

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 status() | 8 |
| 3.1.3.3 submit() | 9 |
| 3.1.3.4 wait() | 10 |
| 3.1.4 Field Documentation | 10 |
| 3.1.4.1 cont | 10 |
| 3.1.4.2 exit_flag | 11 |
| 3.1.4.3 mtx | 11 |
| 3.1.4.4 num_threads | 11 |
| 3.1.4.5 q | 11 |
| 3.1.4.6 state | 11 |
| 3.1.4.7 th | 11 |
| 4 File Documentation | 13 |
| 4.1 main.cpp File Reference | 13 |
| 4.1.1 Function Documentation | 13 |
| 4.1.1.1 main() | 14 |
| 4.2 main.cpp | 14 |
| 4.3 test.cpp File Reference | 15 |
| 4.3.1 Function Documentation | 15 |
| 4.3.1.1 function1() | 16 |
| 4.3.1.2 function2() | 16 |
| 4.3.1.3 test1() | 16 |
| 4.3.1.4 test2() | 17 |
| 4.3.1.5 test4() | 17 |
| 4.4 test.cpp | 17 |
| 4.5 test.hpp File Reference | 18 |
| 4.5.1 Function Documentation | 18 |
| 4.5.1.1 function1() | 19 |

| | |
|-----------------------------------|----|
| 4.5.1.2 function2() | 19 |
| 4.5.1.3 test1() | 19 |
| 4.5.1.4 test2() | 20 |
| 4.5.1.5 test3() | 20 |
| 4.5.1.6 test4() [1/2] | 20 |
| 4.5.1.7 test4() [2/2] | 20 |
| 4.6 test.hpp | 21 |
| 4.7 ThreadPool.cpp File Reference | 21 |
| 4.7.1 Detailed Description | 21 |
| 4.8 ThreadPool.cpp | 22 |
| 4.9 ThreadPool.hpp File Reference | 23 |
| 4.9.1 Detailed Description | 24 |
| 4.10 ThreadPool.hpp | 24 |

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:

| | | |
|----------------|--|----|
| main.cpp | | 13 |
| test.cpp | | 15 |
| test.hpp | | 18 |
| ThreadPool.cpp | Implementation of the ThreadPool class | 21 |
| ThreadPool.hpp | Declaration of a simple ThreadPool class | 23 |

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`

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 33 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

| | |
|--------------------------|-------------------------------------|
| <code>num_threads</code> | 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         while (!this->exit_flag && this->state) {
00029             mtx.lock();
00030
00031             if (!this->q.empty()) {
00032                 task = this->q.front();
00033                 q.pop();
00034
00035                 mtx.unlock();
00036                 task();
00037             } else {
00038                 mtx.unlock();
00039             }
00040         }
00041     };
00042
00043     exit_flag = false;
00044
00045     this->th = new (nothrow) thread[num_threads];
00046
00047     if (this->th != nullptr) {
00048         this->state = true;
00049
00050         try {
00051             for (unsigned int i = 0; i < this->num_threads; i++) {
00052                 this->th[i] = thread(f);
00053             }
00054         } catch (exception&) {
00055             this->state = false;
00056             cout << "The system is unable to start a new thread" << endl;
00057         }
00058     } else {
00059         this->state = false;
00060     }
00061 }
```

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 99 of file [ThreadPool.cpp](#).

```

00099
00100     this->finish();
00101     delete[] this->th;
00102 }
```

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.

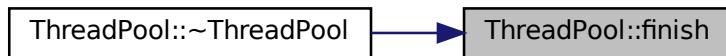
Parameters

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

Definition at line 77 of file [ThreadPool.cpp](#).

```
00077     this->exit_flag = true;
00079
00080     if (secure && this->state) {
00081         for (unsigned int i = 0; i < this->num_threads; i++) {
00082             if (this->th[i].joinable()) {
00083                 try {
00084                     this->th[i].join();
00085                 } catch (...) {
00086                     cout << "Error joining thread " << i << endl;
00087                 }
00088             }
00089         }
00090         this->state = false;
00091     }
00092 }
```

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 68 of file [ThreadPool.cpp](#).

```
00068     {
00069     return this->state;
00070 }
```

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

| | |
|-------------|----------------------------------|
| <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. |

Warning

This method does not notify worker threads.

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

Definition at line 128 of file [ThreadPool.hpp](#).

```
00128
00129     mtx.lock();
00130     this->q.push(
00131         [&]() {
00132             f(args...);
00133         }
00134     );
00135
00136     mtx.unlock();
00137
00138 }
```

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 109 of file [ThreadPool.cpp](#).

