Pierre Godino – Florian Rolland

WORED COIN

Dossier conception en langage c.

Florian Rolland
Projet S4 2017-2018

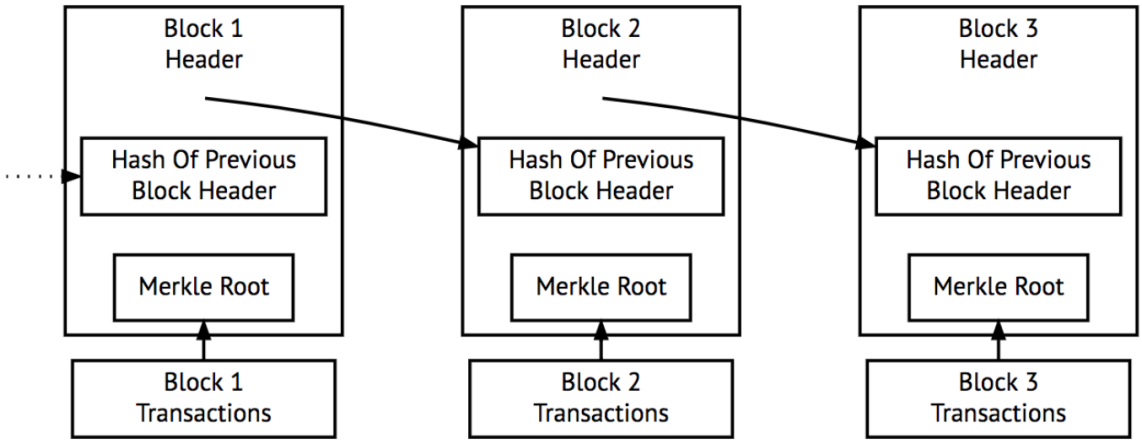
Table des matières

1. Introduction :

Dans ce document, nous allons voir comment le programme a été conçu, c'est-à-dire, qu'elle est sa structure interne, comment il a été ordonné, les différents choix de l'implémentation, un descriptif détailler de son fonctionnement.

Le programme en C de level_1 doit être de générer une BlockChain avec des générations de transactions aléatoires sous la forme d'une chaine de caractère comme ceci : « Source-Destination : 13092823 ». « 13092823 » étant un nombre aléatoire. Chaque bloc va contenir un nombre aléatoire de transactions (dont le nombre maximum est configurable) et sera ensuite miné jusqu'au nombre de blocs voulu atteint. Le programme sera accompagné de 2 utilitaires, 1 permettant de vérifier l'intégrité de la BlockChain, en d'autre mot, que aucun bloc n'a été altéré ou supprimé. Le second quant à lui sera là pour supprimer un block ou une transaction et recalculer entièrement la BlockChain, il s'agira donc d'un cheater. Un terminal de commande permettra d'exécuter le logiciel.

Rapide présentation de la BlockChain à effectuer :



Simplified Bitcoin Block Chain

Chaque bloc va contenir 3 Hash différent, le Hash de son prédécesseur, le Hash issu de l'arbre de Merkle et son Hash courant. Ces 3 Hash permettent de s'assurer de l'intégrité de la BlockChain. Effectivement, si je modifie un block, le Hash issu de l'arbre de Merkle change donc son Hash Courant aussi et le Hash ne correspondra plus avec le Hash Previous du bloc successeur. Il faut donc recalculer les Hash de tous les blocks pour passer inaperçu. Pour éviter ça on va ajouter une difficulté. Cette difficulté doit s'assurer que le temps pour miné un block (Obtenir son Hash Courant) soit conséquent pour éviter de recalculer la BlockChain trop facilement et donc cheater la BlockChain. Donc cette difficulté consiste donc (dans notre cas), à obtenir un nombre de 0 consécutifs au début du hash. Exemple, pour une difficulté de 4 :
00003d04ac9b8e687dce373ba27a91ec8fda114c7fc014ec252758d0805d402e

Nous avons donc les 4 zéros qui correspondent à la difficulté.

