

ThreadPool

Generated by Doxygen 1.9.4

1 Data Structure Index	1
1.1 Data Structures	1
2 File Index	3
2.1 File List	3
3 Data Structure Documentation	5
3.1 ThreadPool Class Reference	5
3.1.1 Detailed Description	6
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 ThreadPool() [1/2]	6
3.1.2.2 ThreadPool() [2/2]	6
3.1.2.3 ~ThreadPool()	7
3.1.3 Member Function Documentation	8
3.1.3.1 finish()	8
3.1.3.2 length()	9
3.1.3.3 status()	9
3.1.3.4 submit()	10
3.1.3.5 wait()	11
3.1.4 Field Documentation	11
3.1.4.1 cv	11
3.1.4.2 cv2	11
3.1.4.3 error	11
3.1.4.4 exit_flag	12
3.1.4.5 mtx	12
3.1.4.6 num_threads	12
3.1.4.7 process_flag	12
3.1.4.8 q	13
3.1.4.9 state	13
3.1.4.10 th	13
4 File Documentation	15
4.1 example.cpp File Reference	15
4.1.1 Function Documentation	15
4.1.1.1 main()	15
4.2 example.cpp	16
4.3 test.hpp File Reference	17
4.3.1 Function Documentation	17
4.3.1.1 function1()	18
4.3.1.2 function2()	18
4.3.1.3 test1()	18
4.3.1.4 test2()	19
4.3.1.5 test3()	19

4.3.1.6 test4() [1/2]	19
4.3.1.7 test4() [2/2]	19
4.4 test.hpp	20
4.5 ThreadPool.hpp File Reference	20
4.5.1 Detailed Description	21
4.6 ThreadPool.hpp	21
4.7 test.cpp File Reference	24
4.7.1 Function Documentation	24
4.7.1.1 function1()	24
4.7.1.2 function2()	25
4.7.1.3 test1()	25
4.7.1.4 test2()	25
4.7.1.5 test4()	26
4.8 test.cpp	26
4.9 ThreadPool.cpp File Reference	26
4.9.1 Detailed Description	27
4.10 ThreadPool.cpp	27
Index	31

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

ThreadPool	Manages a pool of worker threads to execute tasks concurrently	5
----------------------------	--	-------------------

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

example.cpp	15
test.hpp	17
ThreadPool.hpp	
Declaration of a simple ThreadPool class	20
test.cpp	24
ThreadPool.cpp	
Implementation of the ThreadPool class	26

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.
- `unsigned int length ()`
Returns the number of process in queue.
- `void wait ()`
Waits until the task queue becomes empty.
- `~ThreadPool ()`
Destructor.

Data Fields

- `atomic< bool > error =false`

Private Attributes

- thread * [th](#)
- mutex [mtx](#)
- queue< function< void()> > [q](#)
- unsigned int [num_threads](#)
- atomic< bool > [exit_flag](#)
- atomic< bool > * [process_flag](#)
- atomic< bool > [state](#)
- condition_variable [cv](#)
- condition_variable [cv2](#)

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.

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

<i>num_threads</i>	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(unsigned int)> f = [&](unsigned int process_number) {
00025
00026         function<void()> task;
00027
00028         unique_lock<mutex> lock(this->mtx);
00029
00030         while (!this->exit_flag.load() && this->state.load()) {
00031
00032             cv.wait(lock, [&]() {
00033                 return !this->q.empty() || (this->exit_flag.load() || !this->state.load());
00034             });
00035
00036             if (this->exit_flag.load() || !this->state.load()) {
00037                 lock.unlock();
00038                 break;
00039             }
00040
00041             task = move(this->q.front());
00042             this->q.pop();
00043
00044             lock.unlock();
00045
00046             this->process_flag[process_number]=true;
00047             try{
00048                 task();
00049             }catch(...){
00050                 this->error=true;
00051             }
00052             this->process_flag[process_number]=false;
00053             this->cv2.notify_one();
00054             lock.lock();
00055         }
00056     };
00057
00058     this->exit_flag = false;
00059
00060     this->th = new (nothrow) thread[num_threads];
00061     this->process_flag = new (nothrow) atomic<bool>[num_threads];
00062
00063     if (this->th != nullptr && this->process_flag != nullptr) {
00064         this->state = true;
00065
00066         try {
00067             for (unsigned int i = 0; i < this->num_threads; i++) {
00068                 this->th[i] = thread(f,i);
00069                 this->process_flag[i]=false;
00070             }
00071         } catch (exception&) {
00072             this->state = false;
00073             // cout << "The system is unable to start a new thread" << endl;
00074         }
00075     } else {
00076         this->state = false;
00077     }
00078 }
00079
00080 }
00081
00082 }
00083 }
```

3.1.2.3 ~ThreadPool()

