Documentazione Progetto Block Chain

Una blockchain è un tipo speciale di database. Ci si riferisce alle blockchain con il termine “tecnologia a registro distribuito” (o DLT) ma in molti casi, si riferiscono alla stessa cosa.
Una blockchain ha delle proprietà uniche, ci sono regole che governano il modo in cui i dati possono essere aggiunti, e una volta archiviati, è virtualmente impossibile modificarli o cancellarli.
I dati vengono aggiunti nel corso del tempo in strutture chiamate “*blocchi*”.
Ciascun blocco è costruito sopra il precedente e include un pezzo di informazione che lo collega ad esso. Esaminando il blocco più recente, possiamo verificare che sia stato creato dopo il suo precedente. Quindi, se continuiamo fino in fondo alla “catena di blocchi” (da qui il nome *blockchain*) raggiungeremo il primo blocco, denominato *“blocco genesi”*.
La blockchain utilizza le funzioni di hash per collegare i blocchi tra di loro : genera un hash utilizzando i dati inseriti in ogni riga per compilare la riga successiva. Questo comporta che se cambiassimo i dati del primo input otterremmo una combinazione diversa in tutte le altre caselle.
In modo schematico questo mostra come l'hashing sia la colla che tiene insieme i blocchi. Consiste nel prendere dati di qualsiasi dimensione e passarli attraverso una funzione matematica per produrre un output (hash) che ha sempre la stessa lunghezza.
Le hash usate nelle blockchain sono interessanti, in quanto le probabilità di trovare dati differenti che producono lo stesso output sono quasi nulle, e qualsiasi modifica, anche minima, dei dati di input porterà a un output completamente diverso. Un algoritmo hash funziona in una sola direzione: ciò significa che da qualsiasi contenuto possiamo generare il suo hash (la sua "impronta digitale") ma da un hash non c'è modo di generare il contenuto ad esso associato**.**
L'acronimo SHA-256 si riferisce alla funzione hash che è ampiamente utilizzata per il funzionamento di molte criptovalute (anche di Bitcoin) in quanto offre un elevato livello di sicurezza, che la rende perfetta per proteggere e codificare in modo sicuro le proprie informazioni ed in oltre è resistente alle collisioni. Questa funzione hash crittografica genera un'impronta digitale di un file nota come *“checksum”* , cioè un dato di piccole dimensioni derivato da un blocco di dati più grande allo scopo di rilevare gli errori che possono essere stati introdotti durante la trasmissione o l'archiviazione.
All'interno della rete blockchain tutti i nodi avrebbero una copia dell'hash di 64 caratteri che rappresenta l'informazione che, ad esempio, costituisce un intero blocco. Una volta che l'informazione è stata convalidata dalla rete (risulta già registrata nella catena), qualsiasi manipolazione di quell'informazione, che cerchi di modificare qualche carattere dell'hash convalidato, verrebbe rilevata immediatamente e scartata.