Pour ce faire, dans chaque block, nous ajoutons un entier appelé « Nonce » initialiser a 0, et qui fait partie des données Hasher pour le calcul du Hash, une fois calculer, si le Hash ne satisfait pas la difficulté, on incrémente le nonce de 1, et on recommence jusqu'à satisfaire la difficulté.

Dans ce dossier de conception, nous ne parlons pas des fichiers permettant d'effectuer le sha256 pour obtenir le Hash. Ce sont des fichiers extérieurs que nous ne modifierons pas. Il s'agit d'une fonction qui prend une chaine de caractère et qui retourne un Hash en conséquence.

2. La structure du code C :

Le programme est réparti sur 18 fichiers différents et accompagné d'une notice d'utilisation. Sur les 18 fichiers, nous pouvons réduire leur nombre à 10. 8 fichiers .c avec leur .h, un fichier .h de configuration et un fichier .c « main ». Certains ne seront pas détaillé comme dit précédemment.

2.1. Le fichier BlockChain.c/.h

Le fichier BlockChain.c contient un élément principal du programme, la structure de la BlockChain :

```
3  struct sBlockList {
4    Block* block;                //Block
5    struct sBlockList *next;     //Pointeur vers prochain block
6  };
7
8  struct sBlockChain {
9    int nbBlocks;               //Nombre de nbBlock
10   int difficulty;             //Difficulté de la blockchain
11   BlockList *blocklist;        //Liste des blocks
12   BlockList *lastBlockList;    //Dernier block
13   //BlockList firstBlockList;
14 };
```

Cette structure est composée en deux structures : sBlockList va permettre de créer une liste chainée de blocks. Cette structure va contenir un block et l'adresse mémoire de la prochaine structure sBlockList. Ce qui en accédant à cette adresse puis à la prochaine et ainsi de suite que nous pouvons parcourir la liste de nos blocs. La structure sBlockChain va contenir deux informations importantes, la première est le nombre de blocks que contient la BlockChain, et aussi ça difficulté mais nous y reviendrons plus tard là-dessus. Cette structure contient aussi l'adresse de la première structure sBlockList et l'adresse de la dernière structure sBlockList. Savoir à l'instant T l'adresse de la dernière structure n'est pas indispensable, loin de là. Mais elle permet d'évité de parcourir la liste de block pour accéder au premier block.

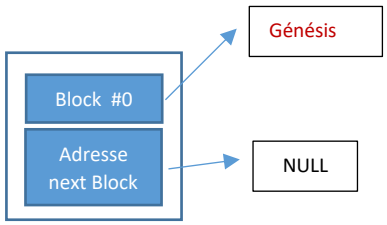
Chose très importante, ces deux structures sont opaques. C'est-à-dire qu'elle ne sont accessible directement que depuis ce fichier. C'est-à-dire que je peux accéder à n'importe quel champ de ma structure pour lire ou modifier celle-ci. Donc pour toute autre fonction contenu dans un autre fichier, les structures seront invisibles. Celle permet notamment de les protéger, et éviter toute modification douteuse. Donc pour pallier à ça, nous avons donc des fonctions pour « interagir » avec ces structures pour générer une BlockChain, y ajouter un block, vérifier sont intégrités et y ajouter un block. Ces fonctions permettent de modifier la BlockChain et d'interagir avec depuis du code extérieur au fichier blockchain.c, cela permet donc de s'assurer du bon fonctionnement de la BlockChain. Par exemple, aucune fonction ou code en dehors de blockchain.c ne pourra modifier le champ difficulty, mais pourra y ajouter un block via la fonction dédier.