```
ThreadPool::~ThreadPool ( )
```

Destructor.

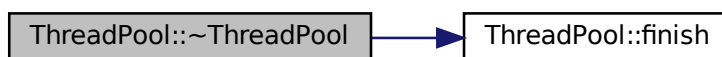
Stops the pool and releases all allocated resources.

Ensures that the pool is stopped and memory is released.

Definition at line 138 of file [ThreadPool.cpp](#).

```
00138         {
00139     this->finish();
00140     try{
00141         delete[] this->th;
00142         delete[] this->process_flag;
00143     }catch(...){
00144     }
00145 }
00146 }
```

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

<i>secure</i>	If true, joins all worker threads.
<i>secure</i>	If true, waits for all threads to finish execution.

Definition at line 113 of file [ThreadPool.cpp](#).

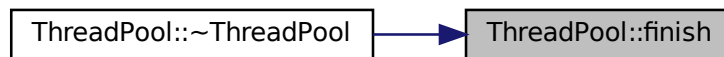
```
00113     {
00114     this->exit_flag = true;
00115
00116     this->cv.notify_all();
00117
00118     if (secure && this->state.load()) {
00119         for (unsigned int i = 0; i < this->num_threads; i++) {
00120             if (this->th[i].joinable()) {
```

```

00121         try {
00122             this->th[i].join();
00123         } catch (...) {
00124             // cout << "Error joining thread " << i << endl;
00125         }
00126     }
00127 }
00128 }
00129     this->state = false;
00130 }
00131 }

```

Here is the caller graph for this function:



3.1.3.2 length()

```
unsigned int ThreadPool::length ( )
```

Returns the number of process in queue.

Provides the number of process in queue.

Returns

integer with the length of the queue.

Definition at line 102 of file [ThreadPool.cpp](#).

```

00102     {
00103         return this->q.size();
00104     }

```

3.1.3.3 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 90 of file [ThreadPool.cpp](#).

```

00090     {
00091         return this->state;
00092     }

```

3.1.3.4 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

<i>Func</i>	Callable object type.
<i>Args</i>	Argument types for the callable.

Parameters

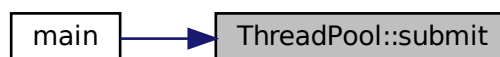
<i>f</i>	Function to execute.
<i>args</i>	Arguments passed to the function.

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

Definition at line 150 of file [ThreadPool.hpp](#).

```
00150                                     {
00151
00152
00153     {
00154         lock_guard<mutex> lock(this->mtx);
00155
00156         this->q.push(
00157             move(
00158                 [&]() {
00159                     f(args...);
00160                 }
00161             )
00162         );
00163     };
00164 }
00165
00166 this->cv.notify_one();
00167 }
```

Here is the caller graph for this function:



3.1.3.5 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.

Definition at line 151 of file [ThreadPool.cpp](#).