Blockchain autorizzata e consenso distribuito

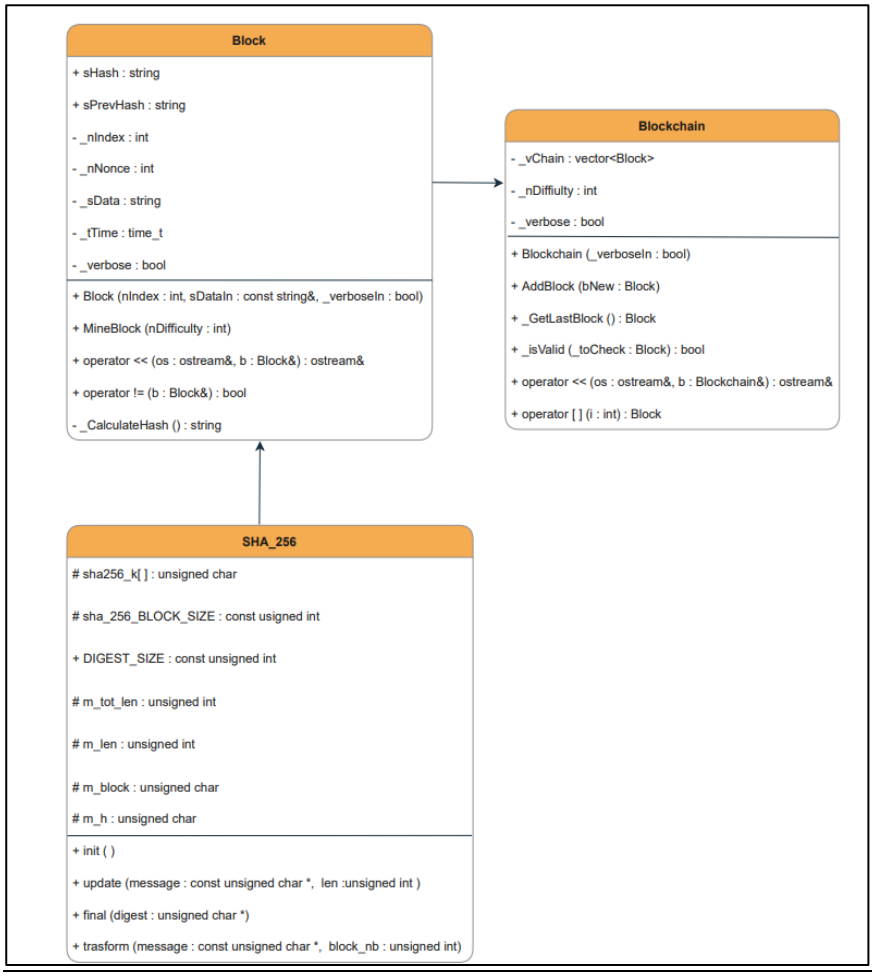
Blockchain è un sistema distribuito peer-to-peer che ha le caratteristiche di alta sicurezza e storage dispersivo con tecnologia di crittografia e tolleranza ai guasti. Ha permeato vari settori della società, in particolare la valuta digitale. Un algoritmo di consenso è la tecnologia di base della blockchain autorizzata. Nel sistema blockchain esistente, Bitcoin utilizza il consenso **Proof of Work** (PoW) ed Ethereum adotta il consenso **Proof of Stake** (PoS). Il sistema Hyperledger utilizza la PBFT (Practical Byzantine Fault Tolerance) per raggiungere accordi. Questi sistemi blockchain sono sistemi pubblici o sindacali e hanno prestazioni di

output basse, ad esempio, Bitcoin elabora 7 transazioni al secondo ed Ethererum elabora 20 transazioni al secondo. La tecnologia di consenso determina in larga misura le prestazioni di output del sistema.

PBFT

L'algoritmo PBFT, derivato dal problema dei generali bizantini, può tollerare nodi difettosi arbitrari e migliorare la disponibilità e l'affidabilità utilizzando i servizi replicati dalla macchina a stati. Il sistema raggiunge il consenso tramite un meccanismo di trasferimento dei messaggi e riesce a risolvere il problema degli attacchi dannosi al sistema. L'algoritmo divide automaticamente ogni livello in base al numero di nodi di consenso nella blockchain e alla dimensione della regione autonoma. Il nodo agente è il nodo chiave del sistema. Agisce come nodo master eseguendo l'algoritmo PBFT all'interno della regione autonoma e raccogliendo il risultato dell'esecuzione. Allo stesso tempo, il nodo agente deve inviare richieste al livello successivo in base al numero di nodi di rete e attende i risultati dell'esecuzione di tutti i nodi nel livello successivo. L'algoritmo presenta i seguenti vantaggi : in primo luogo quando un nuovo nodo di consenso viene aggiunto al sistema, il sistema stesso può modificare la visualizzazione in modo che il nodo possa essere coinvolto nel processo di consenso il prima possibile, per migliorare la scalabilità. In secondo luogo, l'algoritmo non modifica le caratteristiche del database distribuito blockchain decentralizzato e mantiene congiuntamente il database e il consenso. Inoltre, con l'aumento del numero di nodi di consenso, può ottenere migliori prestazioni di output riducendo significativamente il tempo di elaborazione dei messaggi da parte del sistema: viene ridotto il ritardo ed aumentato il throughput del sistema

Diagramma UML :



Segue una possibile implementazione in linguaggio C++ di una blockchain :

Descrizione :

Le prime righe del programma prevedono l'inclusione di alcune librerie indispensabili per il funzionamento del codice . Dato che verranno utilizzati molti oggetti *string* e diversi *cout*, *endl* ecc. si include anche il *namespace standard* per evitare innumerevoli utilizzi dell'operatore di risoluzione di scope *" std:: "* .

```
#include <iostream>
#include <vector>
#include <sstream>
#include <ctime>

using namespace std;
```

