**[Why do we need atomic\_shared\_ptr?](https://www.justsoftwaresolutions.co.uk/threading/why-do-we-need-atomic_shared_ptr.html)**

**Friday, 21 August 2015**

One of the new class templates provided in the upcoming [**Concurrency TS**](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html) is **[atomic\_shared\_ptr](http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4501.html" \l "atomic)**, along with its counterpart atomic\_weak\_ptr. As you might guess, these are the std::shared\_ptr and std::weak\_ptr equivalents of std::atomic<T\*>, but why would one need them? Doesn't std::shared\_ptr already have to synchronize the reference count?

**std::shared\_ptr and multiple threads**

std::shared\_ptr works great in multiple threads, *provided each thread has its own copy or copies*. In this case, the changes to the reference count are indeed synchronized, and everything just works, provided of course that what you do with the shared data is correctly synchronized.

class MyClass;

void thread\_func(std::shared\_ptr<MyClass> sp){

sp->do\_stuff();

std::shared\_ptr<MyClass> sp2=sp;

do\_stuff\_with(sp2);

}

int main(){

std::shared\_ptr<MyClass> sp(new MyClass);

std::thread thread1(thread\_func,sp);

std::thread thread2(thread\_func,sp);

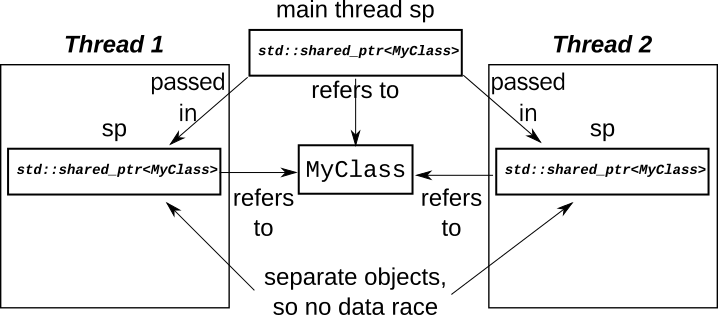
thread2.join();

thread1.join();

}

In this example, you need to ensure that it is safe to call MyClass::do\_stuff() and do\_stuff\_with() from multiple threads concurrently on the same instance, but the reference counts are handled OK.

**Figure 1: Separate shared\_ptr instances**



**The problems come when you try and share a single std::shared\_ptr instance between threads.**

**Sharing a std::shared\_ptr instance between threads**

I could provide a trivial example of a std::shared\_ptr instance shared between threads, but instead we'll look at something a little more interesting, to give a better feel for why you might need it.

Consider for a moment a simple singly-linked list, where each node holds a pointer to the next:

class MyList{

struct Node{

MyClass data;

std::unique\_ptr<Node> next;

};

std::unique\_ptr<Node> head;

// constructors etc. omitted.

};

If we're going to access this from 2 threads, then we have a choice:

1. We could wrap the whole object with a mutex, so only one thread is accessing the list at a time, or
2. We could try and allow concurrent accesses.

For the sake of this article, we're going to assume we're allowing concurrent accesses.

Let's start simply: we want to traverse the list. Writing a traversal function is easy:

class MyList{

void traverse(std::function<void(MyClass)> f){

Node\* p=head.get();

while(p){

f(p->data);

p=p->next;

}

}

};

Assuming the list is immutable, this is fine, but immutable lists are no fun! We want to **remove** items from the front of our list. What changes do we need to make to support that?

**Removing from the front of the list**

If everything was single threaded, removing an element would be easy:

class MyList{

void pop\_front(){

Node\* p=head.get();

if(p){

head=std::move(p->next);

}

}

};

If the list is not empty, the new head is the next pointer of the old head. However, we've got multiple threads accessing this list, so things aren't so straightforward.

Suppose we have a list of 3 elements.

A -> B -> C

If one thread is traversing the list, and another is removing the first element, there is a potential for a race.

1. Thread X reads the head pointer for the list and gets a pointer to A.
2. Thread Y removes A from the list and deletes it.
3. Thread X tries to access the data for node A, but node A has been deleted, so we have a dangling pointer and undefined behaviour.

How can we fix it?

