



UNIVERSIDAD DE BUENOS AIRES  
FACULTAD DE INGENIERÍA  
Año 2020 - 2.<sup>er</sup> cuatrimestre

95.12 ALGORITMOS Y PROGRAMACIÓN II  
TRABAJO PRÁCTICO N.º 0 - PROGRAMACIÓN C++

Estudiantes:

Ochagavia, Lara	100637
<code>lochagavia@fi.uba.ar</code>	
Pintos, Gastón Maximiliano	99711
<code>gmpintos@fi.uba.ar</code>	
Ferreyra, Lucas Ariel	97957
<code>lferreyra@fi.uba.ar</code>	

Profesores:

CALVO, Patricia Mabel  
SANTI, Lucio  
SANTI, Leandro

Fecha de entrega: 12 de Noviembre del año 2020

# Índice

<b>1. Objetivo del proyecto</b>	<b>2</b>
<b>2. Descripción del proyecto</b>	<b>2</b>
2.1. Organización de la ALGOCHAIN . . . . .	2
<b>3. Desarrollo</b>	<b>3</b>
3.1. Consideraciones previas: . . . . .	3
3.2. Representación de datos en la ALGOCHAIN . . . . .	4
3.2.1. Clase Vector . . . . .	4
3.2.2. Clase <i>Block</i> . . . . .	4
3.2.3. Clase Transaction . . . . .	6
3.2.4. Clase Cmdline . . . . .	7
3.2.5. Clase SHA256 . . . . .	7
3.3. Implementación del programa: . . . . .	8
3.4. Compilación del programa: . . . . .	8
3.4.1. Makefile . . . . .	9
3.5. Ejecución del programa . . . . .	9
3.5.1. Corridas de prueba del programa: . . . . .	9
<b>4. Conclusiones</b>	<b>12</b>
<b>5. Código Fuente</b>	<b>12</b>
5.1. main.cc . . . . .	12
5.2. Vector.h . . . . .	15
5.3. Block.h . . . . .	17
5.4. Block.cc . . . . .	18
5.5. Transaction.h . . . . .	21
5.6. Transaction.cc . . . . .	22
5.7. cmdline.h . . . . .	23
5.8. cmdline.cc . . . . .	24
5.9. sha256.h . . . . .	28
5.10. sha256.cc . . . . .	29
5.11. Utilities.h . . . . .	31
5.12. Utilities.cc . . . . .	31

## 1. Objetivo del proyecto

El objetivo del siguiente trabajo práctico es desarrollar un programa que sea capaz de generar un bloque válido de la cadena *Algochain* como *stream* de salida a partir de transacciones ingresadas por un *stream* de entrada, utilizando como herramienta el lenguaje C++. El *stream* de entrada y salida son identificados por el usuario por línea de comando de argumento al momento de ejecutar el programa.

El siguiente informe detalla las tareas a realizar por los alumnos con una descripción de la *Algochain* para una mayor comprensión del trabajo. Están detalladas en el informe las estrategias de programación utilizadas como así también el uso de distintas herramientas propias del lenguaje, como son los *Templates* y el paradigma de programación orientada a objetos. Por último, se presentan las corridas del proyecto junto con las conclusiones y observaciones desarrolladas durante la realización del mismo.

## 2. Descripción del proyecto

El presente trabajo práctico refiere al uso de una estructura denominada *Algochain*. Dicha estructura se basa en la lista enlazada de bloques que contienen información sobre transacciones de criptomonedas *bitcoin* denominada *Blockchain*. Para este trabajo no se tienen en cuenta ciertas validaciones sobre las transacciones, que deben realizarse para detectar si son válidas o no. Estas consideraciones serán abarcadas en los próximos trabajos prácticos.

La *Algochain* se compone de bloques que agrupan transacciones. A su vez, las transacciones constan de una secuencia de *inputs* y otra de *outputs*. Un bloque debe ser válido para ser admitido en la *Algochain*. Esta validez está sujeta al minado de los bloques, tarea que consiste en calcular un doble *hash* del *header* para contrastarlo con la dificultad del minado ingresada. La estructura a alto nivel está representada en la figura 2.1.

Además, para los campos de identificación de cada *input* y *output* de cada transacción son utilizadas funciones de *hash* criptográficas que identifican unívocamente a las partes involucradas por medio de claves. Para el desarrollo de este trabajo práctico se adopta la función *SHA256* provista por la cátedra para generar dichos *hashes*.

### 2.1. Organización de la ALGOCHAIN

La estructura *Algochain* esta conformada de la siguiente manera:

- **Bloque:** Está constituido por un *header* y un *body*. El *header* contiene los campos que identifican al bloque: *prev\_block* contiene el *hash* del bloque completo anterior al bloque actual, *txns\_hash* es un campo que contiene el *hash* de todas las transacciones incluidas en el bloque, el campo de *bits* contiene la dificultad de minado del bloque y el *nonce* es un número arbitrario que se utiliza para poder generar *hashes* sucesivos hasta satisfacer la dificultad del minado. El *body* consta de dos campos: el *txn\_count*, que es la cantidad de transacciones que componen al bloque y *txns*, que almacena todas las transacciones del bloque.
- **Transacciones:** Cada transacción se compone de un casillero que indica el número de *inputs* denominado *n\_tx\_in* y otro que indica el número de *outputs* denominado *n\_tx\_out*. Los *inputs* y los *outputs* están almacenados cada uno en un campo con su mismo nombre.

Cada **input** esta compuesta por tres campos: *tx\_id* es el *hash* de la transacción de donde el *input* toma los fondos, *idx* es el índice sobre la secuencia de *outputs* de la transacción con el *hash* *tx\_id* y el *addr* es un *hash* de la dirección origen de los campos.

Cada **output** esta compuesto de los siguientes campos: *value* es un número de punto flotante correspondiente a la cantidad de *Algocoins* a transferir a ese *output* y *addr* es el *hash* de la dirección de destino de tales fondos.

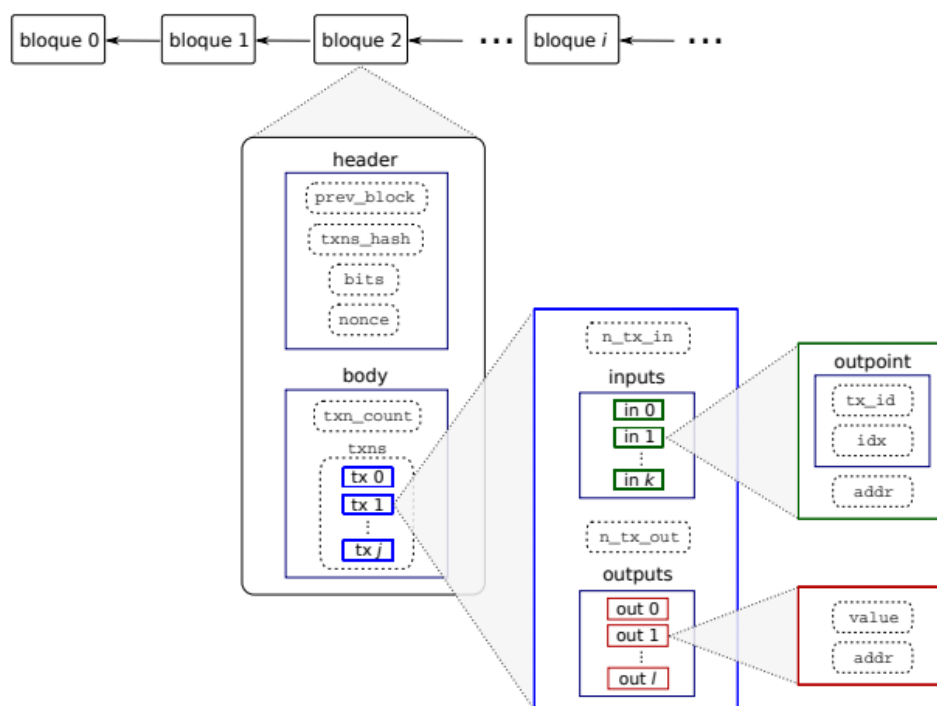


Figura 2.1: Esquema de la estructura Algochain.

### 3. Desarrollo

#### 3.1. Consideraciones previas:

El programa debe de cumplir con el paradigma de programación orientado a objetos. Toda la información tratada a lo largo del programa se maneja a través de objetos pertenecientes a clases. La definición de clases, y la instanciación de objetos, permite manejar la información de manera prolija, eficiente y segura al cumplir con el ocultamiento y encapsulamiento de la información perteneciente a cada objeto.

Para los campos cuya información era resultado de una función de *hash* se implementa el uso de cadenas de caracteres definidas en la clase *string*. Esto es necesario ya que cada *hash* cuenta con 64 caracteres, cada uno representando un número hexadecimal, por lo que el manejo de esta información era mas simple en formato de *string*. El alcance de la definición de un número entero no es suficiente para definir un *hash* y era necesaria la implementación de modificadores y nuevas estructuras que hubieran dificultado el manejo del código.

Los datos que conforman a cada transacción deben de cumplir un formato estricto enunciado en la consigna provista por la cátedra. Ante el no cumplimiento de este formato el código detecta que la información es corrupta. El error detectado se informa por el flujo de errores estándar *cerr*. El mensaje de error especifica el campo de información corrupta.

El formato a respetar es el siguiente: la primera línea debe contener la cantidad de *inputs* a esperar, y esa cantidad de líneas siguientes deben consistir en un *hash* que indica la última transacción en la cual participó el usuario que está transfiriendo fondos, un espacio, un valor entero positivo que indica la posición del *output* de dicha transacción en la cual se hace referencia al usuario que transfiere fondos, un espacio, y finalmente el *hash* del usuario que transfiere fondos. Una vez terminados los *inputs* debe aparecer en una nueva línea la cantidad de *outputs* a esperar, y esa cantidad de líneas siguientes deben consistir en un número flotante positivo que indique el valor a recibir por el usuario que recibe fondos en la presente transferencia, un espacio y el *hash* del usuario que recibe los fondos. Todas las líneas deben estar separadas por '\n' indicando, precisamente, el comienzo de una nueva línea.

La invocación del programa es por línea de argumentos de comando según el siguiente formato: se utilizan los nombres largos de cada opción y se especifican los archivos para el flujo de entrada y salida, se utilizan los nombres cortos de las opciones con el valor por defecto de los flujos de entrada y salida o bien una combinación de todo lo mencionado previamente. Hay que tener en cuenta que el argumento para la opción de dificultad, sea expresando su nombre largo o su nombre corto, es obligatorio.

```
./programa --input Archivo.txt --output Archivo.txt --difficulty dificultad

./programa -i - -o - -d dificultad

./programa --input Archivo.txt -o - -d dificultad
```

## 3.2. Representación de datos en la ALGOCHAIN

Según la estructura presentada en la sección anterior, se procede a ilustrar las estructuras de datos implementadas para poder procesar los datos y almacenarlos de manera que se conforme el bloque de la figura 2.1.

### 3.2.1. Clase Vector

Para los arreglos de elementos vistos en la figura 2.1, fue utilizada la clase *Vector*. Esta clase fue definida como un *template* para que pueda ser utilizada con distintos tipos de datos, y a su vez utilizarse sus métodos a lo largo del código sin necesidad de definir para cada tipo de dato una nueva clase vector.

La clase tiene como atributos a una variable que almacena la cantidad de elementos en el arreglo y a un puntero a los elementos de un arreglo dinámico que contiene datos de tipo T.

Los métodos definidos para esta clase son los necesarios para establecer operaciones con arreglos como lo es la sobrecarga del operador indexación o el operador asignación, junto con los métodos *setters* y *getters* comunes a cada clase.

Cuenta con cuatro tipos de constructores: dos por copia, ya que uno instancia un arreglo de igual cantidad de elementos mientras que el otro genera un nuevo arreglo con la cantidad de elementos según lo ingresado por parámetro de la función, otro por defecto y un último en el que se especifica la cantidad de elementos a almacenar en el arreglo. Todos estos constructores utilizan memoria dinámica para crear el arreglo por medio de la instrucción *new*. Por lo tanto, fue necesario implementar un método destructor para eliminar de forma correcta la memoria asignada.

```
1  template <typename T>
2
3  class Vector
4  {
5  private:
6      int size; // Cuanto va a tener de largo el Array
7      T *ptr; // Puntero al vector dinamico que contiene los datos de tipo T
8
9  public:
10     Vector();
11     Vector(int);
12     Vector(const Vector <T> &);
13     Vector(int, const Vector <T> & );
14     ~Vector();
15
16     Vector<T>& operator=(const Vector <T> &);
17     bool operator==(const Vector <T> &);
18     T & operator[](int);
19
20     int getSize();
21 };
```

### 3.2.2. Clase Block

En la clase *Block* se busco instanciar un objeto que cuente con la misma información detallada en la figura 2.1. En base a eso se definieron dos estructuras denominadas *header* y *body*. En cada una de ellas se encuentran sus respectivos campos de información. Esta forma de encasillar la información en estructuras permite separar de forma ordenada la información tal como se ilustra en la figura. Por lo tanto, la clase cuenta con dos punteros a cada estructura como atributos.

