

# C++ 11 Threads: Make your (multitasking) life easier.

## C++ 11 Threads

### Introduction

This article aims to help the experienced Win32 programmer to understand differences and similarities between C++ 11 threads and synchronization objects, and Win32 threads and synchronization objects.

In Win32, all synchronization object handles are global handles. They can be shared, even duplicated among processes. In C++ 11 all synchronization objects are stack objects, which means they have to be "detached" (if detaching is supported) in order to be able to get destructed by the stack frame. If you do not detach many objects, they will undo their actions and possibly kill your plans (which, if you are a Win32 programmer are "global handle"- oriented).

All C++ 11 synchronization objects have a `native_handle()` member which returns the implementation-specific handle (in Win32, a `HANDLE`).

In all my examples, I give the Win32 pseudocode. Have fun!

## Background Color

0x00000000. That is, nothing. I'm also a C++ 11 thread newbie. You need to know your way about Win32 synchronization through. This is not a tutorial on proper synchronization techniques, but a quick introduction to the C++11 mechanisms for doing what you have already planned in your mind.

## Simplicity makes it perfect

The simple example: Start a thread, then wait for it to finish.

Hide Copy Code

```
void foo()
{
}
void func()
{
    std::thread t(foo); // Starts. Equal to CreateThread.
    t.join(); // Equal to WaitForSingleObject to the thread handle.
}
```

Not like a Win32 thread however, here you can have parameters.

Hide Copy Code

```
void foo(int x, int y)
{
    // x = 4, y = 5.
}
void func()
{
    std::thread t(foo, 4, 5); // Acceptable.
}
```

```
t.join();
}
```

This makes it easy to have a member function as a thread, by passing the hidden 'this' pointer to **std::thread**.

If **std::thread** gets destructed and you haven't called **join()**, it will call abort. To let the thread run without the C++ wrapper:

Hide Copy Code

```
void foo()
{
}
void func()
{
    std::thread t(foo);
    t.detach(); // C++ object detached from Win32 object. Now t.join() would throw std::system_error().
}
```

Along with **join()** and **detach()**, there are also **joinable()**, **get\_id()**, **sleep\_for()**, **sleep\_until()**. Their use should be self-explanatory.

## Use that Mutex

A **std::mutex** is similar to a Win32 critical section. **lock()** is like **EnterCriticalSection**, **unlock()** is like **LeaveCriticalSection**, **try\_lock()** is like **TryEnterCriticalSection**.

Hide Copy Code

```
std::mutex m;
int j = 0;
void foo()
{
    m.lock();
    j++;
    m.unlock();
}
void func()
{
    std::thread t1(foo);
    std::thread t2(foo);
    t1.join();
    t2.join();
    // j = 2;
}
```

As before, you must unlock a **std::mutex** after locking and you must not lock a **std::mutex** if you have already locked it. This is different than Win32, in which, **EnterCriticalSection** does not fail when you are already inside the critical section, but instead increases a counter.

Hey, don't leave. There's **std::recursive\_mutex** (who invented these names ?) that behaves exactly like a critical section:

Hide Copy Code

```
std::recursive_mutex m;
void foo()
{
    m.lock();
    m.lock(); // now valid
    j++;
    m.unlock();
    m.unlock(); // don't forget!
}
```

In addition to these classes, there's also **std::timed\_mutex** and **std::recursive\_timed\_mutex** which also provide a **try\_lock\_for/ try\_lock\_until**. These allow you to wait for a lock until a specific timeout or specific time.

## Thread Local Storage

Similar to **TLS**, this facility allows you to declare a global variable with the **thread\_local** modifier. This means that each thread has it's own instance of that variable, with a common global name. Consider the previous example again:

Hide Copy Code

```
int j = 0;
void foo()
{
    m.lock();
    j++;
    m.unlock();
}
void func()
{
    j = 0;
    std::thread t1(foo);
    std::thread t2(foo);
    t1.join();
    t2.join();
    // j = 2;
}
```

But see this now:

Hide Copy Code

```
thread_local int j = 0;
void foo()
{
    m.lock();
    j++; // j is now 1, no matter the thread. j is local to this thread.
    m.unlock();
}
void func()
{
    j = 0;
    std::thread t1(foo);
    std::thread t2(foo);
    t1.join();
    t2.join();
    // j still 0. The other "j"s were local to the threads
}
```

Thread Local Storage is not yet supported in Visual Studio.

