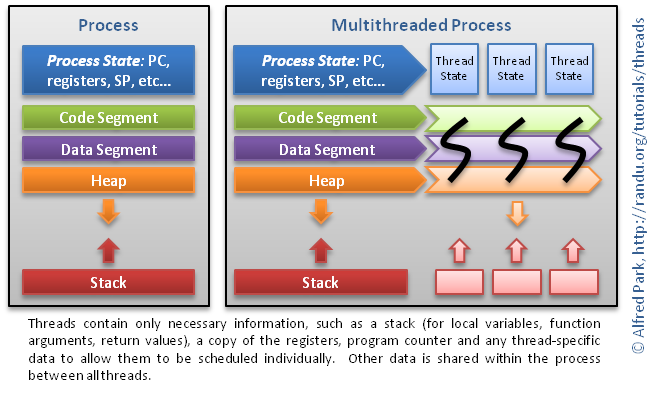
**What is Thread?**

Thread is a lightweight process. It is a part (or small sequence) of programming instructions.

Thread has their own stack space, registers, program control, signal mask and other thread specific data like thread ID, but they share common address space of code, data, heap segment, open files, signal and its handlers.



**Creation of Thread**

From C++11 we have builtin class for threads i.e class thread which is included in #include <thread>.

Before C++11:

#include<pthread.h>

pthread\_t t1;

pthread\_create(pthread\_t \* thread, pthread\_attr\_t \*attr, void \*(\*start\_routie)(void \*), void \* arg);

pthread\_t \*thread: actual thread object which contains thread ID.

pthread\_attr\_t \*attr: attributes apply to the thread.

void \*(\*start\_routie)(void \*) : the function this thread executes.

Void \* arg: arguments to pass to this functions

Pthread\_join(pthread\_t thread, void \*\* value\_ptr);

Void pthread\_exit(void \* value\_ptr);

pthread\_join(t1, NULL); 🡪 it makes main thread to wait for completion of invoked thread.

pthread\_detach(t1, NULL); 🡪 threads are running parallelly

either thread should be joined or detached.

From c++11

How to create a thread?

First, you have to include thread header in your program:

#include <thread>

When you want to create a thread, you have to create an object of a **thread** class.

//this thread does not represent any thread of execution

thread t\_empty;

As you can see, when default constructor of thread class is used, we do not pass any information to the thread. This means, that nothing is executed in this thread. We have to initialize a thread. It can be done in different ways.

Initializing thread with a function

When you create a thread, you can pass a pointer of a function to its constructor. Once thread is created, this function starts its work in a separate thread. Look on an example:

#include <iostream>

#include <thread>

**using** **namespace** std;

**void** threadFunc()

{

cout << "Welcome to Multithreading" << endl;

}

**int** main()

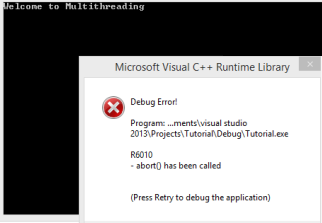
{

//pass a function to thread

thread funcTest1(threadFunc);

}

Try to compile and run this program. It compiles without any errors but you will get a runtime error:



As you can see, main thread creates new thread **funcTest1** with a parameter **threadFunc**. Main thread does not wait for **funcTest1** thread termination. It continues its work. The main thread finishes execution, but **funcTest1** is still running. This causes error. All the threads must be terminated before main thread is terminated.

Join threads

Thread joining is done by using **join()** member function of a thread class:

**void** join();

This function returns only after all the threads are terminated. It means that the main thread will wait until child thread does not finish its execution:

Call join() for the thread, created in the previous example and run the program again:

//pass a function to thread

thread funcTest1(threadFunc);

//main is blocked until funcTest1 is not finished

funcTest1.join();

As you can see, now program is executed successfully.

Joinable and not Joinable threads

After join() returns, thread becomes **not joinable.** A joinable thread is a thread that represents a thread of execution which has not yet been joined.

A thread is not joinable when it is default constructed or is moved/assigned to another thread or join() or detach() member function is called.

Not joinable thread can be destroyed safely.