Se desarrollaron un constructor, un destructor, y métodos para cargar información, denominado *load\_Block* y para imprimir la información contenida en cada bloque denominado *print\_Block*.

Para la instanciación de un objeto *Block* fue necesario pedir memoria, ya que se cuenta con punteros a estructuras como atributo y por este motivo fue necesario el desarrollo explícito de un método destructor para liberar la memoria asignada al objeto.

El método de carga de información, *load\_Block()*, recibe como parámetros el flujo de entrada en donde se encuentra la información necesaria para constituir un bloque y la dificultad con la cual se requiere realizar la tarea de minado. El flujo de entrada debe ser validado ante un archivo corrupto. Dentro de la función se utilizan métodos de la clase *stream* como lo son *getline()* para la lectura del archivo y la asignación de información en cada campo de la estructura.

Para cargar el campo que indica la cantidad de transacciones ingresadas, *txns\_count* se utiliza la función *getline(istream &, string &)* y en la documentación de dicha función se aclara que se extrae la línea leída del *istream* para guardarla en el *string* indicado, por lo que no se puede hacer una lectura del *istream* previa o posterior a la carga del Bloque utilizando esta función. Dicho esto, se realizó el conteo de manera simultánea a la carga de la información, iterativamente.

También se realiza una verificación de la información que contiene el flujo de entrada. En caso de encontrar un archivo vacío se procede a anular la tarea de almacenar las transacciones y el campo *txns\_count* se carga con el valor 0. En el caso de encontrar información que no cumple con el formato establecido se procede a terminar el almacenamiento de información con la función *exit()* previo advertencia e impresión del error identificado en el flujo de errores *cerr*.

Existe una verificación del valor de dificultad. Para esto, se calcula dos veces la función de *hash* a todos los campos que componen el bloque. Una vez obtenido este *hash*, se toma una copia y se recortan la cantidad de caracteres necesarios según el numero de dificultad ingresado. Luego, se realiza una comparación de estos caracteres convertidos a numero binario con una cadena que contiene una cantidad de ceros determinada por el valor de la dificultad del programa. En caso de no cumplir con la dificultad del minado, se realiza nuevamente la operación de doble *hash* pero esta vez variando de forma aleatoria el número que contiene el campo *nonce*. Esta operación se realiza las veces que sean necesario hasta cumplir con la condición de dificultad. Una vez superada la condición de dificultad se completa la operación de almacenamiento del bloque. Para que esta implementación funcione adecuadamente, se valida que la dificultad ingresada en la línea de comandos sea no solo un entero si no que también sea positiva.

El método de impresión, *print\_Block* recibe un flujo de salida previamente validado por donde se imprimirá la información contenida en el bloque.

```
1  struct header_t
2  {
3      string prev_block;
4      string txns_hash;
5      int bits; // Dificultad
6      int nonce; // Para el hash
7  };
8
9  struct body_t
10 {
11     int txns_count;
12     Vector <Transaction> txns;
13 };
14
15 class Block
16 {
17 private:
18     header_t * header;
19     body_t * body;
20
21 public:
22     Block();
23     ~Block();
24     void load_Block(istream *, const int &);
25     void print_Block(ostream *);
26 };
```

### 3.2.3. Clase Transaction

Esta clase se crea para instanciar un objeto que tenga como atributos cada uno de los campos de información que constituyen a cada transferencia.

Para lograr que cada transferencia respete el formato de la figura 2.1 se implementan dos estructuras auxiliares denominadas *input\_t* y *output\_t*, las cuales están formadas por los campos que componen a cada *input* y a cada *output* respectivamente.

Una vez definidas estas estructuras, se forma un Vector de estructuras para almacenar todos los *inputs* y todos los *outputs*. Además, la clase Transacción tiene dos atributos de clase que informan la cantidad de elementos que contienen dichos Vectores. Estos se denominan *n\_tx\_in* para el Vector de estructuras *inputs* y *n\_tx\_out* para el Vector de estructuras *outputs*.

En la parte pública de esta clase se encuentran un constructor y un destructor. En cuanto a los métodos de la clase se encuentran un método para cargar información, *load\_Transaction*, un método para imprimir esta información, *print\_Transaction*, y los métodos *getters* para cada atributo de la clase.

El constructor *Transaction()* genera una instancia de la clase en donde los campos *n\_tx\_in* y *n\_tx\_out* son inicializados con el carácter '0' al ser un tipo de dato *string*. El destructor *~Transaction()* es el destructor por defecto, y es implementado por completitud en la definición de la clase.

Los métodos *getters()* son implementados como solución a la función de carga del *Block*, en la que es necesario acceder a ciertos valores de los atributos de la clase *Transaction*. Con estos métodos se busca respetar la definición de clase, en donde se encapsula la información contenida del objeto y solo se puede acceder a la información de los atributos por medio de los métodos definidos dicha clase.

El método *load\_Transaction* recibe por argumento un flujo de entrada en donde se encuentra la información necesaria para formar cada una de las transacciones y no devuelve ningún valor. El flujo de entrada debe de contener un archivo, previamente validado, que contenga la información de cada transacción respetando el formato enunciado.

Para obtener la información de cada atributo se utiliza la función *getline()* especificando para cada caso el campo de destino y el delimitador. En caso de omisión, el delimitador es el carácter '\n'. Siendo que la cantidad de *inputs* y *outputs* preceden a la información de los mismos, se procedió a rellenar el Vector de estructuras para cada uno respectivamente de forma iterativa y teniendo como condición de corte a la cantidad de elemento previamente informada.

Luego de ingresar la información a cada atributo, se realiza una verificación de errores de formato. En el caso de las cadenas de caracteres, estas deben de contener cadenas de *hash* que tiene un largo de 64 caracteres alfanuméricos y se incurriría en un error al detectar que el largo del atributo o los caracteres que lo componen no cumplen con esta condición.

Para el caso de los campos que contiene valores numéricos, es necesario validar ante la inclusión de caracteres no numéricos ya que esto no respetaría el formato.

El método de impresión *print\_Transaction* recibe un flujo de salida previamente validado por donde se imprimirá la información contenida en cada transacción de forma iterativa.

```
1 struct input_t {
2
3     std::string tx_id;
4     std::string idx;
5     std::string addr;
6 };
7
8 struct output_t {
9
10    std::string value;
11    std::string addr;
12 };
13
14
15 class Transaction
16 {
17 private:
18     std::string n_tx_in;
19     std::string n_tx_out;
20
21     Vector <input_t> inputs;
```

```
22     Vector <output_t> outputs;
23
24 public:
25     Transaction();
26     ~Transaction();
27     void load_Transaction(istream *);
28     void print_Transaction(ostream *);
29     std::string get_n_tx_in(){return n_tx_in;};
30     std::string get_n_tx_out(){return n_tx_out;};
31     input_t getInput(int);
32     output_t getOutput(int);
33
34 };
```

### 3.2.4. Clase Cmdline

Esta clase fue provista por la cátedra. Tiene como objetivo la validación de los argumentos ingresados por línea de comando por el usuario.

Posee como atributo un puntero a una estructura denominada *option\_t*. Esta estructura cuenta con la definición de los campos que componen a cada argumento. Bajo esta definición se puede considerar: si la opción deseada posee un valor o argumento que debe ser ingresado, cuál es el nombre largo y corto con el que se reconoce la opción ingresada por el usuario, con qué función debe ser tratada esta opción (informado a través del puntero a función) y *flags* para indicar si la opción es obligatoria.

Se encuentran un constructor por defecto y un constructor por copia, y, dentro de sus métodos, las funciones para procesar la opción ingresada según se halla utilizado su nombre corto o su nombre largo, y una función de parseo.

En el programa se utiliza esta clase para hacer el tratamiento de las opciones ingresadas por línea de comando. Con estas funciones se pueden validar: el flujo de entrada de la información, que puede ser a través de un archivo especificado por el usuario o bien el flujo de entrada estándar *cin*, el flujo de salida, que puede ser por un archivo especificado por el usuario o bien el flujo de salida estándar *cout* y el grado de dificultad que requiera la tarea de minado. Esta ultima opción es obligatoria para el funcionamiento del programa.

```
1  struct option_t {
2      int has_arg;
3      const char *short_name;
4      const char *long_name;
5      const char *def_value;
6      void (*parse)(std::string const &); // Puntero a funcion de opciones
7      int flags;
8  };
9
10 class cmdline {
11
12     option_t *option_table;
13     cmdline();
14     int do_long_opt(const char *, const char *);
15     int do_short_opt(const char *, const char *);
16
17 public:
18     cmdline(option_t *);
19     void parse(int, char * const []);
20 };
```

### 3.2.5. Clase SHA256

Esta clase fue provista por la cátedra. Su función principal es generar las cadenas de caracteres de *hash* utilizando la función *SHA256* a partir de cadenas de caracteres ingresadas como argumento. El procesamiento de la información y las tareas que realiza esta clase están fuera del alcance de este



trabajo práctico. En el programa se utiliza para generar las cadenas de caracteres para el campo *txn\_hash* dentro del *header* del *Block* y para comprobar la condición del minado al realizar un doble *hash* a la concatenación de todos los campos que forman parte del *header* del *Block*.

```
1  class SHA256
2  {
3  protected:
4      typedef unsigned char uint8;
5      typedef unsigned int uint32;
6      typedef unsigned long long uint64;
7
8      const static uint32 sha256_k[];
9      static const unsigned int SHA224_256_BLOCK_SIZE = (512/8);
10 public:
11     void init();
12     void update(const unsigned char *message, unsigned int len);
13     void final(unsigned char *digest);
14     static const unsigned int DIGEST_SIZE = ( 256 / 8);
15
16 protected:
17     void transform(const unsigned char *message, unsigned int block_nb);
18     unsigned int m_tot_len;
19     unsigned int m_len;
20     unsigned char m_block[2*SHA224_256_BLOCK_SIZE];
21     uint32 m_h[8];
22 };
```

### 3.3. Implementación del programa:

El programa funciona de la siguiente manera: La función principal denominada *main()* recibe como parámetros la línea de comandos ingresada por el usuario. Lo primero que realiza es la instanciación de un objeto *cmdline* para poder procesar las opciones con la función *cmdline::parse()*. Dentro del programa se utilizan variables globales en donde se almacenan los flujos de entrada y salida que fueron seleccionados según las opciones ingresadas.

Una vez inicializados los flujos de entrada y salida que serán utilizados en el programa se procede a formar el bloque de la *Algochain*. Para ello se crea un objeto de la clase *Block* con el constructor sin argumentos. Luego, se utiliza el método *Block::load\_Block()* para almacenar la información que contiene el flujo de entrada en el objeto.

Una vez almacenada la información, se procede a la impresión del objeto con el método de la clase *Block::print\_Block()* utilizando el flujo de salida determinado por el usuario.

### 3.4. Compilación del programa:

Para compilar el proyecto se utiliza la herramienta *make* para lo cual se genera un archivo *Makefile*. El compilador utilizado es el *g++* en su versión 9.3.0. Los *flags* especificados en la línea de compilación son: *-g, -w, -Wall*.

En el directorio */TP0.Grupo14* donde se encuentra el archivo *Makefile*, se encuentra la carpeta *src* que contiene todos los archivos *.h* y *.cc* utilizados en el proyecto. Estos archivos son:

* Block.cc	* Block.h
* Transaction.cc	* Transaction.h
* cmdline.cc	* cmdline.h
* Utilities.cc	* Utilities.h
* sha256.cc	* sha256.h
* main.cc	* Vector.h

Para compilar, se ejecuta la herramienta *make* desde el directorio */TP0\_Grupo14*. El nombre del ejecutable será *TP0\_G14*. Se muestra a continuación el archivo *Makefile* utilizado

### 3.4.1. Makefile

```

1
2  #CC specifies which compiler we're using
3  CC = g++
4
5  #COMPILER_FLAGS specifies the additional compilation options we're using
6  # -w suppresses all warnings
7  COMPILER_FLAGS = -g -w -Wall
8
9  #This is the target that compiles our executable
10
11 SRC_DIR := src
12 OBJ_NAME := TP0_G14
13
14 all :
15     $(CC) $(wildcard $(SRC_DIR)/*.cc) $(COMPILER_FLAGS) -o $(OBJ_NAME)

```

## 3.5. Ejecución del programa

Para ejecutar el programa, el usuario debe situarse en la carpeta donde se encuentre el ejecutable llamado *TP0\_G14*.

