COMP 2404 – Assignment #1

<u>Due:</u> Thursday, October 1 at 11:59 pm

1. Goal

For this assignment, you will write a C++ program to manage the customers of a bank and their bank accounts, and your program will process debit and credit transactions to those accounts. You will practice writing correctly designed code with simple classes in C++, as well as working with dynamically allocated memory.

2. Learning Outcomes

With this assignment, you will:

- write correctly designed code with simple C++ classes, and implement a collection class
- work with dynamically allocated memory and pointers
- write and package a program following standard C++98 and Unix programming conventions

3. Instructions

Your program will implement several new classes representing a bank, and its customers and accounts. The `Bank` object will store a master collection of `Customer` objects, as well as a master collection of `Account` objects. Each customer will have its own separate collection of the accounts that they own. You will note that we will **not** be making copies of the `Account` objects, as that would be a poor design choice.

This program requires the implementation of several new classes. All new classes must follow the implementation conventions that we saw in the course lectures, including but not restricted to the correct separation of code into header and source files, the use of include guards, basic error checking, and the documentation of each class in the class header file. A more comprehensive list of constraints can be found in the Constraints section below.

3.1. **Implement the `Account` class**

You will begin by creating a new class called `Account`. For simplicity, you can start with an existing class from the in-class coding examples posted in *cuLearn*. For example, you can use the `Book` class from section 1.5, program #4, but you will have to make all appropriate changes, including renaming the class and the files.

The `Account` class will contain the following data members:

3.1.1. the account number, which will be a integer

3.1.2. the account balance, which will be a float

3.1.3. the owner of the account, which will be a **pointer** to a `Customer` object; we will be creating the `Customer` class in a future step

The `Account` class will contain the following member functions:

3.1.4. a default constructor that takes an account number and balance as parameters, and initializes all the data members; make sure to specify default values for the parameters

3.1.5. a getter function for the account number

3.1.6. a setter function for the `Customer` pointer

3.1.7. a `bool debit(float)` function that takes an amount as parameter, and deducts that amount from the account balance; this function returns true if no errors occurred, and false otherwise

3.1.8. a `bool credit(float)` function that takes an amount as parameter, and adds that amount to the account balance; this function returns true if no errors occurred, and false otherwise

3.1.9. a `void print()` function that prints to the screen all the account information, including the customer id of the account owner

3.2. **Implement the `Customer` class**

You will implement a new `Customer` class that contains the following data members:

3.2.1. the customer id, which will be a integer

3.2.2. the customer name, which will be a C++ standard library `string` object

3.2.3. the collection of the bank accounts owned by the customer
- (a) this will be a *statically allocated array* of `Account` object **pointers**
- (b) you will define a preprocessor constant for the maximum number of accounts; this can be set to a reasonable number such as 16
- (c) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

3.2.4. the current number of accounts in the array

The `Customer` class will contain the following member functions:

3.2.5. a default constructor that takes a customer id and name as parameters, and initializes all the data members necessary
- (a) it is **not** necessary to initialize each array element to null, since your code should never access any element beyond the current number of accounts
- (b) it is bad form to initialize data members in the class definition; you must do this in the body of the constructor instead

3.2.6. a getter function for the customer id

3.2.7. a `bool addAcct(Account*)` function that adds the given account to the *back* (the end) of the accounts array; this function returns true if no errors occurred, and false otherwise

3.2.8. a `void print()` function that prints to the screen all the customer information, including all the information for each of the customer's accounts; using correct design principles, this function must call an existing function on each `Account` object

Note: The `Customer` and `Account` classes have a *bidirectional association* between them, which means that each instance of one class has one or more instances of the other class. This creates a *circular dependency* between the two class header files that requires a special technique to resolve. You can specify `#include "Customer.h"` in your `Account` header file, and then use a *forward reference* such as `class Account;` at the top of your `Customer` header file. Please note that, while your code will not compile with a `#include "Account.h"` statement in your `Customer` header file, you will likely have to specify that `#include` statement in your `Customer` source file.

