f2 is not virtual, so cannot override

2. Why the statement is not true?
   All virtual functions in an abstract base class must be declared as pure virtual functions.

   No, an abstract base class can include virtual function with implementations

Part III

We will modify previous week Account and CheckingAccount inheritance problem so that we can do polymorphism. In addition to these two accounts, let's add another class call it SavingAccount.

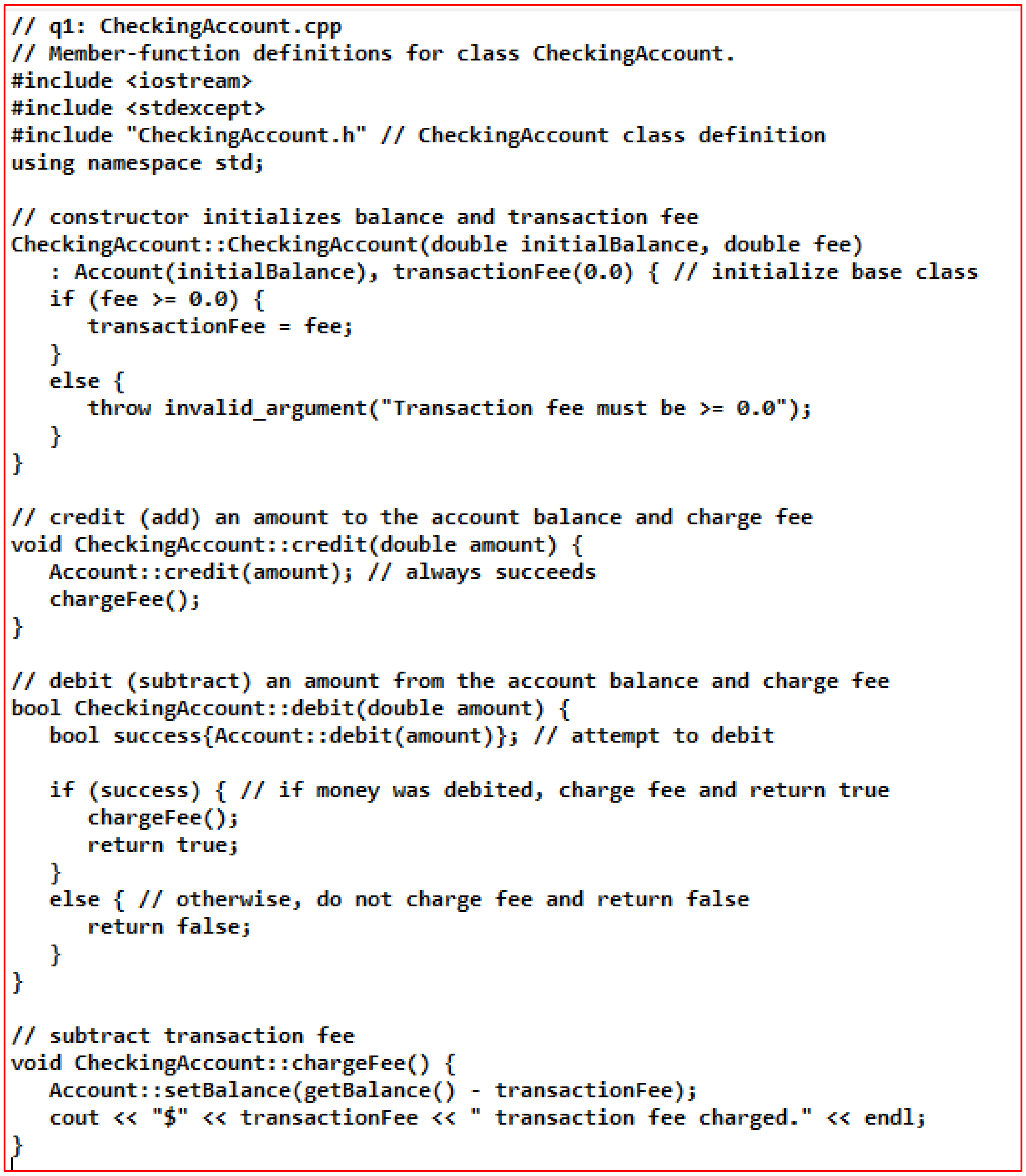
So total three header files and three cpp files for this question.