You can check if a thread is joinable by using joinable() member function:

**bool** joinable()

This function returns true if the thread is joinable and false otherwise. It’s better to check if the thread is joinable before join() function is called:

//pass a function to thread

thread funcTest1(threadFunc);

//check if thread is joinable

**if** (funcTest1.joinable())

{

//main is blocked until funcTest1 is not finished

funcTest1.join();

}

Detaching thread

As we mentioned above, thread becomes not joinable after **detach()** member function is called:

**void** detach()

This function detaches a thread from the parent thread. It allows parent and child threads to be executed independently from each other. After the call of detach() function, the threads are not synchronized in any way:

//detach funcTest1 from main thread

funcTest1.detach();

**if** (funcTest1.joinable())

{

//main is blocked until funcTest1 is not finished

funcTest1.join();

}

**else**

{

cout << "functTest1 is detached" << endl;

}

You will notice that main thread is not waiting for the termination of its child thread.

Initializing thread with an object

You can initialize a thread not only with a function. You can use for this purpose function object (functor) or a member function of a class.

A functor is an object of a class that overloads operator **() -**function call operator.

If you want to initialize a thread with an object of a class, this class should overload operator(). It can be done in the following way:

**class** myFunctor

{

**public**:

**void** **operator**()()

{

cout << "This is my function object" << endl;

}

};

Now you can initialize a thread by passing an object of the class **myFunctor** to the constructor of a thread:

myFunctor myFunc;

thread functorTest(myFunc);

**if** (functorTest.joinable())

functorTest.join();

If you want to initialize a thread with a public member function of a class, you have to specify the identifier of this function and pass an object of the class, which defines this member function:

Add a public member function to **myFunctor** class:

**void** publicFunction()

{

cout << "public function of myFunctor class is called" << endl;

}

And now you can initialize thread with **publicFunction()** of **myFunctor** class:

myFunctor myFunc;

//initializing thread with member function of myFunctor class

thread functorTest(&myFunctor::publicFunction,myFunc);

**if** (functorTest.joinable())

functorTest.join();

Passing arguments to thread

In the previous examples, we used only functions and objects without passing any arguments to these functions and objects.

We can use a function with parameters for thread initialization. Create new function for testing this possibility:

**void** printSomeValues(**int** val, **char**\* str, **double** dval)

{

cout << val << " " << str <<" " << dval << endl;

}

As you can see, this function takes three arguments. If you want to initialize a thread with this function, first you have to pass a pointer to this function, then pass the arguments to the function in the same order as they are in the parameter list of the function:

**char**\* str = "Hello";

//5, str and 3.2 are passed to printSomeValues function

thread paramPass(printSomeValues, 5, str, 3.2);

**if** (paramPass.joinable())

paramPass.join();

When you want to initialize a thread with an object with parameters, we have to add corresponding parameter list to the overloading version of operator ():

**class** myFunctorParam

{

**public**:

**void** **operator**()(**int**\* arr, **int** length)

{

cout << "An array of length " << length << "is passed to thread" << endl;

**for** (**int** i = 0; i != length; ++i)

cout << arr[i] << " " << endl;

cout << endl;

}

};

As you can see, operator () takes two parameters:

**void** **operator**()(**int**\* arr, **int** length)

The initialization of the thread with an object in this case is similar to using a function with parameters:

//these parameters will be passed to thread

**int** arr[5] = { 1, 3, 5, 7, 9 };

myFunctorParam objParamPass;

thread test(objParamPass, arr, 5);

**if** (test.joinable())

test.join();

It is possible to use a member function of a class to pass parameters to thread. Add new public function to myFunctorParam class:

**void** changeSign(**int**\* arr, **int** length)

{

cout << "An arrray of length " << length << "is passed to thread" << endl;

**for** (**int** i = 0; i != length; ++i)

cout << arr[i] << " ";

cout << "Changing sign of all elements of initial array" << endl;

**for** (**int** i = 0; i != length; ++i)

{

arr[i] \*= -1;

cout << arr[i] << " ";

}

}

Passing arguments to member function:

**int** arr2[5] = { -1, 3, 5, -7, 0 };

//initialize thread with member function

thread test2(&myFunctorParam::changeSign, &objParamPass, arr2, 5);

**if** (test2.joinable())

test2.join();

When you pass arguments to the member function of a class, you have to specify arguments in the same order as they are listed in the parameter list of the function. It is done after the second parameter of the thread constructor:

thread test2(&myFunctorParam::changeSign, &objParamPass, arr2, 5);

Thread ID

Every thread has its unique identifier. Class thread has public member function that returns the ID of the thread:

id get\_id()

The returned value is of type id that is specified in thread class.