### 3.5.1. Corridas de prueba del programa:

A continuación se exhibirán distintas corridas del programa que funcionaron como prueba para evaluar el correcto funcionamiento del mismo.

Cuando el bloque se cargue correctamente y se pueda imprimir su contenido, se imprimirá en las corridas de prueba el *hash* del *header* para corroborar que se cumpla con la condición de la dificultad del minado.

- **Corrida del programa utilizando las opciones por defecto:** En este caso el usuario deberá ingresar las transacciones por el flujo de entrada estándar. Para indicar el corte de información de entrada, deberá ingresarse EOF, que, desde la terminal de Ubuntu se consigue con Ctrl+D.

Línea de comando:

```
./TP0_G14 -i - -o - -d 5
```

Resultado:

```

ferre@LUCASFERRERVA: /mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i - -o - -d 5
3
2b2a0ff8f9def3f17d58512ab10d7b1b40e5ae06dabdbddfd4d9278a428a3700c 9 7cadab457ad8d811f134612436daaa5e5914b20dc2502865f714035b0f267680
ab3691d8e45c1f50684b2762fc640afbe61266bf49c34e0a5319044da23af364 4 7c9e7c1494b2684ab7c19d6aff737e460fa9e98d5a234da1310c97ddf5691834
b2471d941bf888366cf43813752ea2ebf08254773cf116187cdd6d0463a50a 1 e9e326d5f3b4741fe5967b5f9f3997e6275331ba18567ef9ef9e0e3a00e78371
2
250.5 ca978112ca1bbdcfac231b39a23dc4da786eff8147c4e72b9807785afee48bb
150.7 3b64db95cb55c763391c707108489ae18b4112d783300de38e033b4c98c3deaf
1
7f82ec7e2862b56289ceaaeb057640d924ec668bbc3262444a41a1e638ff139 4 b2bd1e0493169ae5088d0766603d24bc6660fe188fb62865aa25362b1e2c94da
1
99.9 948721bfc4815b348dc5ccc66804f8841b621bde0ad0054cc7df22df183a9cf2
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
6559112342d7bb4eba14fb1a9b4ee58c8c0b3d08362a74cd4d73870904e3c41d
5
1658549
2
3
2b2a0ff8f9def3f17d58512ab10d7b1b40e5ae06dabdbddfd4d9278a428a3700c 9 7cadab457ad8d811f134612436daaa5e5914b20dc2502865f714035b0f267680
ab3691d8e45c1f50684b2762fc640afbe61266bf49c34e0a5319044da23af364 4 7c9e7c1494b2684ab7c19d6aff737e460fa9e98d5a234da1310c97ddf5691834
b2471d941bf888366cf43813752ea2ebf08254773cf116187cdd6d0463a50a 1 e9e326d5f3b4741fe5967b5f9f3997e6275331ba18567ef9ef9e0e3a00e78371
2
250.5 ca978112ca1bbdcfac231b39a23dc4da786eff8147c4e72b9807785afee48bb
150.7 3b64db95cb55c763391c707108489ae18b4112d783300de38e033b4c98c3deaf
1
7f82ec7e2862b56289ceaaeb057640d924ec668bbc3262444a41a1e638ff139 4 b2bd1e0493169ae5088d0766603d24bc6660fe188fb62865aa25362b1e2c94da
1
99.9 948721bfc4815b348dc5ccc66804f8841b621bde0ad0054cc7df22df183a9cf2

```

Figura 3.1: Resultado de haber corrido el programa con la línea de comando *./TP0\_G14 -i - -o - -d 5*

- **Corrida del programa utilizando archivos fuente:**

Línea de comando:

```
./TP0_G14 -i transacc.txt -o bloque.txt -d 5
```

Resultado:



Figura 3.2: Archivo de entrada 'transacc.txt' y archivo resultante 'bloque.txt'

- **Corrida del programa para un archivo de entrada vacío:** empty.txt es un archivo vacío. En este caso se utilizó una dificultad 10 para el minado, y se imprime por flujo de salida estándar.

Línea de comando:

```
./TP0_G14 -i empty.txt -o - -d 10
```

Resultado:

```
ferre@LUCASFERRER:~/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i empty.txt -o - -d 10
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
cd372fb85148700fa88095e3492d3f9f5beb43e555e5ff26d95f5a6adc36f8e6
10
44580
0
```

Figura 3.3: Resultado impreso por la terminal ante la entrada de un archivo vacío

- **Corrida del programa para un archivo corrupto:** corrupto.txt es un archivo que no respeta el formato en que deben ingresarse las transacciones. Se muestra el resultado por flujo de salida estándar. El resultado muestra distintos tipos de errores que pueden aparecer dependiendo la falla en la entrada.

Línea de comando:

```
./TP0_G14 -i corrupto.txt -o - -d 20
```

Resultado:

```

ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for n_tx in format or correct amount of outputs
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for tx_id format or correct amount of inputs.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for idx format.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for addr format.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for tx_id format or correct amount of inputs.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for n_tx out format or correct amount of inputs.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for value format or correct amount of outputs.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d 20
Invalid input information: check for addr format

```

Figura 3.4: Resultados de los errores al detectar fallas en el formato del archivo de entrada

- **Corrida del programa utilizando Valgrind:** Por último, se utilizó la herramienta *Valgrind* para verificar que no haya fugas de memoria en el programa. Esta herramienta es útil para verificar que se haya liberado la memoria pedida con *new* durante el programa.

Lineas de comando:

```

valgrind --tool=memcheck ./TP0_G14 -i transacc.txt -o - -d 13
valgrind --tool=memcheck ./TP0_G14 -i corrupto.txt -o - -d 13

```

Resultado:

```

ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ valgrind --tool=memcheck ./TP0_G14 -i transacc.txt -o - -d 13
==105== Memcheck, a memory error detector
==105== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==105== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==105== Command: ./TP0_G14 -i transacc.txt -o - -d 13
==105==
==105== error calling PR_SET_PTRACER, vgdb might block
=====
6559112342d7bb4eba14fb1a9b4ee58c8c0b3d08362a74cd4d7387090ae3c41d
13
1348757
2
3
2b2a0ff8f9def3f17d58512ab10d7b1b40e5ae06dabddfd4d9278a428a3700c 9 7cadab457ad8d811f134612436daaa5e5914b20dc2502865f714035b0f267680
ab3691d8e45c1f50684b2762fc640afbe61266bf49c34e0a5319044da23af364 4 7c9e7c1494b2684ab7c19d6aff737e460fa9e98d5a234da1310c97ddf5691834
b2471d941bf888366cf43813752ea2ebf08254773cf116187cdd6d0463a50a 1 e9e326d5f3b4741fe5967b5f9f3997e6275331ba18567ef9ef9e0e3a00e78371
2
250.5 cs078112ca1bbdcfac231b30a23dc4d4786eff8147ce72b9087785afee48bb
150.7 3b64db95cb55c763391c707108489ae1804112d783300de30e033b4c90c3deaf
1
7f82ec7e2862b56289ceae057640d924ec668bbc326244a441a1e638ff139 4 b2bd1e0493169ae5088d8766603d24c6660fe188fb62865aa25362b1e2c94da
1
99.9 948721bfc4815b348dc5ccc66804f8841b621bde0ad0054cc7df22df183a9cf2
==105==
==105== HEAP SUMMARY:
==105==   in use at exit: 0 bytes in 0 blocks
==105==   total heap usage: 3,428 allocs, 3,428 frees, 2,035,355 bytes allocated
==105==
==105== All heap blocks were freed -- no leaks are possible
==105==
==105== For lists of detected and suppressed errors, rerun with: -s
==105== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figura 3.5: Resultado obtenido al ejecutar valgrind con un archivo con formato correcto

```

ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ valgrind --tool=memcheck ./TP0_G14 -i corrupto.txt -o - -d 13
==106== Memcheck, a memory error detector
==106== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==106== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==106== Command: ./TP0_G14 -i corrupto.txt -o - -d 13
==106==
==106== error calling PR_SET_PTRACER, vgdb might block
Invalid input information: check for n_tx_out format or correct amount of inputs.
==106==
==106== HEAP SUMMARY:
==106==   in use at exit: 1,602,488 bytes in 209 blocks
==106==   total heap usage: 213 allocs, 4 frees, 1,693,464 bytes allocated
==106==
==106== LEAK SUMMARY:
==106==   definitely lost: 0 bytes in 0 blocks
==106==   indirectly lost: 0 bytes in 0 blocks
==106==   possibly lost: 0 bytes in 0 blocks
==106==   still reachable: 1,602,488 bytes in 209 blocks
==106==             of which reachable via heuristic:
==106==               newarray      : 1,602,000 bytes in 202 blocks
==106==   suppressed: 0 bytes in 0 blocks
==106== Rerun with --leak-check=full to see details of leaked memory
==106==
==106== For lists of detected and suppressed errors, rerun with: -s
==106== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figura 3.6: Resultado obtenido al ejecutar valgrind con un archivo con formato corrupto

Como puede verse en las imágenes de las figuras 3.5 y 3.6, los resultados obtenidos no son los mismos. Esta diferencia se debe a que cuando se detecta un archivo corrupto, se finaliza al programa con la función *exit()* la cual no invoca a los destructores de los objetos del programa. No obstante, aunque



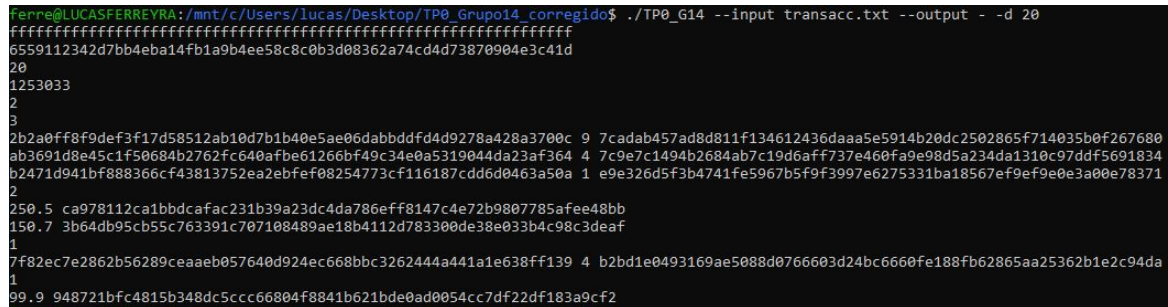
esa memoria podría considerarse perdida, el proceso finaliza y, consecuentemente, toda la memoria que tenía es reclamada por el sistema operativo. De este modo puede considerarse que la memoria 'no se pierde'.

- **Corrida del programa con nombres largos:**

Línea de comando:

```
./TP0_G14 --input transacc.txt --output - -d 20
```

Resultado:



```

ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 --input transacc.txt --output - -d 20
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
6559112342d7bb4eba14fb1a9b4ee58c80b3d08362a74cd4d73870904e3c41d
20
1253033
2
3
2b2a0ff8f9def3f17d58512ab10d7b1b40e5ae06dabddfd4d9278a428a3700c 9 7cadab457ad8d811f134612436daaa5e5914b20dc2502865f714035b0f267680
ab3691d8e45c1f50684b2762fc640afbe61266bf49c34e0a5319044da23af364 4 7c9e7c1494b2684ab7c19d6aff737e460fa9e98d5a234da1310c97ddf5691834
b2471d941bf888366cf43813752ea2ebfef08254773cf116187cdd6d0463a50a 1 e9e326d5f3b4741fe5967b5f9f3997e6275331ba18567ef9e0e3a00e78371
2
250.5 ca978112ca1bbdcfac231b39a23dc4da786eff8147c4e72b9807785afee48bb
150.7 3b64db95cb55c763391c707108489ae18b4112d783300de38e033b4c98c3deaf
1
7f82ec7e2862b56289ceaaeb057640d924ec668bbc326244a441a1e638ff139 4 b2bd1e0493169ae5088d0766603d24bc6660fe188fb62865aa25362b1e2c94da
1
99.9 948721bfc4815b348dc5ccc66804f8841b621bde0ad0054cc7df22df183a9cf2

```

Figura 3.7: Resultado con la línea de comando `./TP0_G14 --input transacc.txt --output - -d 20`

- **Corrida del programa sin argumento de dificultad:** En este caso el usuario deberá ingresar el nombre del ejecutable sin ningún otro de los argumentos que se esperan.