3.3. **Implement the `CustArray` class**

You will implement a new `CustArray` collection class. The only job of any collection class is to hold data and provide operations to manage the data in the collection. Any data-specific operations (in this case, `Customer` specific operations) will **not** be implemented inside the collection class, but in the data class (`Customer`) instead.

Your `CustArray` class will contain the following data members:

3.3.1. the elements in the collection
- (a) this will be a statically allocated array of `Customer` object **pointers**
- (b) you will define a preprocessor constant for the maximum number of customers; this can be set to a reasonable number such as 64
- (c) you can refer to the coding example in section 1.6, program #5, for examples of the four different kinds of arrays

3.3.2. the current number of elements in the collection

Your `CustArray` class will contain the following member functions:

3.3.3. a default constructor that initializes the data members necessary

3.3.4. a destructor that deallocates the dynamically allocated customers contained in the array

3.3.5. a `bool add(Customer*)` function that adds the given customer to the back of the array; this function returns true if no errors occurred, and false otherwise

3.3.6. a `void find(int id, Customer** c)` function that searches the array for the customer with the id specified in parameter `id`, and returns that customer pointer using parameter `c`; you **must** use the parameter to return this data; do not use the return value

3.3.7. a `void print()` function that prints to the screen every `Customer` object contained in the array; using correct design principles, this function must call an existing function on each `Customer` object

3.4. Implement the `Bank` class

You will implement a new `Bank` class that stores two master collections: one collection with all the customers in the bank, and one with all the accounts.

The `Bank` class will contain the following data members:

3.4.1. the bank name, stored as a `string`

3.4.2. the collection of customers in the bank, stored as a `CustArray` object

3.4.3. the collection of accounts in the bank, stored as a statically allocated array of `Account` pointers

3.4.4. the current number of accounts in the accounts array

The `Bank` class will contain the following member functions:

3.4.5. a default constructor that takes a bank name as parameter, and initializes the necessary data members

3.4.6. a destructor that deallocates the dynamically allocated accounts contained in the accounts array

3.4.7. a `bool addCust(Customer*)` function that reuses an existing function to add the given customer to the customers array; the function returns true if the customer was successfully added, and false otherwise

3.4.8. a `bool addAcct(int custId, Account* acct)` function that does the following:
- (a) find the customer owner of the account, which is the `Customer` object with the given customer id; if the customer owner is not found, the account cannot be added
- (b) perform all relevant error checking
- (c) add the given account to the back of the bank's accounts array
- (d) add the given account to the customer owner's accounts array
- (e) set the given account's owner to the customer owner object
- (f) return true if the account was successfully added, and false otherwise

3.4.9. a `bool debit(int acctNum, float amount)` function that finds the `Account` object corresponding to the given account number, and debits the given amount from that account

3.4.10. a `bool credit(int acctNum, float amount)` function that finds the `Account` object corresponding to the given account number, and credits the given amount to that account

3.4.11. a `void print()` function that prints all the accounts and all the customers in the bank

Note #1: You must use correct design principles in the implementation of all the above functions. This means that you must reuse existing functions everywhere possible, and perform all error checking.

Note #2: Do not implement any addtional getter or setter functions, as these are not necessary. However, you may implement other, additional "helper" functions as needed.

3.5. Write the `main()` function

Your `main()` function must test your program thoroughly. To do so, it will declare a bank object with an appropriate name, it will initialize several **different** customers and accounts, it will process several transactions (debits and credits) to different accounts, and it will print the bank data to the screen.

Your program will use two *global functions* to initialize the bank data and to process transactions. You will use the `void initBank(Bank&)` and `void processTransactions(Bank&)` global functions that are posted in *cuLearn* in the `a1-posted.cc` file.