```
00109     {
00110     while (!this->q.empty()) {
00111         // Busy wait
00112     }
00113 }
```

3.1.4 Field Documentation

3.1.4.1 cont

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

Auxiliary counter (currently unused)

Definition at line 56 of file [ThreadPool.hpp](#).

3.1.4.2 `exit_flag`

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

Flag indicating when threads should terminate

Definition at line 50 of file [ThreadPool.hpp](#).

3.1.4.3 `mtx`

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

Mutex protecting access to the task queue

Definition at line 41 of file [ThreadPool.hpp](#).

3.1.4.4 `num_threads`

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

Number of threads managed by the pool

Definition at line 47 of file [ThreadPool.hpp](#).

3.1.4.5 `q`

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

Queue containing pending tasks

Definition at line 44 of file [ThreadPool.hpp](#).

3.1.4.6 `state`

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

Indicates whether the thread pool is active

Definition at line 53 of file [ThreadPool.hpp](#).

3.1.4.7 `th`

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

Pointer to the array of worker threads

Definition at line 38 of file [ThreadPool.hpp](#).

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

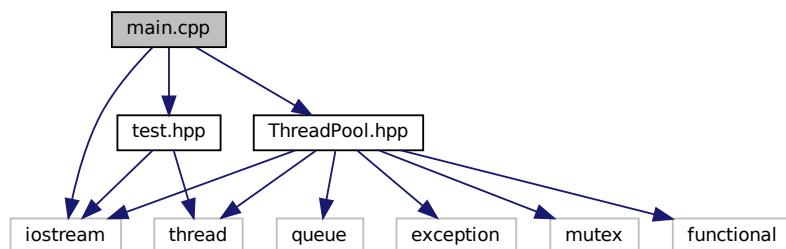
- [ThreadPool.hpp](#)
- [ThreadPool.cpp](#)

Chapter 4

File Documentation

4.1 main.cpp File Reference

```
#include <iostream>
#include "ThreadPool.hpp"
#include "test.hpp"
Include dependency graph for main.cpp:
```



Functions

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

4.1.1 Function Documentation

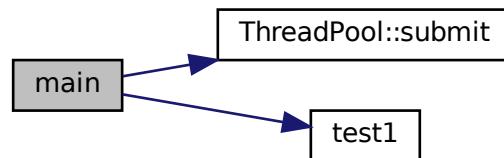
4.1.1.1 main()

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

Definition at line 9 of file [main.cpp](#).

```
00009
00010
00011
00012     ThreadPool thp{10};
00013
00014     cout<<sizeof(mutex)<<endl;
00015
00016     for(int i=0;i<20;i++){
00017         thp.submit(test1);
00018     }
00019
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 main.cpp

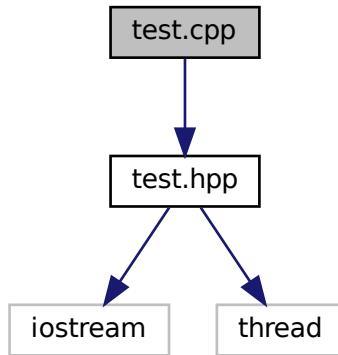
[Go to the documentation of this file.](#)

```
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     cout<<sizeof(mutex)<<endl;
00015
00016     for(int i=0;i<20;i++){
00017         thp.submit(test1);
00018     }
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.cpp File Reference

```
#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)

4.3.1 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+z<<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 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     }
```

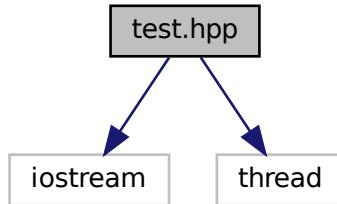
4.4 test.cpp

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

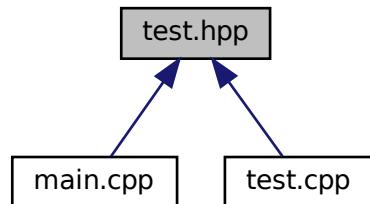
```
00001 #include "test.hpp"
00002
00003
00004
00005 using namespace std;
00006
00007 void test1(){
00008     cout<<this_thread::get_id()<<" Proceso dentro de test1"<<endl;
00009 }
00010
00011
00012 void test2(int x,int y){
00013     cout<<this_thread::get_id()<<" Proceso dentro de test2 con suma "<<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+z<<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 }
```

4.5 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)`

4.5.1 Function Documentation

4.5.1.1 function1()

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

Definition at line 27 of file [test.cpp](#).

