Atomic Smart Pointers, rev. 1

Herb Sutter

Document #:	N4162
Date:	2014-10-06
Reply to:	Herb Sutter (hsutter@microsoft.com)

This is a revision of N4058 to apply SG1 feedback in Redmond to rename `atomic<*_ptr<T>>` to `atomic_*_ptr<T>`, require default initialization to null, and add proposed wording.

Contents

1. Motivation

1.1. Problem

We encourage that modern C++ code should avoid all uses of owning raw pointers and explicit delete. Instead, programmers should use unique_ptr and shared_ptr (with weak_ptr), as this is known to lead to simpler and leak-free memory-safe code. This is especially important when lifetimes are unstructured or nondeterministic, which arises especially in concurrent code, and it has long been well-known that the smart pointers would be useful there; for an example, see [1].

Unfortunately, lock-free code is still mostly forced to use owning raw pointers. Our unique_ptr, shared_ptr, and weak_ptr would directly benefit lock-free code just as they do regular code (see next section), but they are not usable easily or at all in lock-free code because we do not support atomic forms of these pointers. Specifically:

- For shared_ptr we at least have the free functions in [util.smartptr.shared.atomic]. However, as pointed out in [2] and summarized later in this paper, these free functions are strictly inferior in consistency, correctness, and performance to an atomic_shared_ptr<T> type.
- For unique_ptr and weak_ptr we have nothing.

1.2. Motivating example for atomic_unique_ptr<T>: Producer-consumer handoff

Consider a producer that creates a data structure and atomically hands it off to a consumer using a single atomic store operation. Note that the red code is now frowned upon in general as “not modern safe C++.”

```
atomic<X*> p_root{nullptr}; // need init depending on p_root’s scope

void producer() {
    auto temp = make_unique<X>();
    load_from_disk_and_store_in( *temp ); // build data structure
    p_root = temp.release();              // atomically publish it
}
```

In any non-atomic case, we would say that this owning X* and explicit new and delete should be replaced with a unique_ptr<X> and make_unique<X>. Here we can use unique_ptr “partway”—only locally in the function, and then we immediately lose the exception safety and automated lifetime for the rest of the world and the rest of the lifetime of the X object as soon as we pass it to the consumer.

If we had atomic_unique_ptr<T> we could (and should) write the following equivalent code that is safer, no slower, and less error-prone because we can directly express the unique ownership semantics including ownership transfer:

```
atomic_unique_ptr<X> p_root;

void producer() {
    auto temp = make_unique<X>();
    load_from_disk_and_store_in( *temp ); // build data structure
    p_root = move(temp);                  // atomically publish it
}
```

This righteous code should be supported.

1.3. Motivating example for `atomic_shared_ptr<T>`: ABA + robustness + efficiency

"Everyone knows" (at least, I thought I knew until recently) that portable C++ code cannot express many simple high-performance lock-free data structures, such as a concurrent list or stack that allows concurrent insert and erase operations, because of the ABA problem. Even prominent experts commonly teach that the answer is to resort to contortions like hazard pointer libraries, or resort to as-yet-nonstandard extensions like garbage collection or transactional memory.

Yet "everyone" is mostly wrong, because [util.smartptr.shared.atomic] already makes it possible for portable C++ code to avoid the ABA problem (as long as there are no unbroken cycles). See the Appendix for a more complete example of a lock-free stack implemented as a singly linked list without ABA issues in portable C++, thanks to atomic use of `shared_ptr`s.

However, such code is forced to use the free functions in [util.smartptr.shared.atomic], and those are strictly inferior in consistency, correctness, and performance to a real `atomic_shared_ptr<T>`. The fundamental design flaw is that a normal `shared_ptr` and an "atomic `shared_ptr`" are inherently different types, and therefore should be expressed distinctly; and then the latter should have its natural spelling consistent with the existing atomic types.