## Mysterious Variables

Conditional variables are objects that enable threads to wait for a specific condition. In Windows, these objects are user-mode and they cannot be shared with other processes. In Windows, conditional variables are associated with critical sections to acquire or release a lock. **std::condition\_variable** is associated with a **std::mutex** for the same reason.

Hide Copy Code

```
std::condition_variable c;
std::mutex mu; // We use a mutex rather than a recursive_mutex because the lock has to be acquired only and exactly once.
void foo5()
```

```

{
    std::unique_lock lock(mu); // Lock the mutex
    c.notify_one(); // WakeConditionVariable. It also releases the unique lock
}
void func5()
{
    std::unique_lock lock(mu); // Lock the mutex
    std::thread t1(foo5);
    c.wait(lock); // Equal to SleepConditionVariableCS. This unlocks the mutex mu and allows foo5 to lock
    it
    t1.join();
}

```

This is not so innocent as it looks. `c.wait()` might return even when `c.notify_one()` is not called (a situation known as a **spurious wakeup** - [http://msdn.microsoft.com/en-us/library/windows/desktop/ms686301\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms686301(v=vs.85).aspx)). Typically you place the `c.wait()` in a while loop which also checks an external variable to verify the notification.

Conditional Variables are only supported in Vista or later.

## Promise the Future

Consider this scenario. You want a thread to do some work and return you a result. Meanwhile you want to do other work which may or may not take some time. You want the result of the other thread to be available at a certain point.

In Win32, you would:

- Start the thread with `CreateThread()`.
- Inside the thread, do the work and set an event when ready, while storing the result to a global variable.
- In main code, do the other work then `WaitForSingleObject` when you want the result.

In C++ 11 this is done easily by using **`std::future`**, and return any type since it's a template.

Hide Copy Code

```

int GetMyAnswer()
{
    return 10;
}
int main()
{
    std::future<int> GetAnAnswer = std::async(GetMyAnswer); // GetMyAnswer starts background execution
    int answer = GetAnAnswer.get(); // answer = 10;
    // If GetMyAnswer has finished, this call returns immediately.
    // If not, it waits for the thread to finish.
}

```

You also have the **`std::promise`**. This object can provide something that **`std::future`** will later request. If you call **`std::future::get()`** before anything has been put into the promise, get waits until the promised value is there. If **`std::promise::set_exception()`** is called, **`std::future::get()`** throws that exception. If the **`std::promise`** is destroyed and you call **`std::future::get()`**, you get a **`broken_promise`** exception.

Hide Copy Code

```

std::promise<int> sex;
void foo()
{
    // do stuff
    sex.set_value(1); // After this call, future::get() will return this value.
    sex.set_exception(std::make_exception_ptr(std::runtime_error("broken_condom"))); // After this call,
    future::get() will throw this exception
}

```

```
int main()
{
    future<int> makesex = sex.get_future();
    std::thread t(foo);

    // do stuff
    try
    {
        makesex.get();
        hurray();
    }
    catch(...)
    {
        // She dumped us :(
    }
}
```

## Code

The attached CPP file contains all we 've said so far in a ready-to-compile Visual Studio 12 with November 2012 CTP compiler (except the TLS mechanism).

## What's next?

There are a lot of things that deserve to be included, like:

- Semaphores
- Named objects
- Shareable objects across processes.
- [...]

What should you do? Generally, when writing new code, do prefer the standards if they are enough for you. For existing code, I would keep my Win32 calls and, when I need to port them to another platform, then I would implement CreateThread, SetEvent etc with C++ 11 functions.

**GOOD LUCK.**

## History

- 5 - 2 - 2013 : First release.

## License


This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

## Share

## About the Author

### **Michael Chourdakis**

Engineer

Greece 

I'm working in C++, PHP, Java, Windows, iOS and Android.

I've a PhD in Digital Signal Processing and I specialize in Pro Audio applications.

My home page: <http://www.michaelchourdakis.com>