ThreadPool

Generated by Doxygen 1.9.4

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

Data Structure Documentation

3.1 ThreadPool Class Reference

Manages a pool of worker threads to execute tasks concurrently.

```
#include <ThreadPool.hpp>
```

Public Member Functions

- ThreadPool ()=delete

    Deleted default constructor.
- ThreadPool (unsigned int num_threads)

    Constructs a ThreadPool with a fixed number of threads.
- template<class Func , class... Args>
  void submit (Func &&f, Args &&... args)

    Submits a task to the thread pool.
- void finish (bool secure=true)

    Stops the thread pool.
- bool status ()

    Returns the current state of the thread pool.
- void wait ()

    Waits until the task queue becomes empty.
- ∼ThreadPool ()

    Destructor.

Private Attributes

- thread ∗ th
- mutex mtx
- queue< function< void()> > q
- unsigned int num_threads
- bool exit_flag
- bool state
- int cont = 0
- condition_variable cv

3.1.1 Detailed Description

Manages a pool of worker threads to execute tasks concurrently.

The ThreadPool maintains a queue of tasks. Worker threads continuously fetch and execute tasks from this queue until the pool is stopped.

Note

Tasks have no return value.

Warning

This implementation uses busy waiting and does not use condition variables.

Definition at line 35 of file ThreadPool.hpp.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 ThreadPool() `[1/2]`

```
ThreadPool::ThreadPool ( )  [delete]
```

Deleted default constructor.

Forces the user to specify the number of threads.

3.1.2.2 ThreadPool() `[2/2]`

```
ThreadPool::ThreadPool (
            unsigned int num_threads )  [explicit]
```

Constructs a ThreadPool with a fixed number of threads.

Constructs and starts the worker threads.

Parameters

num_threads	Number of worker threads to create.

Each worker thread runs a loop that retrieves tasks from the queue and executes them until termination is requested. Worker thread function.

Continuously checks the task queue and executes tasks while the pool is active and the exit flag is not set.

Definition at line 15 of file ThreadPool.cpp.

```
00016     :  num_threads(num_threads) {
00017
00018 /**
00019 * @brief Worker thread function.
00020 *
00021 * Continuously checks the task queue and executes tasks
00022 * while the pool is active and the exit flag is not set.
00023 */
00024     function<void(void)> f = [&]() {
00025
00026         function<void()> task;
00027
00028         unique_lock<mutex> lock(this->mtx);
00029
00030         while (!this->exit_flag && this->state) {
00031             cv.wait(lock,[&](){return !this->q.empty();});
00032
00033             if (!this->q.empty()) {
00034                 task = move(this->q.front());
00035                 this->q.pop();
00036
00037                 this->mtx.unlock();
00038
00039                 task();
00040             }
00041         }
00042     };
00043
00044     exit_flag = false;
00045
00046     this->th = new (nothrow) thread[num_threads];
00047
00048     if (this->th != nullptr) {
00049         this->state = true;
00050
00051         try {
00052             for (unsigned int i = 0; i < this->num_threads; i++) {
00053                 this->th[i] = thread(f);
00054             }
00055         } catch (exception&) {
00056             this->state = false;
00057             cout « "The system is unable to start a new thread" « endl;
00058         }
00059     } else {
00060         this->state = false;
00061     }
00062 }
```

3.1.2.3 ∼ThreadPool()

```
ThreadPool::∼ThreadPool ( )
```

Destructor.

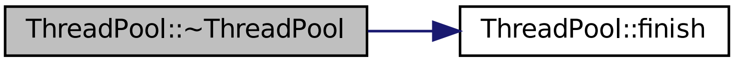
Stops the pool and releases all allocated resources.

Ensures that the pool is stopped and memory is released.

Definition at line 100 of file ThreadPool.cpp.

```
00100                         {
00101     this->finish();
00102     try{
00103         delete[] this->th;
00104     }catch(...){
00105
00106     }
00107 }
```

Here is the call graph for this function:



3.1.3 Member Function Documentation

3.1.3.1 finish()

```
void ThreadPool::finish (
            bool secure = true )
```

Stops the thread pool.

Stops the thread pool and optionally joins threads.

Sets the exit flag and optionally waits for all threads to finish.

