Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2019-2020

Programming Assignment 3

Due date: 30 April 2020, Thursday, 23:55

1 Objectives

This homework aims to help you get familiar with the fundamental C++ programming concepts.

Keywords: *Constructor/Copy Constructor/Destructor, Assignment/Move, Operator Overloading, Memory Management*

2 Problem Definition

Summary of HW3: Implement the methods in the given 3 classes (Transaction, Account, Bank). Banks have Accounts and Accounts have Transactions. You can find more detail in their respective sections.

In this homework you are a programmer and your supervisors want certain features in their Bank system. Your task is to implement Transaction, Account and Bank systems that are provided you with a header file(You will not edit header files). As situation requested shiny features of the latest C++ is not available to you. Therefore, you have to be careful with your programs memory management.

3 Class Definitions

3.1 Transaction

Transaction is the most basic class in this homework. Basically, it holds a certain amount and date of the Transaction.

```
class Transaction {
private:
    double _amount;
    time_t _date;



public:
    /**
     * Empty constructor
     * give -1 to everything
     */
    Transaction();
    /**
     * Constructor
     *
     * @param amount The value of Transaction(Can be negative or positive)
     * @param date Transaction date
    */
    Transaction(double amount, time_t date);
    /**
     * Copy Constructor.
     *
     * @param rhs The Transaction to be copied.
     */
    Transaction(const Transaction& rhs);

    /**
     * Compare two Transaction based on their date
     *
     * @param rhs Compared Transaction
     * @return If current Transaction happened before the given Transaction ↩
        return true
     * else return false
     */
    bool operator<(const Transaction& rhs) const;
    /**
     * Compare two Transaction based on their date
     *
     * @param date Compared date
     * @return If current Transaction happened after the given Transaction return↩
        true
     * else return false
     */
    bool operator>(const Transaction& rhs) const;
    /**
     * Compare a Transaction with a given date
     *
     * @param date Compared date
```

```
     * @return If current Transaction happened before the given date return true
     * else return false
     */
    bool operator<(const time_t date) const;
    /**
     * Compare a Transaction with a given date
     *
     * @param date Compared date
     * @return If current Transaction happened after the given date return true
     * else return false
     */
    bool operator>(const time_t date) const;


    /**
     * Sum the value of two Transaction amounts
     *
     * @param rhs The transaction to sum over
     * @return The output of the summation in double format
     */
    double operator+(const Transaction& rhs);

    /**
     * Sum the value of a Transaction with another double
     *
     * @param add The amount to sum over
     * @return The output of the summation in double format
     */
    double operator+(const double add);


    /**
     * Assignment operator
     *
     * @param rhs Transaction to assign
     * @return this Transaction
     */
    Transaction& operator=(const Transaction& rhs);

    /**
     * Stream overload
     *
     * What to stream:
     * Transaction amount"tab-tab"hour:minute:second-day/month/year(in localtime)
     *
     * @param os Stream to be used.
     * @param transaction Transaction to be streamed.
     * @return the current Stream
     */
    friend std::ostream& operator<<(std::ostream& os, const Transaction& ↩
       transaction);


};
```