Voici donc les autres fonctions :

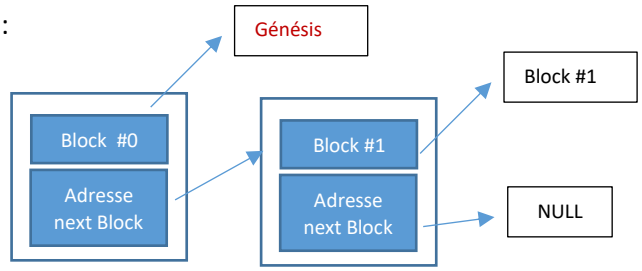
```
17  //Fonction
18  Block* firstBlock();
19  BlockList* addBlockList(Block* block);
20  BlockChain* createBlockChain(int difficulty);
21  BlockList* genBlockList(Block* block);
    Block* getLastBlock(BlockChain* blockChain);
    void addBlock(BlockChain* blockChain, Block* block);
    void removeBlock(BlockChain* blockChain, int indexBlock);
25  bool chainIsValid(BlockChain* blockChain);
26  bool merkleIsValid(BlockChain* blockChain);
27  Block* getBlockInChain(BlockChain* blockChain, int index);
28  int getNbBlock(BlockChain* blockChain);
29  int getDifficulty(BlockChain* blockChain);
```

- Block* firstBlock() ; permet de créer le block Génésis (Block détailler plus tard) et de la miner à une difficulté de 0.
- BlockList* addBlockList(Block* block) ; permet d'ajouter un block passé en paramètre autre que Génésis. Pour ce faire, la fonction accède au dernier block de la chaine. Créer une structure sBlockList avec le block passé en paramètre, puis va ajouter l'adresse mémoire de cette nouvelle structure dans le champs next du dernier block. Il met à jour ensuite l'adresse du dernier dans la structure BlockChain. Voici un schéma récapitulatif :

Avant :



Après :



- BlockChain* createBlockChain(int difficulty) ; permet de créer une BlockChain avec une difficulté donné. Utilise firstBlock() pour créer le premier block et addBlockList() pour l'ajouter.
- void addBlock(BlockChain* blockChain, Block* block) ; ajoute le block dans la blockChain passés en paramètres.
- void removeBlock(BlockChain* blockChain, int indexBlock) ; supprime le block grâce a son index dans la blockChain passés en paramètres.
- bool chainIsValid(BlockChain* blockChain) ; vérifie l'intégrité de la chaine. C'est-à-dire que la fonction va vérifier si les hash de chaque block sont cohérents. En effet chaque block, contient le Hash de son prédécesseur. Donc si le contenu d'un block a été modifier ou qu'un block a été supprimé, en re-minant le block pour vérifier son Hash actuel, il sera forcément différent. Mais nous le verrons plus en détail plus tard.
- bool merkleIsValid(BlockChain* blockChain), vérifie uniquement que les transactions de chaque block n'ont pas été modifié.
- Block* getBlockInChain(BlockChain* blockChain, int index) retourne l'adresse du block numéro « index » dans la blockChain passé en paramètre.

- Les deux autres fonctions restantes servent juste à récupérer la difficulté et le nombre de block dans une blockChain passé en paramètre.

2.2. Le fichier Block.c/.h

Ce fichier va regrouper la structure Block, la fonction de minage de block, et la génération de block.

```
3  struct sBlock{
4
5    int index;                    //Numéro du BlockList
6    char timeStamp[TIMESTAMP_SIZE+1]; //Horodatage du block
7
8    int nbTransaction;            //Nombre de transaction
9    char** transactionList;       //Liste des transactions
10
11   char* hashMerkleRoot; //Hash root de l'arbre de Merkle des transactions
12   char* hashCurrent;    //Hash du block courant
13   char* hashPrevious;   //Hash du block précèdent
14
15   int nonce;                    //Nombre pseudo aléatoire et unique
16
17 };
```

Chaque block contient, un index contenant son numéro, le timeStamp contient là l'horodatage de création du block. Ensuite il y a le nombre de Transaction, un tableau contenant la chaine de caractère de chaque transaction.