```
00151         {
00152             unique_lock<mutex> lock(this->mtx);
00153
00154             this->cv2.wait(lock, [&]() {
00155
00156                 bool flag=true;
00157
00158                 if(this->q.empty()) {
00159
00160                     for(unsigned int i=0;i<this->num_threads;i++){
00161                         if(this->process_flag[i].load()){
00162                             flag=false;
00163                             break;
00164                         }
00165                     }
00166
00167                     return flag?true:false;
00168                 }else{
00169                     return false;
00170                 }
00171             }
00172         );
00173
00174     }
```

3.1.4 Field Documentation

3.1.4.1 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](#).

3.1.4.2 cv2

```
condition_variable ThreadPool::cv2 [private]
```

Condition variable used to wait until all the queue finish

Definition at line 64 of file [ThreadPool.hpp](#).

3.1.4.3 error

```
atomic<bool> ThreadPool::error =false
```

Parameters

<i>Flag</i>	that indicates if any thread has thrown an exception
-------------	--

Definition at line 140 of file [ThreadPool.hpp](#).

3.1.4.4 exit_flag

```
atomic<bool> ThreadPool::exit_flag [private]
```

Flag indicating when threads should terminate

Definition at line 52 of file [ThreadPool.hpp](#).

3.1.4.5 mtx

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

Mutex protecting access to the task queue

Definition at line 43 of file [ThreadPool.hpp](#).

3.1.4.6 num_threads

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

Number of threads managed by the pool

Definition at line 49 of file [ThreadPool.hpp](#).

3.1.4.7 process_flag

```
atomic<bool>* ThreadPool::process_flag [private]
```

Array of flags indicating if a thread is running a process or not

Definition at line 55 of file [ThreadPool.hpp](#).

3.1.4.8 q

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

Queue containing pending tasks

Definition at line 46 of file [ThreadPool.hpp](#).

3.1.4.9 state

```
atomic<bool> ThreadPool::state [private]
```

Indicates whether the thread pool is active

Definition at line 58 of file [ThreadPool.hpp](#).

3.1.4.10 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:

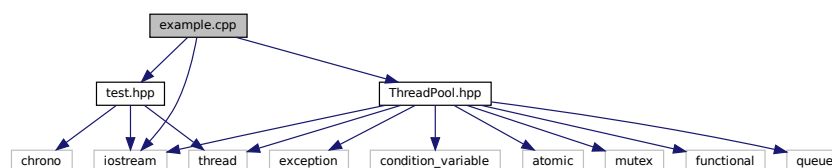
- [ThreadPool.hpp](#)
- [ThreadPool.cpp](#)

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](#).

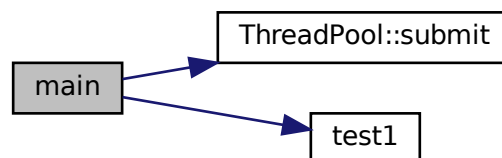
```
00009
00010
00011
00012     ThreadPool thp{4};
```

```

00013
00014     for(int i=0;i<20;i++){
00015         thp.submit(test1,i);
00016         this_thread::sleep_for(20ms);
00017     }
00018
00019     // cout<<"En proceso"<<endl;
00020
00021     thp.wait();
00022
00023     if(thp.status()){
00024         thp.finish();
00025     }
00026
00027     if(!thp.status()){
00028         cout<<"Cerrado correctamente"<<endl;
00029     }
00030
00031     return 0;
00032 }

```

Here is the call graph for this function:



4.2 example.cpp

[Go to the documentation of this file.](#)

```

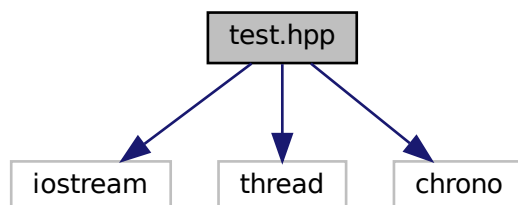
00001 #include <iostream>
00002 #include "ThreadPool.hpp"
00003 #include "test.hpp"
00004
00005
00006 using namespace std;
00007
00008
00009 int main(int argc, char * argv[]){
00010
00011     ThreadPool thp{4};
00012
00013     for(int i=0;i<20;i++){
00014         thp.submit(test1,i);
00015         this_thread::sleep_for(20ms);
00016     }
00017
00018     // cout<<"En proceso"<<endl;
00019
00020     thp.wait();
00021
00022     if(thp.status()){
00023         thp.finish();
00024     }
00025
00026     if(!thp.status()){
00027         cout<<"Cerrado correctamente"<<endl;
00028     }
00029
00030     return 0;
00031 }
00032 }

```

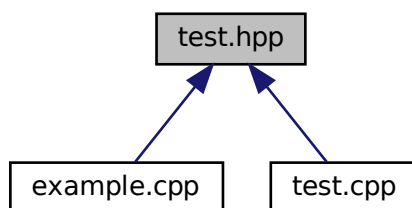
4.3 test.hpp File Reference

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

Include dependency graph for test.hpp:



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



Functions

- void [test1](#) (int)
- 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(*function)(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 28 of file [test.cpp](#).