Una blockchain nasce con l'attivazione del blocco *genesis*, esso è il primo blocco in assoluto registrato sulla rispettiva rete e al quale si concatenano una serie di blocchi che contengono altri dati. Quando un blocco viene trasmesso alla blockchain fa riferimento al blocco precedente, nello specifico ogni blocco contiene una rappresentazione crittografica del blocco precedente, il che significa che diventa molto difficile modificare il contenuto di qualsiasi blocco senza poi dover cambiare ogni blocco successivo (nel caso del blocco genesi, non esiste un blocco precedente a cui fare riferimento).

Poiché le blockchain utilizzano la crittografia, ora sarebbe un buon momento per includere alcune funzionalità crittografiche nella blockchain. Utilizzeremo la tecnica di hashing SHA256 per creare gli hash dei nostri blocchi, potremmo scriverne di nostri ma in realtà, si preferisce sfruttare del software open source che si trova disponibile in rete, già funzionante e testato.

La fonte del file "*sha256.h*" che implementa la funzione C++ sha256 è di **Zedwood** e la licenza si trova nel file *LICENSE.txt* all'interno della cartella del progetto. Essendo un file già pronto e impacchettato non si attenzionano le scelte implementative e il funzionamento del codice.

Segue il blocco di codice che descrive l'oggetto Blocco :

```
#include "sha256.h" //Hash function SHA-256

class Block {

public:
    string sHash;   //hash of the current block
    string sPrevHash;   //link to previous block
    Block(int nIndexIn, const string &sDataIn, bool _verboseIn = false);
    void MineBlock(int nDifficulty);
    friend ostream& operator << (ostream& os, Block& b);
    bool operator != (Block& b){return this->sHash != b.sHash;}

private:
    int _nIndex;
```

```
    int _nNonce;
    string _sData;
    time_t _tTime;
    bool _verbose;
    string _CalculateHash() const;  // inline function
};
```

NB. Si consiglia la scomposizione del progetto in diversi file header (.h) che implementano singolarmente il funzionamento di parti diverse del programma, in modo da favorirne la modularità.

Si denomina la classe **Block** seguita dal modificatore di accesso public che avvisa il compilatore che i campi seguenti risultino visibili all'esterno della classe. Uno di questi attributi è **sPrevHash** , che modella la proprietà di ogni blocco di far riferimento al blocco precedente. La firma del costruttore accetta due parametri in input **nIndexIn** e **sDataIn**. Si noti che la keyword *const* viene utilizzata insieme all'operatore di referenziazione *"&"* in modo che i parametri vengano passati per riferimento ma non possano essere modificati, questo viene fatto per migliorare l'efficienza e risparmiare memoria. La firma del metodo **MineBlock** accetta un parametro intero "**nDifficulty**". Specifichiamo il modificatore *private* seguito dalle variabili **_nIndex** , **_nNonce** , **_sData** , **_sHash**, **_tTime** e **_verbose** in modo che questi campi rimangano privati e accessibili solo all'interno della classe Bock. La firma per _ **CalculateHash** ha anche la parola chiave const , questo per garantire che il metodo non possa modificare nessuna delle variabili nella classe del blocco che è molto utile quando si ha a che fare con una blockchain. Viene previsto poi l'overloading di due operatori , il primo è un operatore di confronto (**!=**) tra oggetti di tipo **Block** che servirà nella classe **Blockchain** per verificare l'appartenenza del blocco alla catena. Il secondo è l'operatore di redirezione dell'output (**<<**), che viene ridefinito per facilitare la stampa a video dell'oggetto.

