abr 16, 19 18:18 **thread.h** Page 1/1

```
#ifndef THREAD_H_
#define THREAD_H_
#include <thread>

class Thread {
    private:
        std::thread thread;

    public:
        Thread();
        void start();
        void join();
        virtual void run() = 0;
        Thread(const Thread&) = delete;
        Thread& operator=(const Thread&) = delete;
        virtual ~Thread();
        Thread(Thread∧ other);
        Thread& operator=(Thread∧ other);
};

#endif
```

abr 16, 19 18:18 **thread.cpp** Page 1/1

```
#include "thread.h"
Thread::Thread() {}

void Thread::start() {
   this→thread = std::thread(&Thread::run, this);
}


void Thread::join() {
   this→thread.join();
}

Thread::~Thread() {}

Thread::Thread(Thread∧ other) {
this→thread = std::move(other.thread);
}
```

abr 16, 19 18:18 **producer.h** Page 1/1

```
#ifndef THREADPOOL_H_
#define THREADPOOL_H_
#include <string>
#include "modo.h"
#include "bf_priority.h"

class Producer: public Modo {
    private:
        int n;

    public:
        explicit Producer(int cant);
        ~Producer();
        //Lee de entrada estandar los parÃ¡metros para la creaciÃ³n de branfucks
        //los agrega a una cola de prioridad bloqueante, para que los
        //n threads del pool (Consumidores) los ejecuten
        //tomando siempre el mÃ¡s prioritario.
        int execute();
};

#endif
```

abr 16, 19 18:18 **producer.cpp** Page 1/1

```
#include <condition_variable>
#include <mutex>
#include <thread>
#include <iostream>
#include <queue>
#include <chrono>
#include <string>
#include <vector>
#include "producer.h"
#include "consumer.h"
#include "command_parcer.h"


Producer::Producer(int cant): n(cant) {}

int Producer::execute() {
    std::priority_queue \
        <BfPriority*, std::vector<BfPriority*>, CompareBf> produced_bfs;
    std::mutex m;
    std::condition_variable cond_var;
    bool done = false;
    bool notified = false;
    bool result = true;
    int state = 0;

    std::vector<Thread*> threads;


    for (int i = 0; i < this→n; i++) {
        Thread* consumer = new Consumer(result, m, produced_bfs, \
                                        done, notified, cond_var, i);
        consumer→start();
        threads.push_back(consumer);
    }

    std::string line;
    while ( std::getline(std::cin, line, ')') ) {
        if (std::cin.eof()) {
            break;
        }
        std::unique_lock<std::mutex> lock(m);
        CommandParcer cp(line);
        BfPriority* bf = cp();
        produced_bfs.push(bf);
        notified = true; //le digo que se cumple
        cond_var.notify_one();
        if (¬result){
            state = 1;
            break;
        }
    }
    done = true; //YA hice todo de todo (por si no habia nadie esperando)
    notified = true;
    cond_var.notify_all();

    for (int i = 0; i < this→n; ++i) {
        threads[i]→join();
        delete threads[i];
    }
    return state;
}


Producer::~Producer(){}

```

abr 16, 19 18:18 **modo.h** Page 1/1

```
1  #ifndef MODO_H_
2  #define MODO_H_
3  #include <iostream>
4  #define COMAND_ERROR 2
5  #define CODE_ERROR 1
6  #define OK 0
7
8  class Modo {
9      public:
10     //Modo();
11     int excecute();
12 };
13
14 #endif
```

abr 16, 19 18:18 **modo.cpp** Page 1/1

```
1  #include "modo.h"
2
3
```

abr 16, 19 18:18 **main.cpp** Page 1/1

```cpp
#include <string>
#include <vector>
#include <cstring>
#include <sstream>
#include <queue>
#include "interprete.h"
#include "producer.h"

#define COMAND_ERROR 2



int main(int argc, char * const argv[]) {
    //verifico la # de comandos
    if ( argc ≠ 3 ) {
            return COMAND_ERROR;
    }

    std::string modo1("interprete");
    std::string modo2("thread-pool");
    if ( modo1.compare(0, strlen(argv[1]), argv[1]) ≡ 0 ) {
        Interprete modo(argv[2]);
        return modo.execute();
    } else if ( modo2.compare(0, strlen(argv[1]), argv[1]) ≡ 0 ) {
        int thread = (int) *argv[2] - 48;
        Producer modo(thread);
        return modo.execute();
    }

    return 1;
}
```

