Part E – Polymorphism

VIRTUAL FUNCTIONS

Workshop 8 (out of 10 marks – 3.75% of your final grade)

In this workshop, you are to implement an abstract definition of behavior for a specific type.

LEARNING OUTCOMES

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

- define a pure virtual function
- code an abstract base class
- implement behavior declared in a pure virtual function
- explain the difference between an abstract base class and a concrete class
- describe 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. You must be present at the lab in order to get credit for the *in-lab* portion.

If you do not attend the lab, you can submit the *in-lab* section along with your *at-home* section (see penalties below). The *at-home* portion of the lab is due on the day that is four days after your scheduled *in-lab* workshop (@23:59) (even if that day is a holiday).

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

You are responsible to back up your work regularly.

Late Submission Penalties

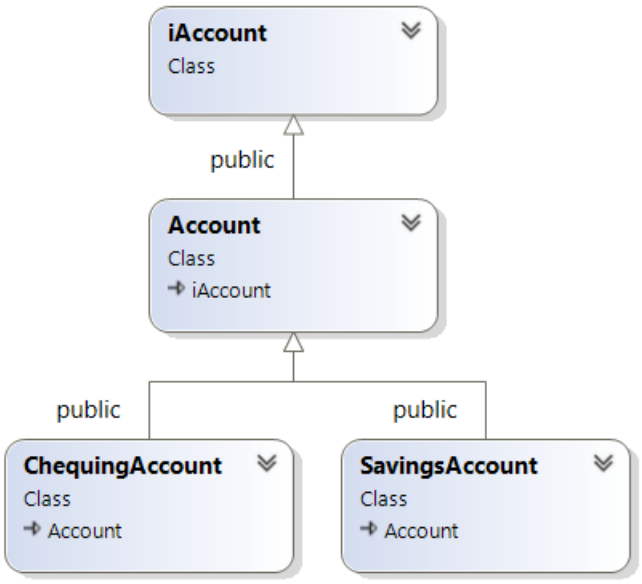
- *In-lab* portion submitted late, with *at-home* portion: **0** for *in-lab*. Maximum of **7**/10 for the entire workshop.
- If any of *in-lab*, *at-home* or *reflection* portions is missing, the mark for the workshop will be **0**/10.

Banking Accounts

In this workshop, you create an inheritance hierarchy for a bank that needs to represent the bank accounts of its clients. All clients at this bank can deposit (i.e., credit) money into their accounts and withdraw (i.e., debit) money from their accounts. Specific types of accounts 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., per a credit or a debit).

Class Hierarchy

The design of your `Account` hierarchy is illustrated in the Figure on the right. An interface named `iAccount` exposes the hierarchy's functionality to a client that uses its features. The abstract base class named `Account` holds the balance for an account, can credit and debit an account transaction and can expose the current balance in the account. Two classes derive from this base class. The `SavingsAccount` and `ChequingAccount` inherit the properties and functionality of the `Account` class.



IN-LAB (30%)

For the *in-lab* part of this workshop, you are to code three classes:

1. `iAccount`: the **interface** to the hierarchy – store it in a file named `iAccount.h`
2. `Account`: the **abstract** base class – store its definition and implementation in files named `Account.h` and `Account.cpp` respectively. In a separate file named `Allocator.cpp` code the function that allocates dynamic memory for an account based on its dynamic type.
3. `SavingsAccount`: a **concrete** class – store its definition and implementation in files named `SavingsAccount.h` and `SavingsAccount.cpp` respectively.

IACCOUNT INTERFACE

The `iAccount` interface includes a destructor and the following ***pure virtual*** public member functions:

`bool credit(double amount)` – adds a positive amount to the account balance. If the parameter is not a positive value, this function does nothing. This function returns `true` if the transaction was successful, `false` otherwise.

`bool debit(double amount)` – subtracts a positive amount from the account balance. If the parameter is not a positive value or if the parameter is bigger than the balance, this function does nothing. This function returns `true` if the transaction was successful, `false` otherwise.

`void monthEnd()` – debits any applicable monthly fees for the account

`void display(ostream& out) const` – inserts account information into the parameter.