Ora è il momento di creare il file di intestazione **Blockchain.h** , che conterrà l'implementazione vera e propria della blockchain :

```
#include "Block.h"
#include <vector>

class Blockchain
{
public:
    Blockchain(bool _verboseIn = false);
    void AddBlock(Block bNew);
    Block _GetLastBlock() const ;
    bool _isValid(Block _toCheck);
    friend ostream& operator<< (ostream& os, Blockchain& b);
    Block operator[] (int i);

private:
    vector<Block> _vChain;  // vettore di Block
    int _nDifficulty;
    bool _verbose;
};
```

Attraverso la prima riga di codice si include innanzitutto l'header file "Block.h" contenente la classe precedentemente creata. Come con la nostra classe block, manteniamo le cose semplici e chiamiamo la

nostra classe blockchain **Blockchain**. La firma del metodo **AddBlock** accetta un parametro **bNew** che deve essere un oggetto della classe Block. Specifichiamo quindi il modificatore *private* seguito dagli attributi privati per **_nDifficulty**, **_vChain** e **_verbose,** nonché dalla firma del metodo **_GetLastBlock** che è anche seguita dalla keyword *const* per indicare che all'oggetto Block restituito dal metodo, non si potrà accedere in scrittura (read-only). Risulta doveroso, in questo caso specifico, assicurarsi che l'oggetto restituito dal metodo non si possa usare a sinistra di un'operazione di assegnamento, al fine di preservare l'integrità dei dati nella blockchain. Il metodo **_isValid** serve per validare un blocco, nello specifico restituisce true se viene trovata una corrispondenza del blocco all'interno della catena, false altrimenti. Anche per la classe Blockchain viene implementato l'overloading dell'operatore di redirezione dell'output, viene però aggiunto anche l'overloading di un altro operatore "**[ ]**", per facilitare l'accesso ad un blocco all'interno della **Blockchain** senza complicare la sintassi.

Una volta data la definizione della struttura delle classi, segue l'implementazione reale dei metodi.

Implementazione

```
    Block(int nIndexIn, const string &sDataIn, bool _verboseIn = false) :
      _nIndex(nIndexIn), _sData(sDataIn), _verbose(_verboseIn)
    {
        _nNonce = 0;
        _tTime = time(nullptr);
        sHash = _CalculateHash();
    }
```

Il costruttore della classe **Block** inizializza gli attributi di classe **_nIndex** e **_sData** tramite lista di inizializzazione, ancora prima di entrare nel corpo della funzione . L'attributo **_verbose** serve a specificare una stampa discorsiva, che avverrà tramite la ridefinizione dell'operatore di redirezione dell'output "<<". La variabile **_nNonce** è impostata su -1 e la variabile **_tTime** è impostata sull'ora corrente.

Quando un utente trasferisce Bitcoin ad un altro utente, una transazione per il trasferimento viene scritta in un blocco sulla blockchain dai nodi della rete Bitcoin. Un nodo è un altro computer che esegue il software Bitcoin e, poiché la rete è *peer-to-peer*, potrebbe essere chiunque in tutto il mondo

NB Un sistema peer-to-peer (P2P) è un sistema distribuito nel quale i nodi hanno simili responsabilità e capacità funzionali. Non vi è centralizzazione di attività in specifici nodi e la cooperazione tra i nodi avviene mediante interazioni tra pari (peers).

Il processo di "*mining*" consiste nella creazione di un nuovo blocco e prevede che il proprietario del nodo venga ricompensato con Bitcoin ogni volta che crea con successo un blocco valido sulla blockchain. Per creare con successo un blocco valido, e quindi essere ricompensato, un *miner* deve creare un hash crittografico del blocco che desidera aggiungere alla blockchain che soddisfi i requisiti di un hash valido in quel momento; questo si ottiene contando il numero di zeri all'inizio dell'hash, se il numero di zeri è uguale o maggiore del livello di difficoltà impostato dalla rete, quel blocco sarà valido. Se l'hash non è valido, viene incrementata una variabile chiamata **nonce** e l'hash viene creato nuovamente;

Tirando le somme diciamo che il processo di mining **serve per validare nuove transazioni e per registrarle sul ledger della blockchain**. Chiarito questo, aggiungiamo l'implementazione del metodo **MineBlock**, sulla quale è incentrato il funzionamento della blockchain :

```
    void MineBlock(int nDifficulty)
```

```
    {
        char cstr[nDifficulty + 1];
        for (int i = 0; i < nDifficulty; i++)
            cstr[i] = '0';
        cstr[nDifficulty] = '\0';
        string str(cstr);   //cstr becomes a standar string

        do
        {
            _nNonce++;
            sHash = _CalculateHash();
        }
        while (sHash.substr(0, nDifficulty) != str);
        cout << "Block mined: " << sHash << endl;
    }
```

La firma del metodo **MineBlock** prende in input un parametro **nDifficulty**. Si crea un array di caratteri con una lunghezza maggiore del valore specificato per **nDifficulty**, successivamente un ciclo for viene utilizzato per riempire l'array con zeri, seguito dall'elemento finale dell'array a cui viene assegnato il carattere di terminazione della stringa (\0) e successivamente l'array di caratteri o *c-string* viene trasformato in una stringa standard.