abr 16, 19 18:18 **interprete.h** Page 1/1

```cpp
#ifndef INTERPRETE_H_
#define INTERPRETE_H_
#include <string>
#include "modo.h"

class Interprete: public Modo {
    private:
        std::string code;
        std::string filename;
        bool readCode();

    public:
        //recibe el nombre de un archivo que contiene un cÃ³digo brainfuck
        explicit Interprete(std::string _filename);
        ~Interprete();
        //intÃ©rprete de brainfuck âM-^@M-^K
        //que deberÃ¡ ejecutar scripts del mencionado lenguaje.
        int execute();
};

#endif
```

abr 16, 19 18:18 **interprete.cpp** Page 1/1

```cpp
#include <sstream>
#include <string>
#include "interprete.h"
#include "brainfuck.h"

Interprete::Interprete(std::string _filename) {
    this→filename = _filename;
}

bool Interprete::readCode() {
    //abro archivo
    std::ifstream script;
    script.open(this→filename);
    if ( ¬script.good() ) {
        return false;
    }
    //leo el archivo y guardo en "code"
    std::stringstream buffer;
    buffer << script.rdbuf();
    this→code = buffer.str();
    script.close();
    return true;
}


int Interprete::execute() {
    bool is_all_ok = true;
    is_all_ok = this→readCode();
    if (¬is_all_ok) {
        return COMAND_ERROR;
    }

    Brainfuck bf(this→code);
    is_all_ok = bf.start();
    if (¬is_all_ok) {
        return CODE_ERROR;
    }
    return OK;
}

Interprete::~Interprete() {}
```