ACCOUNT CLASS

The `Account` class derives from the `iAccount` interface, holds the current balance and includes the following public member functions:

`Account(double balance)`: constructor receives either a double holding the initial account balance or nothing. If the amount received is positive-valued, this function initializes the current account balance to the received amount. If the amount received is not positive-valued or no amount is received, this function initializes the current balance to 0.0.

`bool credit(double amount)`: overrides the function from the interface.
`bool debit(double amount)`: overrides the function from the interface.

The `Account` class includes the following protected member function:

`double balance() const`: returns the current balance of the account.

SavingsAccount Class

The `SavingsAccount` class derives from the `Account` class and holds the interest rate that applies to the account. This class includes the following public member functions:

`SavingsAccount(double balance, double interestRate)`: This custom constructor receives the initial account balance and the interest rate to be applied to the balance. If the interest rate received is positive-valued, this function stores the rate. If not, this function stores 0.0 as the rate to be applied.

`void monthEnd()`: This modifier calculates the interest earned on the current balance and credits the account with that interest.

`void display(ostream& out) const` – This function inserts in the stream received as parameter the content of the current instance in the following format:

```
Account type: Savings
Balance: $xxxx.xx
Interest Rate (%): x.xx
```

Allocator Module

The `Allocator` module defines the accounts rates and charges and the global function that creates the `Account` object for any type of account currently available in the hierarchy. For the in-lab portion, define a constant to hold the interest rate of 5%.

This module contains a single function

`iAccount* CreateAccount(const char* type, double balance)`: A helper function that receives a C-style string identifying the type of account and the initial account balance, dynamically creates the account with the starting balance and returns its address. If the string starts with `'S'`, then this function creates a `SavingsAccount` instance, and returns its address to the client, otherwise this function returns null.

In-Lab Main Module

```
#include <iostream>
#include <cstring>
#include "iAccount.h"

using namespace sict;
using namespace std;

// display inserts account information for client
//
void display(const char* client, iAccount* const acct[], int n)
{
  int lineLength = strlen(client) + 22;
  cout.fill('*');
  cout.width(lineLength);
  cout << "*" << endl;
  cout << "DISPLAY Accounts for " << client << ":" << endl;
  cout.width(lineLength);
  cout << "*" << endl;
  cout.fill(' ');
  for (int i = 0; i < n; ++i)
  {
    acct[i]->display(cout);
    if (i < n - 1)
      cout << "-----------------------" << endl;
  }
  cout.fill('*');
  cout.width(lineLength);
  cout << "**************************" << endl << endl;
  cout.fill(' ');
}

// close a client's accounts
//
void close(iAccount* acct[], int n)
{
  for (int i = 0; i < n; ++i)
  {
    delete acct[i];
    acct[i] = nullptr;
  }
}

int main()
{
  // Create Accounts for Angelina
  iAccount* Angelina[2];

  // initialize Angelina's Accounts
  Angelina[0] = CreateAccount("Savings", 400.0);
  Angelina[1] = CreateAccount("Savings", 400.0);
  display("Angelina", Angelina, 2);
```

```
  cout << "DEPOSIT $2000 into Angelina Accounts ..." << endl;
  for (int i = 0; i < 2; i++)
    Angelina[i]->credit(2000);

  cout << "WITHDRAW $1000 and $500 from Angelina's Accounts ... " << endl;
  Angelina[0]->debit(1000);
  Angelina[1]->debit(500);
  cout << endl;
  display("Angelina", Angelina, 2);

  Angelina[0]->monthEnd();
  Angelina[1]->monthEnd();
  display("Angelina", Angelina, 2);

  close(Angelina, 2);
}
```