Líneas de comando:

```

./TP0_G14
./TP0_G14 -i - -o -d
./TP0_G14 -i - -o -

```

Resultado:



```

ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14
Option -d is mandatory.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o - -d
Option requires argument: -d
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$ ./TP0_G14 -i corrupto.txt -o -
Option -d is mandatory.
ferre@LUCASFERREYRA:/mnt/c/Users/lucas/Desktop/TP0_Grupo14_corregido$

```

Figura 3.8: Resultado de haber corrido el programa sin especificar la dificultad de minado

## 4. Conclusiones

Se puede concluir que se han logrado los objetivos propuestos al comienzo del proyecto. Se ha podido realizar con éxito la generación del bloque propuesto por la cátedra, utilizando conceptos claves como POO, modularización, memoria dinámica y *templates*, entre otros. La estrategia utilizada por el grupo consistió en diagramar previamente las distintas clases y estructuras a utilizar. Esto resultó de vital importancia ya que al momento de implementar el código teníamos una clara visión de como llevarlo a cabo.

Se hizo especial hincapié en la utilización de memoria dinámica. Por ejemplo, para generar los vectores *inputs* y *outputs* con su tamaño adecuado.

## 5. Código Fuente

### 5.1. main.cc

```
1  #include "Block.h"
2  #include "cmdline.h"
3
4  #include <string>
5  #include <string.h>
6  #include <fstream>
7  #include <iomanip>
8  #include <iostream>
9  #include <sstream>
10 #include <cstdlib>
11 #include <cstring>
12 #include <stdio.h>
13
14 using namespace std;
15
16 static void opt_input(string const &);
17 static void opt_output(string const &);
18 static void opt_difficulty(string const &);
19 static void opt_help(string const &);
20
21
22 /***** Elementos globales *****/
23 static option_t options[] = {
24     {1, "d", "difficulty", NULL, opt_difficulty, OPT_MANDATORY},
25     {1, "i", "input", "-", opt_input, OPT_DEFAULT},
26     {1, "o", "output", "-", opt_output, OPT_DEFAULT},
27     {0, "h", "help", NULL, opt_help, OPT_DEFAULT},
28     {0, },
29 };
30
31 static int difficulty;
32 static istream *iss = 0;    // Input Stream (clase para manejo de los flujos de
                             // entrada)
33 static ostream *oss = 0;    // Output Stream (clase para manejo de los flujos
                             // de salida)
34 static fstream ifs;         // Input File Stream (derivada de la clase ifstream
                             // que deriva de istream para el manejo de archivos)
35 static fstream ofs;         // Output File Stream (derivada de la clase ofstream
                             // que deriva de ostream para el manejo de archivos)
36
37
38 /*****
39
40 static void
41 opt_input(string const &arg)
42 {
43     /* Si el nombre del archivos es "-", usaremos la entrada
44     {U+FFFD}ndar De lo contrario, abrimos un archivo en modo
45     de lectura.*/
46
47     if (arg == "-") {
48
49         iss = &cin;
50         std::string line;
51         ofstream file_temp;
52         file_temp.open("input_tmp.txt");
53
54         while (getline(*iss, line).eof() == false)
55         {
56             file_temp << line << endl;
57         }
58     }
```

```

59     file_temp.close();
60     ifs.open("input_tmp.txt", ios::in);
61     iss = &ifs;
62 }
63
64 else {
65     ifs.open(arg.c_str(), ios::in); /*c_str(): Returns a pointer to an
66                                     array that contains a null-terminated
67                                     sequence of characters (i.e., a C-
68                                     string) representing
69                                     the current value of the string
70                                     object.*/
71     iss = &ifs;
72 }
73
74 // Verificamos que el stream este OK.
75 if (!iss->good()) {
76     cerr << "cannot open "
77           << arg
78           << "."
79           << endl;
80     exit(1);
81 }
82
83 static void
84 opt_output(string const &arg)
85 {
86     /* Si el nombre del archivos es "-", usaremos la salida
87        [U+FFFD]ndar De lo contrario, abrimos un archivo en modo
88        de escritura.
89        */
90     if (arg == "-") {
91         oss = &cout;    // Establezco la salida estandar cout como flujo de
92                         salida
93     } else {
94         ofs.open(arg.c_str(), ios::out);
95         oss = &ofs;
96     }
97
98     // Verificamos que el stream este OK.
99     if (!oss->good()) {
100         cerr << "cannot open "
101              << arg
102              << "."
103              << endl;
104         exit(1);    // [U+FFFD]-h/oΔ[U+0000]ndel programa en su totalidad
105     }
106 }
107
108 static void
109 opt_difficulty(string const &arg)
110 {
111     istringstream iss(arg);
112
113     /*Intentamos extraer el difficulty de 1[U+FFFD]neme comandos.
114        Para detectar argumentos que nicamente consistan de
115        nmeros enteros, vamos a verificar que EOF llegue justo
116        [U+FFFD]sde la lectura exitosa del escalar.
117        */
118     if (!(iss >> difficulty) || !iss.eof())
119     {
120         cerr << "non-integer difficulty: "

```

```
118         << arg
119         << "."
120         << endl;
121     exit(1);
122 }
123
124 if(is_int(to_string(difficulty)) == false)
125 {
126     cerr << "non-positive difficulty: "
127           << arg
128           << "."
129           << endl;
130     exit(1);
131 }
132
133 if (iss.bad()) {
134     cerr << "cannot read integer difficulty."
135           << endl;
136     exit(1);
137 }
138 }
139
140 static void
141 opt_help(string const &arg)
142 {
143     cout << "cmdline -d difficulty [-i file] [-o file]"
144           << endl;
145     exit(0);
146 }
147
148
149 int
150 main(int argc, char * const argv[])
151 {
152     cmdline cmdl(options); // Objeto con parametro tipo option_t (struct)
153                             // declarado globalmente. Ver main.cc
154     cmdl.parse(argc, argv); // Metodo de parseo de la clase cmdline
155
156     Block bloq;
157
158     bloq.load_Block(iss, difficulty);
159
160     bloq.print_Block(oss);
161
162     return 0;
163 }
```

## 5.2. Vector.h

```
1  #ifndef VECTOR_H_INCLUDED_
2  #define VECTOR_H_INCLUDED_
3
4  #define DEFECT_SIZE 100
5
6  using namespace std;
7
8  template <typename T>
9
10 class Vector
11 {
12 private:
```



```
13     int size; // Cuanto va a tener de largo el Array
14     T *ptr; // Puntero al vector dinamico que contiene los datos de tipo T
15
16 public:
17     Vector();
18     Vector(int);
19     Vector(const Vector <T> &);
20     Vector(int, const Vector <T> & );
21     ~Vector();
22
23     Vector<T>& operator=(const Vector <T> &);
24     bool operator==(const Vector <T> &);
25     T & operator[](int) const;
26
27     int get_size() const;
28 };
29
30 template <typename T>
31 Vector<T>::Vector()
32 {
33     size = DEFECT_SIZE;
34     ptr = new T[size];
35 }
36
37 template <typename T>
38 Vector<T>::Vector(int x)
39 {
40     size = x;
41
42     if(x!=0)
43         ptr = new T[size];
44     else
45         ptr = 0;
46 }
47
48 template <typename T>
49 Vector<T>::Vector(const Vector <T> & v)
50 {
51     size=v.size;
52
53     ptr = new T[size];
54
55     for (int i=0; i < size ; i++)
56         ptr[i] = v.ptr[i];
57 }
58
59 template <typename T>
60 Vector<T>::Vector(int len,const Vector <T> & v)
61 {
62     size=len;
63
64     if(len !=0)
65     {
66         ptr = new T[size];
67
68         for (int i=0; i < v.size ; i++)
69             ptr[i] = v.ptr[i];
70     }
71 }
72
73 template <typename T>
74 Vector<T>::~~Vector()
75 {
```

```
76     if (ptr) //si el puntero NO apunta a NULL-> liberame la memoria pedida
77         delete [] ptr;
78 }
79
80 template <typename T>
81 Vector<T>& Vector<T>::operator=(const Vector<T> & v2) // v1 = v2
82 {
83     if (this != &v2 )
84     {
85         if (size != v2.size)
86         {
87             T * aux;
88             size = v2.size;
89             delete [] ptr;
90             aux = new T[size];
91             ptr = aux;
92         }
93         for (int i = 0 ; i < size; i++)
94             ptr[i] = v2.ptr[i];
95         return *this; //devuelvo el vector de <T> de la clase Vector por
                        //referencia
96     }
97
98     return *this;
99 }
100
101
102 template <typename T>
103 bool Vector<T>::operator==(const Vector<T> & v2) //v1 == v2
104 {
105     if (size != v2.size)
106         return false;
107     else
108     {
109         for (int i = 0 ; i < size ; i++)
110         {
111             if (ptr[i] != v2.ptr[i])
112                 return false;
113         }
114         return true;
115     }
116 }
117
118
119 template <typename T>
120 T & Vector<T>::operator[](int pos) const // v[pos]
121 {
122     if( pos < size)
123         return ptr[pos];
124 }
125
126 template <typename T>
127 int Vector<T>::get_size() const
128 {
129     return size;
130 }
131
132 #endif //VECTOR_H_INCLUDED_
```

### 5.3. Block.h

```
1  #ifndef _BLOCK_H_INCLUDED_
2  #define _BLOCK_H_INCLUDED_
3
4  #include "Transaction.h"
5  #include "Vector.h"
6  #include "sha256.h"
7  #include "cmdline.h"
8  #include "Utilities.h"
9
10 #include <cstdlib>
11 #include <bitset>
12 #include <time.h>
13 #include <string>
14 #include <string.h>
15 #include <iostream>
16 #include <fstream>
17 #include <sstream>
18 #include <cstdlib>
19 #include <cstring>
20 #include <cmath>
21
22 using namespace std;
23
24 #define PREV_BLOCK "
    ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff "
25 #define NONCE_MIN 10000
26 #define NONCE_MAX 2000000
27
28 struct header_t
29 {
30     string prev_block;
31     string txns_hash;
32     int bits; // Dificultad
33     int nonce; // Para el hash
34 };
35
36 struct body_t
37 {
38     int txns_count;
39     Vector <Transaction> txns;
40 };
41
42 class Block
43 {
44 private:    //Atributos
45     header_t * header;
46     body_t * body;
47     void check_hash_header(int);
48     string & load_header_txns_hash(Transaction &, string &);
49
50 public:
51     Block();
52     ~Block();
53     void load_Block(istream *, const int &);
54     void print_Block(ostream *);
55
56 };
57
58 #endif
```

## 5.4. Block.cc

```
1  #include "Block.h"
2
3  Block::Block()
4  {
5      header = new header_t;
6      body = new body_t;
7  }
8
9  Block::~~Block()
10 {
11     delete header;
12     delete body;
13 }
14
15 string & Block::load_header_txns_hash(Transaction & tx, string &
    header_txnshash)
16 {
17     header_txnshash += tx.get_n_tx_in() + "\n";
18
19     for(int j = 0; j < (stoi(tx.get_n_tx_in())); j++)//Cant de inputs
20     {
21         header_txnshash += tx.get_input(j).tx_id + " ";
22         header_txnshash += tx.get_input(j).idx + " ";
23         header_txnshash += tx.get_input(j).addr + "\n";
24     }
25
26     header_txnshash += tx.get_n_tx_out() + "\n";
27
28     for(int j = 0; j < (stoi(tx.get_n_tx_out())); j++)//Cant de outputs
29     {
30         header_txnshash += tx.get_output(j).value + " ";
31         header_txnshash += tx.get_output(j).addr + "\n";
32     }
33
34     return header_txnshash;
35 }
36
37 void Block::check_hash_header(int dif)
38 {
39     string hash_header;
40     hash_header = header->prev_block + "\n"+ header->txns_hash + "\n" +
        to_string(header->bits)+ "\n" + to_string(header->nonce);
41     string h = sha256(sha256(hash_header));
42
43     //Parte de division:
44     int position=dif/4;      // Nro de caracteres -----> Valor de dividendo
45     int rest=dif%4;
46
47     srand(time(NULL)); //Se usa para generar una semilla para generar numeros
        aleatorios
48
49     string test(h); //Genera una cadena auxiliar con el valor de h
50
51     test.resize(position+1); //Genera una cadena de n bits a testear donde
        cada caracter determinado por position son 4 bits
52
53     test=Hex2Bin(test ); //Queda guardado el numero en binario
54
55     string string_zeros(position*4+rest, '0'); /*Forma una cadena de "0" igual
        a la cantidad de bits cero que necesita
56
        Position*4 porque
        position lo que cuenta
```

```

57                                     son 4 bits
58                                     Rest son bits*/
59     int comparation=(test.compare(0, string_zeros.length(), string_zeros));
60
61     while( comparation != 0)
62     {
63         header->nonce = rand() % NONCE_MAX + NONCE_MIN;        //Genera un nonce
64                             aleatorio
65
66         hash_header = header->prev_block + "\n" + header->txns_hash+ "\n" +
67             to_string(header->bits)+ "\n" + to_string(header->nonce);
68         h = sha256(sha256(hash_header));
69
70         string test(h);
71         test.resize(position+1);
72
73         test=Hex2Bin(test);
74
75         comparation=(test.compare(0, string_zeros.length(), string_zeros));
76     }
77 }
78
79 void Block::load_Block(istream * is, const int & dif )
80 {
81     string header_txnshash = "";
82     (body->txns_count) = 0;
83
84     for(int i = 0; (is->eof()) == false ; i++)
85     {
86         if (body->txns_count == (body->txns).get_size())
87         {
88             Vector <Transaction> aux(2*((body->txns).get_size()), body->txns);
89             body->txns = aux;
90
91             (body->txns)[i].load_Transaction(is); //Cargamos el body
92
93             if (((body->txns)[i].get_n_tx_in() == "0") && ((body->txns)[i].
94                 get_n_tx_out() == "0"))
95                 break;
96
97             (body->txns_count) ++;
98
99             //Cargamos el header
100             header_txnshash += load_header_txnshash((body->txns)[i],
101                 header_txnshash);
102         }
103
104         header->prev_block = PREV_BLOCK;
105         header->txns_hash = sha256(sha256(header_txnshash));
106         header->bits = dif;
107         header->nonce = NONCE_MIN;
108
109         check_hash_header(dif);
110     }
111 }
112
113 void Block::print_Block(ostream * os)
114 {
115     *os<<header->prev_block<<"\n";
116     *os<<header->txns_hash<<"\n";
117     *os<<header->bits<<"\n";
118     *os<<header->nonce<<"\n";

