sep 22, 19 19:33 **Writer.h** Page 1/1

```
1  //
2  // Created by leobellaera on 19/9/19.
3  //
4
5  #ifndef FRAME_OF_REFERENCE_WRITER_H
6  #define FRAME_OF_REFERENCE_WRITER_H
7
8  #include "Thread.h"
9  #include <fstream>
10 #include "BlockingQueue.h"
11 #include <vector>
12
13 class Writer : public Thread {
14     private:
15         std::vector<BlockingQueue*> &queues;
16         std::ofstream stream;
17         bool write_to_stdout;
18         int writeBlock(int index, std::ostream& output);
19     public:
20         Writer(std::vector<BlockingQueue*> &queues, char* outfile_path);
21         virtual void run() override;
22         ~Writer();
23 };
24
25
26 #endif //FRAME_OF_REFERENCE_WRITER_H
```

sep 22, 19 19:33 **Writer.cpp** Page 1/1

```
1  //
2  // Created by leobellaera on 19/9/19.
3  //
4
5  #include "Writer.h"
6  #include <iostream>
7  #include <algorithm>
8  #include <vector>
9  #include <cstring>
10
11 #define QUEUE_CLOSED 1
12 #define SUCCESS 0
13
14 Writer::Writer(std::vector<BlockingQueue*> &queues, char* outfile_path) :
15     queues(queues) {
16     if (strcmp(outfile_path, "-")  ≠ 0) {
17         write_to_stdout = false;
18         stream.open(outfile_path, std::ofstream::binary);
19     } else {
20         write_to_stdout = true;
21     }
22 }
23
24 void Writer::run() {
25     std::ostream& output = write_to_stdout ? std::cout : stream;
26     size_t queues_finished_amount = 0;
27     while (queues_finished_amount ≠ queues.size()) {
28         queues_finished_amount = 0;
29         for (int i = 0; (size_t)i < queues.size(); i++) {
30             if (this→writeBlock(i, output) ≡ QUEUE_CLOSED) {
31                 queues_finished_amount++;
32             }
33         }
34     }
35 }
36
37 int Writer::writeBlock(int index, std::ostream& output) {
38     std::vector<uint8_t> compressed_block;
39     if (queues[index]→pop(compressed_block) ≡ 1) {
40         return QUEUE_CLOSED;
41     }
42     output.write((char*)compressed_block.data(), compressed_block.size());
43     return SUCCESS;
44 }
45
46 Writer::~Writer() {}
```

sep 22, 19 19:33 **Thread.h** Page 1/1

```
1  //
2  // Created by leobellaera on 19/9/19.
3  //
4
5  #ifndef FRAME_OF_REFERENCE_THREAD_H
6  #define FRAME_OF_REFERENCE_THREAD_H
7
8  #include <thread>
9
10 class Thread {
11     private:
12         std::thread thread;
13     public:
14         Thread();
15         void start();
16         void join();
17         virtual void run() = 0;
18         Thread(const Thread&) = delete;
19         Thread& operator=(const Thread&) = delete;
20         Thread(Thread∧ other);
21         Thread& operator=(Thread∧ other);
22         virtual ~Thread();
23 };
24
25 #endif //FRAME_OF_REFERENCE_THREAD_H
```

sep 22, 19 19:33 **Thread.cpp** Page 1/1

```
1  //
2  // Created by leobellaera on 19/9/19.
3  //
4
5  #include "Thread.h"
6
7  Thread::Thread() {}
8
9  void Thread::start() {
10     thread = std::thread(&Thread::run, this);
11 }
12
13 void Thread::join() {
14     thread.join();
15 }
16
17 Thread::Thread(Thread∧ other) {
18     this→thread = std::move(other.thread);
19 }
20
21 Thread& Thread::operator=(Thread∧ other) {
22     this→thread = std::move(other.thread);
23     return *this;
24 }
25
26 Thread::~Thread() {}
```