Ensuite hashMerkleRoot contient de le hash de Merkle. (Détaillé dans le 2.4). hashCurrent contient le hash du block et hashPrevious le hash du bloc précédent.

Cette structure elle aussi opaque, beaucoup de fonction pour lire et modifier les champs de la structure seront créés même si beaucoup sont inutiles. Voici les fonctions :

```
17 //Fonction accès structure block, beaucoup inutile.
18 int getIndexBlock(Block* blockTemp);
19 void setIndexBlock(Block* blockTemp, int index);
20 char* getTimeStampBlock(Block* blockTemp);
21 void setTimeStampBlock(Block* blockTemp, char* timeStamp);
22 int getNbTransationBlock(Block* blockTemp);
23 void setNbTransactionBlock(Block* blockTemp, int nb);
24 char** getListTransationBlock(Block* blockTemp);
25 void setListTransactionBlock(Block* blockTemp, char** transaction);
26 char* getListTransationBlockI(Block* blockTemp, int index);
27 void setListTransactionBlockI(Block* blockTemp, char* transaction, int index);
28 char* getHashMerkleRoot(Block* blockTemp);
29 void setHashMerkleRoot(Block* blockTemp, char* hash);
30 char* getHashPrevious(Block* blockTemp);
31 void setHashPrevious(Block* blockTemp, char* hash);
32 char* getHashCurrent(Block* blockTemp);
33 void setHashCurrent(Block* blockTemp, char* hash);
34 int getNonceBlock(Block* blockTemp);
35 void setNonceBlock(Block* blockTemp, int nonce);
36
37 //Fonction
38 char* getTimeStamp();
39 bool miningOK(char* hasTemp, int difficulty);
40 void miningBlock(Block* blockTemp, int difficulty);
41 bool blockIsValid(Block* blockTemp);
42 Block* GenesisBlock();
43 Block* GenBlock(Block* prevBlock);
44 void freeBlockTransI(Block* temp, int i);
```

On va juste détailler les fonctions de la ligne 38 à 44.

- Char* getTimeSamp() ; permet donc de récupéré l'horodatage a l'instant T.
- Bool miningOK(Char* hasTemp, int difficulty) ; permet de vérifier si le hash obtenu en minant, correspond à la difficulté.
- Bool blockIsValid(Block* blockTemp) ; permet de vérifier si le block n'a pas été altéré. La fonction va vérifier le hashCurrent du block, et le nouveau hash obtenu en re-minant le block, si ils sont différents, le block a été altéré.
- Void miningBlock(Block* blockTemp, int difficulty) ;

La fonction de minage va miner le block, c'est à dire que on va obtenir un hash correspondent à la difficulté de la blockChain. Le hash sera calculé à partir d'une chaine de caractère qui sera issu de toute les informations du block sauf hashCurrent et nonce nommé A. A partir de ce moment-là, on va dupliquer A pour créer B, puis on concatène le nonce initialisé à 0 a B. Ensuite on calcule la hash à partir de B. Si minningOk renvoi « true », alors le hash correspond à la difficulté. Si la fonction renvoi « false », alors on re-duplique A dans B, on concatène la nonce + 1 a B et on recalcule le hash jusqu'à ce que le hash satisfait la difficulté. Puis on ajoute le hash obtenu dans hashCurrent.

La fonction miningBlock serra utilisé à plusieurs reprises pour vérifier la blockChain et aussi pour recalculer tous les hash pour cheater la blockChain. Cependant, lors de la vérification, on calcule le

hash directement avec le nonce indiqué dans le block. Donc le code de 90% de la fonction serra « copier » et adapter dans d'autre fonction.

