

abr 16, 19 18:18

thread.h

Page 1/1

```
1 #ifndef THREAD_H_
2 #define THREAD_H_
3 #include <thread>
4
5 class Thread {
6     private:
7         std::thread thread;
8
9     public:
10         Thread();
11         void start();
12         void join();
13         virtual void run() = 0;
14         Thread(const Thread&) = delete;
15         Thread& operator=(const Thread&) = delete;
16         virtual ~Thread();
17         Thread(Thread^ other);
18         Thread& operator=(Thread^ other);
19 };
20
21 #endif
```

abr 16, 19 18:18

thread.cpp

Page 1/1

```
1 #include "thread.h"
2 Thread::Thread() {}
3
4 void Thread::start() {
5     this->thread = std::thread(&Thread::run, this);
6 }
7
8
9 void Thread::join() {
10     this->thread.join();
11 }
12
13 Thread::~Thread() {}
14
15 Thread::Thread(Thread^ other) {
16     this->thread = std::move(other.thread);
17 }
```

abr 16, 19 18:18

## producer.h

Page 1/1

```

1  #ifndef THREADPOOL_H_
2  #define THREADPOOL_H_
3  #include <string>
4  #include "modo.h"
5  #include "bf_priority.h"
6
7  class Producer: public Modo {
8      private:
9          int n;
10
11      public:
12          explicit Producer(int cant);
13          ~Producer();
14          //Lee de entrada estandar los parámetros para la creación de branfucks
15          //los agrega a una cola de prioridad bloqueante, para que los
16          //n threads del pool (Consumidores) los ejecuten
17          //tomando siempre el más prioritario.
18          int execute();
19  };
20
21 #endif

```

abr 16, 19 18:18

## producer.cpp

Page 1/1

```

1  #include <condition_variable>
2  #include <mutex>
3  #include <thread>
4  #include <iostream>
5  #include <queue>
6  #include <chrono>
7  #include <string>
8  #include <vector>
9  #include "producer.h"
10 #include "consumer.h"
11 #include "command_parcer.h"
12
13
14 Producer::Producer(int cant): n(cant) {}
15
16 int Producer::execute() {
17     std::priority_queue \
18         <BfPriority*, std::vector<BfPriority*>, CompareBf> produced_bfs;
19     std::mutex m;
20     std::condition_variable cond_var;
21     bool done = false;
22     bool notified = false;
23     bool result = true;
24     int state = 0;
25
26     std::vector<Thread*> threads;
27
28
29     for (int i = 0; i < this->n; i++) {
30         Thread* consumer = new Consumer(result, m, produced_bfs, \
31                                         done, notified, cond_var, i);
32         consumer->start();
33         threads.push_back(consumer);
34     }
35
36     std::string line;
37     while ( std::getline(std::cin, line, '\n') ) {
38         if (std::cin.eof()) {
39             break;
40         }
41         std::unique_lock<std::mutex> lock(m);
42         CommandParcer cp(line);
43         BfPriority* bf = cp();
44         produced_bfs.push(bf);
45         notified = true; //le digo que se cumple
46         cond_var.notify_one();
47         if (!result) {
48             state = 1;
49             break;
50         }
51     }
52     done = true; //YA hice todo de todo (por si no habia nadie esperando)
53     notified = true;
54     cond_var.notify_all();
55
56     for (int i = 0; i < this->n; ++i) {
57         threads[i]->join();
58         delete threads[i];
59     }
60     return state;
61 }
62
63
64
65 Producer::~Producer(){}
66

```

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

```

1  #include <string>
2  #include <vector>
3  #include <cstring>
4  #include <sstream>
5  #include <queue>
6  #include "interprete.h"
7  #include "producer.h"
8
9  #define COMAND_ERROR 2
10
11
12
13 int main(int argc, char * const argv[]) {
14     //verifico la # de comandos
15     if ( argc != 3 ) {
16         return COMAND_ERROR;
17     }
18
19     std::string modo1("interprete");
20     std::string modo2("thread-pool");
21     if ( modo1.compare(0, strlen(argv[1]), argv[1]) == 0 ) {
22         Interprete modo(argv[2]);
23         return modo.execute();
24     } else if ( modo2.compare(0, strlen(argv[1]), argv[1]) == 0 ) {
25         int thread = (int) *argv[2] - 48;
26         Producer modo(thread);
27         return modo.execute();
28     }
29
30     return 1;
31 }

