Middle East Technical University | Department of Computer Engineering

CENG 242

Programming Language Concepts

Spring '2017-2018

Programming Assignment 3

Due date: 27 April 2017, Friday, 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

TL;DR: Implement the methods in the given 3 classes (Koin, Blockchain, Miner). Blockchains have Koins, Miners have Blockchains. The details are on their way.

....
Searching for a hyped keyword to construct the homework upon...
Found 1 result(s): Blockchain.
...
Initiating environment...
Kicking in Google-fu...
Browsing StackOverflow...
Watching cute cat videos... Oh, is that a "don't laugh challenge"? I'm interested!
Populating classes... Wait a second, is this a linked list? Ugh.
Brawling with C++ Memory Manager... (wish I didn't say "I would like to speak with your manager")
Preparing test cases...
Ready.

As an aspiring Computer Engineering student, your next task -*if you accept*- is to create a simplified version of a Blockchain. So, you are going to overload some operators, keep track of the memory, fight with faulty segments *(it's not you, it's the segment.)*, calculate your Blockchain's and your Miner's values, and some housekeeping.

A standard C++ homework, with the addition of being rich **locally**. Because why would anyone pay fiat money for some bytes in another person's computer? I wouldn't.
(I actually did, but that's not in the scope of this homework.)

3 Class Definitions

3.1 Koin

Koin is the most basic class in this homework. Basically it's the **node** in a **linked-list**, but with the addition of Blockchain, they would create *the Voltron*. No? Okay, linked list it is.

```
class Koin {

private:
    double value;
    Koin *next;

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PRIVATE METHODS/PROPERTIES BELOW
public:
    /**
     * Constructor.
     *
     * @param value The Koin's value.
     */
    Koin(double value);
    /**
     * Copy Constructor.
     *
     * @param rhs The koin to be copied.
     */
    Koin(const Koin& rhs);
    /**
     * Assignment.
     *
     * @param rhs The koin to assign into this koin.
     * @return The assigned koin.
     */
    Koin& operator=(const Koin& rhs);
    ~Koin();
    double getValue() const;
    Koin* getNext() const;
    /**
     * Sets the next chain for this Koin.
     *
     * Important: Koin does NOT "own" next.
     *
     * @param next The next Koin.
     */
    void setNext(Koin *next);
    /**
     * Equality overload.
     *
     * The Koins are equal if their values are equal, and their next Koins, and ↩
        the next Koins' next koins... are also equal.
     *
     * Important: Since the values are double, you should refrain to use equality↩
        between two doubles.
     * Instead, they should be compared within a sensitivity, given in Utilities.↩
        h.
     *
```

```
     * @param rhs The Koin to compare.
     * @return True if they're equal, false otherwise.
     */
    bool operator==(const Koin& rhs) const;
    /**
     * Inequality overload.
     *
     * Explained in upper part.
     *
     * @param rhs The Koin to compare.
     * @return True if they're not equal, false otherwise.
     */
    bool operator!=(const Koin& rhs) const;
    /**
     * Multiply the value of the Koin with given multiplier.
     *
     * @param multiplier The value to multiply the Koin's value.
     * @return The current Koin.
     */
    Koin& operator*=(int multiplier);
    /**
     * Divide the value of the Koin with given divisor.
     *
     * @param divisor The value to divide the Koin's value.
     * @return The current Koin.
     */
    Koin& operator/=(int divisor);
    /**
     * Stream overload.
     *
     * What to stream:
     * koinValue—nextKoin's value—nextKoins' nextKoin's value--...--| (pipe at ↩
         the end)
     *
     * Example:
     * 0.0156--0.0156--0.0156--|
     *
     * Important: The precision required is given to you in Utilizer.h.
     * For this example, it's 3.
     *
     * @param os Stream to be used.
     * @param koin Koin to be streamed.
     * @return The current Stream.
     */
    friend std::ostream& operator<<(std::ostream& os, const Koin& koin);

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PUBLIC METHODS/PROPERTIES BELOW
};
```