- Block* GenesisBlock() ; Cette fonction créé le block Genesis, soit le premier block de la blockChain, ce block contient une seul transaction « Genesis » et un nonce de 0. En effet, le block Génésis ne doit pas obligatoirement satisfaire la difficulté de la blockChain. C'est pour ça qu'il serra miné a une difficulté dite de zéro. Le block Génésis n'ayant pas de block avant lui, le previousHash sera « 0 ». Mais tout comme les autres blocks, il contient un timeStamp et un merkleHash
- Block* GenBlock(Block* prevBlock) ; Cette fonction va créer un block en récupérant en récupérant le hashCurrent de prevBlock pour le mettre de previousHash du nouveau block, l'index du block est celui de prevBlock +1, on récupère le timeStamp a l'instant T, on génère des transaction et on les inclut dans le block, on calcule le hashMerkle et on mine le block.

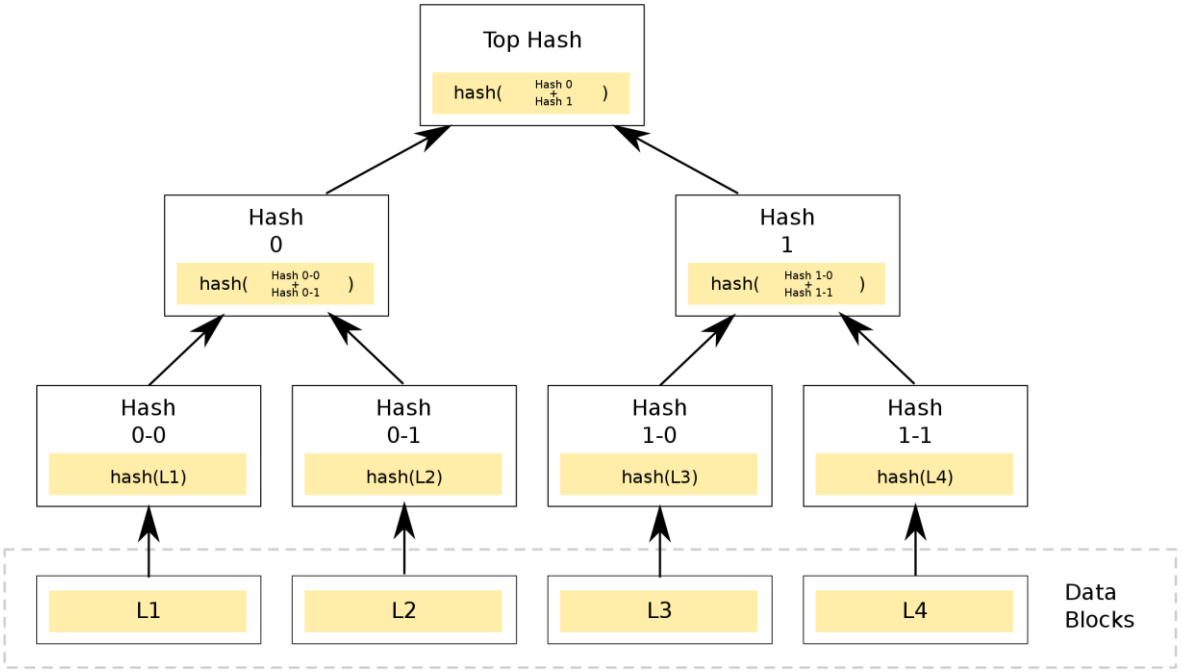
2.3. Le fichier transaction.c/.h

Ce fichier qui contient une petite fonction (void genTransaction(Block* block)) qui permet de générer des transactions. Tout d'abords on génère un nombre aléatoire. Ce nombre sera le nombre de transaction à générer. A noté que l'on peut définir le nombre maximum de transaction. Ensuite on va générer un nombre aléatoire pour chaque transaction lui aussi on peut y définir son maximum. Et on va créer chaque transaction sous la forme

« Source-Destination: 292929292 » sous forme de chaine de caractère. La fonction va ensuite modifier le nombre de transaction dans le block, et y ajouter les transactions. Cette fonction est donc appelé par la fonction genBlock().