IN-LAB EXPECTED OUTPUT

```
*****************************
DISPLAY Accounts for Angelina:
*****************************
Account type: Savings
Balance: $400.00
Interest Rate (%): 5.00
----------------------
Account type: Savings
Balance: $400.00
Interest Rate (%): 5.00
*****************************

DEPOSIT $2000 into Angelina Accounts ...
WITHDRAW $1000 and $500 from Angelina's Accounts ...

*****************************
DISPLAY Accounts for Angelina:
*****************************
Account type: Savings
Balance: $1400.00
Interest Rate (%): 5.00
----------------------
Account type: Savings
Balance: $1900.00
Interest Rate (%): 5.00
*****************************

*****************************
DISPLAY Accounts for Angelina:
*****************************
Account type: Savings
Balance: $1470.00
Interest Rate (%): 5.00
----------------------
Account type: Savings
```

```
Balance: $1995.00
Interest Rate (%): 5.00
****************************
```

IN-LAB SUBMISSION

To test and demonstrate execution of your program use the same data as the output example above.

If not on matrix already, upload `iAccount.h`, `Account.h`, `Account.cpp`, `Allocator.cpp`, `SavingsAccount.h`, `SavingsAccount.cpp`, and `w8_in_lab.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

Then, run the following command from your account (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace XXX, i.e., SAA, SBB, etc.):

~profname.proflastname/submit 244XXX_w8_lab<ENTER>

and follow the instructions.

IMPORTANT: Please note that a successful submission does not guarantee full credit for this workshop. If your professor is not satisfied with your implementation, your professor may ask you to resubmit. Resubmissions will attract a penalty.

AT-HOME (30%)

For the at-home portion of the workshop, you will add a new class to the hierarchy. Copy the `iAccount.h`, `Account.h`, `Account.cpp`, `Allocator.cpp`, `SavingsAccount.h`, and `SavingsAccount.cpp` files from the in-lab portion of your solution.

CHEQUINGACCOUNT CLASS:

The `ChequingAccount` class derives from the `Account` class and holds the transaction fee and month-end fee to be applied to the account. This class includes the following public member functions:

`ChequingAccount(double balance, double transFee, double monthlyFee)`: constructor receives the initial account balance, the transaction fee to be applied to the balance and the month-end fee to be applied to the account. If the transaction fee received is positive-valued, this function stores the fee. If not, this function stores 0.0 as the fee to be applied. If the monthly fee received is positive-valued, this function stores the fee. If not, this function stores 0.0 as the fee to be applied.

`bool credit(double amount)`: this modifier credits the balance by the amount received and if successful debits the transaction fee from the balance. This function returns `true` if the transaction succeeded; `false` otherwise.

`bool debit(double amount)`: this modifier debits the balance by the amount received and if successful debits the transaction fee from the balance. This function returns `true` if the transaction succeeded; `false` otherwise.

`void monthEnd()`: this modifier debits the monthly fee from the balance.

`void display(ostream& out) const`: Inserts in the stream the content of the current instance in the following format:

```
Account type: Chequing
Balance: $xxxx.xx
Per Transaction Fee: x.xx
Monthly Fee: x.xx
```

ALLOCATOR MODULE

The `Allocator` module pre-defines the accounts rates and charges and defines the global function that creates the `Account` object from the types of account currently available. The rates and charges are:

- interest rate 5%
- transaction fee 0.50
- monthly fee 2.00

Modify the allocation function to include the following specification:

`iAccount* CreateAccount(const char* type, double balance)`: this function receives the address of a C-style string that identifies the type of account to be created and the initial balance in the account and returns its address to the calling function. If the initial character of the string is `'S'`, this function creates a savings account in dynamic memory. *If the initial character of the string is* `'C'`*, this function creates a chequing account in dynamic memory.* If the string does not identify a type that is available, this function returns null.