sep 22, 19 19:33 **SamplesPacker.h** Page 1/1

```
1  //
2  // Created by leobellaera on 13/9/19.
3  //
4
5
6  #ifndef TP2_SAMPLESPACKER_H
7  #define TP2_SAMPLESPACKER_H
8
9  #include <stdint.h>
10 #include <string>
11 #include <vector>
12 #include "Block.h"
13
14 class SamplesPacker {
15     private:
16         int size;
17         uint8_t convertBinaryByteToNumb(const char* binary);
18         void getSamplesPackedAsString(Block& block, std::string &str);
19     public:
20         explicit SamplesPacker(int block_size);
21         uint8_t getBitsPerSample(Block& block);
22         void packSamples(Block& block, std::vector<uint8_t> &compression_buf);
23         ~SamplesPacker();
24 };
25
26 #endif //TP2_SAMPLESPACKER_H
```

sep 22, 19 19:33 **SamplesPacker.cpp** Page 1/1

```
1  //
2  // Created by leobellaera on 13/9/19.
3  //
4
5  #include "SamplesPacker.h"
6  #include <bitset>
7  #include <math.h>
8  #include <vector>
9  #include <string>
10 #include <cstring>
11 #define UINT32_SIZE 4
12
13 SamplesPacker::SamplesPacker(int block_size) :
14         size(block_size) {}
15
16 uint8_t SamplesPacker::getBitsPerSample(Block& block) {
17     uint32_t max = block.getMax();
18     if (max ≡ 0) {
19         return 0;
20     }
21     return (uint8_t)log2(max)+1;
22 }
23
24 void SamplesPacker::packSamples(Block& block,
25         std::vector<uint8_t> &compression_buf) {
26     std::string compressed_block;
27     this→getSamplesPackedAsString(block, compressed_block);
28     const char* aux = compressed_block.c_str();
29     size_t i = 0;
30     while (i < compressed_block.length()) {
31         uint8_t numb = this→convertBinaryByteToNumb(&aux[i]);
32         compression_buf.push_back(numb);
33         i+=8;
34     }
35 }
36
37 void SamplesPacker::getSamplesPackedAsString(Block &block,
38         std::string &string) {
39     int bits_per_sample = this→getBitsPerSample(block);
40     for (int i = 0; i < size; i++) {
41         std::bitset<32> bitset_numb(block.getNumber(i));
42         std::string aux = bitset_numb.to_string<char,
43                 std::string::traits_type,std::string::allocator_type>();
44         aux = aux.substr(32 − bits_per_sample, bits_per_sample);
45         string.append(aux);
46     }
47     while (string.length() % 8 ≠ 0) {
48         string.append("0");
49     }
50 }
51
52 uint8_t SamplesPacker::convertBinaryByteToNumb(const char* binary) {
53     uint8_t sum = 0;
54     for (int i = 0; i < 8; i++) {
55         if (binary[i] ≡ '1') {
56             sum += pow(2, 7−i);
57         }
58     }
59     return sum;
60 }
61
62 SamplesPacker::~SamplesPacker() {}
63
```

sep 22, 19 19:33 **main.cpp** Page 1/1

```cpp
//
// Created by leobellaera on 16/9/19.
//
#include <iostream>
#include <string>
#include "FrameOfReference.h"

#define INVALID_ARGS_AMOUNT_MSG "Invalid number of parameters inserted.\n"

int main(int argc, char *argv[]) {
    if (argc ≠ 6) {
        std::cerr << INVALID_ARGS_AMOUNT_MSG;
        return 1;
    }
    int block_size = std::stoi(argv[1]);
    int threads_processors_amount = std::stoi(argv[2]);
    int q_size = std::stoi(argv[3]);

    FrameOfReference compressor(block_size, threads_processors_amount,
            q_size, argv[4], argv[5]);
    compressor.compress();
    return 0;
}
```