abr 16, 19 18:18 **executor.h** Page 1/1

```cpp
#ifndef EXCECUTOR_H_
#define EXCECUTOR_H_
#include <string>
#include <vector>
#include <fstream>


class Executor {
    private:
        int data_pos;
        int code_pos;
        std::vector<char> data_memory;
        std::string code_memory;
        std::string in;
        std::string out;

        //Mueve el puntero de datos a la izquierda
        void back();

        //Mueve el puntero de datos a la derecha
        void next();

        //Incrementa el dato apuntado por el puntero de datos en 1 unidad
        void plus();

        //Decrementa el dato apuntado por el puntero de datos en 1 unidad
        void less();

        //Imprime el caracter apuntado por el puntero de datos
        void write(std::ostream* file);

        //Lee un dato de la entrada y lo coloca en la posiciÃ³n
        //apuntada por el puntero de datos
        void read(std::istream* file);

        //Si el dato apuntado por el puntero de datos es igual a cero,
        //salta hasta la instrucciÃ³n siguiente al siguiente âM-^@M-^K ']'
        void loopStart();

        //Vuelve al âM-^@M-^K '[' âM-^@M-^K anterior
        void loopEnd();

    public:
        void print_code();
        explicit Executor(std::string& str);
        Executor(std::string& str, std::string& _in, std::string& _out);
        ~Executor();
        //Abre los archivos de entrada y salida en caso de no ser los estandars
        //Y ejecuta el cÃ³digo bf.
        void start();
};

#endif
```

```
1  #include <iostream>
2   #include <string>
3  #include "executor.h"
4  #define MAX_LEN_CODE_MEMORY 20000
5  #define MAX_LEN_DATA_MEMORY 200
6
7  //&std::cin, &std::cout
8  Executor::Executor(std::string& str) {
9      this→in = std::string("");
10     this→out = std::string("");
11     this→data_memory.resize(MAX_LEN_DATA_MEMORY);
12     this→code_memory = str;
13     std::fill(this→data_memory.begin(), this→data_memory.end(), 0);
14
15     this→data_pos = 0;
16     this→code_pos = 0;
17
18     //this->in = std::cin;
19     //this->out = std::cout;
20 }
21
22 Executor::Executor(std::string& str, std::string& _in, std::string& _out):\
23      in(_in), out(_out) {
24     this→data_memory.resize(MAX_LEN_DATA_MEMORY);
25     this→code_memory = str;
26     std::fill(this→data_memory.begin(), this→data_memory.end(), 0);
27
28     this→data_pos = 0;
29     this→code_pos = 0;
30 }
31
32 //<
33 void Executor::back() {
34     this→data_pos--;
35 }
36
37 //>
38 void Executor::next() {
39     this→data_pos++;
40 }
41
42 //+
43 void Executor::plus() {
44     this→data_memory[this→data_pos]++;
45 }
46
47 //-
48 void Executor::less() {
49     this→data_memory[this→data_pos]--;
50 }
51
52 //.
53 void Executor::write(std::ostream* file) {
54     *file << this→data_memory[this→data_pos];
55 }
56
57 //,
58 void Executor::read(std::istream* file) {
59     this→data_memory[this→data_pos] = file→get();
60     if (this→data_memory[this→data_pos] ≡ -1) {
61         this→data_memory[this→data_pos] = 0;
62     }
63 }
64
65 //[
66 void Executor::loopStart() {
```



```
67     size_t count = 0;
68     bool keep_looking = true;
69     if ( this→data_memory[this→data_pos] ) return;
70     while ( keep_looking ) {
71         this→code_pos++;
72         if (this→code_memory.compare(this→code_pos, 1,"[") ≡ 0) {
73             count++;
74         }
75         if (this→code_memory.compare(this→code_pos, 1,"]") ≡ 0) {
76             if (count ≡ 0) {
77                 keep_looking = false;
78             } else {
79                 count--;
80             }
81         }
82     }
83 }
84
85
86 //]
87 void Executor::loopEnd() {
88     size_t count = 0;
89     bool keep_looking = true;
90
91     while (keep_looking) {
92         this→code_pos--;
93         if (this→code_memory.compare(this→code_pos, 1,"]") ≡ 0) {
94             count++;
95         }
96         if (this→code_memory.compare(this→code_pos, 1,"[") ≡ 0) {
97             if (count ≡ 0) {
98                 keep_looking = false;
99             } else {
100                count--;
101            }
102        }
103    }
104    this→code_pos--;
105 }
106 void Executor::print_code() {
107     std::cout << this→code_memory << '\n';
108 }
109
110 void Executor::start() {
111     std::istream* in_file = &std::cin;
112     std::ostream* out_file = &std::cout;
113     std::ofstream _out_file;
114     std::ifstream _in_file;
115
116     if (this→in.compare("") ≠ 0 ∧
117         this→out.compare("") ≠ 0) {
118         _in_file.open(this→in);
119         _out_file.open(this→out,
120                        std::ofstream::out | std::ofstream::trunc);
121         in_file = &_in_file;
122         out_file = &_out_file;
123     }
124
125     while ( this→code_memory[this→code_pos] ) {
126         switch ( this→code_memory[this→code_pos] ) {
127             case '<':
128                 this→back();
129                 break;
130             case '>':
131                 this→next();
132                 break;
```

abr 16, 19 18:18 **executor.cpp** Page 3/3

```
            case '+':
                this→plus();
                break;
            case '-':
                this→less();
                break;
            case '.':
                this→write(out_file);
                break;
            case ',':
                this→read(in_file);
                break;
            case '[':
                this→loopStart();
                break;
            case ']':
                this→ loopEnd();
                break;
        }
        this→code_pos++;
    }
}

Executor::~Executor() {
    //do nothing
}
```