The following are Account.h, Account.cpp, CheckingAccount.h and CheckingAccount.cpp from last week.

```
// q1: Account.h
// Definition of Account class.
#ifndef ACCOUNT_H
#define ACCOUNT_H

class Account {
public:
   Account(double); // constructor initializes balance
   void credit(double); // add an amount to the account balance
   bool debit(double); // subtract an amount from the account balance
   void setBalance(double); // sets the account balance
   double getBalance(); // return the account balance
private:
   double balance; // data member that stores the balance
};

#endif
```

```
// q1: CheckingAccount.h
// Definition of CheckingAccount class.
#ifndef CHECKING_H
#define CHECKING_H

#include "Account.h" // Account class definition

class CheckingAccount : public Account {
public:
   // constructor initializes balance and transaction fee
   CheckingAccount(double, double);

   void credit(double); // redefined credit function
   bool debit(double); // redefined debit function
private:
   double transactionFee; // fee charged per transaction

   // utility function to charge fee
   void chargeFee();
};

#endif
```

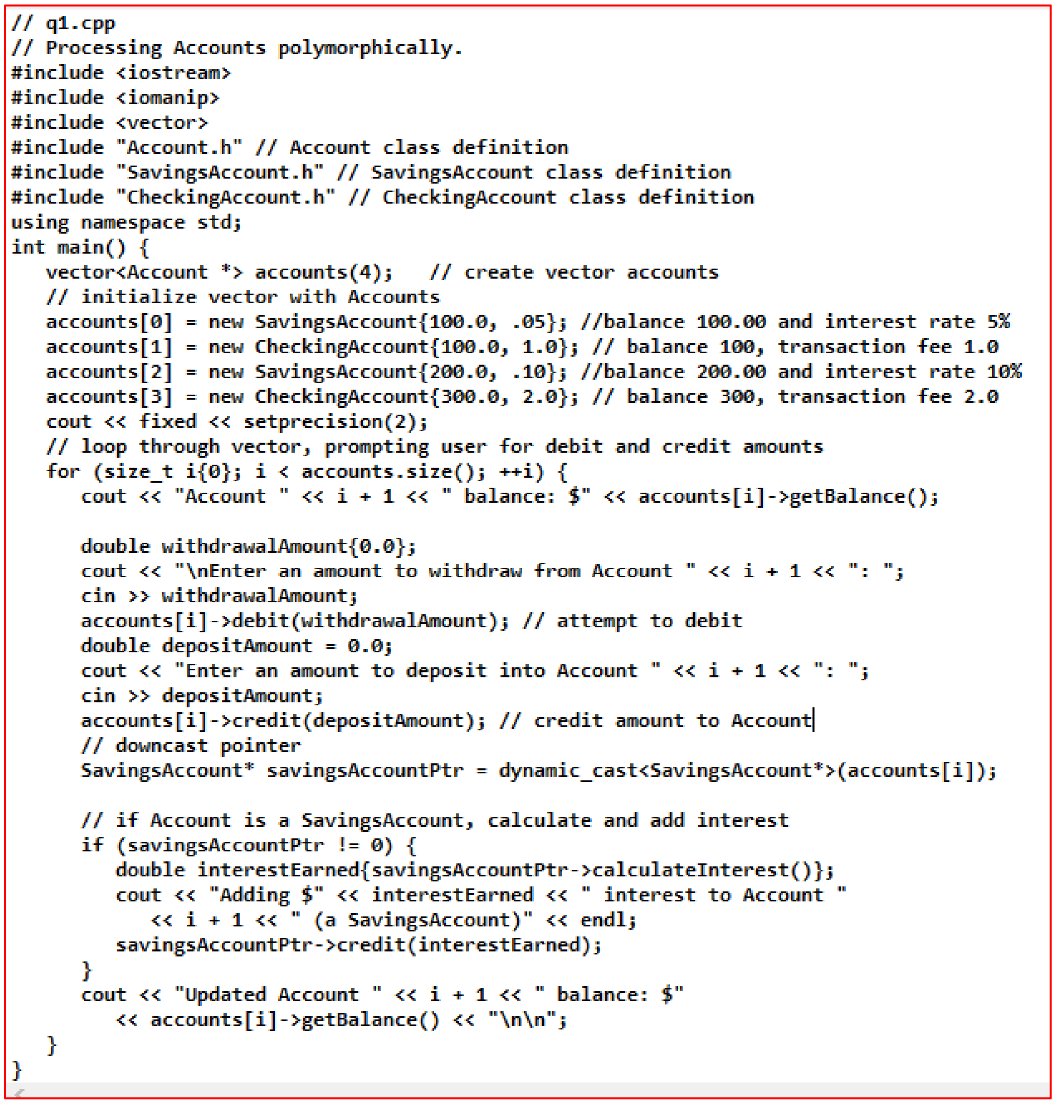
```
// q1: Account.cpp
// Member-function definitions for class Account.
#include <stdexcept>
#include <iostream>
#include "Account.h" // include definition of class Account
using namespace std;

// Account constructor initializes data member balance
Account::Account(double initialBalance) : balance(0.0) {
   if (initialBalance >= 0.0) {
      balance = initialBalance;
   }
   else {
      throw invalid_argument("Initial balance must be positive");
   }
}

// credit (add) an amount to the account balance
void Account::credit(double amount) {
   balance = balance + amount; // add amount to balance
}

// debit (subtract) an amount from the account balance
// return true if money was debited
bool Account::debit(double amount) {
   if (amount > balance) { // debit amount exceeds balance
      cout << "Debit amount exceeded account balance." << endl;
      return false;
   }
   else { // debit amount does not exceed balance
      balance = balance - amount;
      return true;
   }
}

void Account::setBalance(double newBalance) {
   balance = newBalance;
}

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