sep 22, 19 19:33 **FrameOfReference.h** Page 1/1

```cpp
//
// Created by leobellaera on 20/9/19.
//

#ifndef FRAME_OF_REFERENCE_FRAMEOFREFERENCE_H
#define FRAME_OF_REFERENCE_FRAMEOFREFERENCE_H

#include <vector>
#include "BlockingQueue.h"
#include "Thread.h"
#include "FileReader.h"
#include "BlocksProcessor.h"
#include "Writer.h"

class FrameOfReference {
    private:
        FileReader file_reader;
        std::vector<BlockingQueue*> queues;
        std::vector<Thread*> threads;
    public:
        FrameOfReference(int block_size, int processors_amount,
                int q_size, char* infile_path, char* outfile_path);
        void compress();
        ~FrameOfReference();
};


#endif //FRAME_OF_REFERENCE_FRAMEOFREFERENCE_H
```

sep 22, 19 19:33 **FrameOfReference.cpp** Page 1/1

```cpp
//
// Created by leobellaera on 20/9/19.
//

#include "FrameOfReference.h"

FrameOfReference::FrameOfReference(int block_size, int processors_amount,
        int q_size, char* infile_path, char* outfile_path) :
    file_reader(infile_path, block_size) {
    for (int i = 0; i < processors_amount; ++i) {
        queues.push_back(new BlockingQueue(q_size));
        threads.push_back(new BlocksProcessor(queues[i],
                file_reader, processors_amount, i, block_size));
    }
    threads.push_back(new Writer(queues, outfile_path));
}

void FrameOfReference::compress() {
    int t_size = threads.size();
    int q_size = queues.size();
    for (int i = 0; i < t_size; i++) {
        threads[i]→start();
    }
    for (int i = 0; i < t_size; i++) {
        threads[i]→join();
    }
    for (int i = 0; i < t_size; i++) {
        delete threads[i];
        if (i < q_size) {
            delete queues[i];
        }
    }
}

FrameOfReference::~FrameOfReference() {}
```

sep 22, 19 19:33 **FileReader.h** Page 1/1

```cpp
//
// Created by leobellaera on 12/9/19.
//

#ifndef TP2_FILE_H
#define TP2_FILE_H

#include <fstream>
#include <mutex>
#include <stdint.h>
#include <vector>

class FileReader {
    private:
        std::ifstream stream;
        int block_size;
        bool read_from_stdin;
        std::mutex m;
        int readSample(std::vector<uint32_t> &destin, std::istream &input);
    public:
        FileReader(char* path, int block_size);
        int readBlock(std::vector<uint32_t> &destin, int block_to_read);
        ~FileReader();
};

#endif //TP2_FILE_H
```

sep 22, 19 19:33 **FileReader.cpp** Page 1/1

```cpp
//
// Created by leobellaera on 12/9/19.
//

#include "FileReader.h"
#include <iostream>
#include <string.h>
#include <cstring>
#include <vector>
#include <endian.h>

#define UINT32_SIZE 4
#define SUCCESS 0
#define EOF_REACHED 1
#define NO_BLOCK_TO_READ -1

FileReader::FileReader(char* path, int block_size) :
    block_size(block_size) {
    if (strcmp(path, "-")  ≠ 0) {
        read_from_stdin = false;
        stream.open(path, std::ifstream::binary);
    } else {
        read_from_stdin = true;
    }
}

int FileReader::readBlock(std::vector<uint32_t> &destin, int block_to_read) {
    std::unique_lock<std::mutex> lock(m);
    std::istream& input = read_from_stdin ? std::cin : stream;
    input.clear();
    input.seekg(UINT32_SIZE * block_size * block_to_read);
    if (input.fail()) {
        return NO_BLOCK_TO_READ;
    }
    for (int i = 0; i < block_size; i++) {
        if (this→readSample(destin, input) ≡ EOF_REACHED) {
            if (i ≡ 0) {
                return NO_BLOCK_TO_READ;
            } else {
                return EOF_REACHED;
            }
        }
    }
    return SUCCESS;
}

int FileReader::readSample(std::vector<uint32_t> &destin, std::istream& input){
    char buf[UINT32_SIZE];
    uint32_t numb;
    input.read(buf, UINT32_SIZE);
    if (input.eof()) {
        return EOF_REACHED;
    } else {
        std::memcpy(&numb, buf, UINT32_SIZE);
        numb = be32toh(numb);
        destin.push_back(numb);
        return SUCCESS;
    }
}

FileReader::~FileReader() {}
```