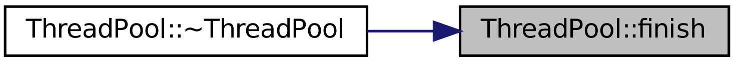
Parameters

secure	If true, joins all worker threads.
secure	If true, waits for all threads to finish execution.

Definition at line 78 of file ThreadPool.cpp.

```
                                        {
    this->exit_flag = true;

    if (secure && this->state) {
        for (unsigned int i = 0; i < this->num_threads; i++) {
            if (this->th[i].joinable()) {
                try {
                    this->th[i].join();
                } catch (...)  {
                    cout « "Error joining thread " « i « endl;
                }
            }
        }
        this->state = false;
    }
}
```

Here is the caller graph for this function:



3.1.3.2 status()

```
bool ThreadPool::status ( )
```

Returns the current state of the thread pool.

Checks whether the thread pool is running.

Returns

True if the pool is running, false otherwise.

True if the pool is active, false otherwise.

Definition at line 69 of file ThreadPool.cpp.

```
                             {
    return this->state;
}
```

3.1.3.3 submit()

```
template<class Func , class...  Args>
void ThreadPool::submit (
            Func && f,
            Args &&...  args )
```

Submits a task to the thread pool.

Adds a new task to the task queue.

The task is stored in the internal queue and will be executed by one of the worker threads.

Template Parameters

Func	Callable object type.
Args	Argument types for the callable.

Parameters

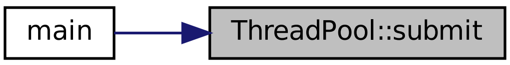
f	Function to execute.
args	Arguments passed to the function.

Locks the mutex, pushes the task into the queue, and then unlocks the mutex.

Definition at line 132 of file ThreadPool.hpp.

```
                                                      {

    {
        lock_guard<mutex> lock(this->mtx);

        this->q.push(
            move(
                [&]() {
                    f(args...);
                }
            )
        );
    }

    this->cv.notify_one();
}
```

Here is the caller graph for this function:



3.1.3.4 wait()

```
void ThreadPool::wait ( )
```

Waits until the task queue becomes empty.

Waits until all queued tasks have been processed.

Note

This function performs a busy wait.

Warning

This function uses busy waiting and may waste CPU time.

Definition at line 114 of file ThreadPool.cpp.

```
                           {
    while (!this->q.empty()) {
        // Busy wait
    }
}
```

3.1.4 Field Documentation

3.1.4.1 cont

```
int ThreadPool::cont = 0  [private]
```

Auxiliary counter (currently unused)

Definition at line 58 of file ThreadPool.hpp.

3.1.4.2 cv

```
condition_variable ThreadPool::cv  [private]
```

Condition variable used to stop the threads until a new process is pushed

Definition at line 61 of file ThreadPool.hpp.

3.1.4.3 exit_flag

```
bool ThreadPool::exit_flag  [private]
```

Flag indicating when threads should terminate

Definition at line 52 of file ThreadPool.hpp.

3.1.4.4 mtx

```
mutex ThreadPool::mtx  [private]
```

Mutex protecting access to the task queue

Definition at line 43 of file ThreadPool.hpp.

3.1.4.5 num_threads

```
unsigned int ThreadPool::num_threads  [private]
```

Number of threads managed by the pool

Definition at line 49 of file ThreadPool.hpp.

3.1.4.6 q

```
queue<function<void()> > ThreadPool::q  [private]
```

Queue containing pending tasks

Definition at line 46 of file ThreadPool.hpp.

3.1.4.7 state

```
bool ThreadPool::state  [private]
```

Indicates whether the thread pool is active

Definition at line 55 of file ThreadPool.hpp.

3.1.4.8 th

```
thread* ThreadPool::th  [private]
```

Pointer to the array of worker threads

Definition at line 40 of file ThreadPool.hpp.