```

abr 16, 19 18:18

interprete.h

Page 1/1

```

1  #ifndef INTERPRETE_H_
2  #define INTERPRETE_H_
3  #include <string>
4  #include "modo.h"
5
6  class Interprete: public Modo {
7      private:
8          std::string code;
9          std::string filename;
10         bool readCode();
11
12     public:
13         //recibe el nombre de un archivo que contiene un código brainfuck
14         explicit Interprete(std::string _filename);
15         ~Interprete();
16         //intÃrprete de brainfuck âM-^@M-^K
17         //que deberÃ ejecutar scripts del mencionado lenguaje.
18         int execute();
19     };
20
21 #endif

```

abr 16, 19 18:18

interprete.cpp

Page 1/1

```

1  #include <sstream>
2  #include <string>
3  #include "interprete.h"
4  #include "brainfuck.h"
5
6  Interprete::Interprete(std::string _filename) {
7      this->filename = _filename;
8  }
9
10 bool Interprete::readCode() {
11     //abro archivo
12     std::ifstream script;
13     script.open(this->filename);
14     if ( !script.good() ) {
15         return false;
16     }
17     //leo el archivo y guardo en "code"
18     std::stringstream buffer;
19     buffer << script.rdbuf();
20     this->code = buffer.str();
21     script.close();
22     return true;
23 }
24
25 int Interprete::execute() {
26     bool is_all_ok = true;
27     is_all_ok = this->readCode();
28     if (!is_all_ok) {
29         return COMAND_ERROR;
30     }
31
32     Brainfuck bf(this->code);
33     is_all_ok = bf.start();
34     if (!is_all_ok) {
35         return CODE_ERROR;
36     }
37     return OK;
38 }
39
40
41 Interprete::~Interprete() {}

```

abr 16, 19 18:18

executor.h

Page 1/1

```

1  #ifndef EXECUTOR_H_
2  #define EXECUTOR_H_
3  #include <string>
4  #include <vector>
5  #include <fstream>
6
7
8  class Executor {
9      private:
10         int data_pos;
11         int code_pos;
12         std::vector<char> data_memory;
13         std::string code_memory;
14         std::string in;
15         std::string out;
16
17         //Mueve el puntero de datos a la izquierda
18         void back();
19
20         //Mueve el puntero de datos a la derecha
21         void next();
22
23         //Incrementa el dato apuntado por el puntero de datos en 1 unidad
24         void plus();
25
26         //Decrementa el dato apuntado por el puntero de datos en 1 unidad
27         void less();
28
29         //Imprime el caracter apuntado por el puntero de datos
30         void write(std::ostream* file);
31
32         //Lee un dato de la entrada y lo coloca en la posiciÃ³n
33         //apuntada por el puntero de datos
34         void read(std::istream* file);
35
36         //Si el dato apuntado por el puntero de datos es igual a cero,
37         //salta hasta la instrucciÃ³n siguiente al siguiente âM-^@M-^K ']'
38         void loopStart();
39
40         //Vuelve al âM-^@M-^K '[' âM-^@M-^K anterior
41         void loopEnd();
42
43     public:
44         void print_code();
45         explicit Executor(std::string& str);
46         Executor(std::string& str, std::string& _in, std::string& _out);
47         ~Executor();
48         //Abre los archivos de entrada y salida en caso de no ser los estandars
49         //Y ejecuta el cÃ³digo bf.
50         void start();
51     };
52
53 #endif

```

abr 16, 19 18:18

executor.cpp

Page 1/3

```

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() {

```

abr 16, 19 18:18

executor.cpp

Page 2/3

```

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

```

133         case '+':
134             this->plus();
135             break;
136         case '-':
137             this->less();
138             break;
139         case ' ':
140             this->write(out_file);
141             break;
142         case ',':
143             this->read(in_file);
144             break;
145         case '[':
146             this->loopStart();
147             break;
148         case ']':
149             this-> loopEnd();
150             break;
151     }
152     this->code_pos++;
153 }
154 }
155
156 Executor::~Executor() {
157     //do nothing
158 }
159

```

abr 16, 19 18:18

consumer.h

Page 1/1