```
00028         {
00029     cout<<x+y<<x<<endl;
00030
00031 }
```

4.3.1.2 function2()

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

Definition at line 34 of file [test.cpp](#).

```
00034         {
00035
00036     cout<<"function2 " <<endl;
00037     funcion(2,3,4);
00038     funcion(args...);
00039 }
```

4.3.1.3 test1()

```
void test1 (
    int x )
```

Definition at line 7 of file [test.cpp](#).

```
00007     {
00008     // this_thread::sleep_for(1000ms);
00009     cout<<this_thread::get_id()<<" Proceso dentro de test1 "<<x<<endl;
00010 }
```

Here is the caller graph for this function:



4.3.1.4 test2()

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

Definition at line 13 of file [test.cpp](#).

```
00013     {
00014     cout<<this_thread::get_id()<<" Proceso dentro de test2 con suma "<<x+y<<endl;
00015 }
```

4.3.1.5 test3()

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

Definition at line 14 of file [test.hpp](#).

```
00014     {
00015     cout<<this_thread::get_id()<<" Proceso dentro de test3 con suma "<<x+y<<endl;
00016 }
```

4.3.1.6 test4() [1/2]

```
void test4 ( )
```

Definition at line 23 of file [test.cpp](#).

```
00023     {
00024     cout<<this_thread::get_id()<<" Proceso dentro de test4 sin argumentos"<<endl;
00025 }
```

4.3.1.7 test4() [2/2]

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

Definition at line 19 of file [test.hpp](#).

```
00019     {
00020     cout<<this_thread::get_id()<<" Proceso dentro de test4 con argumento "<<x<<endl;
00021 }
```

4.4 test.hpp

[Go to the documentation of this file.](#)

```

00001 #ifndef TEST_HPP
00002 #define TEST_HPP
00003
00004 #include <iostream>
00005 #include <thread>
00006 #include <chrono>
00007
00008 using namespace std;
00009
00010 void test1(int);
00011 void test2(int,int);
00012
00013 template <class T>
00014 void test3(T x,T y){
00015     cout<<this_thread::get_id()<<" Proceso dentro de test3 con suma "<<x+y<<endl;
00016 }
00017
00018 template <class... Args>
00019 void test4(int x,Args... args){
00020     cout<<this_thread::get_id()<<" Proceso dentro de test4 con argumento "<<x<<endl;
00021 }
00022
00023 void test4();
00024
00025
00026 void function1(int x,int y,int z);
00027
00028 template <class... Args>
00029 void function2(void (*funcion)(int,int,int),Args... args);
00030
00031
00032 #endif

```

4.5 ThreadPool.hpp File Reference

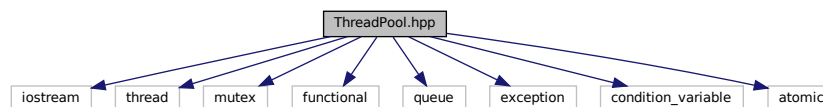
Declaration of a simple [ThreadPool](#) class.