Your `main()` function will do the following:

3.5.1. declare a bank object with an appropriate name

3.5.2. initialize the bank data by calling the `initBank()` function provided in the posted code

3.5.3. print out the bank data, including all account and customer information

3.5.4. perform some banking transactions by calling the `processTransactions()` function provided in the posted code

3.5.5. print out the bank data again

3.6. **Packaging**

Every assignment in this course is required to follow the conventional packaging rules for Unix-based systems:

3.6.1. Your code must be correctly separated into header and source files, as seen in class.

3.6.2. You must provide a Makefile that compiles and links all your code into a working executable.

(a) **do not** use a generic Makefile; it must be specific to this program

3.6.3. You must provide a README file that contains a preamble (program author, purpose, list of source, header, data files), as well as compiling and launching instructions.

3.6.4. **Do not submit** any additional files, including object files, executables, or supplementary files or directories (macOS users must remove their additional hidden directories).

3.7. **Test the program**

You must provide code that tests your program thoroughly. For this program, the use of the provided global functions will be sufficient. Specifically:

3.7.1. Make sure that the data you provide exercises all your functions thoroughly. Failure to do this will result in major deductions, *even if the program appears to be working correctly*.

3.7.2. Check that the bank information is correct when it is printed at the end of the program.

3.7.3. Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

4. Constraints

Your program must comply with all the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

4.1. The code must be written in C++98, and it must compile and execute in the default course VM. It must not require the installation of libraries or packages or any software not already provided in the default VM.

4.2. Your program must not use any classes, containers, or algorithms from the C++ standard template library (STL), unless explicitly permitted in the instructions.

4.3. Your program must follow basic OO programming conventions, including the following:

4.3.1. Do not use any global variables or any global functions other than the ones explicitly permitted.

4.3.2. Do not use `struct`s. You must use classes instead.

4.3.3. Objects must always be passed by reference, never by value.

4.3.4. Existing functions must be reused everywhere possible.

4.3.5. All basic error checking must be performed.

4.3.6. All dynamically allocated memory must be explicitly deallocated.

4.4. All classes must be thoroughly documented in every class definition, as indicated in the course material, section 1.3.

5. Submission

5.1. You will submit in *cuLearn*, before the due date and time, the following:

5.1.1. One `tar` or `zip` file that includes:

- (a) all source and header files
- (b) a Makefile
- (c) a README file that includes:
  - (i) a preamble (program author, purpose, list of source and header files)
  - (ii) compilation and launching instructions

NOTE: Do not include object files, executables, hidden files, or duplicate files or directories in your submission.

5.2. Late submissions will be subject to the late penalty described in the course outline. No exceptions will be made, including for technical and/or connectivity issues. Do not wait until the last day to submit.

5.3. Only files uploaded into *cuLearn* will be graded. Submissions that contain incorrect, corrupt, or missing files will receive a grade of zero. Corrections to submissions will not be accepted after the due date and time, for any reason.

6. Grading

6.1. **Marking components:**

- 16 marks: correct implementation of `Account` class
- 14 marks: correct implementation of `Customer` class
- 22 marks: correct implementation of `CustArray` class
- 43 marks: correct implementation of `Bank` class
- 5 marks: correct implementation of `main()` function

6.2. **Execution requirements:**

6.2.1. all marking components must be called and execute successfully in order to earn marks

6.2.2. all data handled must be printed to the screen for marking components to earn marks

6.3. **Deductions:**

6.3.1. Packaging errors:

- (a) 10 marks for missing Makefile
- (b) 5 marks for missing README
- (c) up to 10 marks for failure to correctly separate code into header and source files
- (d) up to 10 marks for bad style or missing documentation

6.3.2. Major design and programming errors:

- (a) 50% of a marking component that uses global variables or `struct`s
- (b) 50% of a marking component that consistently fails to use correct design principles
- (c) 50% of a marking component that uses prohibited library classes or functions
- (d) up to 100% of a marking component where Constraints listed are not followed
- (e) up to 10 marks for bad style
- (f) up to 10 marks for memory leaks

6.3.3. Execution errors:

- (a) 100% of any marking component that cannot be tested because:
  - (i) it doesn't compile or execute in the course VM, or
  - (ii) the feature is not used in the code, or
  - (iii) data cannot be printed to the screen, or
  - (iv) insufficient datafill is provided.