```

```
115     *os<<body->txns_count<<"\n";
116
117     for(int i = 0; i < body->txns_count; i++)//Cant de Transacciones
118     {
119         (body->txns)[i].print_Transaction(os);
120     }
121
122     string hash_header;
123     hash_header = header->prev_block + "\n" + header->txns_hash + "\n" +
124         to_string(header->bits) + "\n" + to_string(header->nonce);
125     string h = sha256(sha256(hash_header));
126     //(*os)<<"hash_header:"<<sha256(sha256(hash_header))<<endl;
```

## 5.5. Transaction.h

```
1  #ifndef _TRANSACTION_H_INCLUDED_
2  #define _TRANSACTION_H_INCLUDED_
3  #include "Vector.h"
4  #include "cmdline.h"
5  #include "Utilities.h"
6
7  #include <cstring>
8  #include <ostream>
9
10 #include <string>
11 #include <string.h>
12 #include <string_view>
13 #include <iostream>
14 #include <istream>
15 #include <fstream>
16 #include <iomanip>
17 #include <sstream>
18 #include <cstdlib>
19
20 #define HASH_LENGTH 64
21
22 using namespace std;
23
24 struct input_t {
25
26     string tx_id;
27     string idx;
28     string addr;
29 };
30
31 struct output_t {
32
33     string value;
34     string addr;
35 };
36
37
38 class Transaction
39 {
40 private:
41     //Atributos
42     string n_tx_in;
43     string n_tx_out;
44
45     Vector <input_t> inputs;
```

```
46     Vector <output_t> outputs;
47
48 public:
49
50     Transaction();
51     ~Transaction();
52     void load_Transaction(istream *);
53     void print_Transaction(ostream *);
54     string get_n_tx_in() const {return n_tx_in;};
55     string get_n_tx_out() const {return n_tx_out;};
56     input_t get_input(int position) const {return inputs[position];};
57     output_t get_output(int position) const {return outputs[position];};
58
59 };
60
61 #endif
```

## 5.6. Transaction.cc

```
1  #include "Transaction.h"
2
3  Transaction::Transaction()
4  {
5      n_tx_in = "0";
6      n_tx_out = "0";
7  }
8
9  Transaction::~~Transaction()
10 {
11 }
12
13 void Transaction::load_Transaction(istream * is)
14 {
15     char delim = ' ';
16     getline(*is, n_tx_in);
17     if (is_int(n_tx_in) == false)
18     {
19         cerr << "Invalid input information: check for n_tx_in format or correct
20             amount of outputs"<< endl;
21         exit(1);
22     }
23     if (is->eof() == true)
24     {
25         n_tx_in="0";
26         n_tx_out="0";
27         return;
28     }
29     Vector <input_t> aux(stoi(n_tx_in));
30     inputs = aux;
31     for (int i = 0; i < stoi (n_tx_in); i++)
32     {
33         getline(*is, inputs[i].tx_id, delim);
34         if ( is_alphanumeric((inputs[i].tx_id)) == false ||(inputs[i].tx_id).
35             length() != HASH_LENGTH)
36         {
37             cerr << "Invalid input information: check for tx_id format or
38                 correct amount of inputs."<< endl;
39             exit(1);
40         }
41         getline(*is, inputs[i].idx, delim);
42         if (is_int(inputs[i].idx) == false)
```

```

40     {
41         cerr << "Invalid input information: check for idx format."<< endl;
42         exit(1);
43     }
44     getline(*is, inputs[i].addr);
45     if (is_alphanumeric(inputs[i].addr) == false || (inputs[i].addr).length
46         () != HASH_LENGTH)
47     {
48         cerr << "Invalid input information: check for addr format."<< endl;
49         exit(1);
50     }
51     getline(*is, n_tx_out);
52     if (is_int(n_tx_out) == false)
53     {
54         cerr << "Invalid input information: check for n_tx_out format or
55             correct amount of inputs."<< endl;
56         exit(1);
57     }
58     Vector <output_t> aux1(stoi(n_tx_out));
59     outputs = aux1;
60     for (int j = 0; j < stoi (n_tx_out); j++)
61     {
62         getline(*is, outputs[j].value, delim);
63         if (is_float_or_double(outputs[j].value) == false)
64         {
65             cerr << "Invalid input information: check for value format or
66                 correct amount of outputs."<< endl;
67             exit(1);
68         }
69         getline(*is, outputs[j].addr);
70         if ( is_alphanumeric(outputs[j].addr) == false || (outputs[j].addr).
71             length() != HASH_LENGTH)
72         {
73             cerr << "Invalid input information: check for addr format"<< endl;
74             exit(1);
75         }
76     }
77 }
78
79 void Transaction::print_Transaction(ostream *os)
80 {
81     *os<<n_tx_in<<"\n";
82     for (int i = 0; i < stoi (n_tx_in); i++)
83     {
84         *os<<inputs[i].tx_id<<" ";
85         *os<<inputs[i].idx<<" ";
86         *os<<inputs[i].addr<<"\n";
87     }
88     *os<<n_tx_out<<"\n";
89     for (int i = 0; i < stoi (n_tx_out); i++)
90     {
91         *os<<outputs[i].value<<" ";
92         *os<<outputs[i].addr<<"\n";
93     }
94 }

```

## 5.7. cmdline.h

```

1  #ifndef _CMDLINE_H_INCLUDED_
2  #define _CMDLINE_H_INCLUDED_

```



```
3
4 #include <string>
5 #include <cstring>
6 #include <string.h>
7 #include <iostream>
8
9 using namespace std;
10
11 #define OPT_DEFAULT 0
12 #define OPT_SEEN 1
13 #define OPT_MANDATORY 2
14
15 struct option_t {
16     int has_arg;
17     const char *short_name;
18     const char *long_name;
19     const char *def_value;
20     void (*parse)(string const &); // P[00FFFD] a/oΔ[U+0000]de opciones
21     int flags;
22 };
23
24 class cmdline {
25     /* Este atributo apunta a la tabla que describe todas
26        las opciones a procesar. Por el [U+FFFD] a/oΔ[U+0000] debe
27        ser modificado mediante constructor, y debe finalizar
28        con un elemento nulo.
29     */
30     option_t *option_table;
31
32     /* El constructor por defecto cmdline::cmdline(), es
33        privado, para evitar construir "parsers" (analizador
34        a[U+FFFD]ctico que recibe una palabra y lo interpreta en
35        [U+FFFD] a/oΔ[U+0000] dependiendo su significado para el programa)
36        sin opciones. Es decir, objetos de esta clase sin opciones.
37     */
38
39     cmdline();
40     int do_long_opt(const char *, const char *);
41     int do_short_opt(const char *, const char *);
42 public:
43     cmdline(option_t *);
44     void parse(int, char * const []);
45 };
46
47 #endif
```

## 5.8. cmdline.cc

```
1 #include <string>
2 #include <cstdlib>
3 #include <iostream>
4 #include <cstring>
5
6 #include "cmdline.h"
7
8 using namespace std;
9
10 cmdline::cmdline()
11 {
12 }
13
```

```

14 cmdline::cmdline(option_t *table) : option_table(table)
15 {
16     /*
17     - Lo mismo que hacer:
18
19     option_table = table;
20
21     Siendo "option_table" un atributo de la clase cmdline
22     y table un puntero a objeto o struct de "option_t".
23
24     Se crea una instancia de la clase cmdline
25     cargandole los datos que se hayan en table (la table con
26     las opciones) en el main.cc)
27
28     */
29 }
30
31 void
32 cmdline::parse(int argc, char * const argv[])
33 {
34     #define END_OF_OPTIONS(p) \
35         ((p)->short_name == 0 \
36          && (p)->long_name == 0 \
37          && (p)->parse == 0)
38     /* Primer pasada por la secuencia de opciones: marcamos
39     todas las opciones, como no procesadas abajo.
40
41     */
42     for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op)
43         op->flags &= ~OPT_SEEN;
44
45     /* Recorremos el arreglo argv. En cada paso, vemos
46     si se trata de una opción corta, o larga. Luego,
47     llamamos al parseo correspondiente.
48
49     */
50     for (int i = 1; i < argc; ++i) {
51         /* Todos los metros de este programa deben
52         pasarse en forma de opciones. Encontrar un
53         error.
54
55         */
56         if (argv[i][0] != '-') {
57             cerr << "Invalid non-option argument: "
58                 << argv[i]
59                 << endl;
60             exit(1);
61         }
62
63         /* Usamos "--" para marcar el fin de las
64         opciones; todo los argumentos que puedan
65         estar ahí no son interpretados
66         como opciones.
67
68         */
69         if (argv[i][1] == '-'
70             && argv[i][2] == 0)
71             break;
72
73         /* Finalmente, vemos si se trata o no de una
74         larga; y llamamos a todo lo que se
75         encarga de cada caso.
76
77         */
78         if (argv[i][1] == '-')
79             i += do_long_opt(&argv[i][2], argv[i + 1]);
80         else

```

```

77         i += do_short_opt(&argv[i][1], argv[i + 1]);
78     }
79
80     /* Segunda pasada: procesamos aquellas opciones que,
81        (1) no hayan figurado en la tabla de comandos, y (2) tengan valor por defecto.
82     */
83
84     for (option_t *op = option_table; !END_OF_OPTIONS(op); ++op) {
85 #define OPTION_NAME(op) \
86     (op->short_name ? op->short_name : op->long_name)
87         if (op->flags & OPT_SEEN)
88             continue;
89         if (op->flags & OPT_MANDATORY) {
90             cerr << "Option "
91                  << "-"
92                  << OPTION_NAME(op)
93                  << " is mandatory."
94                  << "\n";
95             exit(1);
96         }
97         if (op->def_value == 0)
98             continue;
99         op->parse(string(op->def_value));
100     }
101 }
102
103 int
104 cmdline::do_long_opt(const char *opt, const char *arg)
105 {
106     /* Recorremos la tabla de opciones, y buscamos la
107        entrada larga que se corresponde con la opt. De no encontrarse, indicamos el
108        error.
109     */
110
111     for (option_t *op = option_table; op->long_name != 0; ++op) {
112         if (string(opt) == string(op->long_name)) {
113             /* Marca la opción como usada en
114                forma larga para evitar tener
115                que inicializarla con el valor por
116                defecto.
117             */
118             op->flags |= OPT_SEEN;
119
120             if (op->has_arg) {
121                 /* Como se traduce a una línea con argumento, verificamos que
122                    el mismo haya sido provisto.
123                 */
124                 if (arg == 0) {
125                     cerr << "Option requires argument: "
126                          << "-"
127                          << opt
128                          << "\n";
129                     exit(1);
130                 }
131                 op->parse(string(arg));
132                 return 1;
133             } else {
134                 /* Sin argumento.
135                 */
136                 op->parse(string(""));
137                 return 0;
138             }
139         }
140     }
141 }