```

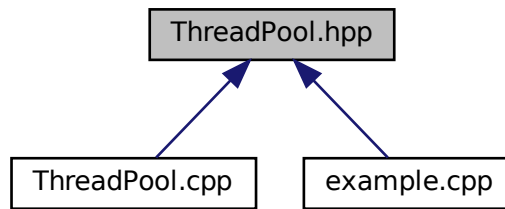
#include <iostream>
#include <thread>
#include <mutex>
#include <functional>
#include <queue>
#include <exception>
#include <condition_variable>
#include <atomic>

```

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-ascii

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.](#)

```

00001
00002 /**
00003  * @file ThreadPool.hpp
00004  * @author qwert-ascii
00005  * @brief Declaration of a simple ThreadPool class.
00006  *
00007  * This file contains the declaration of a basic thread pool
00008  * that executes tasks using a fixed number of worker threads.
00009  */
00010
00011 #ifndef THREADPOOL_HPP
00012 #define THREADPOOL_HPP
00013
00014 #include <iostream>
00015 #include <thread>
00016 #include <mutex>
00017 #include <functional>
00018 #include <queue>
00019 #include <exception>
00020 #include <condition_variable>
00021 #include <atomic>
00022

```

```

00023
00024 using namespace std;
00025
00026 /**
00027  * @class ThreadPool
00028  * @brief Manages a pool of worker threads to execute tasks concurrently.
00029  *
00030  * The ThreadPool maintains a queue of tasks. Worker threads continuously
00031  * fetch and execute tasks from this queue until the pool is stopped.
00032  *
00033  * @note Tasks have no return value.
00034  */
00035 class ThreadPool {
00036
00037 private:
00038
00039 /** Pointer to the array of worker threads */
00040 thread* th;
00041
00042 /** Mutex protecting access to the task queue */
00043 mutex mtx;
00044
00045 /** Queue containing pending tasks */
00046 queue<function<void()>> q;
00047
00048 /** Number of threads managed by the pool */
00049 unsigned int num_threads;
00050
00051 /** Flag indicating when threads should terminate */
00052 atomic<bool> exit_flag;
00053
00054 /** Array of flags indicating if a thread is running a process or not */
00055 atomic<bool> * process_flag;
00056
00057 /** Indicates whether the thread pool is active */
00058 atomic<bool> state;
00059
00060 /** Condition variable used to stop the threads until a new process is pushed */
00061 condition_variable cv;
00062
00063 /** Condition variable used to wait until all the queue finish */
00064 condition_variable cv2;
00065
00066 public:
00067
00068 /**
00069  * @brief Deleted default constructor.
00070  *
00071  * Forces the user to specify the number of threads.
00072  */
00073 ThreadPool() = delete;
00074
00075 /**
00076  * @brief Constructs a ThreadPool with a fixed number of threads.
00077  *
00078  * @param num_threads Number of worker threads to create.
00079  */
00080 explicit ThreadPool(unsigned int num_threads);
00081
00082 /**
00083  * @brief Submits a task to the thread pool.
00084  *
00085  * The task is stored in the internal queue and will be executed
00086  * by one of the worker threads.
00087  *
00088  * @tparam Func Callable object type.
00089  * @tparam Args Argument types for the callable.
00090  * @param f Function to execute.
00091  * @param args Arguments passed to the function.
00092  */
00093 template <class Func, class... Args>
00094 void submit(Func&& f, Args&&... args);
00095
00096
00097 /**
00098  * @brief Stops the thread pool.
00099  *
00100  * Sets the exit flag and optionally waits for all threads to finish.
00101  *
00102  * @param secure If true, joins all worker threads.
00103  */
00104 void finish(bool secure = true);
00105
00106 /**
00107  * @brief Returns the current state of the thread pool.
00108  *
00109  * @return True if the pool is running, false otherwise.

```