2.4. Le fichier merkel.c/.h

Le fichier a été appelé merkel au lieu de merkle par erreur. Donc, ce fichier contient une seule et unique fonction (char* getMerkelRoot(char** tabTransaction, int nbTransaction)) qui retourne le hashMerkle obtenu. Donc le hashMerkleRoot est une technique pour vérifier l'intégralité d'un fichier, en effet, il s'agit de plusieurs hash concaténés entre eux puis re-hash pour etre re-concaténé en boucle jusqu'à obtenir un seul et unique hash. Une image sera plus explicite :



Nota Bene : Si le nombre de hash n'est pas paire, on duplique le dernier.

L1,L2…LN correspond à chaque transaction du bloc et Top Hash est le hash retourné par la fonction. Cette méthode permet donc de vérifier que aucune transaction a été altéré.

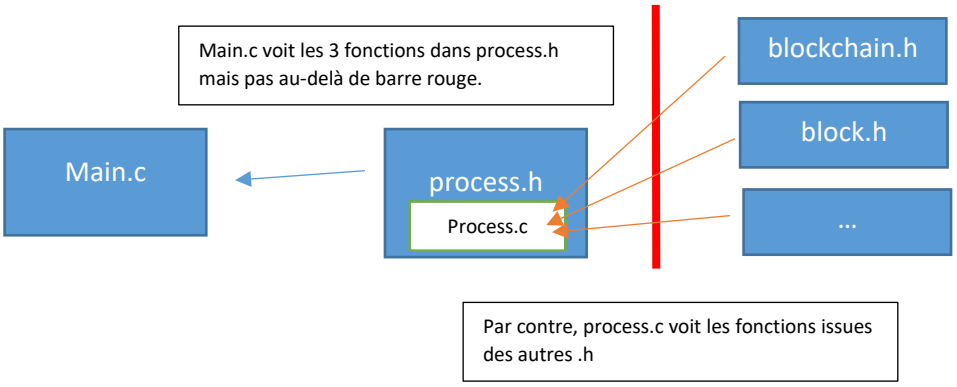
2.5. Le fichier cheater.c/.h

Ce fichier contient deux fonctions qui permettent de supprimer un block et de supprimer une transaction d'un block.

- void alteredTransactionBlock(BlockChain* blockChain, Block* blockTemp, int removeTransaction); va supprimer la transaction de blockTemp dont le numéro est passé en paramètre, modifier le nombre de transaction du block en conséquence. Ensuite la fonction va miner tous les blocks à partir du blockTemp et modifier les hashPrevious issu des nouveaux hashCurrent obtenu via le minage.
- void alteredRemoveBlock(BlockChain* blockChain, int index); exactement la même chose sauf que l'on supprime pas une transaction mais un block dont le numéro est en paramètre.

2.6. Le fichier process.c/h

Le fichier process.c/h est très important. Nous avons parlé de structures opaques plus haut, et bien nous allons parler de fonction « opaques ». Effectivement, nous ne pouvons pas mettre dans notre fichier principal, notre fonction firstBlock() ou genTransaction() ni même la fonction qui permet de modifier un hash issu de block.c. Donc pour pallier ça, nous allons créer un fichier entre notre main.c et tous les autres. Le fichier process va définir et limiter les fonctions accessibles par l'utilisateur ou un développeur extérieur, cela permet de s'assurer de bon fonctionnement de nos fonctions surtout si ce programme doit être ajouter dans un autre programme. On peut imaginer ça, comme un distributeur de billet, donc le distributeur a des fonctions accessibles par l'utilisateur mais pas toutes, imaginé on est accès a la fonction qui actionne la sortie des billets sans utilisé sa carte bleue. C'est un peu là même ici, il faut protéger son programme. Voici un petit schéma :




```
21  void printBlockChain(BlockChain* blockChain);
    int hackBlockChain(BlockChain* blockChain, int i, int transaction);
    BlockChain* genBlockChain(int difficulty, int nbBlock);

25  //Fonction pour pouvoir rajouter des actions plus spécifique. Pas utile dans l'état actuel du programme.
26
27  //void createBlock(BlockChain* temp);
28  //void printBlock(Block* blockTemp);
29  //bool verifBlockChain(BlockChain* blockChain);
30  //bool verifHashRoot(BlockChain* blockChain);
```