Look on the following example:

//create 3 different threads

thread t1(showMessage);

thread t2(showMessage);

thread t3(showMessage);

//get id of all the threads

thread::id id1 = t1.get\_id();

thread::id id2 = t2.get\_id();

thread::id id3 = t3.get\_id();

//join all the threads

**if** (t1.joinable())

{

t1.join();

cout << "Thread with id " << id1 << " is terminated" << endl;

}

**if** (t2.joinable())

{

t2.join();

cout << "Thread with id " << id2 << " is terminated" << endl;

}

**if** (t3.joinable())

{

t3.join();

cout << "Thread with id " << id3 << " is terminated" << endl;

}

Every thread prints its unique identifier after it finishes its execution:

Thread with id 8228 is terminated  
Thread with id 10948 is terminated  
Thread with id 9552 is terminated

this\_thread  Namespace

this\_thread namespace from thread header offers possibilities to work with current thread. This namespace contains four useful functions:

1. **id\_get\_id()** – returns the id of current thread.

2. **template**  
    **void sleep\_until (const chrono::time\_point<Clock,Duration>& abs\_time)** – blocks current thread until abs\_time is not reached.

3. **template**  
    **void sleep\_for (const chrono::duration<Rep,Period>& rel\_time);** – thread is blocked during time span specified by rel\_time.

4. **void yield()** – current thread allows implementation to reschedule the execution of thread. It used to avoid blocking.

This is an example of using these functions:

#include <iostream>

#include <iomanip>

#include <thread>

#include <chrono>

#include <ctime>

**using** **namespace** std;

**using** std::chrono::system\_clock;

**int** main()

{

cout << "The id of current thread is " << this\_thread::get\_id << endl;

//sleep while next minute is not reached

//get current time

time\_t timet = system\_clock::to\_time\_t(system\_clock::now());

//convert it to tm struct

**struct** tm \* time = localtime(&timet);

cout << "Current time: " << put\_time(time, "%X") << '\n';

std::cout << "Waiting for the next minute to begin...\n";

time->tm\_min++; time->tm\_sec = 0;

//sleep until next minute is not reached

this\_thread::sleep\_until(system\_clock::from\_time\_t(mktime(time)));

cout << std::put\_time(time, "%X") << " reached!\n";

//sleep for 5 seconds

this\_thread::sleep\_for(chrono::seconds(5));

//get current time

timet = system\_clock::to\_time\_t(system\_clock::now());

//convert it to tm struct

time = std::localtime(&timet);

cout << "Current time: " << put\_time(time, "%X") << '\n';

}

You will get an output depending on your current time:

The id of current thread is 009717C6  
Current time: 15:28:35  
Waiting for the next minute to begin...  
15:29:00 reached!  
Current time: 15:29:05

Concurrent access to resources

Multithreading programming faces a problem with concurrent access to a shared resource. Simultaneous access to the same resource can leads to a lot of errors and chaos in the program.

Have a look at below example:

vector<int> vec;

**void** push()

{

**for** (**int** i = 0; i != 10; ++i)

{

cout << "Push " << i << endl;

\_sleep(500);

vec.push\_back(i);

}

}

**void** pop()

{

**for** (**int** i = 0; i != 10; ++i)

{

**if** (vec.size() > 0)

{

**int** val = vec.back();

vec.pop\_back();

cout << "Pop "<< val << endl;

}

\_sleep(500);

}

}

**int** main()

{

//create two threads

thread push(push);

thread pop(pop);

**if** (push.joinable())

push.join();

**if** (pop.joinable())

pop.join();

}

As you can see, there is a global vector **vec** of integer values. Two threads **push**and **pop** try to access this vector simultaneously: the first thread pushes an element to the vector and the second one tries to pop an element from the vector.

The access to the vector is not synchronized. Threads are accessing vector non-continuously. Because of simultaneous access to shared data many errors can appear.