Viene quindi utilizzato un ciclo do-while per incrementare **_nNonce** e ad **_sHash** viene assegnato l'output di **_CalculateHash**. Il prefisso (esattamente i primi *nDifficulty* caratteri) dell'hash viene quindi confrontata con la stringa di zeri che abbiamo appena creato; se non viene trovata alcuna corrispondenza, il ciclo viene ripetuto finché non ne viene trovata una. Una volta trovata una corrispondenza, viene inviato un messaggio al buffer di output per dire che il blocco è stato minato con successo, sinonimo del fatto che è stata aggiunta una transazione valida all'interno della blockchain.

L'abbiamo visto menzionato alcune volte prima, quindi ora aggiungiamo il metodo **_CalculateHash** :

```
inline string Block::_CalculateHash() const
{
    stringstream ss;
    ss << _nIndex << sPrevHash << _tTime << _sData << _nNonce;
    return sha256(ss.str());
}
```

La keyword *inline* indica una funzione per la quale si intende scorporare la definizione del metodo dall'implementazione. L'utilizzo di funzioni inline rende la loro esecuzione più veloce in quanto elimina l'overhead associato alle chiamate di funzione, poiché il compilatore inserisce le istruzioni del metodo in linea ovunque si trovi il metodo chiamato; Viene quindi creato un flusso di stringhe(stringstream), seguito dall'aggiunta dei valori per **_nIndex** , **_tTime** , **_sData** , **_nNonce** e **sPrevHash** al flusso. Concludiamo restituendo l'output generato dalla funzione di hash **sha256** a partire dal flusso di stringhe appena generato.

C++ mette a disposizione dell'utente la possibilità di ridefinire il comportamento di alcuni operatori, tra i quali anche l'operatore di redirezione dell'output "<<"

```
friend ostream& operator << (ostream& os, Block& b)
{
    if(!b._verbose)
        return os << "Hash: " << b.sHash << endl;
    os << "Index: " << b._nIndex << endl;
    os << "Data: " << b._sData << endl;
    os << "nNonce: " << b._nNonce << endl;
    os << "Previous: " << (b.sPrevHash != "" ? b.sPrevHash : "NULL") << endl;
    return os << "Hash: " << b.sHash << endl;
}
```

L'overloading viene specificato come funzione membro della classe Block. Si prevede l'utilizzo della keyword ***friend*** in modo da dare accesso alla classe ostream ai contenuti privati della classe Block. La firma prende in input un riferimento ad ostream ed uno ad oggetto Block. L'implementazione prevede che vengano fornite in output le informazioni riguardanti l'oggetto ed in oltre è possibile specificare, al momento dell'istanziazione, se la stampa dovrà avvenire in maniera discorsiva (con informazioni altrimenti omissibili) attraverso l'utilizzo dell'attributo **_verbose**.

Abbastanza semplice ed intuitivo è il comportamento dell'operatore " **!= "** applicato a due oggetti **Block**

```
bool operator != (Block& b){return this->sHash != b.sHash;}
```

La firma del metodo prende in input un reference a **Block** e restituisce true se l'istanza in questione (a cui si accede tramite il puntatore di classe **this**) presenta lo stesso hash crittografico dell'istanza che si trova a destra dell'operatore.

Abbiamo a disposizione abbastanza materiale per concludere implementazione blockchain. Iniziamo includendo l'header file "Block.h" che servirà in quanto le due entità sono legate da una di quelle che nella OOP si definiscono *"relazioni part-of" ,* in particolare una relazione di composizione, ovvero una blockchain si compone di oggetti di tipo Block.