3.2 Blockchain

Blockchain consists of an id (which is unique in a Miner) and a head Koin. You are going to be traversing the Koin chain, mostly. Also, in order to deal with the memory, it's important to decide that which Koins are under the ownership of a Blockchain, which are not.

```
class Blockchain {

private:
    Koin *head;
    const int id;

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PRIVATE METHODS/PROPERTIES BELOW
public:
    /**
     * Constructor.
     *
     * @param id The ID of the Blockchain.
     */
    Blockchain(int id);
    /**
     * Secondary Constructor.
     *
     * Important: Blockchain "owns" this Koin.
     *
     * @param id The ID of the Blockchain.
     * @param head The starting point of the Blockchain.
     */
    Blockchain(int id, Koin *head);
    /**
     * Copy Constructor.
     *
     * The head should be deep copied.
     *
     * @param rhs The blockchain to be copied.
     */
    Blockchain(const Blockchain& rhs);
    /**
     * Move Operator.
     *
     * RHS will relinquish the rights of the headKoin to LHS.
     * LHS's ID will not (cannot) be changed.
     *
     * Careful about memory-leaks!
     *
     * @param rhs The blockchain to be moved into this blockchain.
     * @return The modified blockchain.
     */
    Blockchain& operator=(Blockchain&& rhs) noexcept;
    /**
     * No assignment with const rhs.
     *
     * @param rhs -noNeed-
     * @return -noNeed-
     */
    Blockchain& operator=(const Blockchain& rhs) = delete;
```

```
~Blockchain();
int getID() const;
Koin* getHead() const;
/**
 * Calculate the value of the blockchain.
 *
 * @return Total value of the Koins in the blockchain.
 */
double getTotalValue() const;
/**
 * Calculate the length of the koins end-to-end.
 *
 * @return The length of the blockchain.
 */
long getChainLength() const;
/**
 * Prefix addition.
 *
 * Mine and insert the mined Koin at the end of the chain.
 */
void operator++();
/**
 * Prefix decrement.
 *
 * Remove/destroy the last inserted Koin to the chain.
 * no-op if the head is nullptr.
 *
 * Important note:
 *
 * If a blockchain is created by a soft fork, its head cannot be deleted.
 */
void operator--();
/**
 * Multiplication overload.
 *
 * Multiply the value of every Koin in the blockchain.
 *
 * @param multiplier The value to be multiplied with the values of the Koins.
 * @return The current blockchain.
 */
Blockchain& operator*=(int multiplier);
/**
 * Division.
 *
 * Divide the value of every Koin in the blockchain.
 *
 * @param divisor The value to divide the values of the Koins.
 * @return The current blockchain.
 */
Blockchain& operator/=(int divisor);
/**
 * Stream overload.
 *
 * What to stream:
 *
 * Block -blockchainID-: headKoinStream (see Koin for stream example) (-↩
    totalValue-)
```

```
     *
     * Example:
     *
     * Block 6: 0.707--0.390--0.984--|(2.080)
     *
     * Edge case : Blockchain without head
     *
     * Block -blockchainID-: Empty.
     *
     * @param os Stream to be used.
     * @param blockchain Blockchain to be streamed.
     * @return The current stream.
     */
    friend std::ostream& operator<<(std::ostream& os, const Blockchain& ↩
        blockchain);

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PUBLIC METHODS/PROPERTIES BELOW
};
```