```

1  #ifndef BFCONSUMER_H_
2  #define BFCONSUMER_H_
3  #include <string>
4  #include <condition_variable>
5  #include <mutex>
6  #include <thread>
7  #include <iostream>
8  #include <queue>
9  #include <chrono>
10 #include <vector>
11 #include "thread.h"
12 #include "bf_priority.h"
13 // #include "compare_bf.h"
14
15 class CompareBf {
16     public:
17         bool operator()(const BfPriority* a, const BfPriority* b) {
18             return a->getPriority() < b->getPriority();
19             //queria hacer a<b pero no funciona el operador
20         }
21 };
22
23
24 class Consumer : public Thread {
25     private:
26         bool& result;
27         std::mutex& m;
28         std::priority_queue \
29             <BfPriority*, std::vector<BfPriority*>, CompareBf> \
30             &produced_bfs;
31         bool& done;
32         bool& notified;
33         std::condition_variable& cond_var;
34         int i;
35
36     public:
37         //Recibe una cola bloqueante
38         //para ejecutar siempre el bf de mayor prioridad
39         Consumer(bool& _result, std::mutex& _m, std::priority_queue \
40             <BfPriority*, std::vector<BfPriority*>, CompareBf> \
41             & _produced_bfs, \
42             bool& _done, bool& _notified, \
43             std::condition_variable& _cond_var, int _i);
44
45         //Espera la notificaciÃ³n de que hay brainfucks para ejecutar
46         //y los ejecuta.
47         virtual void run() override;
48     };
49
50
51
52 #endif

```

abr 16, 19 18:18

consumer.cpp

Page 1/1

```

1  #include <vector>
2  #include "consumer.h"
3
4  Consumer::Consumer(bool& _result, std::mutex& _m, std::priority_queue \
5      <BfPriority*, std::vector<BfPriority*>, CompareBf> \
6      & _produced_bfs, \
7      bool& _done, bool& _notified, \
8      std::condition_variable& _cond_var, int _i) :
9      result(_result), \
10     m(_m), \
11     produced_bfs(_produced_bfs), \
12     done(_done), notified(_notified), \
13     cond_var(_cond_var), i(_i) \
14     {}
15
16 void Consumer::run() {
17     std::unique_lock<std::mutex> lock(this->m);
18     BfPriority* bf;
19     while (true) {
20         while (!this->notified) { // loop to avoid spurious wakeups
21             if (this->done) break; //to avoid neverending wait
22             this->cond_var.wait(lock);
23         }
24         notified = false;
25         if (this->done ^ produced_bfs.empty()) break; //same shit
26
27         bf = this->produced_bfs.top();
28         this->produced_bfs.pop();
29
30         //libero el mutex pa que otros hilos puedan seguir desencolando
31         lock.unlock();
32         this->result = bf->start();
33         delete bf;
34         //lo bloqueo porque wait lo desbloquea
35         lock.lock();
36     }
37 }

```

abr 16, 19 18:18

compiler.h

Page 1/1

```

1  #ifndef COMPILER_H_
2  #define COMPILER_H_
3  #include <string>
4
5  class Compiler {
6      private:
7          std::string code;
8
9      public:
10         //Recibe un código brainfuck
11         explicit Compiler(std::string& str);
12
13         ~Compiler();
14         //Parcea el código brainfuck y verifica el correcto uso
15         //de los corchetes ('[' y ']')
16         bool start();
17     };
18
19 #endif

```



abr 16, 19 18:18

compiler.cpp

Page 1/1