abr 16, 19 18:18 **consumer.h** Page 1/1

```
#ifndef BFCONSUMER_H_
#define BFCONSUMER_H_
#include <string>
#include <condition_variable>
#include <mutex>
#include <thread>
#include <iostream>
#include <queue>
#include <chrono>
#include <vector>
#include "thread.h"
#include "bf_priority.h"
//#include "compare_bf.h"

class CompareBf {
    public:
        bool operator()(const BfPriority* a, const BfPriority* b) {
            return a→getPriority() < b→getPriority();
            //queria hacer a<b pero no funciona el operador
        }
};


class Consumer : public Thread {
    private:
        bool& result;
        std::mutex& m;
        std::priority_queue \
            <BfPriority*, std::vector<BfPriority*>,CompareBf> \
            &produced_bfs;
        bool& done;
        bool& notified;
        std::condition_variable& cond_var;
        int i;

    public:
        //Recibe una cola bloqueante
        //para ejecutar siempre el bf de mayor prioridad
        Consumer(bool& _result, std::mutex& _m, std::priority_queue \
                    <BfPriority*, std::vector <BfPriority*>, CompareBf> \
                    & _produced_bfs, \
                     bool& _done, bool& _notified,  \
                     std::condition_variable& _cond_var, int _i);

        //Espera la notificaciÃ³n de que hay brainfucks para ejecutar
        //y los ejecuta.
        virtual void run() override;
};


#endif
```

abr 16, 19 18:18 **consumer.cpp** Page 1/1

```cpp
#include <vector>
#include "consumer.h"

Consumer::Consumer(bool& _result, std::mutex& _m, std::priority_queue \
            <BfPriority*, std::vector<BfPriority*>, CompareBf> \
            & _produced_bfs, \
            bool& _done, bool& _notified,  \
            std::condition_variable& _cond_var, int _i) :
    result(_result), \
    m(_m), \
    produced_bfs(_produced_bfs), \
    done(_done), notified(_notified), \
    cond_var(_cond_var), i(_i) \
    {}

void Consumer::run() {
    std::unique_lock<std::mutex> lock(this→m);
    BfPriority* bf;
    while (true) {
        while (¬this→notified) {  // loop to avoid spurious wakeups
            if (this→done) break; //to evoid neverending wait
            this→cond_var.wait(lock);
        }
        notified = false;
        if (this→done ∧ produced_bfs.empty()) break; //same shit

        bf = this→produced_bfs.top();
        this→produced_bfs.pop();

        //libero el mutex pa que otros hilos puedan seguir desencolando
        lock.unlock();
        this→result = bf→start();
        delete bf;
        //lo bloqueo porque wait lo desbloquea
        lock.lock();
    }
}
```

abr 16, 19 18:18 **compiler.h** Page 1/1

```cpp
#ifndef COMPILER_H_
#define COMPILER_H_
#include <string>

class Compiler {
    private:
    std::string code;

    public:
    //Recibe un cÃ³digo brainfuck
    explicit Compiler(std::string& str);

    ~Compiler();
    //Parcea el cÃ³gigo brainfuck y verifica el correcto uso
    //de los corcetes ('[' y ']')
    bool start();
};

#endif
```

abr 16, 19 18:18 **compiler.cpp** Page 1/1

```cpp
#include <iostream>
#include <string>
#include <stack>
#include "compiler.h"
#define ERROR_CODE 2
#define OK_CODE 0


Compiler::Compiler(std::string& str) {
    this→code = str;
}

Compiler::~Compiler() {
    //do nothing
}

bool Compiler::start() {
    char current = this→code[0];
    size_t pos = 0;
    std::stack<char> my_stack;
    while ( current ) {
        if ( current ≡ '[' ) {
            my_stack.push(current);
        }
        if ( current ≡ ']' ) {
            if (my_stack.empty()) {
                return false;
            }
            my_stack.pop();
        }
        pos ++;
        current = this→code[pos];
    }

    if ( ¬my_stack.empty() ) {
        return false;
    }
    return true;
}
```

abr 16, 19 18:18 **command_parcer.h** Page 1/1

```cpp
#ifndef COMMANPARCER_H_
#define COMMANPARCER_H_
#include <string>
#include "bf_priority.h"

class CommandParcer {
    private:
        std::string in_filename;
        std::string out_filename;
        std::string code;
        int priority;
    public:
        //Parcea un comando de tipo:
        //(nombre_del_script, prioridad, archivo_entrada,
        // archivo_salida, codigo_brainfuck)
        //
        //Y guarda los Ãºltimos cuatro parÃ¡metros.
        explicit CommandParcer(std::string& str);

        //crea un brainfuck a partir de los datos obtenidos en la creaciÃ³n.
        BfPriority* operator()();
};

#endif
```

