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. 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 that is two days before your next scheduled workshop (23:59:59).

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:

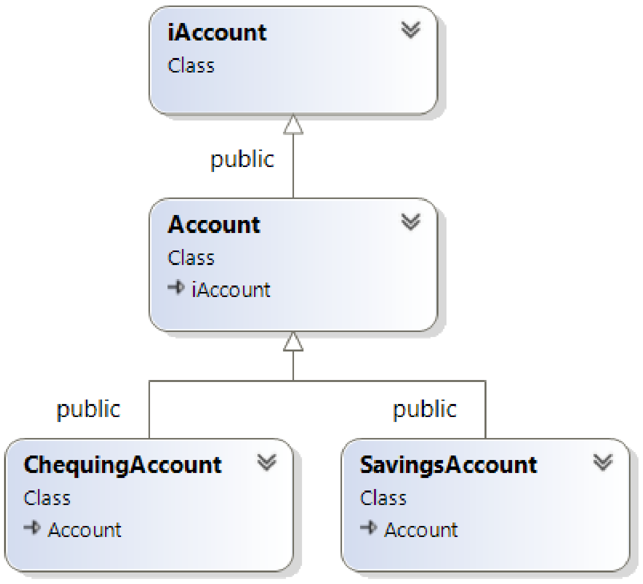
- *In-lab* portion submitted late, with *at-home* portion: **0** for *in-lab*. Maximum of 7/10 for the entire workshop.
- If any of *in-lab*, *at-home* or *reflection* portions is missing, the mark for the workshop will be **0**/10.

## INTRODUCTION – 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 the following pure virtual public member functions:

- `bool credit(double)` – adds a positive amount to the account balance
- `bool debit(double)` – subtracts a positive amount from the account balance
- `void monthEnd()` – applies month-end transactions to the account
- `void display(std::ostream&) const` – inserts account information into an `ostream` object

This interface declares the following helper function:

- `iAccount* CreateAccount(const char*, double)` – receives a C-style string identifying the type of account and the initial account balance, creates the account with the starting balance and returns its address.

## Account Class:

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

- `Account(double)` – 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)` – receives an amount to be credited (added) to the account balance and returns the success of the transaction. If the amount received is positive-valued, the transaction succeeds and this function adds the amount received to the current balance. If the amount is not positive-valued, the transaction fails and this function does not add the amount received.
- `bool debit(double)` – receives an amount to be debited (subtracted) from the account balance and returns the success of the transaction. If the amount received is positive-valued, the transaction succeeds and this function subtracts the amount

received from the current balance. If the amount is not positive-valued, the transaction fails and this function does not s the amount received.

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

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

## Account Class - 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. The rates and charges are

- interest rate 0.05 (5%)
- other rates and charges added with other accounts

The allocator function has the following specification:

- `iAccount* CreateAccount (const char*, double)` – 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 string does not identify a type that is available, this function returns nullptr.

You will upgrade this module as you add new account types to the hierarchy with corresponding rates and charges.

## 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, double)` – constructor receives a double holding the initial account balance and a double holding 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(std::ostream&) const` – receives a reference to an `ostream` object and inserts the following output on separate lines to the object. The values marked in red are fixed format with 2 decimal places and no fixed field width:
    - `Account type: Savings`
    - `Balance: $xxxx.xx`
    - `Interest Rate (%): x.xx`

The following source code uses your hierarchy:

```
// Workshop 8 - Virtual Functions and Abstract Base Classes
// File: w8_in_lab.cpp
// Version: 2.0
// Date: 2017/12/11
// Author: Chris Szalwinski, based on previous work by Heidar Davoudi
// Description:
// This file tests in_lab section of your workshop
///////////////////////////////////////////////////

#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);
}
```

The following is the exact output of this tester program:

```
******************************
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

If not on matrix already, upload **iAccount.h**, **Account.h**, **Account.cpp.**, **Allocator.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 244_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%):

This part of Wworkshop 8 adds a **ChequingAccount** to your **Account** hierarchy.

Copy the **iAccount.h**, **Account.h**, **Account.cpp.**, **Allocator.cpp**, **SavingsAccount.h,** and **SavingsAccount.cpp** files from the in_lab part 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, double, double)` – constructor receives a double holding the initial account balance, a double holding the transaction fee to be applied to the balance and a double holding 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)` – 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)` – 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(std::ostream&) const` – receives a reference to an `ostream` object and inserts the following output on separate lines to the object. The values marked in red are fixed format with 2 decimal places and no fixed field width:
  - `Account type: Chequing`
  - `Balance: $xxxx.xx`
  - `Per Transaction Fee: x.xx`
  - `Monthly Fee: x.xx`

## Account Class - 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 0.05 (5%)
- transaction fee 0.50
- monthly fee 2.00

Modify the allocation function to include the following specification:

- `iAccount* CreateAccount (const char*, double)` – 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 nullptr.

The following code uses your hierarchy:

```
// Workshop 8 - Virtual Functions and Abstract Base Classes
// File: w8_at_home.cpp
// Version: 2.0
// Date: 2017/12/11
// Author: Chris Szalwinski, based on previous work by Heidar Davoudi
// Description:
// This file tests at_home section of your workshop
///////////////////////////////////////////////////

#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);
}
```

The following is the exact output of the tester program:

```
******************************
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: $1896.50
Per Transaction Fee: 0.50
Monthly Fee: 2.00
******************************
```

## REFLECTION (40%)

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. Identify the functions that shadow functions of the same name in your solution?
3. Explain what have you learned in this workshop.

### 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

If not on matrix already, upload **w8_at_home.cpp**, **iAccount.h**, **Account.h**, **Account.cpp, Allocator.cpp, SavingsAccount.h, SavingsAccount.cpp, ChequingAccount.h, ChequingAccount.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 244_w8_home <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.