```

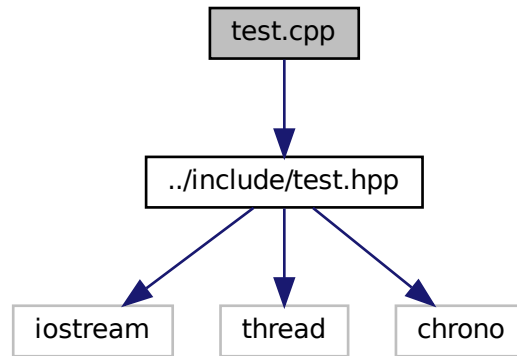
00110 */
00111     bool status();
00112
00113
00114 /**
00115  * @brief Returns the number of process in queue.
00116  *
00117  * @return integer with the length of the queue.
00118  */
00119     unsigned int length();
00120
00121 /**
00122  * @brief Waits until the task queue becomes empty.
00123  *
00124  * @note This function performs a busy wait.
00125  */
00126     void wait();
00127
00128 /**
00129  * @brief Destructor.
00130  *
00131  * Stops the pool and releases all allocated resources.
00132  */
00133     ~ThreadPool();
00134
00135
00136 /**
00137  * @param Flag that indicates if any thread has thrown an exception
00138  *
00139  */
00140     atomic<bool> error=false;
00141 };
00142
00143 /**
00144  * @brief Adds a new task to the task queue.
00145  *
00146  * Locks the mutex, pushes the task into the queue,
00147  * and then unlocks the mutex.
00148  */
00149 template <class Func, class... Args>
00150 void ThreadPool::submit(Func&& f, Args&&... args) {
00151
00152
00153     {
00154         lock_guard<mutex> lock(this->mtx);
00155
00156         this->q.push(
00157             move(
00158                 [&]() {
00159                     f(args...);
00160                 }
00161             )
00162         );
00163     };
00164 }
00165
00166     this->cv.notify_one();
00167 }
00168
00169 #endif

```

4.7 test.cpp File Reference

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

Include dependency graph for test.cpp:



Functions

- void [test1](#) (int x)
- void [test2](#) (int x, int y)
- void [test4](#) ()
- void [function1](#) (int x, int y, int z)
- template<class... Args>
void [function2](#) (void(*function)(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 28 of file [test.cpp](#).

```
00028      {  
00029          cout<<x+y+x<<endl;  
00030      }  
00031 }
```

4.7.1.2 function2()

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

Definition at line 34 of file [test.cpp](#).

```
00034                                     {
00035
00036     cout<<"function2 " <<endl;
00037     funcion(2,3,4);
00038     funcion(args...);
00039 }
```

4.7.1.3 test1()

```
void test1 (
    int x )
```

Definition at line 7 of file [test.cpp](#).

```
00007     {
00008     // this_thread::sleep_for(1000ms);
00009     cout<<this_thread::get_id()<<" Proceso dentro de test1 "<<x<<endl;
00010 }
```

Here is the caller graph for this function:



4.7.1.4 test2()

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

Definition at line 13 of file [test.cpp](#).

```
00013     {
00014     cout<<this_thread::get_id()<<" Proceso dentro de test2 con suma "<<x+y<<endl;
00015 }
```

4.7.1.5 test4()

```
void test4 ( )
```

Definition at line 23 of file [test.cpp](#).

```
00023 {
00024     cout<<this_thread::get_id()<<" Proceso dentro de test4 sin argumentos"<<endl;
00025 }
```

4.8 test.cpp

[Go to the documentation of this file.](#)

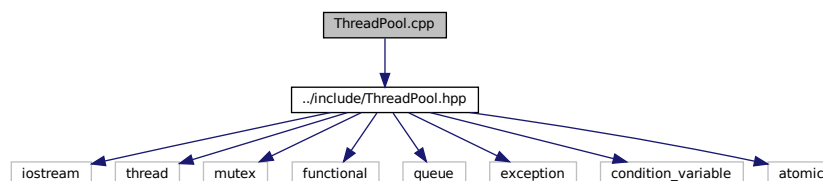
```
00001 #include "../include/test.hpp"
00002
00003
00004
00005 using namespace std;
00006
00007 void test1(int x){
00008     // this_thread::sleep_for(1000ms);
00009     cout<<this_thread::get_id()<<" Proceso dentro de test1 "<<x<<endl;
00010 }
00011
00012
00013 void test2(int x,int y){
00014     cout<<this_thread::get_id()<<" Proceso dentro de test2 con suma "<<x+y<<endl;
00015 }
00016
00017
00018
00019
00020
00021
00022
00023 void test4(){
00024     cout<<this_thread::get_id()<<" Proceso dentro de test4 sin argumentos"<<endl;
00025 }
00026
00027
00028 void function1(int x,int y,int z){
00029     cout<<x+y+x<<endl;
00030
00031 }
00032
00033 template <class... Args>
00034 void function2(void (*funcion)(int,int,int),Args... args){
00035
00036     cout<<"function2 "<<endl;
00037     funcion(2,3,4);
00038     funcion(args...);
00039 }
```

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.](#)