The documentation for this class was generated from the following files:

- ThreadPool.hpp
- ThreadPool.cpp

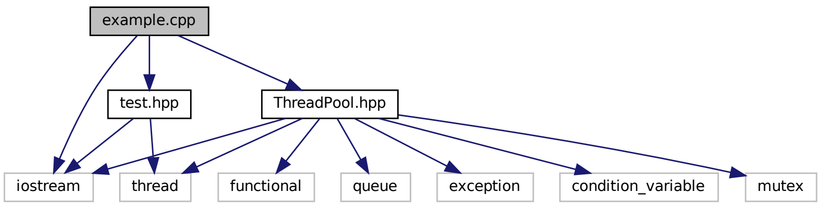
Chapter 4

File Documentation

4.1 example.cpp File Reference

```
#include <iostream>
#include "ThreadPool.hpp"
#include "test.hpp"
```

Include dependency graph for example.cpp:



Functions

- int main (int argc, char ∗argv[ ])

4.1.1 Function Documentation

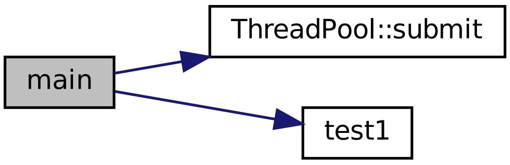
4.1.1.1 main()

```
int main (
              int argc,
              char * argv[] )
```

Definition at line 9 of file example.cpp.

```
00009                                       {
00010 
00011 
00012     ThreadPool thp{10};
00013 
00014     for(int i=0;i<20;i++){
00015         thp.submit(test1);
00016     }
00017 
00018 
00019     // thp.wait();
00020 
00021     if(thp.status()){
00022         thp.finish();
00023     }
00024 
00025     if(!thp.status()){
00026         cout«"Cerrado correctamente"«endl;
00027     }
00028 
00029     return 0;
00030 }
```

Here is the call graph for this function:



4.2 example.cpp

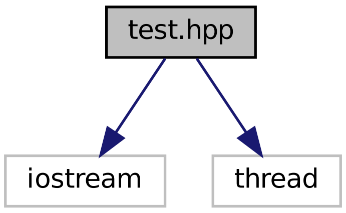
Go to the documentation of this file.

```
00001 #include <iostream>
00002 #include "ThreadPool.hpp"
00003 #include "test.hpp"
00004 
00005 
00006 using namespace std;
00007 
00008 
00009 int main(int argc, char * argv[]){
00010 
00011 
00012     ThreadPool thp{10};
00013 
00014     for(int i=0;i<20;i++){
00015         thp.submit(test1);
00016     }
00017 
00018 
00019     // thp.wait();
00020 
00021     if(thp.status()){
        thp.finish();
    }

    if(!thp.status()){
        cout«"Cerrado correctamente"«endl;
    }

    return 0;
}
```

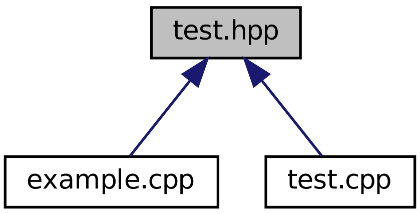
4.3 test.hpp File Reference

```
#include <iostream>
#include <thread>
```

Include dependency graph for test.hpp:



This graph shows which files directly or indirectly include this file:



Functions

- void test1 ()
- void test2 (int, int)
- template<class T >
  void test3 (T x, T y)
- template<class... Args>
  void test4 (int x, Args... args)

- void test4 ()
- void function1 (int x, int y, int z)
- template<class... Args>
  void function2 (void(∗funcion)(int, int, int), Args... args)

4.3.1 Function Documentation

4.3.1.1 function1()

```
void function1 (
            int x,
            int y,
            int z )
```

Definition at line 27 of file test.cpp.

```
00027                              {
00028     cout«x+y+x«endl;
00029
00030 }
```

4.3.1.2 function2()

```
template<class...  Args>
void function2 (
            void(∗)(int, int, int) funcion,
            Args...  args )
```

Definition at line 33 of file test.cpp.

```
00033                                                     {
00034
00035     cout«"function2 "«endl;
00036     funcion(2,3,4);
00037     funcion(args...);
00038 }
```

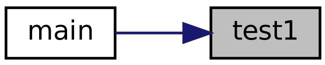
4.3.1.3 test1()

```
void test1 ( )
```

Definition at line 7 of file test.cpp.

```
00007           {
00008     cout«this_thread::get_id()«" Proceso dentro de test1"«endl;
00009 }
```

Here is the caller graph for this function:



4.3.1.4 test2()

```
void test2 (
             int x,
             int y )
```

Definition at line 12 of file test.cpp.

```
00012                          {
00013     cout«this_thread::get_id()«" Proceso dentro de test2 con suma "«x+y«endl;
00014 }
```