```

```

140     }
141
142     /* [U+FFFD]-h/oΔ[U+0000] no reconocida. Imprimimos un mensaje
143     // de error, y finalizamos [U+FFFD]-h/oΔ[U+0000] el programa.
144     cerr << "Unknown option: "
145         << "--"
146         << opt
147         << "."
148         << endl;
149     exit(1);
150
151     /* Algunos compiladores se quejan con funciones que
152     [U+FFFD]-h/oΔ[U+0000] gicamente pueden terminar, y que no devuelven
153     un valor en esta ltima parte.
154     */
155     return -1;
156 }
157
158 int
159 cmdline::do_short_opt(const char *opt, const char *arg)
160 {
161     option_t *op;
162
163     /* Recorremos la tabla de opciones, y buscamos la
164     entrada corta que se corresponde [U+FFFD]-h/oΔ[U+0000] de
165     [U+FFFD] a los comandos. De no encontrarse, indicamos el
166     error.
167     */
168     for (op = option_table; op->short_name != 0; ++op) {
169         if (string(opt) == string(op->short_name)) {
170             /* Marca [U+FFFD]-h/oΔ[U+0000] como usada en
171             forma [U+FFFD] para evitar tener
172             que inicializarla con el valor por
173             defecto.
174             */
175             op->flags |= OPT_SEEN;
176
177             if (op->has_arg) {
178                 /* Como se trat [U+FFFD]-h/oΔ[U+0000] n
179                 con argumento, verificamos que
180                 el mismo haya sido provisto.
181                 */
182                 if (arg == 0) {
183                     cerr << "Option requires argument: "
184                         << "--"
185                         << opt
186                         << "\n";
187                     exit(1);
188                 }
189                 op->parse(string(arg));
190                 return 1;
191             } else {
192                 /* [U+FFFD]-h/oΔ[U+0000] sin argumento.
193                 op->parse(string(""));
194                 return 0;
195             }
196         }
197     }
198
199     /* [U+FFFD]-h/oΔ[U+0000] no reconocida. Imprimimos un mensaje
200     de error, y finalizamos [U+FFFD]-h/oΔ[U+0000] el programa.
201     */
202     cerr << "Unknown option: "

```

```
203         << "-"
204         << opt
205         << "."
206         << endl;
207     exit(1);
208
209     /* Algunos compiladores se quejan con funciones que
210 [U+FFED]-h/oΔ[U+0000]gicamnt pueden terminar, y que no devuelven
211     un valor en esta ltima parte.
212     */
213     return -1;
214 }
```

## 5.9. sha256.h

```
1  #ifndef SHA256_H_INCLUDED_
2  #define SHA256_H_INCLUDED_
3  #include <string.h>
4  #include <string>
5  #include <cstring>
6
7  class SHA256
8  {
9  protected:
10     typedef unsigned char uint8;
11     typedef unsigned int uint32;
12     typedef unsigned long long uint64;
13
14     const static uint32 sha256_k[];
15     static const unsigned int SHA224_256_BLOCK_SIZE = (512/8);
16 public:
17     void init();
18     void update(const unsigned char *message, unsigned int len);
19     void final(unsigned char *digest);
20     static const unsigned int DIGEST_SIZE = ( 256 / 8);
21
22 protected:
23     void transform(const unsigned char *message, unsigned int block_nb);
24     unsigned int m_tot_len;
25     unsigned int m_len;
26     unsigned char m_block[2*SHA224_256_BLOCK_SIZE];
27     uint32 m_h[8];
28 };
29
30 std::string sha256(std::string input);
31
32 #define SHA2_SHFR(x, n) ((x >> n))
33 #define SHA2_ROTR(x, n) (((x >> n) | (x << ((sizeof(x) << 3) - n)))
34 #define SHA2_ROTL(x, n) (((x << n) | (x >> ((sizeof(x) << 3) - n)))
35 #define SHA2_CH(x, y, z) ((x & y) ^ (~x & z))
36 #define SHA2_MAJ(x, y, z) ((x & y) ^ (x & z) ^ (y & z))
37 #define SHA256_F1(x) (SHA2_ROTR(x, 2) ^ SHA2_ROTR(x, 13) ^ SHA2_ROTR(x, 22))
38 #define SHA256_F2(x) (SHA2_ROTR(x, 6) ^ SHA2_ROTR(x, 11) ^ SHA2_ROTR(x, 25))
39 #define SHA256_F3(x) (SHA2_ROTR(x, 7) ^ SHA2_ROTR(x, 18) ^ SHA2_SHFR(x, 3))
40 #define SHA256_F4(x) (SHA2_ROTR(x, 17) ^ SHA2_ROTR(x, 19) ^ SHA2_SHFR(x, 10))
41 #define SHA2_UNPACK32(x, str) \
42 { \
43     *((str) + 3) = (uint8) ((x >> 24)); \
44     *((str) + 2) = (uint8) ((x >> 16)); \
45     *((str) + 1) = (uint8) ((x >> 8)); \
46     *((str) + 0) = (uint8) ((x >> 0)); \
47 }
```

```
47 }
48 #define SHA2_PACK32(str, x) \
49 { \
50     *(x) = ((uint32) *((str) + 3) \
51             | ((uint32) *((str) + 2) << 8) \
52             | ((uint32) *((str) + 1) << 16) \
53             | ((uint32) *((str) + 0) << 24)); \
54 }
55 #endif
```

## 5.10. sha256.cc

```
1  #include <cstring>
2  #include <fstream>
3  #include "sha256.h"
4
5  const unsigned int SHA256::sha256_k[64] = //UL = uint32
6      {0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5,
7       0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5,
8       0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3,
9       0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174,
10      0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,
11      0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da,
12      0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7,
13      0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967,
14      0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13,
15      0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,
16      0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3,
17      0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070,
18      0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5,
19      0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3,
20      0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,
21      0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2};
22
23 void SHA256::transform(const unsigned char *message, unsigned int block_nb)
24 {
25     uint32 w[64];
26     uint32 wv[8];
27     uint32 t1, t2;
28     const unsigned char *sub_block;
29     int i;
30     int j;
31     for (i = 0; i < (int) block_nb; i++) {
32         sub_block = message + (i << 6);
33         for (j = 0; j < 16; j++) {
34             SHA2_PACK32(&sub_block[j << 2], &w[j]);
35         }
36         for (j = 16; j < 64; j++) {
37             w[j] = SHA256_F4(w[j - 2]) + w[j - 7] + SHA256_F3(w[j - 15]) + w[j - 16];
38         }
39         for (j = 0; j < 8; j++) {
40             wv[j] = m_h[j];
41         }
42         for (j = 0; j < 64; j++) {
43             t1 = wv[7] + SHA256_F2(wv[4]) + SHA2_CH(wv[4], wv[5], wv[6])
44                 + sha256_k[j] + w[j];
45             t2 = SHA256_F1(wv[0]) + SHA2_MAJ(wv[0], wv[1], wv[2]);
46             wv[7] = wv[6];
47             wv[6] = wv[5];
48             wv[5] = wv[4];
```

```
49         wv[4] = wv[3] + t1;
50         wv[3] = wv[2];
51         wv[2] = wv[1];
52         wv[1] = wv[0];
53         wv[0] = t1 + t2;
54     }
55     for (j = 0; j < 8; j++) {
56         m_h[j] += wv[j];
57     }
58 }
59 }
60
61 void SHA256::init()
62 {
63     m_h[0] = 0x6a09e667;
64     m_h[1] = 0xbb67ae85;
65     m_h[2] = 0x3c6ef372;
66     m_h[3] = 0xa54ff53a;
67     m_h[4] = 0x510e527f;
68     m_h[5] = 0x9b05688c;
69     m_h[6] = 0x1f83d9ab;
70     m_h[7] = 0x5be0cd19;
71     m_len = 0;
72     m_tot_len = 0;
73 }
74
75 void SHA256::update(const unsigned char *message, unsigned int len)
76 {
77     unsigned int block_nb;
78     unsigned int new_len, rem_len, tmp_len;
79     const unsigned char *shifted_message;
80     tmp_len = SHA224_256_BLOCK_SIZE - m_len;
81     rem_len = len < tmp_len ? len : tmp_len;
82     memcpy(&m_block[m_len], message, rem_len);
83     if (m_len + len < SHA224_256_BLOCK_SIZE) {
84         m_len += len;
85         return;
86     }
87     new_len = len - rem_len;
88     block_nb = new_len / SHA224_256_BLOCK_SIZE;
89     shifted_message = message + rem_len;
90     transform(m_block, 1);
91     transform(shifted_message, block_nb);
92     rem_len = new_len % SHA224_256_BLOCK_SIZE;
93     memcpy(m_block, &shifted_message[block_nb << 6], rem_len);
94     m_len = rem_len;
95     m_tot_len += (block_nb + 1) << 6;
96 }
97
98 void SHA256::final(unsigned char *digest)
99 {
100     unsigned int block_nb;
101     unsigned int pm_len;
102     unsigned int len_b;
103     int i;
104     block_nb = (1 + ((SHA224_256_BLOCK_SIZE - 9)
105                     < (m_len % SHA224_256_BLOCK_SIZE)));
106     len_b = (m_tot_len + m_len) << 3;
107     pm_len = block_nb << 6;
108     memset(m_block + m_len, 0, pm_len - m_len);
109     m_block[m_len] = 0x80;
110     SHA2_UNPACK32(len_b, m_block + pm_len - 4);
111     transform(m_block, block_nb);
```

```
112     for (i = 0 ; i < 8; i++) {
113         SHA2_UNPACK32(m_h[i], &digest[i << 2]);
114     }
115 }
116
117 std::string sha256(std::string input)
118 {
119     unsigned char digest[SHA256::DIGEST_SIZE];
120     memset(digest,0,SHA256::DIGEST_SIZE);
121
122     SHA256 ctx = SHA256();
123     ctx.init();
124     ctx.update( (unsigned char*)input.c_str(), input.length());
125     ctx.final(digest);
126
127     char buf[2*SHA256::DIGEST_SIZE+1];
128     buf[2*SHA256::DIGEST_SIZE] = 0;
129     for (unsigned int i = 0; i < SHA256::DIGEST_SIZE; i++)
130         sprintf(buf+i*2, "%02x", digest[i]);
131     return std::string(buf);
132 }
```

## 5.11. Utilities.h

```
1  #ifndef UTILITIES_H_INCLUDED_
2  #define UTILITIES_H_INCLUDED_
3
4  #include <cstring>
5  #include <ostream>
6
7  #include <string>
8  #include <string.h>
9  #include <string_view>
10 #include <iostream>
11 #include <istream>
12 #include <fstream>
13 #include <iomanip>
14 #include <sstream>
15 #include <cstdlib>
16 #include <bitset>
17 #include <time.h>
18
19 using namespace std;
20
21 string Hex2Bin(const string&);
22 bool is_float_or_double(const string &str);
23 bool is_alphabetic(const string &str);
24 bool is_alphanumeric(const string &str);
25 bool is_int(const string &str);
26 bool is_alphanumeric_point(const string &str);
27
28 #endif
```

## 5.12. Utilities.cc

```
1  #include "Utilities.h"
2
3  string Hex2Bin(const string& s)//transforma de hexa a binario una string,
    maximo 4 bytes, 8 char y 32 bits
```



```
4 {
5     string bin_number="";
6
7     for(size_t i=0; i< s.length(); i++)
8     {
9         stringstream ss;
10        unsigned n;
11        ss << hex << s[i];
12        ss >> n;
13
14        bitset<4> b(n);      //32 es el maximo por el unsigned ----> Nuestra
                             dificultad maxima seria 32 bits teoricamente!
15        bin_number.append( b.to_string() );//.substr(32 - 4*(s.length()));
16    }
17
18    return bin_number;
19 }
20
21 bool is_float_or_double(const string &str)
22 {
23     return str.find_first_not_of("0123456789.") == string::npos;
24 }
25
26 bool is_alphanumeric(const string &str)
27 {
28     return str.find_first_not_of("0123456789qwertyuiopasdfghjklzxcvbnm") ==
        string::npos;
29 }
30
31 bool is_alphabetic(const string &str)
32 {
33     return str.find_first_not_of("qwertyuiopasdfghjklzxcvbnm") == string::npos;
34 }
35
36 bool is_alphanumeric_point(const string &str)
37 {
38     return str.find_first_not_of("0123456789qwertyuiopasdfghjklzxcvbnm._-") ==
        string::npos;
39 }
40
41 bool is_int(const string &str)
42 {
43     return str.find_first_not_of("0123456789") == string::npos;
44 }
```

# 75.04/95.12 Algoritmos y Programación II

## Trabajo práctico 0: programación C++

Universidad de Buenos Aires - FIUBA  
Segundo cuatrimestre de 2020

### 1. Objetivos

Ejercitar conceptos básicos de programación C++, implementando un programa y su correspondiente documentación que resuelva el problema descripto más abajo.