3.2 Account

The account is defined for a single user and users have their respective ids. Accounts also hold Transaction information of their user in a sorted manner.

```
class Account {
private:
    int _id;
    Transaction** _activity;
    int* _monthly_activity_frequency;


public:
    /**
     * Empty constructor
     * give the id as -1
     * give nullptr for pointers
     */
    Account();
    /**
     * Constructor
     *
     *
     * Note: The given activity array will have 12 Transaction*
     * Each of these Transaction* will represent a month from the 2019
     * Basicaly activity[0] will represent January
     *          activity[11] will represent February
     *          activity[11] will represent March
     *                      ...
     *          activity[10] will represent November
     *          activity[11] will represent December
     * activity[0] will only contain Transactions happened in January
     * However, be careful that Transactions inside of activity[i] will not be in↩
         sorted order
     * For Example: We are certain that activity[0] is containing Transactions ↩
        happened in January 2019
     * But we are not sure which of them happened first.
     * I strongly suggest you to use a sorting algorithm while storing these ↩
        Transaction to your object.
     * (Sorting by the date, So that you can directly use them in stream overload↩
        )
     * (You can use bubble sort)
     *
     * @param id id of this Account
     * @param activity 2d Transaction array first layers lenght is 12 for each ↩
        month
     * @param monthly_activity_frequency how many transactions made in each month
    */
    Account(int id, Transaction** const activity, int* monthly_activity_frequency↩
        );

    /**
     * Destructor
     *
     * Do not forget to free the space you have created(This assignment does not ↩
        use smart pointers)
     */
```

```
~Account();

/**
 * Copy constructor(Deep copy)
 *
 * @param other The Account to be copied
 */
Account(const Account& rhs);

/**
 * Copy constructor(Deep copy)
 *
 * This copy constructors takes two time_t elements
 * Transactions of the old Account will be copied to new Account
 * if and only if they are between these given dates
 * Given dates will not be included.
 *
 * @param rhs The Account to be copied
 * @param start_date Starting date for transaction to be copied.
 * @param end_date Ending date for transactions to be copied.
 */
Account(const Account& rhs, time_t start_date, time_t end_date);

/**
 * Move constructor
 *
 * @param rhs Account which you will move the resources from
 */
Account(Account&& rhs);

/**
 * Move assignment operator
 *
 * @param rhs Account which you will move the resources from
 * @return this account
 */
Account& operator=(Account&& rhs);

/**
 * Assignment operator
 * deep copy
 *
 * @param rhs Account to assign
 * @return this account
 */
Account& operator=(const Account& rhs);
/**
 * Equality comparison overload
 *
 * This operator checks only id of the Account
 *
 * @param rhs The Account to compare
 * @return returns true if both ids are same false othervise
 */

bool operator==(const Account& rhs) const;
/**
```

```
 * Equality comparison overload
 *
 * This operator checks only id of the Account
 *
 * @param id to compare
 * @return returns true if both ids are same false othervise
 */
bool operator==(int id) const;


/**
 * sum and equal operator
 * Add Transactions of two Accounts
 * You have to add transactions in correct places in your _activity array
 * Note: Remember that _activity[0] is always January and _activity[11] is ↩
    always December
 * (This information also holds for every other month)
 *
 * You can have Transactions with the same date
 *
 * @param rhs Account which take new Transactions from
 * @return this Account after adding new Transactions
 */
Account& operator+=(const Account& rhs);

/**
 * How much money Account has(Sum of Transaction amounts)
 *
 *
 * @return total amount of the money of the account
 */
double balance();

/**
 * How much money Account has at the end of given date
 *
 * Given date will not be included.
 * @param end_date You will count the amounts until this given date(not ↩
    inclusive)
 * @return Total amount the Account has until given date
 */
double balance(time_t end_date);


/**
 * How much money Account between given dates
 * Given dates will not be included.
 *
 * @param end_date You will count the amounts between given dates(not ↩
    inclusive)
 * @return Total amount the Account has between given dates
 * You will only count a Transaction amount if and only if it occured between↩
    given dates
 */
double balance(time_t start_date, time_t end_date);
```

```
    /**
     * Stream overload.
     *
     *
     *
     * What to stream
     * Id of the user
     * Earliest Transaction amount"tab"-"tab"hour:minute:second-day/month/year(in↩
        localtime)
     * Second earliest Transaction amount"tab"-"tab"hour:minute:second-day/month/↩
        year(in localtime)
     * ...
     * Latest Transaction amount"tab-tab"hour:minute:second-day/month/year(in ↩
        localtime)
     *
     * Note: _activity array will only contain dates from January 2019 to ↩
        December 2019
     * Note: Transactions should be in order by date
     * Note: either of _monthly_activity_frequency or _activity is nullptr
     * you will just stream
     * -1
     * @param os Stream to be used.
     * @param Account to be streamed.
     * @return the current Stream
     */

    friend std::ostream& operator<<(std::ostream& os, const Account& account);
};
```