Mutex

Class **mutex** is a synchronization primitive that is used to protect shared data from simultaneous access. A mutex can be locked and unlocked. Once a mutex is locked, current thread owns mutex until it is not unlocked. It means that no other thread can execute any instructions from the block of code surrounded by mutex until thread that owns mutex unlocks it. If you want to use mutex, you have to include mutex header in the program:

#include <mutex>

After this, you have to create a global variable of **mutex**type. It will be used to synchronize access to the shared data:

Once you want that a portion of program to be executed only by one thread in the same period, you have to “lock” it using mutex:

**void** push()

{

m.**lock**();

**for** (**int** i = 0; i != 10; ++i)

{

cout << "Push " << i << endl;

\_sleep(500);

vec.push\_back(i);

}

m.unlock();

}

**void** pop()

{

m.**lock**();

**for** (**int** i = 0; i != 10; ++i)

{

**if** (vec.size() > 0)

{

**int** val = vec.back();

vec.pop\_back();

cout << "Pop " << val << endl;

}

\_sleep(500);

}

m.unlock();

}

Operations of pushing and popping elements to the vector are locked using mutex. Therefore, if a thread enters a block of instructions and locks the mutex, no any thread can execute this code until mutex is unlocked. Try to execute this program again:

//create two threads

thread push(push);

thread pop(pop);

**if** (push.joinable())

push.join();

**if** (pop.joinable())

pop.join();

Now, the access to vector is synchronized:

Push 0  
Push 1  
Push 2  
Push 3  
Push 4  
Push 5  
Push 6  
Push 7  
Push 8  
Push 9  
Pop 9  
Pop 8  
Pop 7  
Pop 6  
Pop 5  
Pop 4  
Pop 3  
Pop 2  
Pop 1  
Pop 0

We can examine another example of mutex usage. Imagine the following situation:

“A lot of people run to a call-box to talk to their friend. The first person to catch the door-handle of the call box is the only one who is allowed to use the phone. He must keep holding on to the handle of the door as long as he uses the call box. Otherwise, someone else will catch hold of the handle, throw him out and talk to his friend. There is no queue system as in real life. When the person finishes his call, exits the call-box and leaves the door handle, the next person that gets hold of the door handle will be allowed to use the phone.”

In this case, you have to imagine a problem of simultaneous access to data in the following way:

A **thread** is a person.  
The **mutex** is the door handle.  
The **lock** is the person's hand.  
The **resource** is the phone.

Any thread which has to execute some lines of code which should not be executed by other threads at the same time (using the phone to talk to his friend), has to first acquire a lock on a mutex (clutching the door handle of the call-box). Only then, a thread will be able to run those lines of code (making the phone call).

Once the thread finishes executing that code, it should release the lock on the mutex so that another thread can acquire a lock on the mutex (other people being able to access the phone booth).

This is an example of this situation written using mutex:

std::mutex m;//door handle

**void** makeACall()

{

m.**lock**();//person enters the call box and locks the door

//now it can talk to his friend without any interruption

cout << " Hello my friend, this is " << this\_thread::get\_id() << endl;

//this person finished to talk to his friend

m.unlock();//and he leaves the call box and unlock the door

}

**int** main()

{

//create 3 persons who want to make a call from call box

thread person1(makeACall);

thread person2(makeACall);

thread person3(makeACall);

**if** (person1.joinable())

{

person1.join();

}

**if** (person2.joinable())

{

person2.join();

}

**if** (person3.joinable())

{

person3.join();

}

}

The access to makeACall function will be synchronized. You will get an output that is similar to this one:

Hello my friend, this is 3636  
Hello my friend, this is 5680  
Hello my friend, this is 928

In C++11, C++ introduced built-in support for threads std::thread. Since then, starting a new thread in C++ is as easy as defining an object. However, dynamically terminating a running C++ thread is still very tricky, especially for the joined/detached thread. There are plenty of discussions of this topic, and the conclusion is that

[“terminate 1 thread + forcefully (target thread doesn’t cooperate) + pure C++11 = No way”](https://stackoverflow.com/questions/12207684/how-do-i-terminate-a-thread-in-c11).