La classe è dotata, come già visto, di due attributi privati : **_vChain** , che implementa la struttura come un vettore di oggetti Block e **_nDifficulty** che rappresenta la complessità del processo di *Proof of Work*.

```
Blockchain(bool _verboseIn = false) : _verbose(_verboseIn)
{
    _vChain.emplace_back(Block(0, "Genesis Block", _verbose));
    _nDifficulty = 4;
}
```

La firma del costruttore prevede un'istanza anche senza l'utilizzo di parametri, infatti in caso di assenza di parametri, il costruttore inizializza di default a *false* l'attributo **_verbose**. Man mano che i blocchi vengono aggiunti alla blockchain, devono fare riferimento al blocco precedente utilizzando il suo hash, ma poiché la blockchain deve iniziare da qualche parte, si deve creare un blocco a cui farà riferimento il blocco successivo, lo si denomina ovviamente *Blocco di Genesi*, il quale viene posizionato sul vettore **_vChain**. Quindi si imposta il livello **_nDifficulty** a seconda di quanto si vuole rendere difficile il processo di validazione del blocco creato.

NB Una difficoltà maggiore comporta un tempo di esecuzione più lungo.

Ora è il momento di scrivere il blocco di codice che consente di aggiungere un blocco alla **blockchain**, attraverso le seguenti righe :

```
    void AddBlock(Block bNew)
    {
        bNew.sPrevHash = _GetLastBlock().sHash;
        bNew.MineBlock(_nDifficulty);
        _vChain.push_back(bNew);
    }
```

La firma del metodo AddBlock , intuitivamente, prevede che venga preso in input un parametro formale Block. Segue l'inizializzazione del campo **sPrevHash** del nuovo blocco con l'hash del blocco che lo precede nella blockchain, che si ottiene attraverso **_GetLastBlock**. Il blocco viene quindi validato utilizzando il metodo **MineBlock** e successivamente registrato nel ledger della struttura attraverso l'aggiunta del blocco al vettore **_vChain**, completando così il processo di aggiunta del blocco alla blockchain.

Adesso serve un modo per verificare se un blocco sia valido all'interno della Blockchain . Il metodo **_isValid** implementa questo *consenso*, di cui si è largamente discusso nella trattazione teorica dell'argomento.

```
    bool _isValid(Block _toCheck)
    {
        for(auto i = _vChain.begin(); i != _vChain.end(); i++)
        {
            if(_toCheck.sHash == (*i).sHash)
                return true;
        }
        return false;
    }
```

La firma prende in input un riferimento a **Block** che simula il blocco per il quale bisogna effettuare la verifica. Un ciclo for scorre l'intera catena a partire dal blocco genesi e restituisce true solo se si verifica che un altro blocco all'interno della catena abbia lo stesso identico hash del blocco preso in esame. La verifica avviene mediante l'operatore di confronto **!=** , implementato in precedenza nella classe **Block** . Non c'è possibilità che due blocchi diversi generino lo stesso hash crittografico, quindi è praticamente impossibile che un malintenzionato riesca a verificare un blocco "estraneo" alla blockchain, vedendosi validata la transazione.

Viene previsto anche un metodo *getter* che restituisce in sola lettura l'ultimo blocco che è stato inserito all'interno della catena.

```
Block _GetLastBlock() const {return _vChain.back();}
```

Ovviamente, come detto anche sopra, si rende doveroso il modificatore const in modo da impedire modifiche al di fuori della classe.

Anche qui, come per la classe Block, viene prevista la ridefinizione dell'operatore di redirezione dell'output " << ".

```
    friend ostream& operator<< (ostream& os, Blockchain& b)
    {
        os << "\nBlockchain:\n" << endl;
        for(auto i = (b._vChain).begin(); i != (b._vChain).end(); i++)
            os << *i << endl;
        return os << endl;
    }
```

Non viene analizzata troppo l'implementazione, in quanto è perfettamente allineata con quella seguita per la classe Block, ma si sposta l'attenzione sulla ridefinizione di un altro operatore " **[ ]** ", utile per l'accesso ad un blocco specifico nella catena, attraverso un indice posizionale.

```
    Block operator[] (int i){return _vChain.at(i);}
```