4.3.1.5 test3()

```
template<class T >
void test3 (
             T x,
             T y )
```

Definition at line 13 of file test.hpp.

```
00013                      {
00014     cout«this_thread::get_id()«" Proceso dentro de test3 con suma "«x+y«endl;
00015 }
```

4.3.1.6 test4() [1/2]

```
void test4 ( )
```

Definition at line 22 of file test.cpp.

```
00022               {
00023     cout«this_thread::get_id()«" Proceso dentro de test4 sin argumentos"«endl;
00024 }
```

4.3.1.7 test4() [2/2]

```
template<class...  Args>
void test4 (
             int x,
             Args...  args )
```

Definition at line 18 of file test.hpp.

```
00018                               {
00019     cout«this_thread::get_id()«" Proceso dentro de test4 con argumento "«x«endl;
00020 }
```

4.4 test.hpp

Go to the documentation of this file.

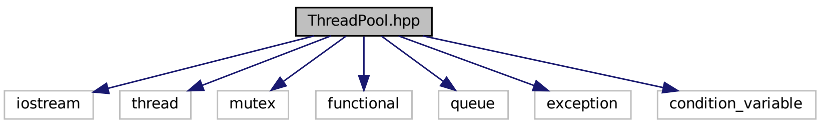
```
#ifndef TEST_HPP
#define TEST_HPP

#include <iostream>
#include <thread>

using namespace std;

void test1();
void test2(int,int);

template <class T>
void test3(T x,T y){
    cout«this_thread::get_id()«" Proceso dentro de test3 con suma "«x+y«endl;
}

template <class... Args>
void test4(int x,Args... args){
    cout«this_thread::get_id()«" Proceso dentro de test4 con argumento "«x«endl;
}

void test4();


void function1(int x,int y,int z);

template <class... Args>
void function2(void (*funcion)(int,int,int),Args... args);


#endif
```

4.5 ThreadPool.hpp File Reference

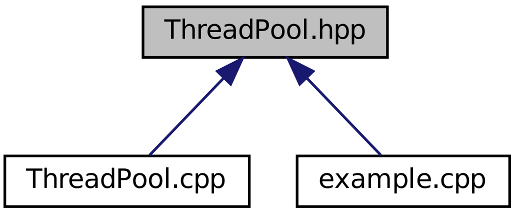
Declaration of a simple ThreadPool class.

```
#include <iostream>
#include <thread>
#include <mutex>
#include <functional>
#include <queue>
#include <exception>
#include <condition_variable>
```

Include dependency graph for ThreadPool.hpp:



This graph shows which files directly or indirectly include this file:



Data Structures

- class ThreadPool

  Manages a pool of worker threads to execute tasks concurrently.

4.5.1 Detailed Description

Declaration of a simple ThreadPool class.

Author

qwert-asci

This file contains the declaration of a basic thread pool that executes tasks using a fixed number of worker threads.

Definition in file ThreadPool.hpp.

4.6 ThreadPool.hpp

Go to the documentation of this file.