```
00027
00028     cout<<x+y+z<<endl;
00029
00030 }
```

4.5.1.2 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 }
```

4.5.1.3 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:



4.5.1.4 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     }
```

4.5.1.5 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     }
```

4.5.1.6 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     }
```

4.5.1.7 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     }
```

4.6 test.hpp

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

```

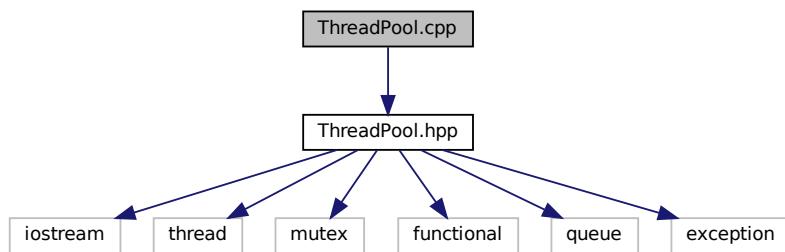
00001 #ifndef TEST_HPP
00002 #define TEST_HPP
00003
00004 #include <iostream>
00005 #include <thread>
00006
00007 using namespace std;
00008
00009 void test1();
00010 void test2(int,int);
00011
00012 template <class T>
00013 void test3(T x,T y){
00014     cout<<this_thread::get_id()<<" Proceso dentro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

```

4.7 ThreadPool.cpp File Reference

Implementation of the [ThreadPool](#) class.

```
#include "ThreadPool.hpp"
Include dependency graph for ThreadPool.cpp:
```



4.7.1 Detailed Description

Implementation of the [ThreadPool](#) class.

Definition in file [ThreadPool.cpp](#).

4.8 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 "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(void)> f = [&]() {
00025
00026         function<void()> task;
00027
00028         while (!this->exit_flag && this->state) {
00029             mtx.lock();
00030
00031             if (!this->q.empty()) {
00032                 task = this->q.front();
00033                 q.pop();
00034
00035                 mtx.unlock();
00036                 task();
00037             } else {
00038                 mtx.unlock();
00039             }
00040         }
00041     };
00042
00043     exit_flag = false;
00044
00045     this->th = new (nothrow) thread[num_threads];
00046
00047     if (this->th != nullptr) {
00048         this->state = true;
00049
00050         try {
00051             for (unsigned int i = 0; i < this->num_threads; i++) {
00052                 this->th[i] = thread(f);
00053             }
00054         } catch (exception&) {
00055             this->state = false;
00056             cout << "The system is unable to start a new thread" << endl;
00057         }
00058     } else {
00059         this->state = false;
00060     }
00061 }
00062
00063 /**
00064 * @brief Checks whether the thread pool is running.
00065 *
00066 * @return True if the pool is active, false otherwise.
00067 */
00068 bool ThreadPool::status() {
00069     return this->state;
00070 }
00071
00072 /**
00073 * @brief Stops the thread pool and optionally joins threads.
00074 *
00075 * @param secure If true, waits for all threads to finish execution.
00076 */
00077 void ThreadPool::finish(bool secure) {
00078     this->exit_flag = true;
00079
00080     if (secure && this->state) {
00081         for (unsigned int i = 0; i < this->num_threads; i++) {
00082             if (this->th[i].joinable())

```

```

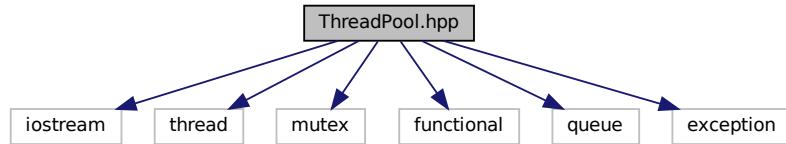
00083         try {
00084             this->th[i].join();
00085         } catch (...) {
00086             cout << "Error joining thread " << i << endl;
00087         }
00088     }
00089 }
00090     this->state = false;
00091 }
00092 }
00093
00094 /**
00095 * @brief Destructor.
00096 *
00097 * Ensures that the pool is stopped and memory is released.
00098 */
00099 ThreadPool::~ThreadPool() {
00100     this->finish();
00101     delete[] this->th;
00102 }
00103
00104 /**
00105 * @brief Waits until all queued tasks have been processed.
00106 *
00107 * @warning This function uses busy waiting and may waste CPU time.
00108 */
00109 void ThreadPool::wait() {
00110     while (!this->q.empty()) {
00111         // Busy wait
00112     }
00113 }

```

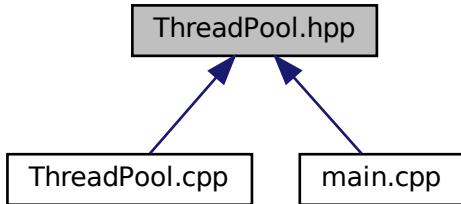
4.9 ThreadPool.hpp File Reference

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

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

4.9.1 Detailed Description

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

Author

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](#).

4.10 ThreadPool.hpp

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

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