Conclusa l'implementazione di quest'ultima classe si giunge al test della struttura appena realizzata attraverso la funzione main, basta includere l'heade file "Blockchain.h" tramite la solita direttiva del preprocessore *#include.*

```
#include "Blockchain.h"

int main()
{
    Blockchain bChain = Blockchain(true);

    cout << "\nMining block 1..." << endl;
    bChain.AddBlock(Block(1, "Block 1 Data", true));

    cout << "\nMining block 2..." << endl;
    bChain.AddBlock(Block(2, "Block 2 Data", true));

    cout << "\nMining block 3..." << endl;
    bChain.AddBlock(Block(3, "Block 3 Data", true));

    cout << bChain << endl;

    //Blocco valido
    Block good = bChain[1];
    cout << "First block to check: " << endl;
    cout << good << endl;
    cout << (bChain._isValid(good) ? "Block valid" : "Block is not valid") << endl;

    //Blocco non valido
    Block bad(5, "Block bad Data"); // stampa non discorsiva
    cout << "\nSecond block to check: " << endl;
    cout << bad << endl;
    cout << (bChain._isValid(bad) ? "Block valid" : "Block is not valid") << endl;
}
```

Si istanzia un oggetto di classe **Blockchain** mediante l'invocazione del metodo costruttore. Questo crea una nuova blockchain e informa l'utente che un blocco viene estratto stampando nel buffer di output quindi crea un nuovo blocco e lo aggiunge alla catena; il processo per l'inserimento di quel blocco non terminerà quindi fino a quando esso non verrà verificato tramite un hash valido. Una volta inserito il blocco, il processo viene ripetuto per altri due blocchi. Viene poi stampata l'intera blockchain (in modo discorsivo), e si può valutare che in effetti ogni blocco è correttamente collegato a quello "minato" in precedenza tramite il corrispettivo hash. Segue un test di validazione di due blocchi : il primo valido in quanto copia di un blocco già esistente nella catena, avvenuta tramite l'operatore di accesso **[ ]** , il secondo simula un blocco estraneo alla blockchain che viene correttamente rifiutato dal sistema. Questo Toy-Example mostra il corretto funzionamento della struttura realizzata. Seguono alcuni esempi di output generati dal codice a diversi livelli di difficoltà :

```
Mining block 1...
Block mined: 000000d47ae999af5d2a7113d0e2e11b80f72f83de5cc43372d7e269f7dce690
Mining block 2...
Block mined: 00000036bdfbc4c735cb2be2cb151593b49500363c27258db0357be313497b64
Mining block 3...
Block mined: 0000003137bd0eaf7984684fb3b487e81b5109ba3c04554224a1d5f920b172e9
```

```
Mining block 1...
Block mined: 000065eeb9e12b92fb126b795bf7ebbe29743229d6f4b56652175a7d1e324294
Mining block 2...
Block mined: 00008d35d873c774a2ec0c9e428f1f954c67fd13c0f5d95d6c225cec22e1ce6f
Mining block 3...
Block mined: 000088d949a468c6bde581b2b3b2623bbbabb465b224f59fc66d2600c1eb081b
```

La stampa dell'intera catena :

```
Blockchain:

Index: 0
Data: Genesis Block
nNonce: 0
Previous: NULL
Hash: c86eb5f73f17c989d15dd2fabca358d9a963d0b7e02dcd307d4520e409e017b7

Index: 1
Data: Block 1 Data
nNonce: 126698
Previous: c86eb5f73f17c989d15dd2fabca358d9a963d0b7e02dcd307d4520e409e017b7
Hash: 00000bcd1d48599d5e6588106cbe843fdab372beb36c50e7df4def4ee31bd890

Index: 2
Data: Block 2 Data
nNonce: 127542
Previous: 00000bcd1d48599d5e6588106cbe843fdab372beb36c50e7df4def4ee31bd890
Hash: 000025dfc684dcabbfc003beffa4ea0d654681cdc649c844855f2b31d7d7354f

Index: 3
Data: Block 3 Data
nNonce: 17796
Previous: 000025dfc684dcabbfc003beffa4ea0d654681cdc649c844855f2b31d7d7354f
Hash: 00003fc868276919ad39fa837f8109b120e504316aa8108f1eeadb34226add80
```

Processo di consenso con esito differente per due blocchi :

```
First block to check:
Index: 1
Data: Block 1 Data
nNonce: 126698
Previous: c86eb5f73f17c989d15dd2fabca358d9a963d0b7e02dcd307d4520e409e017b7
Hash: 00000bcd1d48599d5e6588106cbe843fdab372beb36c50e7df4def4ee31bd890

Block valid

Second block to check:
Hash: f4f05913203e60034f4622cc7ef987a524e07ad02f0c4140ca0688546e0071ab

Block is not valid
```