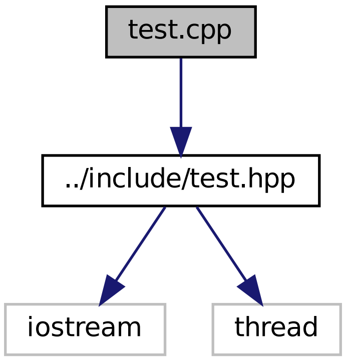
```

/**
* @file ThreadPool.hpp
* @author qwert-asci
* @brief Declaration of a simple ThreadPool class.
*
* This file contains the declaration of a basic thread pool
* that executes tasks using a fixed number of worker threads.
*/

#ifndef THREADPOOL_HPP
#define THREADPOOL_HPP

#include <iostream>
#include <thread>
#include <mutex>
#include <functional>
#include <queue>
#include <exception>
#include <condition_variable>

using namespace std;
```

```cpp

/**
* @class ThreadPool
* @brief Manages a pool of worker threads to execute tasks concurrently.
*
* The ThreadPool maintains a queue of tasks.  Worker threads continuously
* fetch and execute tasks from this queue until the pool is stopped.
*
* @note Tasks have no return value.
* @warning This implementation uses busy waiting and does not use
*          condition variables.
*/
class ThreadPool {

private:

/** Pointer to the array of worker threads */
    thread* th;

/** Mutex protecting access to the task queue */
    mutex mtx;

/** Queue containing pending tasks */
    queue<function<void()» q;

/** Number of threads managed by the pool */
    unsigned int num_threads;

/** Flag indicating when threads should terminate */
    bool exit_flag;

/** Indicates whether the thread pool is active */
    bool state;

/** Auxiliary counter (currently unused) */
    int cont = 0;

/** Condition variable used to stop the threads until a new process is pushed */
    condition_variable cv;

public:

/**
* @brief Deleted default constructor.
*
* Forces the user to specify the number of threads.
*/
    ThreadPool() = delete;

/**
* @brief Constructs a ThreadPool with a fixed number of threads.
*
* @param num_threads Number of worker threads to create.
*/
    explicit ThreadPool(unsigned int num_threads);

/**
* @brief Submits a task to the thread pool.
*
* The task is stored in the internal queue and will be executed
* by one of the worker threads.
*
* @tparam Func Callable object type.
* @tparam Args Argument types for the callable.
* @param f Function to execute.
* @param args Arguments passed to the function.
*
*/
    template <class Func, class...  Args>
    void submit(Func&& f, Args&&...  args);

/**
* @brief Stops the thread pool.
*
* Sets the exit flag and optionally waits for all threads to finish.
*
* @param secure If true, joins all worker threads.
*/
    void finish(bool secure = true);

/**
* @brief Returns the current state of the thread pool.
*
* @return True if the pool is running, false otherwise.
*/
    bool status();

/**
* @brief Waits until the task queue becomes empty.
*
* @note This function performs a busy wait.
*/
    void wait();

/**
* @brief Destructor.
*
* Stops the pool and releases all allocated resources.
*/
    ~ThreadPool();
};

/**
* @brief Adds a new task to the task queue.
*
* Locks the mutex, pushes the task into the queue,
* and then unlocks the mutex.
*/
template <class Func, class...  Args>
void ThreadPool::submit(Func&& f, Args&&...  args) {

    {
        lock_guard<mutex> lock(this->mtx);


        this->q.push(
            move(
                [&]() {
                    f(args...);
                }
            )
        );
    }

    this->cv.notify_one();
}

#endif
```

4.7 test.cpp File Reference

`#include "../include/test.hpp"`

Include dependency graph for test.cpp:



Functions

- void test1 ()

- void test2 (int x, int y)
- void test4 ()
- void function1 (int x, int y, int z)
- template<class... Args>
  void function2 (void(∗funcion)(int, int, int), Args... args)

4.7.1 Function Documentation

4.7.1.1 function1()

```
void function1 (
            int x,
            int y,
            int z )
```

Definition at line 27 of file test.cpp.

```
00027                             {
00028     cout«x+y+x«endl;
00029 
00030 }
```

4.7.1.2 function2()

```
template<class... Args>
void function2 (
            void(*)(int, int, int) funcion,
            Args... args )
```

Definition at line 33 of file test.cpp.

```
00033                                                 {
00034 
00035     cout«"function2 "«endl;
00036     funcion(2,3,4);
00037     funcion(args...);
00038 }
```

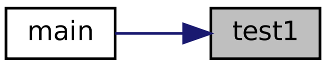
4.7.1.3 test1()

```
void test1 ( )
```

Definition at line 7 of file test.cpp.

```
00007           {
00008     cout«this_thread::get_id()«" Proceso dentro de test1"«endl;
00009 }
```

Here is the caller graph for this function:



4.7.1.4 test2()

```
void test2 (
              int x,
              int y )
```

Definition at line 12 of file test.cpp.

```
00012                            {
00013     cout«this_thread::get_id()«" Proceso dentro de test2 con suma "«x+y«endl;
00014 }
```

4.7.1.5 test4()

```
void test4 ( )
```

Definition at line 22 of file test.cpp.

```
00022                 {
00023     cout«this_thread::get_id()«" Proceso dentro de test4 sin argumentos"«endl;
00024 }
```

4.8 test.cpp

Go to the documentation of this file.