This post is not aimed to provide a magic to kill C++ thread with built-in C++ APIs - apologize for it. Instead, I am going to explain how to kill a C++ thread using native(OS/compiler-dependent) function.

The complete code in this post can be found on [*my github*](https://github.com/bo-yang/terminate_cpp_thread), which is built with GCC 4.8.5 and verified on CentOS 7.

### **1. std::thread destructor not work**

The destructor [std::thread::~thread()](http://en.cppreference.com/w/cpp/thread/thread/~thread) can only terminate the thread when the thread is still [joinable](http://en.cppreference.com/w/cpp/thread/thread/joinable). A C++ thread is joinable after it is started and before calling join or detach(see [this example](http://en.cppreference.com/w/cpp/thread/thread/joinable)). So for joined/detached thread, the std::thread destructor cannot terminate the thread at all.

Following is an class to dynamically start and stop a thread. The std::unordered\_map<std::string, std::thread> is used to record thread name and thread. And in stop\_thread(), the std::thread destructor is directly called.

**class** **Foo** {

**public**:

**void** sleep\_for(**const** std::string &tname, **int** num)

{

prctl(PR\_SET\_NAME,tname.c\_str(),**0**,**0**,**0**);

sleep(num);

}

**void** start\_thread(**const** std::string &tname)

{

std::**thread** thrd = std::**thread**(&Foo::sleep\_for, **this**, tname, **3600**);

thrd.detach();

tm\_[tname] = std::move(thrd);

std::cout << "Thread " << tname << " created:" << std::endl;

}

**void** stop\_thread(**const** std::string &tname)

{

ThreadMap::const\_iterator it = tm\_.find(tname);

**if** (it != tm\_.end()) {

it->second.std::**thread**::~**thread**(); // thread not killed

tm\_.erase(tname);

std::cout << "Thread " << tname << " killed:" << std::endl;

}

}

**private**:

**typedef** std::unordered\_map<std::string, std::**thread**> ThreadMap;

ThreadMap tm\_;

};

To show the running threads, function show\_thread() is called everytime starting/stopping a thread.

**void** **show\_thread**(**const** std::string &keyword)

{

std::string cmd("ps -T | grep ");

cmd += keyword;

system(cmd.c\_str());

}

**int** **main**()

{

Foo foo;

std::string keyword("test\_thread");

std::string tname1 = keyword + "1";

std::string tname2 = keyword + "2";

// create and kill thread 1

foo.start\_thread(tname1);

show\_thread(keyword);

foo.stop\_thread(tname1);

show\_thread(keyword);

// create and kill thread 2

foo.start\_thread(tname2);

show\_thread(keyword);

foo.stop\_thread(tname2);

show\_thread(keyword);

**return** **0**;

}

The output of the the above test code is like:

$ g++ -Wall -std=c++11 kill\_cpp\_thread.cc -o kill\_cpp\_thread -pthread -lpthread

$ ./kill\_cpp\_thread

Thread test\_thread1 created:

29469 29470 pts/5 00:00:00 test\_thread1

Thread test\_thread1 killed:

29469 29470 pts/5 00:00:00 test\_thread1

Thread test\_thread2 created:

29469 29470 pts/5 00:00:00 test\_thread1

29469 29477 pts/5 00:00:00 test\_thread2

Thread test\_thread2 killed:

29469 29470 pts/5 00:00:00 test\_thread1

29469 29477 pts/5 00:00:00 test\_thread2

Obviously, the test threads could not be killed in Foo::stop\_thread().

### **2. std::thread::id vs. pthread\_t**

To terminate threads with OS/compiler-dependent functions, we need to know how to get the native thread data type from C++ std::thread. Fortunately, std::thread provides an API native\_handle() to get the thread’s native handle type - before calling join() or detach(). And this native handle can be passed to native OS thread terminate function, e.g. pthread\_cancel().

Here’s a demo code to show the fact that the std::thread::native\_handle(), std::thread::get\_id() and [pthread\_self()](http://pubs.opengroup.org/onlinepubs/9699919799/) return the same pthread\_t to handle a C++ thread for Linux/GCC. You also can [try it out online](http://coliru.stacked-crooked.com/view?id=44c988ce133260ee).