abr 16, 19 18:18 **command_parcer.cpp** Page 1/1

```cpp
#include <string>
#include <algorithm>
#include <iterator>
#include <sstream>
#include <vector>
#include "command_parcer.h"
#include <iostream>

CommandParcer::CommandParcer(std::string& line) {
    std::vector<std::string> container;
  std::string buff{""};
  int i = 0;
    while ( line[i] ) {
    if(line[i] ≡ ',' ∧ line[i+1] ≡ ' ' ∧ buff ≠ "") {
            container.push_back(buff); buff = "";
            i++;
        } else {
             buff+=line[i];
        }
        i++;
  }
  if (buff ≠ "") container.push_back(buff);

    this→in_filename = container[2];
    this→out_filename = container[3];

    container[0][0] = ' ';

    this→code = container[4];
    this→priority= std::atoi(container[1].c_str());
}


BfPriority* CommandParcer::operator()() {
    return new BfPriority(this→code, this→in_filename,\
                          this→out_filename, this→priority);
}
```

abr 16, 19 18:18 **brainfuck.h** Page 1/1

```cpp
#ifndef BRAINFUCK_H_
#define BRAINFUCK_H_
#include <string>
#include "compiler.h"
#include "executor.h"

class Brainfuck {
    private:
        Compiler compiler;
        Executor executor;

    public:
        Brainfuck(std::string& str, std::string& _in, std::string& _out);
        explicit Brainfuck(std::string& str);
        ~Brainfuck();
        bool start();
};

#endif
```

abr 16, 19 18:18 **brainfuck.cpp** Page 1/1

```cpp
#include <iostream>
#include <string>
#include "brainfuck.h"


Brainfuck::Brainfuck(std::string& str):
    compiler(str),
    executor(str) {
        //std::cout << "RECIEN CREADO: " << '\n';
        //excecutor.print_code();
    }

Brainfuck::Brainfuck(std::string& str, std::string& _in, std::string& _out):
    compiler(str),
    executor(str, _in, _out) {
        //std::cout << "RECIEN CREADO: " << '\n';
        //excecutor.print_code();
    }

bool Brainfuck::start() {
    bool is_all_ok = true;
    //std::cout << "ANTES DE COMPILAR: " << '\n';
    //excecutor.print_code();
    is_all_ok = compiler.start();
    if (¬ is_all_ok) return false;
    //std::cout << "HAGO START: " << '\n';
    executor.start();
    //excecutor.print_code();
    return true;
}
/*
void Brainfuck::print_code() {
    excecutor.print_code();
}
*/
Brainfuck::~Brainfuck() {}
```

abr 16, 19 18:18 **bf_priority.h** Page 1/1

```cpp
#ifndef BFPRIORITY_H_
#define BFPRIORITY_H_
#include <string>
#include "brainfuck.h"


//Clase brainfuck con prioridad
class BfPriority {
    private:
        Brainfuck bf;
        int priority;

    public:
        //crea un brainfuck y le asigna su prioridad
        explicit BfPriority(std::string& str,std::string& _in,\
                            std::string& _out, int _i);
        ~BfPriority();

        //inicia el bf
        bool start();

        //sobre carga el operador less than
        bool operator<(BfPriority& other);

        int getPriority() const;
};

#endif
```

abr 16, 19 18:18 **bf_priority.cpp** Page 1/1

```cpp
#include "bf_priority.h"
#include <iostream>
#include <string>

BfPriority::BfPriority(std::string& str,std::string& _in,\
                       std::string& _out, int _i):
    bf(str, _in, _out), priority(_i) {}

bool BfPriority::start(){
    return this→bf.start();
}

bool BfPriority::operator<(BfPriority& other) {
    std::cout << this→priority << "<" << other.priority;
    bool aux = this→priority < other.priority;
    if (aux) {
        std::cout << " TRUE \n";
    } else {
        std::cout << " FALSE \n";
    }
    return aux;
}

int BfPriority::getPriority() const {
    return this→priority;
}

BfPriority::~BfPriority() {}
```

abr 16, 19 18:18 **Table of Content** Page 1/1