Donc en état actuel de l'interface du programme, seules les fonctions de la ligne 21 à 23 sont accessibles au main.c, les fonctions lignes 27 à 30 sont occultés pour le main car l'interface ne permet mais pas de les utilisés mais par contre, elles sont utilisées par les 3 fonctions juste au-dessus pour compensé leur absence d'accès via l'interface.

- Void printBlockChain(BlockChain* blockChain) ; va afficher la blockChain sur le terminal de commande en affichant le numéro du block, son nombre de transaction et ses 3 hash.
- Int hackBlockChain(BlockChain* blockChain, int i, int transaction) ; « i » représente le numéro de block. « transaction » lui va donner le numéro de transaction a supprimer, si il est égale à -1, on supprime le block, sinon on supprime la transaction du block. Après avoir cheater et recalculer la blockChain, on va vérifier son intégrité et afficher le résultat, et ensuite on affiche la blockChain en entier. De plus, on affiche le temps mis pour cheater la blockChain.

- BlockChain* genBlockChain(int difficulty, int nbBlock) ; on va générer une blockChain avec le nombre de blocks et la difficulté passé en paramètre, on affiche chaque block une fois le block généré et miné, puis à la fin, on affiche le temps mis pour tout générer.

2.7. Le fichier config.h

Ce fichier est très important, c'est là où toute les constantes de paramètres sont réunis :

```
4  //Constante du programme
5  // ATTENTION : toujours vérifier que les varibles Nonce/MAX_VALUE_RAND et nombre de block ne dépasse
6  // pas en nombre de caractère, sinon ajuster les varibles caract en conséquence
7  #define MAX_BLOCK 3 //nb de caract du Nombre
8  #define MAX_TRANSACTION_CHAR 3 //nb de caract du Nombre de Transaction
9  #define MAX_NONCE_CHAR 10 //Nb caract max de la nonce
10 #define TRANSACTION_SIZE 30 // Taille de la chaine de caractère d'une transaction
11 #define TIMESTAMP_SIZE 25 //nb de caract du timeStamp
12
13 #define MAX_TRANSACTION 100 //nb max de transaction
14 #define HASH_SIZE 64 //Taille du hash
15 #define MAX_NONCE 9999999 //Nonce max
16 #define MAX_VALUE_RAND 1000000 //Max nombre random pour simuler transaction
```

On y défini la taille des tableaux qui sont allouer dynamiquement pendant l'exécution du programme, mais aussi le nombre de transaction et le nonce max par exemple.

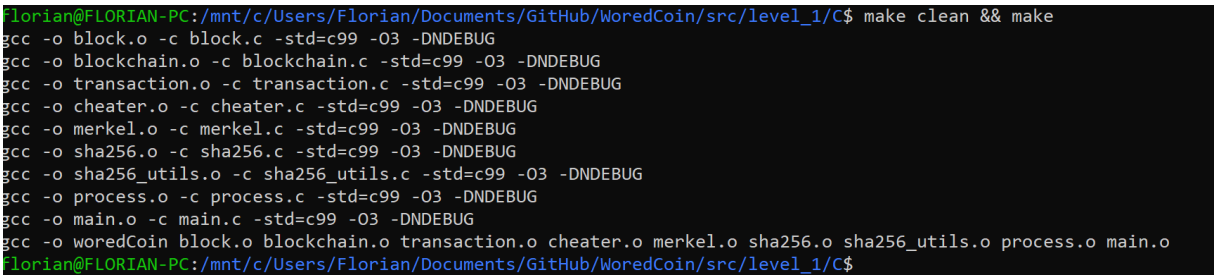
- MAX_BLOCK -> cette constante défini le nombre de caractères (chiffre) du nombre de block. Il doit toujours être supérieur ou égale (exemple : MAX_BLOCK 3 pour un nb de block de 150, et MAX_BLOCK 4 pour 1230). S'il est inférieur, le tableau qui contient la chaine de caractère concaténé de la fonction miningBlock sera trop petit, et le programme s'arrêteras de manière inattendu.
- MAX_TRANSACTION_CHAR -> même chose mais pour le nombre de transaction
- MAX_NONCE_CHAR -> même chose mais pour le nonce.
- MAX TRANSACTION_SIZE -> même chose mais prend en compte « Source-Destination: » et le nombre généré aléatoirement. Soit 20 caractères + le nombre de caractères max du nombre généré aléatoirement.
- TIMESTAMP_SIZE -> taille de la chaine de caractère du TimeStamp.
- MAX_TRANSACTION -> nombre maximum de transaction par block.
- HASH_SIZE -> taille du hash.
- MAX_NONCE-> nonce maximum.
- MAX_VALUE_RAND -> nombre maximum pour simuler la transaction.