### 2. Alcance

Este Trabajo Práctico es de elaboración grupal, evaluación individual, y de carácter obligatorio para todos alumnos del curso.

### 3. Requisitos

El trabajo deberá ser entregado a través del campus virtual, en la fecha estipulada, con una carátula que contenga los datos completos de todos los integrantes, un informe impreso de acuerdo con lo que mencionaremos en la Sección 5, y con una copia digital de los archivos fuente necesarios para compilar el trabajo.

### 4. Descripción

**Bitcoin** es, posiblemente, la criptomoneda más importante de la actualidad. Los trabajos prácticos de este cuatrimestre están destinados a comprender los detalles técnicos más relevantes detrás de Bitcoin –en particular, la tecnología de **blockchain**. Para ello, trabajaremos con **ALGOCHAIN**, una simplificación de la blockchain orientada a capturar los conceptos esenciales de la tecnología.

En este primer acercamiento al problema, nos abocaremos a leer y procesar **transacciones** y ensamblar un **bloque** a partir de estas. Estos conceptos serán debidamente introducidos en la Sección 4.1, donde daremos una breve introducción a Bitcoin y blockchain. Una vez hecho esto, presentaremos la **ALGOCHAIN** en la Sección 4.2, destacando al mismo tiempo las similitudes y diferencias más importantes con la blockchain propiamente dicha. Las tareas a realizar en el presente trabajo se detallan en la Sección 4.3.

## 4.1. Introducción a Bitcoin y blockchain

Una *criptomoneda* es un activo digital que actúa como medio de intercambio utilizando tecnología criptográfica para asegurar la autenticidad de las transacciones. Bitcoin es, tal vez, la criptomoneda más importante en la actualidad. Propuesta en 2009 por una persona (o grupo de personas) bajo el seudónimo *Satoshi Nakamoto* [2], se caracterizó por ser la primera criptomoneda descentralizada que propuso una solución al problema de *double-spending* sin involucrar una tercera parte de confianza<sup>1</sup>. La idea esencial (y revolucionaria) que introdujo Bitcoin se basa en un registro descentralizado de todas las transacciones procesadas en el que cualquiera puede asentar operaciones. Este registro, replicado y distribuido en cada nodo de la red, se conoce como **blockchain**.

La blockchain no es otra cosa que una lista enlazada de *bloques*<sup>2</sup>. Los bloques agrupan *transacciones* y son la unidad básica de información de la blockchain (i.e., son los nodos de la lista). Cuando un usuario introduce una nueva transacción  $t$  en la red, las propiedades de la blockchain garantizan una detección eficiente de cualquier otra transacción que extraiga los fondos de la misma operación referenciada por  $t$ . En caso de que esto sucediera, se considera que el usuario está intentando hacer *double-spending* y la transacción es consecuentemente invalidada por los nodos de la red.

En lo que sigue describiremos los conceptos más importantes detrás de la blockchain. La Figura 1 provee un resumen visual de todo estos conceptos en el marco de la ALGOCHAIN.

### 4.1.1. Funciones de hash criptográficas

Una *función de hash criptográfica* es un algoritmo matemático que toma una cantidad arbitraria de bytes y computa una tira de bytes de una longitud fija (en adelante, un *hash*). Es importante que dichas funciones sean *one-way*, en el sentido de que sea computacionalmente inviable hacer una “ingeniería reversa” sobre la salida para reconstruir una posible entrada. Otra propiedad que suelen tener dichas funciones es un *efecto avalancha* en el que cambios incluso en bits aislados de la entrada derivan en hashes significativamente diferentes.

Bitcoin emplea la función de hash SHA256, ampliamente utilizada en una gran variedad de protocolos de autenticación y encriptación [3]. Esta genera una salida de 32 bytes que usualmente se representa mediante 64 dígitos hexadecimales. A modo de ejemplo, el valor de  $\text{SHA256}(\text{'Sarasa.'})$  es

9c231858fa5fef160c1e7ecfa333df51e72ec04e9c550a57c59f22fe8bb10df2

### 4.1.2. Direcciones y firmas digitales

Una *dirección* de Bitcoin es básicamente un hash de 160 bits de la clave pública de un usuario. Mediante algoritmos criptográficos asimétricos, los usuarios pueden generar pares de claves mutuamente asociadas (pública y privada). La *clave privada* se emplea para *firmar*

<sup>1</sup>El doble gasto (o *double-spending*) ocurre cuando el emisor del dinero crea más de una transacción a partir de una misma operación previa. Naturalmente, sólo una de las nuevas transacciones debería ser válida puesto que, de lo contrario, el emisor estaría multiplicando dinero.

<sup>2</sup>Técnicamente, la blockchain es más bien un árbol de bloques, pero esto será abordado en el contexto del siguiente trabajo práctico.

los mensajes que desean transmitirse. Cualquier receptor puede luego verificar que la firma es válida utilizando la *clave pública* asociada. Esta clase de métodos criptográficos ofrecen garantías de que es computacionalmente difícil reconstruir la clave privada a partir de la información públicamente disponible.

#### 4.1.3. Transacciones

Una *transacción* en Bitcoin está definida por una lista de entradas (*inputs*) y otra de salidas (*outputs*). Un *output* se representa a través de un par (*value, addr*), donde *value* indica la cantidad de bitcoins que recibirá el destinatario y *addr* es el hash criptográfico de la clave pública del destinatario. Por otro lado, un *input* puede entenderse como una tupla (*tx\_id, idx, key, sig*) tal que:

- *tx\_id* indica el hash de una transacción previa de la que esta nueva transacción toma fondos,
- *idx* es un índice dentro de la secuencia de *outputs* de dicha transacción (los fondos de este *input*, luego, provienen de dicho *output*),
- *key* es la clave pública asociada a tal *output*, y
- *sig* es la firma digital del hash de la transacción usando la clave privada asociada a la clave pública del *output*.

En consecuencia, cada *input* hace referencia a un *output* anterior en la blockchain (los campos *tx\_id* y *idx* suelen agruparse en una estructura común bajo el nombre de *outpoint*). Para verificar que el uso de dicho *output* es legítimo, se calcula el hash de la clave pública y se verifica que sea igual a la que figura en el *output* utilizado. Luego, basta con verificar la firma digital con esa clave pública para asegurar la autenticidad de la operación.

Para garantizar la validez de una transacción, es importante verificar no sólo que cada *input* es válido sino también que la suma de los *outputs* referenciados sea mayor o igual que la suma de los *outputs* de la transacción. La diferencia entre ambas sumas, en caso de existir, es lo que se conoce como *transaction fee*. Este valor puede ser reclamado por quien agrega la transacción a la blockchain (como retribución por suministrar poder de cómputo para realizar el minado de un nuevo bloque).

Naturalmente, una vez que un *output* de una transacción haya sido utilizado, este no podrá volver a utilizarse en el futuro. En otras palabras, cada nueva transacción sólo puede referenciar *outputs* que no fueron utilizados previamente. Estos últimos se conocen como *unspent transaction outputs* (UTXOs).

#### 4.1.4. Bloques

Toda transacción de Bitcoin pertenece necesariamente a un *bloque*. Cada bloque está integrado por un encabezado (*header*) y un cuerpo (*body*). En el header se destaca la siguiente información:

- El hash del bloque antecesor en la blockchain (*prev\_block*),

- El hash de todas las transacciones incluidas en el bloque (`txns_hash`),
- La *dificultad* con la cual este bloque fue ensamblado (`bits`), y
- Un campo en el que se puede poner datos arbitrarios, permitiendo así alterar el hash resultante (`nonce`).

El cuerpo de un bloque, por otro lado, incluye la cantidad total de transacciones (`txn_count`) seguido de la secuencia conformada por dichas transacciones (`txns`).

#### 4.1.5. Minado de bloques

Para que un bloque sea válido y pueda en consecuencia ser aceptado por la red de Bitcoin, debe contar con una prueba de trabajo (*proof-of-work*) que debe ser difícil de calcular y, en simultáneo, fácil de verificar. El mecanismo detrás de este proceso se conoce como *minado*. Las entidades encargadas de agrupar transacciones y ensamblar bloques válidos son los *mineros*.

Los mineros obtienen una recompensa en bitcoins cuando agregan un bloque a la blockchain. Esto último se logra calculando la *proof-of-work* del nuevo bloque, lo cual a su vez se realiza con poder de cómputo. La *proof-of-work* de un bloque consiste en un hash  $h = \text{SHA256}(\text{SHA256}(\text{header}))$  tal que su cantidad de ceros en los bits más significativos es mayor o igual que un valor derivado del campo `bits` del header del bloque. A los efectos prácticos, consideraremos que, si el campo `bits` indica un valor  $d$ , la cantidad de ceros en los bits más significativos de  $h$  debe ser  $\geq d$ .

El esfuerzo necesario para ensamblar un bloque que cumpla con la dificultad de la red crece exponencialmente con la cantidad de ceros requerida. Esto se debe a que agregar un cero extra a la dificultad disminuye a la mitad la cantidad de hashes que cumplan con dicha restricción. No obstante esto, para verificar que un bloque cumple con esta propiedad, basta con computar dos veces la función de hash SHA256. De esta forma, se puede comprobar fácilmente que un minero realizó una cierta cantidad de trabajo para hallar un bloque válido.

## 4.2. Algochain: la blockchain de Algoritmos II

La blockchain simplificada con la que estaremos trabajando a lo largo del cuatrimestre es la ALGOCHAIN. Al igual que la blockchain, la ALGOCHAIN se compone de bloques que agrupan transacciones. A su vez, las transacciones constan de una secuencia de *inputs* y otra de *outputs* que siguen los mismos lineamientos esbozados más arriba. La Figura 1 muestra un esquema de alto nivel de la ALGOCHAIN.

### 4.2.1. Direcciones

Una de las diferencias más importantes con la blockchain radica en una simplificación intencional del proceso de verificación y validación de direcciones al momento de procesar las transacciones: la ALGOCHAIN no utiliza firmas digitales ni claves públicas. En su lugar, tanto los *inputs* como los *outputs* de las transacciones referencian directamente direcciones de origen y destino de los fondos, respectivamente. Esta *dirección* la interpretaremos como un hash SHA256 de una cadena de caracteres arbitraria que simbolice la dirección propiamente dicha

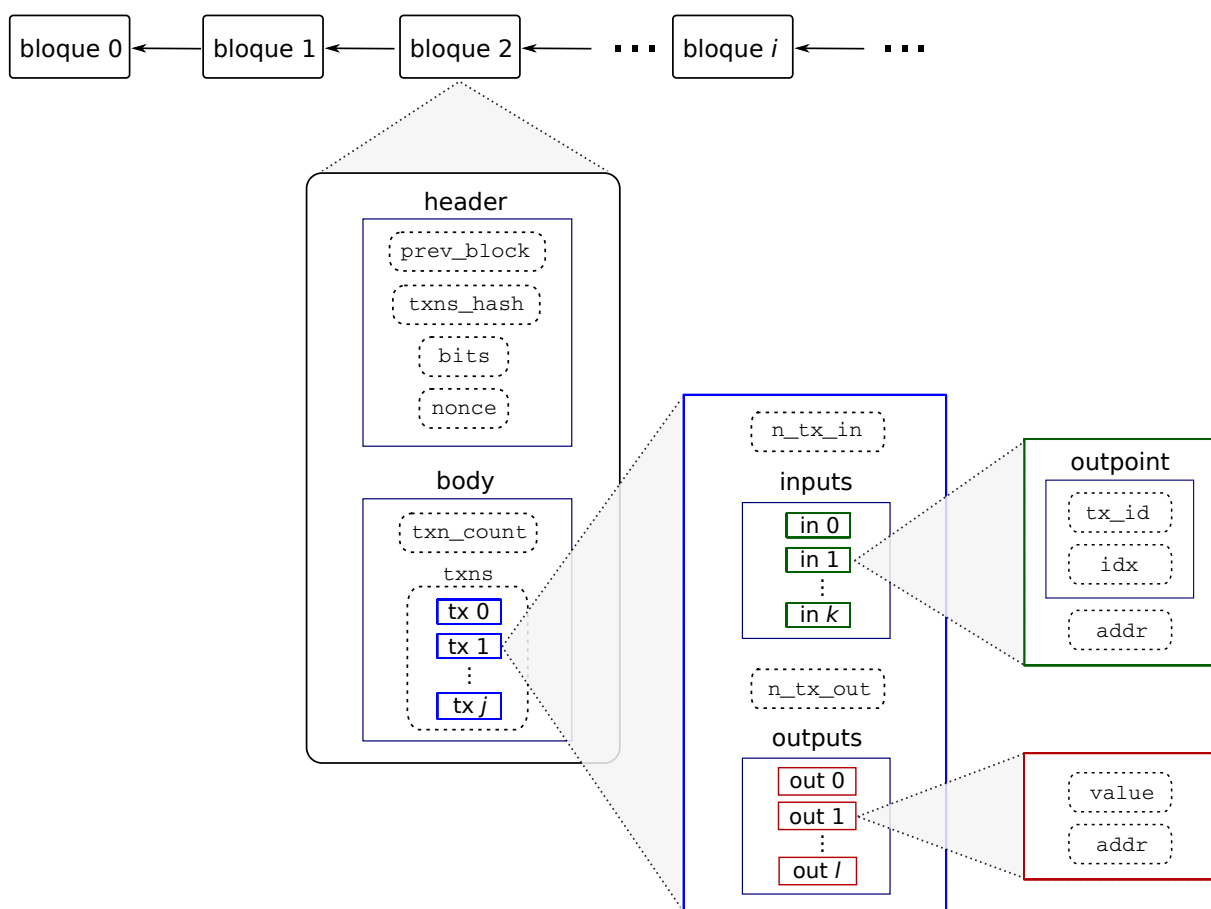


Figura 1: Esquema de alto nivel de la ALGOCHAIN

del usuario. A la hora de procesar una nueva transacción  $t$ , simplemente deberá validarse que la dirección `addr` de cada *input* de  $t$  coincida exactamente con el valor del `addr` especificado en el *output* referenciado en dicho *input*.