```cpp
// q1: CheckingAccount.cpp
// Member-function definitions for class CheckingAccount.
#include <iostream>
#include <stdexcept>
#include "CheckingAccount.h" // CheckingAccount class definition
using namespace std;

// constructor initializes balance and transaction fee
CheckingAccount::CheckingAccount(double initialBalance, double fee)
   : Account(initialBalance), transactionFee(0.0) { // initialize base class
   if (fee >= 0.0) {
      transactionFee = fee;
   }
   else {
      throw invalid_argument("Transaction fee must be >= 0.0");
   }
}

// credit (add) an amount to the account balance and charge fee
void CheckingAccount::credit(double amount) {
   Account::credit(amount); // always succeeds
   chargeFee();
}

// debit (subtract) an amount from the account balance and charge fee
bool CheckingAccount::debit(double amount) {
   bool success{Account::debit(amount)}; // attempt to debit

   if (success) { // if money was debited, charge fee and return true
      chargeFee();
      return true;
   }
   else { // otherwise, do not charge fee and return false
      return false;
   }
}

// subtract transaction fee
void CheckingAccount::chargeFee() {
   Account::setBalance(getBalance() - transactionFee);
   cout << "$" << transactionFee << " transaction fee charged." << endl;
}
```

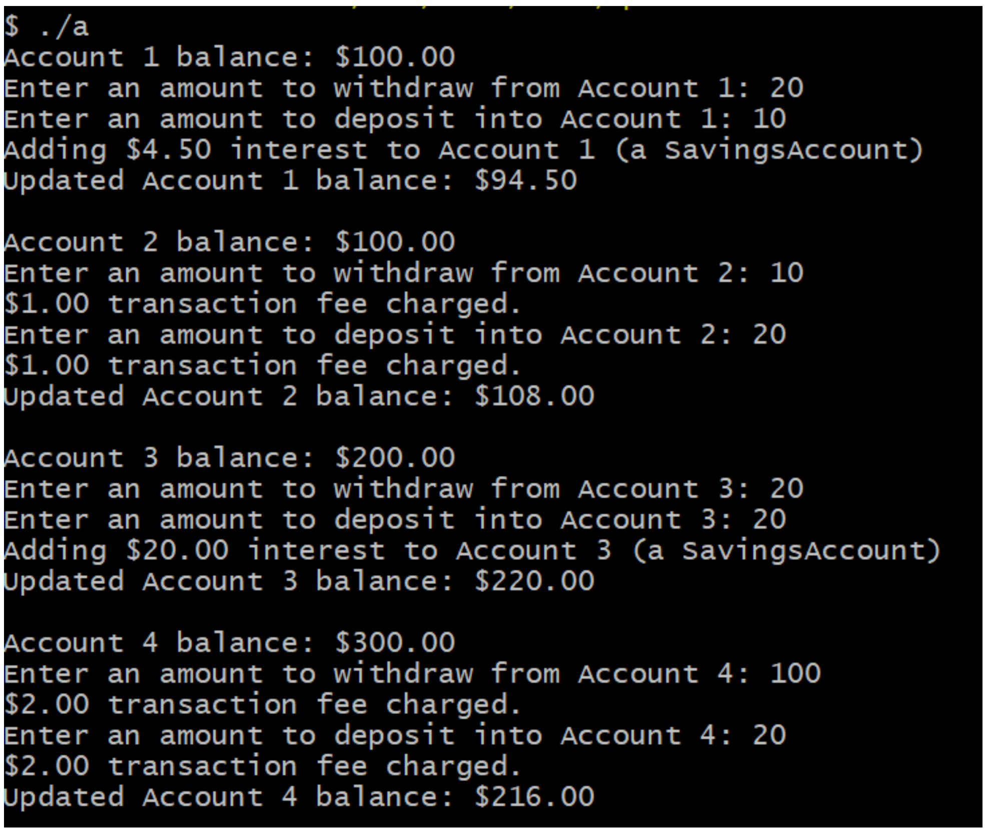
The class SavingAccount shall inherit class Account. In addition to the initial balance, the SavingAccount has an attribute call interestRate. This class contains a constructor with two parameters. One for balance and the other for interest rate. It also has a member function call calculateInterest() which will return the interest earned by multiply balance with interest rate.

After modify the implementation of these classes, we wrote a driver to test our implementation. The following is the test program and the result.