Consistency. As far as I know, the [util.smartptr.shared.atomic] functions are the only atomic operations in the standard that are not available via an `atomic` type. And for all types besides `shared_ptr`, we teach programmers to use `atomic` types in C++, not `atomic_*` C-style functions. And that's in part because of...

Correctness. Using the free functions makes code error-prone and racy by default. It is far superior to write `atomic` once on the variable declaration itself and know all accesses will be atomic, instead of having to remember to use the `atomic_*` operation on *every* use of the object, even apparently-plain reads. The latter style is error-prone; for example, "doing it wrong" means simply writing whitespace (e.g., `head` instead of `atomic_load(&head)`), so that in this style every use of the variable is "wrong by default." If you forget to write the `atomic_*` call in even one place, your code will still successfully compile without any errors or warnings, it will "appear to work" including likely pass most testing, but will still contain a silent race with undefined behavior that usually surfaces as intermittent hard-to-reproduce failures, often/usually in the field, and I expect also in some cases exploitable vulnerabilities. These classes of errors are eliminated by simply declaring the variable `atomic`, because then it's safe by default and to write the same set of bugs requires explicit non-whitespace code (sometimes explicit `memory_order_*` arguments, and usually `reinterpret_cast`ing).

Performance. `atomic_shared_ptr<>` as a distinct type has an important efficiency advantage over the functions in [util.smartptr.shared.atomic]—it can simply store an additional `atomic_flag` (or similar) for the internal spinlock as usual for `atomic<bigstruct>`. In contrast, the existing standalone functions are required to be usable on any arbitrary `shared_ptr` object, even though the vast majority of `shared_ptr`s will never be used atomically. This makes the free functions inherently less efficient; for example, the implementation could require every `shared_ptr` to carry the overhead of an internal spinlock variable (better concurrency, but significant overhead per `shared_ptr`), or else the library must maintain a lookaside data structure to store the extra information for `shared_ptr`s that are actually used atomically, or (worst and apparently common in practice) the library must use a global spinlock.

We should extend the existing consistent and superior practice of providing a distinct `atomic` type, to be available also for existing functionality that is already in the standard in [util.smartptr.shared.atomic].

1.4. Motivating example for `atomic_weak_ptr<T>`: Swinging a `weak_ptr`

Many atomic uses of `weak_ptr` are already supported just because most uses of a `weak_ptr` require first converting it to a `shared_ptr` using `lock()`, after which you use the `shared_ptr`.

However, we don't have a way to atomically reseat an existing `weak_ptr` to refer to a different object.

Consider the following code that remembers the last object seen, but only wants to hold a weak reference to later possibly observe the X object, but not keep it alive:

```
weak_ptr<X> p_last;

void use( const shared_ptr<X>& x ) {
    do_something_with( *x );
    p_last = x;              // remember last X seen
}
```

To make this safe for concurrent use today would require adding an indirection to store the `weak_ptr` itself on the heap and using an `atomic<weak_ptr<X>*>`. For example:

```
atomic<weak_ptr<X>*> p_last{nullptr}; // need init depending on scope

void use( const shared_ptr<X>& x ) {
    auto temp = make_unique<weak_ptr<X>>( x );
    do_something_with( *x );
    p_last.exchange( temp );  // remember last X seen
}
```

Instead we should be able to directly write the much simpler and less error-prone:

```
atomic_weak_ptr<X> p_last;

void use( const shared_ptr<X>& x ) {
    do_something_with( *x );
    p_last = x;              // remember last X seen
}
```

This righteous code should be supported.

2. Proposal

2.1. Add `atomic_shared_ptr<T>`

Specify an `atomic_shared_ptr<T>` type that is pure syntactic sugar for existing functionality—that supports exactly and only those operations already in [util.smartptr.shared.atomic], and not additional functions such as `fetch_add` which don't make sense for `shared_ptr`s anyway. Default construction initializes to `nullptr`.

This makes it clear that this proposal is not adding any new functionality and builds on known existing practice. If additional functions are desired in the future they can be added later.