```

1  #include <iostream>
2  #include <string>
3  #include <stack>
4  #include "compiler.h"
5  #define ERROR_CODE 2
6  #define OK_CODE 0
7
8
9  Compiler::Compiler(std::string& str) {
10     this->code = str;
11 }
12
13 Compiler::~Compiler() {
14     //do nothing
15 }
16
17 bool Compiler::start() {
18     char current = this->code[0];
19     size_t pos = 0;
20     std::stack<char> my_stack;
21     while ( current ) {
22         if ( current == '[' ) {
23             my_stack.push(current);
24         }
25         if ( current == ']' ) {
26             if (my_stack.empty()) {
27                 return false;
28             }
29             my_stack.pop();
30         }
31         pos++;
32         current = this->code[pos];
33     }
34
35     if ( !my_stack.empty() ) {
36         return false;
37     }
38     return true;
39 }

```

abr 16, 19 18:18

command\_parcer.h

Page 1/1

```

1  #ifndef COMMANPARCER_H_
2  #define COMMANPARCER_H_
3  #include <string>
4  #include "bf_priority.h"
5
6  class CommandParcer {
7     private:
8         std::string in_filename;
9         std::string out_filename;
10        std::string code;
11        int priority;
12    public:
13        //Parcea un comando de tipo:
14        //(nombre_del_script, prioridad, archivo_entrada,
15        // archivo_salida, codigo_brainfuck)
16        //
17        //Y guarda los Ãºltimos cuatro parÃ¡metros.
18        explicit CommandParcer(std::string& str);
19
20        //crea un brainfuck a partir de los datos obtenidos en la creaciÃ³n.
21        BfPriority* operator()();
22    };
23
24 #endif

```

abr 16, 19 18:18

command\_parcer.cpp

Page 1/1

```

1  #include <string>
2  #include <algorithm>
3  #include <iterator>
4  #include <sstream>
5  #include <vector>
6  #include "command_parcer.h"
7  #include <iostream>
8
9  CommandParcer::CommandParcer(std::string& line) {
10     std::vector<std::string> container;
11     std::string buff{" "};
12     int i = 0;
13     while ( line[i] ) {
14         if(line[i] == ',' ^ line[i+1] == ' ' ^ buff != " ") {
15             container.push_back(buff); buff = " ";
16             i++;
17         } else {
18             buff+=line[i];
19         }
20         i++;
21     }
22     if (buff != " ") container.push_back(buff);
23
24     this->in_filename = container[2];
25     this->out_filename = container[3];
26
27     container[0][0] = ' ';
28
29     this->code = container[4];
30     this->priority= std::atoi(container[1].c_str());
31 }
32
33 BfPriority* CommandParcer::operator()() {
34     return new BfPriority(this->code, this->in_filename,\
35                          this->out_filename, this->priority);
36 }
37

```

abr 16, 19 18:18

brainfuck.h

Page 1/1

```

1  #ifndef BRAINFUCK_H_
2  #define BRAINFUCK_H_
3  #include <string>
4  #include "compiler.h"
5  #include "executor.h"
6
7  class Brainfuck {
8     private:
9         Compiler compiler;
10         Executor executor;
11
12     public:
13         Brainfuck(std::string& str, std::string& _in, std::string& _out);
14         explicit Brainfuck(std::string& str);
15         ~Brainfuck();
16         bool start();
17     };
18
19 #endif

```

abr 16, 19 18:18

brainfuck.cpp

Page 1/1

```

1  #include <iostream>
2  #include <string>
3  #include "brainfuck.h"
4
5
6  Brainfuck::Brainfuck(std::string& str):
7      compiler(str),
8      executor(str) {
9      //std::cout << "RECIEN CREADO: " << '\n';
10     //executor.print_code();
11 }
12
13 Brainfuck::Brainfuck(std::string& str, std::string& _in, std::string& _out):
14     compiler(str),
15     executor(str, _in, _out) {
16     //std::cout << "RECIEN CREADO: " << '\n';
17     //executor.print_code();
18 }
19
20 bool Brainfuck::start() {
21     bool is_all_ok = true;
22     //std::cout << "ANTES DE COMPILAR: " << '\n';
23     //executor.print_code();
24     is_all_ok = compiler.start();
25     if (!is_all_ok) return false;
26     //std::cout << "HAGO START: " << '\n';
27     executor.start();
28     //executor.print_code();
29     return true;
30 }
31 /*
32 void Brainfuck::print_code() {
33     executor.print_code();
34 }
35 */
36 Brainfuck::~Brainfuck() {}

```

abr 16, 19 18:18

bf\_priority.h

Page 1/1