The first thing to change is to make all our std::unique\_ptrs into std::shared\_ptrs, and have our traversal function take a std::shared\_ptr copy rather than using a raw pointer. That way, node A won't be deleted immediately, since our traversing thread still holds a reference.

class MyList{

struct Node{

MyClass data;

std::shared\_ptr<Node> next;

};

std::shared\_ptr<Node> head;

void traverse(std::function<void(MyClass)> f){

std::shared\_ptr<Node> p=head;

while(p){

f(p->data);

p=p->next;

}

}

void pop\_front(){

std::shared\_ptr<Node> p=head;

if(p){

head=std::move(p->next);

}

}

// constructors etc. omitted.

};

Unfortunately that only fixes that race condition. There is a second race that is just as bad.

**The second race condition**

The second race condition is on head itself. In order to traverse the list, thread X has to read head. In the mean time, thread Y is updating head. This is the very definition of a *data race*, and is thus **undefined behaviour**.

We therefore need to do something to fix it.

We could use a mutex to protect head. This would be more fine-grained than a whole-list mutex, since it would only be held for the brief time when the pointer was being read or changed. However, we don't need to: we can use std::experimental::atomic\_shared\_ptr instead.

The implementation is allowed to use a mutex internally with atomic\_shared\_ptr, in which case we haven't gained anything with respect to performance or concurrency, but we *have* gained by reducing the maintenance load on our code. We don't have to have an explicit mutex data member, and we don't have to remember to lock it and unlock it correctly around every access to head. Instead, we can defer all that to the implementation with a single line change:

class MyList{

std::experimental::atomic\_shared\_ptr<Node> head;

};

Now, the read from head no longer races with a store to head: the implementation of atomic\_shared\_ptr takes care of ensuring that the load gets either the new value or the old one without problem, and ensuring that the reference count is correctly updated.

Unfortunately, the code is still not bug free: what if 2 threads try and remove a node at the same time.

**Race 3: double removal**

As it stands, pop\_front assumes it is the only modifying thread, which leaves the door wide open for race conditions. If 2 threads call pop\_front concurrently, we can get the following scenario:

1. Thread X loads head and gets a pointer to node A.
2. Thread Y loads head and gets a pointer to node A.
3. Thread X replaces head with A->next, which is node B.
4. Thread Y replaces head with A->next, which is node B.

So, two threads call pop\_front, but only one node is removed. That's a bug.

The fix here is to use the ubiquitous compare\_exchange\_strong function, a staple of any programmer who's ever written code that uses atomic variables.

class MyList{

void pop\_front(){

std::shared\_ptr<Node> p=head;

while(p &&

!head.compare\_exchange\_strong(p,p->next));

}

};

If head has changed since we loaded it, then the call to compare\_exchange\_strong will return false, and reload p for us. We then loop again, checking that p is still non-null.

This will ensure that two calls to pop\_front removes two nodes (if there are 2 nodes) without problems either with each other, or with a traversing thread.

Experienced lock-free programmers might well be thinking "what about the ABA problem?" right about now. Thankfully, we don't have to worry!

**What no ABA problem?**

That's right, pop\_front does not suffer from the ABA problem. Even assuming we've got a function that adds new values, we can never get a new value of head the same as the old one. This is an additional benefit of using std::shared\_ptr: the old node is kept alive as long as one thread holds a pointer. So, thread X reads head and gets a pointer to node A. This node is now kept alive until thread X destroys or reassigns that pointer. That means that no new node can be allocated with the same address, so if head is equal to the value p then it really must be the same node, and not just some imposter that happens to share the same address.

**Lock-freedom**

I mentioned earlier that implementations *may* use a mutex to provide the synchronization in atomic\_shared\_ptr. They may also manage to make it lock-free. This can be tested using the is\_lock\_free() member function common to all the C++ atomic types.

The advantage of providing a lock-free atomic\_shared\_ptr should be obvious: by avoiding the use of a mutex, there is greater scope for concurrency of the atomic\_shared\_ptr operations. In particular, multiple concurrent reads can proceed unhindered.

The downside will also be apparent to anyone with any experience with lock-free code: the code is more complex, and has more work to do, so may be slower in the uncontended cases. The other big downside of lock-free code (maintenance and correctness) is passed over to the implementor!

It is my belief that the scalability benefits outweight the complexity, which is why [**Just::Thread**](http://www.stdthread.co.uk/) v2.2 will ship with a lock-free atomic\_shared\_ptr implementation.