#include <mutex>

#include <iostream>

#include <chrono>

#include <cstring>

#include <pthread.h>

std::mutex iomutex;

**void** **f**(**int** num)

{

std::this\_thread::sleep\_for(std::chrono::seconds(**1**));

std::lock\_guard<std::mutex> lk(iomutex);

std::cout << "Thread " << num << " pthread\_t " << pthread\_self() << std::endl;

}

**int** **main**()

{

std::**thread** t1(f, **1**), t2(f, **2**);

//t1.join(); t2.join();

//t1.detach(); t2.detach();

std::cout << "Thread 1 thread id " << t1.get\_id() << std::endl;

std::cout << "Thread 2 thread id " << t2.get\_id() << std::endl;

std::cout << "Thread 1 native handle " << t1.native\_handle() << std::endl;

std::cout << "Thread 2 native handle " << t2.native\_handle() << std::endl;

t1.join(); t2.join();

//t1.detach(); t2.detach();

}

Build and run it get the following result:

$ g++ -Wall -std=c++11 cpp\_thread\_pthread.cc -o cpp\_thread\_pthread -pthread -lpthread

$ ./cpp\_thread\_pthread

Thread 1 thread id 140109390030592

Thread 2 thread id 140109381637888

Thread 1 native handle 140109390030592

Thread 2 native handle 140109381637888

Thread 1 pthread\_t 140109390030592

Thread 2 pthread\_t 140109381637888

However, after calling join() or detach(), the C++ thread loses the info of native handle type:

**int** **main**()

{

std::**thread** t1(f, **1**), t2(f, **2**);

t1.join(); t2.join();

//t1.detach(); t2.detach();

std::cout << "Thread 1 thread id " << t1.get\_id() << std::endl;

std::cout << "Thread 2 thread id " << t2.get\_id() << std::endl;

std::cout << "Thread 1 native handle " << t1.native\_handle() << std::endl;

std::cout << "Thread 2 native handle " << t2.native\_handle() << std::endl;

}

$ ./cpp\_thread\_pthread

Thread 1 pthread\_t 139811504355072

Thread 2 pthread\_t 139811495962368

Thread 1 thread id thread::id of a non-executing thread

Thread 2 thread id thread::id of a non-executing thread

Thread 1 native handle 0

Thread 2 native handle 0

So if you call pthread\_cancel(t1.native\_handle()) after t1 joined/detached, your program will definitely coredump.

### **3. The Terminator**

So in summary, to effectively call native thread termination function(e.g. pthread\_cancel), you need to save the native handle before calling std::thread::join() or std::thread::detach(). So that your native terminator always has a valid native handle to use.

Following is the revised Foo class that can stop C++ thread:

**class** **Foo** {

**public**:

**void** sleep\_for(**const** std::string &tname, **int** num)

{

prctl(PR\_SET\_NAME,tname.c\_str(),**0**,**0**,**0**);

sleep(num);

}

**void** start\_thread(**const** std::string &tname)

{

std::**thread** thrd = std::**thread**(&Foo::sleep\_for, **this**, tname, **3600**);

tm\_[tname] = thrd.native\_handle();

thrd.detach();

std::cout << "Thread " << tname << " created:" << std::endl;

}

**void** stop\_thread(**const** std::string &tname)

{

ThreadMap::const\_iterator it = tm\_.find(tname);

**if** (it != tm\_.end()) {

pthread\_cancel(it->second);

tm\_.erase(tname);

std::cout << "Thread " << tname << " killed:" << std::endl;

}

}

**private**:

**typedef** std::unordered\_map<std::string, pthread\_t> ThreadMap;

ThreadMap tm\_;

};

And the result is:

$ g++ -Wall -std=c++11 kill\_cpp\_thread.cc -o kill\_cpp\_thread -pthread -lpthread

$ ./kill\_cpp\_thread

Thread test\_thread1 created:

30332 30333 pts/5 00:00:00 test\_thread1

Thread test\_thread1 killed:

Thread test\_thread2 created:

30332 30340 pts/5 00:00:00 test\_thread2

Thread test\_thread2 killed:

Synchronisation:

1. Join

|  |  |
| --- | --- |
| // example for thread::join  #include <iostream> // std::cout  #include <thread> // std::thread, std::this\_thread::sleep\_for  #include <chrono> // std::chrono::seconds    void pause\_thread(int n)  {  std::this\_thread::sleep\_for (std::chrono::seconds(n));  std::cout << "pause of " << n << " seconds ended\n";  }    int main()  {  std::cout << "Spawning 3 threads...\n";  std::thread t1 (pause\_thread,1);  std::thread t2 (pause\_thread,2);  std::thread t3 (pause\_thread,3);  std::cout << "Done spawning threads. Now waiting for them to join:\n";  t1.join();  t2.join();  t3.join();  std::cout << "All threads joined!\n";  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/thread/thread/join/) |

1. Mutex:

To get mutex related library, include<mutex>

|  |  |
| --- | --- |
| // mutex example  #include <iostream> // std::cout  #include <thread> // std::thread  #include <mutex> // std::mutex  std::mutex mtx; // mutex for critical section  void print\_block (int n, char c) {  // critical section (exclusive access to std::cout signaled by locking mtx):  mtx.lock();  for (int i=0; i<n; ++i) { std::cout << c; }  std::cout << '\n';  mtx.unlock();  }  int main ()  {  std::thread th1 (print\_block,50,'\*');  std::thread th2 (print\_block,50,'$');  th1.join();  th2.join();  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/condition_variable/condition_variable/) |

Locks has two types:

1. Lock\_guard: by default lock\_gaurd is suggested to use on first priority. Lock guard doesn’t have lock(), unlock() function. The  std::**lock\_guard** will be locked only once on construction and unlocked on destruction.

std::lock\_guard<mutex\_type> lock(this->my\_mutex);

// lock\_guard example

#include <iostream> // std::cout

#include <thread> // std::thread

#include <mutex> // std::mutex, std::lock\_guard

#include <stdexcept> // std::logic\_error

std::mutex mtx;

void print\_even (int x) {

if (x%2==0) std::cout << x << " is even\n";

else throw (std::logic\_error("not even"));

}

void print\_thread\_id (int id) {

try {

// using a local lock\_guard to lock mtx guarantees unlocking on destruction / exception:

std::lock\_guard<std::mutex> lck (mtx);

print\_even(id);

}

catch (std::logic\_error&) {

std::cout << "[exception caught]\n";

}

}

int main ()

{

std::thread threads[10];

// spawn 10 threads:

for (int i=0; i<10; ++i)

threads[i] = std::thread(print\_thread\_id,i+1);

for (auto& th : threads) th.join();

return 0;

}

1. Unique\_lock:

Need std::unique\_lock for the condition variable.

It has lock(), unlock() public member function.

std::unique\_lock<std::mutex> lck (mtx);

// unique\_lock example

#include <iostream> // std::cout

#include <thread> // std::thread

#include <mutex> // std::mutex, std::unique\_lock

std::mutex mtx; // mutex for critical section

void print\_block (int n, char c) {

// critical section (exclusive access to std::cout signaled by lifetime of lck):

std::unique\_lock<std::mutex> lck (mtx);

for (int i=0; i<n; ++i) { std::cout << c; }

std::cout << '\n';

}

int main ()

{

std::thread th1 (print\_block,50,'\*');

std::thread th2 (print\_block,50,'$');

th1.join();

th2.join();

return 0;

}

1. Condition variable:

|  |  |
| --- | --- |
| #include <iostream> // std::cout  #include <thread> // std::thread  #include <mutex> // std::mutex, std::unique\_lock  #include <condition\_variable> // std::condition\_variable  std::mutex mtx;  std::condition\_variable cv;  bool ready = false;  void print\_id (int id) {  std::unique\_lock<std::mutex> lck(mtx);  while (!ready) cv.wait(lck);  // ...  std::cout << "thread " << id << '\n';  }  void go() {  std::unique\_lock<std::mutex> lck(mtx);  ready = true;  cv.notify\_all();  }  int main ()  {  std::thread threads[10];  // spawn 10 threads:  for (int i=0; i<10; ++i)  threads[i] = std::thread(print\_id,i);  std::cout << "10 threads ready to race...\n";  go(); // go!  for (auto& th : threads) th.join();  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/condition_variable/condition_variable/) |