sep 22, 19 19:33 **BlocksProcessor.h** Page 1/1

```cpp
//
// Created by leobellaera on 20/9/19.
//

#ifndef FRAME_OF_REFERENCE_BLOCKSPROCESSOR_H
#define FRAME_OF_REFERENCE_BLOCKSPROCESSOR_H

#include "Thread.h"
#include "FileReader.h"
#include "BlockingQueue.h"
#include "BlockCompressor.h"

class BlocksProcessor : public Thread {
    private:
        BlockingQueue* queue;
        FileReader &file_reader;
        BlockCompressor block_compressor;
        int processors_amount;
        int slot;
        int block_size;
        int process_block(int block_to_read);
    public:
        BlocksProcessor(BlockingQueue* queue,
                FileReader &fr, int n, int slot, int block_size);
        virtual void run() override;
        ~BlocksProcessor();
};


#endif //FRAME_OF_REFERENCE_BLOCKSPROCESSOR_H
```

sep 22, 19 19:33 **BlocksProcessor.cpp** Page 1/1

```cpp
//
// Created by leobellaera on 20/9/19.
//

#include "BlocksProcessor.h"
#include <vector>

#define REFERENCE_SIZE 4
#define SUCCESS 0
#define EOF_REACHED 1
#define NO_BLOCK_TO_READ -1

BlocksProcessor::BlocksProcessor(BlockingQueue* queue,
        FileReader &fr, int n, int slot, int block_size) :
    queue(queue),
    file_reader(fr),
    block_compressor(block_size),
    processors_amount(n),
    slot(slot),
    block_size(block_size) {}

void BlocksProcessor::run() {
    int state = SUCCESS;
    int block_to_read = slot;
    while (state == SUCCESS) {
        state = process_block(block_to_read);
        block_to_read += processors_amount;
    }
    return;
}

int BlocksProcessor::process_block(int block_to_read) {
    std::vector<uint32_t> numbs;
    int state = file_reader.readBlock(numbs, block_to_read);
    if (state == NO_BLOCK_TO_READ) {
        queue->close();
        return NO_BLOCK_TO_READ;
    }
    Block block(numbs, block_size);
    std::vector<uint8_t> compressed_block(REFERENCE_SIZE);
    block_compressor.compressBlock(block, compressed_block);
    queue->push(compressed_block);
    if (state == EOF_REACHED) {
        queue->close();
    }
    return state;
}

BlocksProcessor::~BlocksProcessor() {}
```

sep 22, 19 19:33 **BlockingQueue.h** Page 1/1

```cpp
//
// Created by leobellaera on 19/9/19.
//

#ifndef FRAME_OF_REFERENCE_BLOCKINGQUEUE_H
#define FRAME_OF_REFERENCE_BLOCKINGQUEUE_H

#include <queue>
#include <vector>
#include <stdint.h>
#include <mutex>
#include <condition_variable>

class BlockingQueue {
    private:
        std::queue<std::vector<uint8_t>> q;
        std::mutex m;
        std::condition_variable cond_var;
        size_t max_size;
        bool closed;
    public:
        explicit BlockingQueue(size_t max_size);
        void push(std::vector<uint8_t> &elem);
        int pop(std::vector<uint8_t> &elem);
        void close();
        ~BlockingQueue();
};

#endif //FRAME_OF_REFERENCE_BLOCKINGQUEUE_H
```

sep 22, 19 19:33 **BlockingQueue.cpp** Page 1/1

```cpp
//
// Created by leobellaera on 20/9/19.
//

#include "BlockingQueue.h"
#include <vector>

#define QUEUE_CLOSED 1
#define SUCCESS 0

BlockingQueue::BlockingQueue(size_t max_size) :
    max_size(max_size),
    closed(false) {}

void BlockingQueue::push(std::vector<uint8_t> &elem) {
    std::unique_lock<std::mutex> lock(m);
    while (q.size() ≡ max_size) {
        cond_var.wait(lock);
    }
    q.push(std::move(elem));
    cond_var.notify_all();
}

int BlockingQueue::pop(std::vector<uint8_t> &elem) {
    std::unique_lock<std::mutex> lock(m);
    while (q.empty() ∧ ¬closed) {
        cond_var.wait(lock);
    }
    if (closed ∧ q.empty()) {
        return QUEUE_CLOSED;
    }
    elem = std::move(q.front());
    q.pop();
    cond_var.notify_all();
    return SUCCESS;
}

void BlockingQueue::close() {
    //no lock, there is no race condition
    closed = true;
    cond_var.notify_all();
}

BlockingQueue::~BlockingQueue() {}
```