```

1  #ifndef BFPRIORITY_H_
2  #define BFPRIORITY_H_
3  #include <string>
4  #include "brainfuck.h"
5
6
7  //Clase brainfuck con prioridad
8  class BfPriority {
9      private:
10         Brainfuck bf;
11         int priority;
12
13     public:
14         //crea un brainfuck y le asigna su prioridad
15         explicit BfPriority(std::string& str, std::string& _in, \
16                             std::string& _out, int _i);
17         ~BfPriority();
18
19         //inicia el bf
20         bool start();
21
22         //sobre carga el operador less than
23         bool operator<(BfPriority& other);
24
25         int getPriority() const;
26     };
27
28 #endif

```

abr 16, 19 18:18

## bf\_priority.cpp

Page 1/1

```

1  #include "bf_priority.h"
2  #include <iostream>
3  #include <string>
4
5  BfPriority::BfPriority(std::string& str, std::string& _in, \
6                      std::string& _out, int _i):
7      bf(str, _in, _out), priority(_i) {}
8
9  bool BfPriority::start(){
10     return this->bf.start();
11 }
12
13 bool BfPriority::operator<(BfPriority& other) {
14     std::cout << this->priority << "<" << other.priority;
15     bool aux = this->priority < other.priority;
16     if (aux) {
17         std::cout << " TRUE\n";
18     } else {
19         std::cout << " FALSE\n";
20     }
21     return aux;
22 }
23
24 int BfPriority::getPriority() const {
25     return this->priority;
26 }
27
28 BfPriority::~BfPriority() {}

```

abr 16, 19 18:18

## Table of Content

Page 1/1

1	Table of Contents				
2	1 thread.h.....	sheets	1 to	1 ( 1) pages	1- 1 22 lines
3	2 thread.cpp.....	sheets	1 to	1 ( 1) pages	2- 2 18 lines
4	3 producer.h.....	sheets	2 to	2 ( 1) pages	3- 3 22 lines
5	4 producer.cpp.....	sheets	2 to	2 ( 1) pages	4- 4 67 lines
6	5 modo.h.....	sheets	3 to	3 ( 1) pages	5- 5 15 lines
7	6 modo.cpp.....	sheets	3 to	3 ( 1) pages	6- 6 4 lines
8	7 main.cpp.....	sheets	4 to	4 ( 1) pages	7- 7 32 lines
9	8 interprete.h.....	sheets	4 to	4 ( 1) pages	8- 8 22 lines
10	9 interprete.cpp.....	sheets	5 to	5 ( 1) pages	9- 9 42 lines
11	10 executor.h.....	sheets	5 to	5 ( 1) pages	10- 10 54 lines
12	11 executor.cpp.....	sheets	6 to	7 ( 2) pages	11- 13 160 lines
13	12 consumer.h.....	sheets	7 to	7 ( 1) pages	14- 14 53 lines
14	13 consumer.cpp.....	sheets	8 to	8 ( 1) pages	15- 15 38 lines
15	14 compiler.h.....	sheets	8 to	8 ( 1) pages	16- 16 20 lines
16	15 compiler.cpp.....	sheets	9 to	9 ( 1) pages	17- 17 40 lines
17	16 command_parcer.h....	sheets	9 to	9 ( 1) pages	18- 18 25 lines
18	17 command_parcer.cpp..	sheets	10 to	10 ( 1) pages	19- 19 38 lines
19	18 brainfuck.h.....	sheets	10 to	10 ( 1) pages	20- 20 20 lines
20	19 brainfuck.cpp.....	sheets	11 to	11 ( 1) pages	21- 21 37 lines
21	20 bf_priority.h.....	sheets	11 to	11 ( 1) pages	22- 22 29 lines
22	21 bf_priority.cpp.....	sheets	12 to	12 ( 1) pages	23- 23 29 lines