Attention, il faut absolument prendre en compte le caractère « \0 » dans les calculs, qui est le caractère de fin de chaine. Donc, faire +1.

3. L'exécution du programme :

3.1. La compilation :

Le programme est fourni avec un makefile qui permet de compiler le programme sous une machine Unix (Distribution linux et macOs), il est possible de le compiler sous Windows en utilisant WSL (Windows SubSystem for Linux) qui permet d'accéder au terminal d'une distribution Linux dans Windows 10 même (Ce que j'utilise). Ce n'est pas non plus une machine virtuelle et ni un émulateur, le système fonctionne en « symbiose » avec Windows. Très performant donc, mais uniquement sous un terminal. On peut aussi le compiler a la main sur Windows en utilisant minGW par exemple.



Un « make clean && make » permet de compiler le programme. Le programme compilé aura pour nom « woredCoin ».

3.2. L'exécution

Une fois le programme compilé, on peut exécuter le programme. Pour lancer le programme, il nécessite des arguments :

- -b -> le nombre de blocks
- -d -> la difficulté
- -cb -> numéro de block à supprimer (optionnel)
- -ct -> numéro de block à supprimer (optionnel mais nécessite -cb)

Exemple : Commande-> ./woredCoin –b 10 –d 4

On génère 10 blocks avec une difficulté de 4.

```
florian@FLORIAN-PC:/mnt/c/Users/Florian/Documents/GitHub/WoredCoin/src/level_1/C$ ./woredCoin -b 10 -d 4
### GENERATION BLOCKCHAIN ###

### GENERATION BLOCK n° 1 ###
### BLOCK -> 1 ### NB TRANSACTION -> 84
### HASH PREV -> 376fbc0f97df64f95a644479e2f203f8d8aca457c0511c92afb7af5fdb019d13
### HASH MERKLE ROOT -> 26b5e9110a9d99e38507fa104365bfb872aa06e806ef4589fe21ab5b7b352765
### HASH CURRENT -> 0000d79c09a7768c23f914c81e61e72d7c8b2bfe6196bde2021a3b8f6d37bb39
```

...

```
### GENERATION BLOCK n° 10 ###
### BLOCK -> 10 ### NB TRANSACTION -> 29
### HASH PREV -> 0000598a2a5b693f3ac58ec31dec1f180c3fe8635acb27a779ab8bca969f3a62
### HASH MERKLE ROOT -> f5041e84bc3a3fa68f36b4b645c8be13f3fedebce7ff57e392ebe94e65f1f24a
### HASH CURRENT -> 00003d04ac9b8e687dce373ba27a91ec8fda114c7fc014ec252758d0805d402e

### GENERATION BLOCKCHAIN TERMINEE EN 2.562500s ###

florian@FLORIAN-PC:/mnt/c/Users/Florian/Documents/GitHub/WoredCoin/src/level_1/C$
```