```cpp
// q1.cpp
// Processing Accounts polymorphically.
#include <iostream>
#include <iomanip>
#include <vector>
#include "Account.h" // Account class definition
#include "SavingsAccount.h" // SavingsAccount class definition
#include "CheckingAccount.h" // CheckingAccount class definition
using namespace std;
int main() {
   vector<Account *> accounts(4);   // create vector accounts
   // initialize vector with Accounts
   accounts[0] = new SavingsAccount{100.0, .05}; //balance 100.00 and interest rate 5%
   accounts[1] = new CheckingAccount{100.0, 1.0}; // balance 100, transaction fee 1.0
   accounts[2] = new SavingsAccount{200.0, .10}; //balance 200.00 and interest rate 10%
   accounts[3] = new CheckingAccount{300.0, 2.0}; // balance 300, transaction fee 2.0
   cout << fixed << setprecision(2);
   // loop through vector, prompting user for debit and credit amounts
   for (size_t i{0}; i < accounts.size(); ++i) {
      cout << "Account " << i + 1 << " balance: $" << accounts[i]->getBalance();

      double withdrawalAmount{0.0};
      cout << "\nEnter an amount to withdraw from Account " << i + 1 << ": ";
      cin >> withdrawalAmount;
      accounts[i]->debit(withdrawalAmount); // attempt to debit
      double depositAmount = 0.0;
      cout << "Enter an amount to deposit into Account " << i + 1 << ": ";
      cin >> depositAmount;
      accounts[i]->credit(depositAmount); // credit amount to Account
      // downcast pointer
      SavingsAccount* savingsAccountPtr = dynamic_cast<SavingsAccount*>(accounts[i]);

      // if Account is a SavingsAccount, calculate and add interest
      if (savingsAccountPtr != 0) {
         double interestEarned{savingsAccountPtr->calculateInterest()};
         cout << "Adding $" << interestEarned << " interest to Account "
            << i + 1 << " (a SavingsAccount)" << endl;
         savingsAccountPtr->credit(interestEarned);
      }
      cout << "Updated Account " << i + 1 << " balance: $"
         << accounts[i]->getBalance() << "\n\n";
   }
}
```

The test run results



```
$ ./a
Account 1 balance: $100.00
Enter an amount to withdraw from Account 1: 20
Enter an amount to deposit into Account 1: 10
Adding $4.50 interest to Account 1 (a SavingsAccount)
Updated Account 1 balance: $94.50

Account 2 balance: $100.00
Enter an amount to withdraw from Account 2: 10
$1.00 transaction fee charged.
Enter an amount to deposit into Account 2: 20
$1.00 transaction fee charged.
Updated Account 2 balance: $108.00

Account 3 balance: $200.00
Enter an amount to withdraw from Account 3: 20
Enter an amount to deposit into Account 3: 20
Adding $20.00 interest to Account 3 (a SavingsAccount)
Updated Account 3 balance: $220.00

Account 4 balance: $300.00
Enter an amount to withdraw from Account 4: 100
$2.00 transaction fee charged.
Enter an amount to deposit into Account 4: 20
$2.00 transaction fee charged.
Updated Account 4 balance: $216.00
```