```

00001 /**
00002  * @file ThreadPool.cpp
00003  * @brief Implementation of the ThreadPool class.
00004  */
00005
00006 #include "../include/ThreadPool.hpp"
00007
00008 /**
00009  * @brief Constructs and starts the worker threads.
00010  *
00011  * Each worker thread runs a loop that retrieves tasks
00012  * from the queue and executes them until termination
00013  * is requested.
00014  */
00015 ThreadPool::ThreadPool(unsigned int num_threads)
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(unsigned int)> f = [&](unsigned int process_number) {
00025
00026         function<void()> task;
00027
00028         unique_lock<mutex> lock(this->mtx);
00029
00030         while (!this->exit_flag.load() && this->state.load()) {
00031
00032             cv.wait(lock, [&]() {
00033                 return !this->q.empty() || (this->exit_flag.load() || !this->state.load());
00034             });
00035
00036             task = move(this->q.front());
00037             this->q.pop();
00038
00039             lock.unlock();
00040
00041             if (this->exit_flag.load() || !this->state.load()) {
00042                 break;
00043             }
00044
00045             task = move(this->q.front());
00046             this->q.pop();
00047
00048             lock.unlock();
00049
00050             this->process_flag[process_number]=true;
00051             try{
00052                 task();
00053             }catch(...){
00054                 this->error=true;
00055             }
00056             this->process_flag[process_number]=false;
00057             this->cv2.notify_one();
00058             lock.lock();
00059         }
00060     };
00061
00062     this->exit_flag = false;
00063
00064     this->th = new (nothrow) thread[num_threads];
00065     this->process_flag = new (nothrow) atomic<bool>[num_threads];
00066
00067     if (this->th != nullptr && this->process_flag != nullptr) {
00068         this->state = true;
00069     }
00070 }

```

```

00070
00071     try {
00072         for (unsigned int i = 0; i < this->num_threads; i++) {
00073             this->th[i] = thread(f,i);
00074             this->process_flag[i]=false;
00075         }
00076     } catch (exception&) {
00077         this->state = false;
00078         // cout << "The system is unable to start a new thread" << endl;
00079     }
00080 } else {
00081     this->state = false;
00082 }
00083 }
00084
00085 /**
00086 * @brief Checks whether the thread pool is running.
00087 *
00088 * @return True if the pool is active, false otherwise.
00089 */
00090 bool ThreadPool::status() {
00091     return this->state;
00092 }
00093
00094
00095
00096
00097 /**
00098 * @brief Provides the number of process in queue.
00099 *
00100 * @return integer with the length of the queue.
00101 */
00102 unsigned int ThreadPool::length() {
00103     return this->q.size();
00104 }
00105
00106
00107
00108 /**
00109 * @brief Stops the thread pool and optionally joins threads.
00110 *
00111 * @param secure If true, waits for all threads to finish execution.
00112 */
00113 void ThreadPool::finish(bool secure) {
00114     this->exit_flag = true;
00115
00116     this->cv.notify_all();
00117
00118     if (secure && this->state.load()) {
00119         for (unsigned int i = 0; i < this->num_threads; i++) {
00120             if (this->th[i].joinable()) {
00121                 try {
00122                     this->th[i].join();
00123                 } catch (...) {
00124                     // cout << "Error joining thread " << i << endl;
00125                 }
00126             }
00127         }
00128     }
00129     this->state = false;
00130 }
00131 }
00132
00133 /**
00134 * @brief Destructor.
00135 *
00136 * Ensures that the pool is stopped and memory is released.
00137 */
00138 ThreadPool::~ThreadPool() {
00139     this->finish();
00140     try{
00141         delete[] this->th;
00142         delete[] this->process_flag;
00143     }catch(...){
00144     }
00145 }
00146 }
00147
00148 /**
00149 * @brief Waits until all queued tasks have been processed.
00150 */
00151 void ThreadPool::wait() {
00152     unique_lock<mutex> lock(this->mtx);
00153
00154     this->cv2.wait(lock, [&]() {
00155
00156         bool flag=true;

```