AT-HOME MAIN MODULE

```
#include <iostream>
#include <cstring>
#include "iAccount.h"

using namespace sict;
using namespace std;

// display inserts account information for client
//
void display(const char* client, iAccount* const acct[], int n)
{
  int lineLength = strlen(client) + 22;
  cout.fill('*');
  cout.width(lineLength);
  cout << "*" << endl;
  cout << "DISPLAY Accounts for " << client << ":" << endl;
  cout.width(lineLength);
  cout << "*" << endl;
  cout.fill(' ');
  for (int i = 0; i < n; ++i)
  {
    acct[i]->display(cout);
    if (i < n - 1)
      cout << "-----------------------" << endl;
  }
  cout.fill('*');
  cout.width(lineLength);
  cout << "**************************" << endl << endl;
  cout.fill(' ');
}

// close a client's accounts
//
```

```
void close(iAccount* acct[], int n)
{
  for (int i = 0; i < n; ++i)
  {
    delete acct[i];
    acct[i] = nullptr;
  }
}

int main ()
{
  // Create Accounts for Angelina
  iAccount* Angelina[2];

  // initialize Angelina's Accounts
  Angelina[0] = CreateAccount("Savings", 400.0);
  Angelina[1] = CreateAccount("Chequing", 400.0);
  display("Angelina", Angelina, 2);

  cout << "DEPOSIT $2000 into Angelina Accounts ..." << endl ;
  for(int i = 0 ; i < 2 ; i++)
    Angelina[i]->credit(2000);

  cout << "WITHDRAW $1000 and $500 from Angelina's Accounts ... " << endl ;
  Angelina[0]->debit(1000);
  Angelina[1]->debit(500);
  cout << endl;
  display("Angelina", Angelina, 2);

  Angelina[0]->monthEnd();
  Angelina[1]->monthEnd();
  display("Angelina", Angelina, 2);

  close(Angelina, 2);
}
```

AT-HOME EXPECTED OUTPUT

```
*****************************
DISPLAY Accounts for Angelina:
*****************************
Account type: Savings
Balance: $400.00
Interest Rate (%): 5.00
-----------------------
Account type: Chequing
Balance: $400.00
Per Transaction Fee: 0.50
Monthly Fee: 2.00
*****************************

DEPOSIT $2000 into Angelina Accounts ...
WITHDRAW $1000 and $500 from Angelina's Accounts ...
```

```
****************************
DISPLAY Accounts for Angelina:
****************************
Account type: Savings
Balance: $1400.00
Interest Rate (%): 5.00
----------------------
Account type: Chequing
Balance: $1899.00
Per Transaction Fee: 0.50
Monthly Fee: 2.00
****************************

****************************
DISPLAY Accounts for Angelina:
****************************
Account type: Savings
Balance: $1470.00
Interest Rate (%): 5.00
----------------------
Account type: Chequing
Balance: $1897.00
Per Transaction Fee: 0.50
Monthly Fee: 2.00
****************************
```

REFLECTION (40%)

Study your final solution, reread the related parts of the course notes, and make sure that you have understood the concepts covered by this workshop. **This should take no less than 30 minutes of your time.**

Create a file named `reflect.txt` that contains your **detailed description of the topics that you have learned** in completing this workshop and mention any issues that caused you difficulty. Include in your explanation—**but do not limit it to**—the following points:

1. What is the difference between an abstract base class and a concrete class?
2. Take a look to the main module—how is it possible that this module can work with chequing and savings accounts but doesn't know about the `SavingsAccount` or `ChequingAccount` classes?
3. Note that the interface has a destructor, even if doesn't store any attributes. Why is that destructor necessary?
4. In the context of a hierarchy of classes, what is the difference between overloading, overriding and shadowing?

Quiz Reflection

Add a section to `reflect.txt` called Quiz X Reflection. Replace the X with the number of the last quiz that you received and list the numbers of all questions that you answered incorrectly.

Then for each incorrectly answered question write your mistake and the correct answer to that question. If you have missed the last quiz, then write all the questions and their answers.

AT-HOME SUBMISSION

To submit the *at-home* section, demonstrate execution of your program with the exact output as in the example above.

Upload `reflect.txt`, `iAccount.h`, `Account.h`, `Account.cpp`, `Allocator.cpp`, `SavingsAccount.h`, `SavingsAccount.cpp`, `ChequingAccount.h`, `ChecquingAccount.cpp` and `w8_at_home.cpp` to your matrix account. Compile and run your code and make sure everything works properly.

To submit, run the following command from your account (use your professor's Seneca userid to replace `profname.proflastname`, and your section ID to replace `XXX`, i.e., SAA, SBB, etc.):

~profname.proflastname/submit 244XXX_w8_home<ENTER>

and follow the instructions.

IMPORTANT: 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.