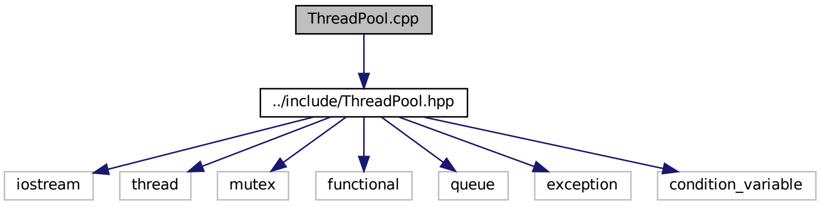
```
00001 #include "../include/test.hpp"
00002
00003
00004
00005 using namespace std;
00006
00007 void test1(){
00008     cout«this_thread::get_id()«" Proceso dentro de test1"«endl;
00009 }
00010
00011
00012 void test2(int x,int y){
00013     cout«this_thread::get_id()«" Proceso dentro de test2 con suma "«x+y«endl;
00014 }
00015
00016
00017
00018
00019
00020
00021
00022 void test4(){
00023     cout«this_thread::get_id()«" Proceso dentro de test4 sin argumentos"«endl;
00024 }
00025
00026
00027 void function1(int x,int y,int z){
00028     cout«x+y+x«endl;
00029
00030 }
00031
00032 template <class...  Args>
00033 void function2(void (*funcion)(int,int,int),Args...  args){
00034
00035     cout«"function2 "«endl;
00036     funcion(2,3,4);
00037     funcion(args...);
00038 }
```

4.9 ThreadPool.cpp File Reference

Implementation of the ThreadPool class.

```
#include "../include/ThreadPool.hpp"
```

Include dependency graph for ThreadPool.cpp:



4.9.1 Detailed Description

Implementation of the ThreadPool class.

Definition in file ThreadPool.cpp.

4.10 ThreadPool.cpp

Go to the documentation of this file.

```
/**
* @file ThreadPool.cpp
* @brief Implementation of the ThreadPool class.
*/

#include "../include/ThreadPool.hpp"

/**
* @brief Constructs and starts the worker threads.
*
* Each worker thread runs a loop that retrieves tasks
* from the queue and executes them until termination
* is requested.
*/
ThreadPool::ThreadPool(unsigned int num_threads)
    :  num_threads(num_threads) {

/**
* @brief Worker thread function.
*
* Continuously checks the task queue and executes tasks
* while the pool is active and the exit flag is not set.
*/
    function<void(void)> f = [&]() {

        function<void()> task;

        unique_lock<mutex> lock(this->mtx);

        while (!this->exit_flag && this->state) {
            cv.wait(lock,[&](){return !this->q.empty();});

            if (!this->q.empty()) {
                task = move(this->q.front());
                this->q.pop();

                this->mtx.unlock();

                task();
            }
        }
    };

    exit_flag = false;

    this->th = new (nothrow) thread[num_threads];

    if (this->th != nullptr) {
        this->state = true;

        try {
            for (unsigned int i = 0; i < this->num_threads; i++) {
                this->th[i] = thread(f);
            }
        } catch (exception&) {
            this->state = false;
            cout « "The system is unable to start a new thread" « endl;
        }
    } else {
        this->state = false;
    }
}

/**
* @brief Checks whether the thread pool is running.
*
* @return True if the pool is active, false otherwise.
*/
bool ThreadPool::status() {
    return this->state;
}

/**
* @brief Stops the thread pool and optionally joins threads.
*
* @param secure If true, waits for all threads to finish execution.
*/
void ThreadPool::finish(bool secure) {
    this->exit_flag = true;

    if (secure && this->state) {
        for (unsigned int i = 0; i < this->num_threads; i++) {
            if (this->th[i].joinable()) {
                try {
                    this->th[i].join();
                } catch (...)  {
                    cout « "Error joining thread " « i « endl;
                }
            }
        }
        this->state = false;
    }
}

/**
* @brief Destructor.
*
* Ensures that the pool is stopped and memory is released.
*/
ThreadPool::~ThreadPool() {
    this->finish();
    try{
        delete[] this->th;
    }catch(...){

    }
}

/**
* @brief Waits until all queued tasks have been processed.
*
* @warning This function uses busy waiting and may waste CPU time.
*/
void ThreadPool::wait() {
    while (!this->q.empty()) {
        // Busy wait
    }
}
```

Index