```
00157
00158         if(this->q.empty()){
00159
00160             for(unsigned int i=0;i<this->num_threads;i++){
00161                 if(this->process_flag[i].load()){
00162                     flag=false;
00163                     break;
00164                 }
00165             }
00166
00167             return flag?true:false;
00168         }else{
00169             return false;
00170         }
00171     }
00172 };
00173
00174 }
```


Index

- [~ThreadPool](#)
 - [ThreadPool](#), [7](#)
- [cv](#)
 - [ThreadPool](#), [11](#)
- [cv2](#)
 - [ThreadPool](#), [11](#)
- [error](#)
 - [ThreadPool](#), [11](#)
- [example.cpp](#), [15](#), [16](#)
 - [main](#), [15](#)
- [exit_flag](#)
 - [ThreadPool](#), [12](#)
- [finish](#)
 - [ThreadPool](#), [8](#)
- [function1](#)
 - [test.cpp](#), [24](#)
 - [test.hpp](#), [17](#)
- [function2](#)
 - [test.cpp](#), [24](#)
 - [test.hpp](#), [18](#)
- [length](#)
 - [ThreadPool](#), [9](#)
- [main](#)
 - [example.cpp](#), [15](#)
- [mtx](#)
 - [ThreadPool](#), [12](#)
- [num_threads](#)
 - [ThreadPool](#), [12](#)
- [process_flag](#)
 - [ThreadPool](#), [12](#)
- [q](#)
 - [ThreadPool](#), [12](#)
- [state](#)
 - [ThreadPool](#), [13](#)
- [status](#)
 - [ThreadPool](#), [9](#)
- [submit](#)
 - [ThreadPool](#), [9](#)
- [test.cpp](#), [24](#), [26](#)
 - [function1](#), [24](#)
 - [function2](#), [24](#)
- [test1](#), [25](#)
- [test2](#), [25](#)
- [test4](#), [25](#)
- [test.hpp](#), [17](#), [20](#)
 - [function1](#), [17](#)
 - [function2](#), [18](#)
 - [test1](#), [18](#)
 - [test2](#), [18](#)
 - [test3](#), [19](#)
 - [test4](#), [19](#)
- [test1](#)
 - [test.cpp](#), [25](#)
 - [test.hpp](#), [18](#)
- [test2](#)
 - [test.cpp](#), [25](#)
 - [test.hpp](#), [18](#)
- [test3](#)
 - [test.hpp](#), [19](#)
- [test4](#)
 - [test.cpp](#), [25](#)
 - [test.hpp](#), [19](#)
- [th](#)
 - [ThreadPool](#), [13](#)
- [ThreadPool](#), [5](#)
 - [~ThreadPool](#), [7](#)
 - [cv](#), [11](#)
 - [cv2](#), [11](#)
 - [error](#), [11](#)
 - [exit_flag](#), [12](#)
 - [finish](#), [8](#)
 - [length](#), [9](#)
 - [mtx](#), [12](#)
 - [num_threads](#), [12](#)
 - [process_flag](#), [12](#)
 - [q](#), [12](#)
 - [state](#), [13](#)
 - [status](#), [9](#)
 - [submit](#), [9](#)
 - [th](#), [13](#)
 - [ThreadPool](#), [6](#)
 - [wait](#), [10](#)
- [ThreadPool.cpp](#), [26](#), [27](#)
- [ThreadPool.hpp](#), [20](#), [21](#)
- [wait](#)
 - [ThreadPool](#), [10](#)