sep 22, 19 19:33 **Block.h** Page 1/1

```cpp
//
// Created by leobellaera on 16/9/19.
//

#ifndef FRAME_OF_REFERENCE_BLOCK_H
#define FRAME_OF_REFERENCE_BLOCK_H

#include <stdint.h>
#include <vector>

class Block {
    private:
        int size;
        std::vector<uint32_t> numbers;
    public:
        Block(std::vector<uint32_t> &numbs, int size);
        void subtractMin();
        uint32_t getNumber(int position);
        uint32_t getMax();
        uint32_t getMin();
        ~Block();
};

#endif //FRAME_OF_REFERENCE_BLOCK_H
```

sep 22, 19 19:33 **Block.cpp** Page 1/1

```cpp
//
// Created by leobellaera on 16/9/19.
//

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

Block::Block(std::vector<uint32_t> &numbs, int size):
    size(size),
    numbers(std::move(numbs)) {
    int i = numbers.size();
    while (i < size) {
        numbers.push_back(numbers[i-1]);
        i++;
    }
}

uint32_t Block::getNumber(int position) {
    return numbers[position];
}

void Block::subtractMin() {
    uint32_t min = this→getMin();
    for (int i = 0; i < size; i++) {
        numbers[i] -= min;
    }
}

uint32_t Block::getMin() {
    uint32_t min = numbers[0];
    for (int i = 0; i < size; i++) {
        if (numbers[i] < min) {
            min = numbers[i];
        }
    }
    return min;
}

uint32_t Block::getMax() {
    uint32_t max = numbers[0];
    for (int i = 0; i < size; i++) {
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max;
}

Block::~Block() {}
```

sep 22, 19 19:33 **BlockCompressor.h** Page 1/1

```cpp
//
// Created by leobellaera on 13/9/19.
//

#ifndef FRAME_OF_REFERENCE_BLOCKCOMPRESSOR_H
#define FRAME_OF_REFERENCE_BLOCKCOMPRESSOR_H

#include "Block.h"
#include <vector>
#include <stdint.h>
#include "SamplesPacker.h"


class BlockCompressor {
    private:
        SamplesPacker samples_packer;
        int size;
    public:
        explicit BlockCompressor(int block_size);
        void compressBlock(Block &block,
                std::vector<uint8_t> &compressed_block);
        ~BlockCompressor();
};

#endif
```

sep 22, 19 19:33 **BlockCompressor.cpp** Page 1/1

```
1  //
2  // Created by leobellaera on 13/9/19.
3  //
4
5  #include "BlockCompressor.h"
6  #include <cstring>
7  #include <vector>
8  #include <endian.h>
9
10 #define UINT32_SIZE 4
11
12 BlockCompressor::BlockCompressor(int block_size) :
13     samples_packer(block_size),
14     size(block_size) {}
15
16 void BlockCompressor::compressBlock(Block &block,
17         std::vector<uint8_t> &compressed_block) {
18     uint32_t min = htobe32(block.getMin());
19     block.subtractMin();
20     uint8_t bits_per_sample = samples_packer.getBitsPerSample(block);
21
22     std::memcpy(compressed_block.data(), &min, UINT32_SIZE);
23     compressed_block.push_back(bits_per_sample);
24
25     if (bits_per_sample ≠ 0) {
26         this→samples_packer.packSamples(block, compressed_block);
27     }
28 }
29
30 BlockCompressor::~BlockCompressor() {}
```

sep 22, 19 19:33 **Table of Content** Page 1/1