3.3 Bank

The bank keeps track of accounts.

```
class Bank {
private:
    std::string _bank_name;
    int _user_count;
    Account* _users;
public:

    /**
     * Empty constructor
     * give the bank_name as "not_defined"
     * give nullptr for pointers
     * give 0 as _users_count
     */
    Bank();
    /**
     * Constructor
     *
     *
     * @param bank_name name of this bank
     * @param users pointer to hold users of this bank
     * @param user_count number of users this bank has
```

```
*/
Bank(std::string bank_name, Account* const users, int user_count);
/**
 * Destructor
 *
 * Do not forget to free the space you have created(This assignment does not use smart pointers)
 */
~Bank();
/**
 * Copy constructor(Deep copy)
 *
 * @param rhs The Bank to be copied
 */
Bank(const Bank& rhs);
/**
 * You should deep copy the content of the second bank
 * Merge two banks
 * If both banks has a user with the same id, Transactions of these users will be merged in to the same Account
 * For example:
 * Bank1 has [1,2] id users
 * Bank2 has [2,3] id users
 *
 * Bank1 after += operator will have [1,2,3] id users
 * User with id 2 will have its transactions histories merged
 *
 * Transactions with of the users with the same id should be merged and updated
 * @param rhs Merged Bank
 * @return this Bank
 */
Bank& operator+=(const Bank& rhs);

/**
 * Add a new account to Bank
 *
 * If the newly added user already exists in this Bank merge their Transactions
 *
 * @param new_acc new account to add to bank
 * @return this Bank
 */
Bank& operator+=(const Account& new_acc);

/** Indexing operator overload
 *
 * Return the Account with the given id
 *
 * If there is no Account with the given id return the first element
 *
 * @param account_id id of the Account
 * @return if given id exist in the bank return the account, else return the first account
 *
 */
Account& operator[](int account_id);
```

```
    /**
     * Stream overload.
     * all the accounts will be between 01-01-2019 and 31-12-2019
     * What to stream
     * bank_name"tab"number of users who are eligible for a loan"tab"total balance of the bank
     *
     * A user is safe for a loan if and only if that user did not have any negative balance for 2 or more consecutive months
     * For example, let's say our bank named as "banana" has two users
     *
     * User A's balance for each month is as given
     *
     * January - 0
     * February - 0
     * March - 100
     * April - -20
     * May - -30
     * June - 40
     * July - 60
     * August - 0
     * September - 0
     * October - 0
     * November - 0
     * December - 0
     *
     * This user is not eligible because in April and May his/her balance was negative(consecutive)
     * You still have to add 150 to the total balance of the bank
     * User B's balance for each month is as given
     *
     * January - 0
     * February - 0
     * March - 100
     * April - -20
     * May - 40
     * June - -30
     * July - 60
     * August - 0
     * September - 0
     * October - 0
     * November - 0
     * December - 0
     *
     * This user is eligible because negative balances were not consecutive
     * You will also add 150 to the total balance of the bank
     *
     * your output will be as
     * banana   1   300
     */
    friend std::ostream& operator<<(std::ostream& os, const Bank& bank);
};
```

4 Extras

You have to check for memory leaks in your code. Any memory leak in certain class will result in point reduction.

You can test your code for any memory leak by using valgrind. (Provided MakeFile has valgrind option)

You can also test your implementation by using transaction_test.cpp, account_test.cpp, bank_test.cpp, and compare your results with provided corresponding example_*.txt files. (Some text editors might display tab character in a different format. To look at the output in the best way use gedit in Ubuntu, TextEdit in Mac or just open it with vi)

Note: You can test your classes with (ouputs of these runs also given to you)

```
$ make transaction
$ make run
$ make valgrind
$ make account
$ make run
$ make valgrind
$ make bank
$ make run
$ make valgrind
```

5 Grading

- Full grade for Transaction class implementation **15** points.
- Full grade for Account class implementation **60** points.
- Full grade for Bank class implementation **25** points.

To get a full grade from each part your code should not have any memory leak. This will be checked with Valgrind. If any of your classes have memory leak your grade will be reduced by 10 points from the overall points you get for that class. (Let say your Account class has memory leak because of some of your functions, in this case even if all your functions run correctly and passes all the tests you will get 50) If your code gives segmentation error in any part of the testing process you will not be graded for the remaining parts of the test.

Please remember that these classes require each other to function properly. If your Transaction functions have problems, your Account and Bank Functions can also have problems. This will be effective in the testing process. For example, if your Account constructor does not work in a correct way(can be segmentation or any other run time error) we can not grade you for remaining functions.

6 Regulations

- **Programming Language:** You must code your program in C++ (11). Your submission will be compiled with `g++` with `-std=c++11` flag on department lab machines.
- **Allowed Libraries:** You can only use libraries provided inside of headers. Use of any other library (especially the external libraries found on the internet) is forbidden.
- **Memory Management:** When an instance of a class is destructed, the instance must free all of its owned/used heap memory. Any heap block, which is not freed at the end of the program will result in grade deduction. Please check your codes using valgrind –leak-check=full for memory-leaks.

- **Late Submission:** You have a total of 10 days for late submission. You can spend this credit for any of the assignments or distribute it for all. For each assignment, you can use at most 3 days-late.
- **Cheating:** In case of cheating, the university regulations will be applied.
- **Newsgroup:** It's your responsibility to follow the cow forums for discussions and possible updates on a daily basis.

7 Submission

Submission will be done via CengClass. Create a zip file named `hw3.zip` that contains:

- Transaction.cpp
- Account.cpp
- Bank.cpp

Do not submit a file that contains a `main` function. Such a file will be provided and your code will be compiled with it. Also, do not submit a Makefile.

Note: The submitted zip file should not contain any directories! The following command sequence is expected to run your program on a Linux system:

```
$ unzip hw3.zip
$ make clean
$ make all
$ make run
$ -optional- make valgrind
```