```
00023 * @class ThreadPool
00024 * @brief Manages a pool of worker threads to execute tasks concurrently.
00025 *
00026 * The ThreadPool maintains a queue of tasks. Worker threads continuously
00027 * fetch and execute tasks from this queue until the pool is stopped.
00028 *
00029 * @note Tasks have no return value.
00030 * @warning This implementation uses busy waiting and does not use
00031 *           condition variables.
00032 */
00033 class ThreadPool {
00034
00035     private:
00036
00037     /** Pointer to the array of worker threads */
00038     thread* th;
00039
00040     /** Mutex protecting access to the task queue */
00041     mutex mtx;
00042
00043     /** Queue containing pending tasks */
00044     queue<function<void()>> q;
00045
00046     /** Number of threads managed by the pool */
00047     unsigned int num_threads;
00048
00049     /** Flag indicating when threads should terminate */
00050     bool exit_flag;
00051
00052     /** Indicates whether the thread pool is active */
00053     bool state;
00054
00055     /** Auxiliary counter (currently unused) */
00056     int cont = 0;
00057
00058     public:
00059
00060     /**
00061     * @brief Deleted default constructor.
00062     */
00063     * Forces the user to specify the number of threads.
00064     */
00065     ThreadPool() = delete;
00066
00067 /**
00068 * @brief Constructs a ThreadPool with a fixed number of threads.
00069 */
00070 * @param num_threads Number of worker threads to create.
00071 */
00072     explicit ThreadPool(unsigned int num_threads);
00073
00074 /**
00075 * @brief Submits a task to the thread pool.
00076 *
00077 * The task is stored in the internal queue and will be executed
00078 * by one of the worker threads.
00079 *
00080 * @tparam Func Callable object type.
00081 * @tparam Args Argument types for the callable.
00082 * @param f Function to execute.
00083 * @param args Arguments passed to the function.
00084 *
00085 * @warning This method does not notify worker threads.
00086 */
00087     template <class Func, class... Args>
00088     void submit(Func&& f, Args&&... args);
00089
00090 /**
00091 * @brief Stops the thread pool.
00092 *
00093 * Sets the exit flag and optionally waits for all threads to finish.
00094 *
00095 * @param secure If true, joins all worker threads.
00096 */
00097     void finish(bool secure = true);
00098
00099 /**
00100 * @brief Returns the current state of the thread pool.
00101 *
00102 * @return True if the pool is running, false otherwise.
00103 */
00104     bool status();
00105
00106 /**
00107 * @brief Waits until the task queue becomes empty.
00108 */
00109 * @note This function performs a busy wait.
```

```
00110 */
00111     void wait();
00112
00113 /**
00114 * @brief Destructor.
00115 *
00116 * Stops the pool and releases all allocated resources.
00117 */
00118     ~ThreadPool();
00119 };
00120
00121 /**
00122 * @brief Adds a new task to the task queue.
00123 *
00124 * Locks the mutex, pushes the task into the queue,
00125 * and then unlocks the mutex.
00126 */
00127 template <class Func, class... Args>
00128 void ThreadPool::submit(Func&& f, Args&&... args) {
00129     mtx.lock();
00130     this->q.push(
00131         [&]() {
00132             f(args...);
00133         }
00134     );
00135     mtx.unlock();
00136 }
00137
00138 #endif
```

Index

~ThreadPool
 ThreadPool, 7

cont
 ThreadPool, 10

exit_flag
 ThreadPool, 10

finish
 ThreadPool, 8

function1
 test.cpp, 15
 test.hpp, 18

function2
 test.cpp, 16
 test.hpp, 19

main
 main.cpp, 13

main.cpp, 13
 main, 13

mtx
 ThreadPool, 11

num_threads
 ThreadPool, 11

q
 ThreadPool, 11

state
 ThreadPool, 11

status
 ThreadPool, 8

submit
 ThreadPool, 9

test.cpp, 15
 function1, 15
 function2, 16
 test1, 16
 test2, 16
 test4, 17

test.hpp, 18
 function1, 18
 function2, 19
 test1, 19
 test2, 19
 test3, 20
 test4, 20

test1
 test.cpp, 16
 test.hpp, 19

test2
 test.cpp, 16
 test.hpp, 19

test3
 test.hpp, 20

test4
 test.cpp, 17
 test.hpp, 20

th
 ThreadPool, 11

ThreadPool, 5
 ~ThreadPool, 7
 cont, 10
 exit_flag, 10
 finish, 8
 mtx, 11
 num_threads, 11
 q, 11
 state, 11
 status, 8
 submit, 9
 th, 11
 ThreadPool, 6
 wait, 10

ThreadPool.cpp, 21

ThreadPool.hpp, 23

wait
 ThreadPool, 10