3.3 Miner

Miner is the owner of the blockchains in its arsenal. Also, it's able to **fork** the blockchains as requested. The details are in the code itself below.

```
class Miner {

private:
    /**
     * Find the next ID for a newly created/forked blockchain.
     * Blockchain IDs are unique belonging to Miner, and strictly increasing (↩
         started from 0).
     *
     * @return The next available ID for a new blockchain.
     */
    int getNextAvailableBlockchainID() const;

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PRIVATE METHODS/PROPERTIES BELOW
public:
    /**
     * Constructor.
     *
     * @param name Name of the miner.
     */
    Miner(std::string name);
    ~Miner();
    /**
     * Create a new blockchain with the next available ID.
     */
    void createNewBlockchain();
    /**
     * Start mining with the given count.
     * For every count, a single Koin will be produced in each blockchain.
     *
     * Operations must be ordered according to the blockchainIDs.
     *
```

```
     * @param cycleCount The count which the blockchains will be run for.
     */
    void mineUntil(int cycleCount);
    /**
     * Start de-mining with the given count.
     * For every count, the last Koin in the blockchain must be destroyed.
     * no-op for Blockchain if it doesn't have any (left).
     *
     * Operations must be ordered according to the blockchainIDs.
     *
     * @param cycleCount The count which the blockchains will be run for.
     */
    void demineUntil(int cycleCount);
    /**
     * Calculate the value of the miner, which is the sum of the blockchains' ↩
        value.
     * Soft-forks DO NOT constitute for the total value of the miner.
     *
     * @return Total value of the miner.
     */
    double getTotalValue() const;
    /**
     * Return the count of the blockchains.
     *
     * @return Total count of the blockchains.
     */
    long getBlockchainCount() const;
    /**
     * Indexing.
     *
     * Find the blockchain with the given id.
     *
     * @return The blockchain with the given id. nullptr if not exists.
     */
    Blockchain* operator[](int id) const;
    /**
     *  Shallow-copy the blockchain with given ID.
     *
     *  How-To:
     *  1) Fetch the blockchain in the miner with given ID.
     *  2) If not exists, no-op.
     *  3) Fetch the next available ID for the blockchain.
     *  4) Create a new blockchain with the new ID, and with the head as the ↩
        original of the last Koin of the blockchain.
     *  5) Hence, a modification to the newly created blockchain will also affect↩
         the old blockchain, but only ...
     *  6) ... after the head (head included). And vice versa.
     *  7) Save the newly created blockchain in the miner.
     *
     *  @param blockchainID The blockchain ID to be forked.
     *
     *  @return true if the blockchain with given ID exists, otherwise false.
    */
    bool softFork(int blockchainID);
    /**
     *  Deep-copy the blockchain with given ID.
     *
```

```
     *  How-To:
     *  1) Fetch the blockchain in the miner with given ID.
     *  2) If not exists, no-op.
     *  3) Else, fetch the next available ID for the blockchain.
     *  4) Create a new blockchain with the new ID, and with the head as a copy ↩
         of the last Koin of the blockchain.
     *  5) Any changes made in the new blockchain will NOT affect the old ↩
         blockchain. And vice versa.
     *  6) Save the newly created blockchain in the miner.
     *
     *  @param blockchainID The blockchain ID to be forked.
     *
     *  @return true if the blockchain with given ID exists, otherwise false.
     */
    bool hardFork(int blockchainID);
    /**
     * Stream overload.
     *
     * What to stream:
     *
     * -- BEGIN MINER --
     * Miner name: -minerName-
     * Blockchain count: -getBlockchainCount()-
     * Total value: -getTotalValue()-
     *
     * Block -blockchainID-: headKoin (see Blockchain for stream example)
     * .
     * .
     * .
     * Block -lastBlockchainID-: headKoin (see Blockchain for stream example)
     *
     * -- END MINER --
     *
     * Example:
     *
     * -- BEGIN MINER --
     * Miner name: BTCMiner
     * Blockchain count: 5
     * Total value: 2.519
     *
     * Block 0: 0.491--0.103--0.529--|(1.123)
     * Block 1: Empty.
     * Block 2: Empty.
     * Block 3: Empty.
     * Block 4: 0.400--0.924--0.072--|(1.396)
     *
     * -- END MINER --
     *
     * @param os Stream to be used.
     * @param miner Miner to be streamed.
     * @return The current stream.
     */
    friend std::ostream& operator<<(std::ostream& os, const Miner& miner);

    // DO NOT MODIFY THE UPPER PART
    // ADD OWN PUBLIC METHODS/PROPERTIES BELOW
};
```

4 Extras

While producing Koins, you need to produce a random double value for its worth. To do this, you **MUST** use Utilizer class. It'll seed a Random Number Generator and give you the values accordingly. This RNG is OS-Agnostic, but the grading will only be done in Ubuntu (ineks to be exact).

The implementation of the Mersenne and the Utilizer is already provided to you. Hence, you just need to do this:

```
Koin* newKoin = new Koin(Utilizer::fetchRandomValue());
```

While comparing/printing the Koins, you **MUST** use the Utilizer class again.

To compare values, use `Utilizer::doubleSensitivity`.
To print, use `Utilizer::printPrecision`.

The summary of the memory ownership:

- A Koin **WILL NOT** own its `nextKoin`.
- A Blockchain **WILL** own the `headKoin` when constructed with.
- A Blockchain **WILL** own the **COPY** of the `headKoin` when copy constructed.
- A Blockchain **WILL** own the `headKoin` when moved from RHS. RHS **WILL** relinquish *(TR: feragat etmek/vazgecmek)* its ownership from the `headKoin`.
- A Miner **WILL** own the `blockchain` when `createNewBlockchain, softFork:, hardFork:` is called and produced a `blockchain`.
- Owning a `koin` also means owning the `nextKoin`, and the `nextKoin` of the `nextKoin`, ...

5 Grading

For your convenience, there are multiple partial tests already given to you in **StudentPack**. The rest of the points can be earned with proper memory management and proper outputs. The example I/O will be shared on COW, and a simple one is in **StudentPack**.

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 may include and use C++ Standard Library. 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. Class-wise memory management is explained to you in source-code. 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 newsgroup (news.ceng.metu.edu.tr) 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:

- Koin.h
- Koin.cpp
- Blockchain.h
- Blockchain.cpp
- Miner.h
- Miner.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
```

Good luck, have fun.

- engin