A modo de ejemplo, si la dirección real de un usuario de nuestra ALGOCHAIN fuese Segurola y Habana, los campos `addr` de las transacciones que involucren a dicho usuario deberían contener el valor `addr = SHA256('Segurola y Habana')`, que equivale a

485c8c85be20ebb6a9f6dd586b0f9eb6163aa0db1c6e29185b3c6cd1f7b15e9e

#### 4.2.2. Hashes de bloques y transacciones

Tal como ocurre en blockchain, y como explicamos en la Sección 4.1, en ALGOCHAIN identificamos unívocamente bloques y transacciones mediante hashes SHA256 dobles.

El campo `prev_block` del header de un bloque  $b$  indica el hash del bloque antecesor  $b'$  en la ALGOCHAIN. De este modo, `prev_block = SHA256(SHA256( $b'$ ))`. Dicho hash lo calcularemos sobre una concatenación secuencial de todos los campos de  $b'$  respetando exactamente el formato de bloque que describiremos en la Sección 4.4.

De forma análoga, el campo `tx_id` de los *inputs* de las transacciones lo calcularemos con un doble hash SHA256 sobre una concatenación de todos los campos de la transacción correspondiente.

Finalmente, el campo `txns_hash` del header de un bloque  $b$  contendrá también un doble hash SHA256 de todas las transacciones incluidas en  $b$ . En el contexto de este trabajo práctico, dicho hash lo calcularemos sobre una concatenación de todas las transacciones respetando exactamente el formato que describiremos en la Sección 4.4. En otras palabras, dadas las transacciones  $t_0, t_1 \dots, t_j$  del bloque  $b$ ,

$$\text{txns\_hash} = \text{SHA256}(\text{SHA256}(t_0 t_1 \dots t_j))$$

### 4.3. Tareas a realizar

Para apuntalar los objetivos esenciales de este trabajo práctico, esbozados en la Sección 1, deberemos escribir un programa que reciba transacciones por un *stream* de entrada y ensamble un bloque con todas ellas una vez finalizada la lectura. Dicho bloque deberá escribirse en un *stream* de salida. De este modo, al estar trabajando con único bloque, por convención dejaremos fijo el valor del campo `prev_block` en su header. Dicho campo debe instanciarse en

```
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

Naturalmente, nuestros bloques deben satisfacer los requisitos de validez delineados en la Sección 4.1.5. En particular, nos interesa exhibir la correspondiente *proof-of-work* para poder reclamar las eventuales recompensas derivadas del minado. Para ello, nuestros programas recibirán como parámetro el valor de la *dificultad*  $d$  esperada. En otras palabras, debemos garantizar que la cantidad de ceros en los bits más significativos de nuestro hash  $h$  es  $\geq d$ , siendo  $h = \text{SHA256}(\text{SHA256}(\text{header}))$ . Recordar que, en caso de no encontrar un hash  $h$  válido, es posible intentar sucesivas veces modificando el campo `nonce` del header del bloque (la Sección 4.4 describe en detalle el formato de dicho header). Este campo puede instanciarse con valores numéricos arbitrarios tantas veces como sea necesario hasta dar con un hash válido.

Para simplificar el proceso de desarrollo, la cátedra suministrará código C++ para calcular hashes SHA256.

### 4.4. Formatos de la Algochain

En esta Sección detallaremos el formato de las transacciones y bloques de la ALGOCHAIN. Tener en cuenta que es sumamente importante **respetar de manera estricta** este formato. Mostraremos algunos ejemplos concretos en la Sección 4.6.

#### 4.4.1. Transacciones

Toda transacción de la ALGOCHAIN debe satisfacer el siguiente formato:

- Empieza con una línea que contiene el campo entero `n_tx_in`, que indica la cantidad total de *inputs*.

- Luego siguen los *inputs*, uno por línea. Cada *input* consta de tres campos separados entre sí por un único espacio:
  - *tx\_id*, el hash de la transacción de donde este *input* toma fondos,
  - *idx*, un valor entero no negativo que sirve de índice sobre la secuencia de *outputs* de la transacción con hash *tx\_id*, y
  - *addr*, la dirección de origen de los fondos (que debe coincidir con la dirección del *output* referenciado).
- Luego de la secuencia de *inputs*, sigue una línea con el campo entero *n\_tx\_out*, que indica la cantidad total de *outputs* en la transacción.
- Las *n\_tx\_out* líneas siguientes contienen la secuencia de *outputs*, uno por línea. Cada *output* consta de los siguientes campos, separados por un único espacio:
  - *value*, un número de punto flotante que representa la cantidad de Algos a transferir en este *output*, y
  - *addr*, la dirección de destino de tales fondos.

#### 4.4.2. Bloques

Como indicamos en la Sección 4.1.4, todo bloque arranca con un header. El formato de nuestros headers es el siguiente:

- El primer campo es *prev\_block*, que contiene el hash del bloque completo que antecede al bloque actual en la ALGOCHAIN.
- Luego sigue el campo *txns\_hash*, que contiene el hash de todas las transacciones incluidas en el bloque. El cálculo de este hash debe realizarse de acuerdo a las instrucciones de la Sección 4.2.2.
- A continuación sigue el campo *bits*, un valor entero positivo que indica la dificultad con la que fue minada este bloque.
- El último campo del header es el *nonce*, un valor entero no negativo que puede contener valores arbitrarios. El objetivo de este campo es tener un espacio de prueba modificable para poder generar hashes sucesivos hasta satisfacer la dificultad del minado.

Todos estos campos deben aparecer en líneas independientes. En la línea inmediatamente posterior al *nonce* comienza la información del body del bloque:

- La primera línea contiene el campo *txn\_count*, un valor entero positivo que indica la cantidad total de transacciones incluidas en el bloque.
- A continuación siguen una por una las *txn\_count* transacciones. Todas ellas deben respetar el formato de transacción de la Sección anterior.



## 4.5. Interfaz

La interacción con nuestros programas se dará a través de la línea de comando. Las opciones a implementar en este trabajo práctico son las siguientes:

- `-d`, o `--difficulty`, que indica la dificultad esperada  $d$  del minado del bloque. En otras palabras, el hash  $h = \text{SHA256}(\text{SHA256}(\text{header}))$  debe ser tal que la cantidad de ceros en sus bits más significativos sea  $\geq d$ . Esta opción es de carácter obligatorio (i.e., el programa no puede continuar en su ausencia).
- `-i`, o `--input`, que permite controlar el stream de entrada de las transacciones. El programa deberá recibir las transacciones a partir del archivo con el nombre pasado como argumento. Si dicho argumento es `"-"`, el programa las leerá de la entrada standard, `std::cin`.
- `-o`, o `--output`, que permite direccionar la salida al archivo pasado como argumento o, de manera similar a la anterior, a la salida standard `-std::cout` si el argumento es `"-"`.

## 4.6. Ejemplos

Consideremos la siguiente transacción  $t$ :

```
1
48df0779 2 d4cc51bb
1
250.5 842f33e9
```

Por una cuestión de espacio, los hashes involucrados en estos ejemplos aparecen representados por sus últimos 8 bytes. De este modo, el hash `tx_id` del *input* de  $t$  y las direcciones *addr* referenciadas en el *input* y en el *output* son, respectivamente,

```
26429a356b1d25b7d57c0f9a6d5fed8a290cb42374185887dcd2874548df0779
f680e0021dcaf15d161604378236937225eeecae85cc6cda09ea85fad4cc51bb
0618013fa64ac6807bdea212bbdd08ffc628dd440fa725b92a8b534a842f33e9
```

Esta transacción consta de un único *input* y un único *output*. El *input* toma fondos de alguna supuesta transacción  $t'$  cuyo hash es `48df0779`. En particular, los fondos provienen del tercer *output* de  $t'$  (observar que `idx` es 2). La dirección de origen de los fondos es `d4cc51bb`. Por otra parte, el *output* de  $t$  deposita 250,5 Algos en la dirección `842f33e9`.

Supongamos ahora que contamos con un archivo de transacciones que contiene la información de  $t$ :

```
$ cat txns.txt
1
48df0779 2 d4cc51bb
1
250.5 842f33e9
```

La siguiente invocación solicita ensamblar un nuevo bloque con esta transacción:

```
$ ./tp0 -i txns.txt -o block.txt -d 3
```

Notar que el bloque debe escribirse al archivo `block.txt`. Además, la dificultad de minado sugerida es 3: esto nos dice que el hash del header de nuestro bloque debe comenzar con 3 o más bits nulos. Una posible salida podría ser la siguiente:

```
$ cat block.txt
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
155dc94b29dce95bb2f940cdd2d7e0bce66dca9370c3ed96d50a30b3d84f8c4c
3
12232
1
48df0779 2 d4cc51bb
1
250.5 842f33e9
```

Observar que el valor del nonce es 12232. Si bien dicho nonce permite encontrar un hash del header satisfactorio, esta elección por supuesto no es única. El hash del header, bajo esta elección, resulta

```
045b22553f86219b1ecb68bc34a623ecff7fe1807be806a3ccfa9f1b3df5cfc0
```

Como puede verse, el hash comienza con cuatro bits nulos.

Un ejemplo aún más simple (y curioso) consiste en invocar nuestros programas con una entrada vacía:

```
$ touch empty.txt
$ ./tp0 -i empty.txt -d 3
ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
cd372fb85148700fa88095e3492d3f9f5beb43e555e5ff26d95f5a6adc36f8e6
3
59329
0
```

En primer lugar, notemos que, al no especificar un *stream* de salida, el programa dirige la escritura del bloque a la salida estándar. El bloque construido, si bien no incluye ninguna transacción, contiene información válida en su header. Notar que el campo `txns_hash` se calcula en este caso a partir del doble hash SHA256 de una cadena de caracteres vacía.

## 4.7. Portabilidad

Es deseable que la implementación desarrollada provea un grado mínimo de portabilidad. Sugerimos verificar nuestros programas en alguna versión reciente de UNIX: BSD o Linux.

## 5. Informe

El informe deberá incluir, como mínimo:

- Una carátula que incluya los nombres de los integrantes y el listado de todas las entregas realizadas hasta ese momento, con sus respectivas fechas.
- Documentación relevante al diseño e implementación del programa.
- Documentación relevante al proceso de compilación: cómo obtener el ejecutable a partir de los archivos fuente.
- Las corridas de prueba, con los comentarios pertinentes.
- El código fuente, en lenguaje C++.
- Este enunciado.

## 6. Fechas

La última fecha de entrega es el **jueves 12 de noviembre de 2020**.

## Referencias

- [1] Wikipedia, "Bitcoin Wiki." [https://en.bitcoin.it/wiki/Main\\_Page](https://en.bitcoin.it/wiki/Main_Page).
- [2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2009.
- [3] Wikipedia, "SHA-2." <https://en.wikipedia.org/wiki/SHA-2>.