2.2. Deprecate [util.smartptr.shared.atomic]

The [util.smartptr.shared.atomic] free functions are so inefficient and error-prone that they should not be used in cases where a proper `atomic_shared_ptr<T>` can do the same job.

It appears that `atomic_shared_ptr<T>` is a complete replacement. If so, the free functions should be deprecated to encourage use of the better tool.

2.3. Add `atomic_weak_ptr<T>`

Specify an `atomic_weak_ptrT<>` that offers the appropriate subset of operations supported by `weak_ptr`. Default construction initializes to `nullptr`.

2.4. Add `atomic_unique_ptr<T>`

Specify an `atomic_unique_ptrT<>` partial specialization that offers the appropriate subset of operations supported by `unique_ptr`, with the addition of `.get()` to enable getting a (non-owning) raw pointer without moving ownership out of the `atomic_unique_ptr`. Default construction initializes to `nullptr`.

3. Proposed Wording

The proposed wording below was derived as follows:

- Created the synopsis for `atomic_shared_ptr<T>` from a copy of the synopsis of `atomic<T*>`, removing all and only those functions that did not correspond to [util.smartptr.shared.atomic]. There were two kinds of removed functions: the `volatile`-qualified functions, and the pointer arithmetic functions.
- Created the synopsis of `atomic_unique_ptr<T>` and `atomic_weak_ptr<T>` from `atomic_shared_ptr<T>`.
- Added default initialization to null.
- Added `.get` for `atomic_unique_ptr<T>`.

3.1. Changes to 29.2

In 29.2, add the following synopsis:

// 29.6.x, operations on atomic smart pointer types

```
template <class T> struct atomic_unique_ptr;
template <class T> struct atomic_shared_ptr;
template <class T> struct atomic_weak_ptr;
```

3.2. Changes to 29.6

Add the following subclause 29.6.x:

29.6.x Operations on atomic smart pointer types [atomics.types.operations.smart-ptr]

```
template <class T> struct atomic_unique_ptr {
```

```
    bool is_lock_free() const noexcept;
    void store(unique_ptr<T>, memory_order = memory_order_seq_cst) noexcept;
    unique_ptr<T> load(memory_order = memory_order_seq_cst) const noexcept;
    T* get(memory_order = memory_order_seq_cst) const noexcept;
    operator unique_ptr<T>() const noexcept;
    unique_ptr<T> exchange(unique_ptr<T>, memory_order = memory_order_seq_cst)
    noexcept;
    bool compare_exchange_weak(unique_ptr<T>&, unique_ptr<T>, memory_order,
    memory_order) noexcept;
    bool compare_exchange_strong(unique_ptr<T>&, unique_ptr<T>, memory_order,
    memory_order) noexcept;
    atomic_unique_ptr() noexcept = default;
    constexpr atomic_unique_ptr( unique_ptr<T> ) noexcept;
    atomic_unique_ptr(const atomic_unique_ptr&) = delete;
    atomic_unique_ptr& operator=(const atomic_unique_ptr&) = delete;
    unique_ptr<T> operator=(unique_ptr<T>) noexcept;
};

template <class T> struct atomic_shared_ptr {
    bool is_lock_free() const noexcept;
    void store(shared_ptr<T>, memory_order = memory_order_seq_cst) noexcept;
    shared_ptr<T> load(memory_order = memory_order_seq_cst) const noexcept;
    operator shared_ptr<T>() const noexcept;
    shared_ptr<T> exchange(shared_ptr<T>, memory_order = memory_order_seq_cst)
    noexcept;
    bool compare_exchange_weak(shared_ptr<T>&, shared_ptr<T>, memory_order,
    memory_order) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, shared_ptr<T>, memory_order,
    memory_order) noexcept;
    atomic_shared_ptr() noexcept = default;
    constexpr atomic_shared_ptr( shared_ptr<T> ) noexcept;
    atomic_shared_ptr(const atomic_shared_ptr&) = delete;
    atomic_shared_ptr& operator=(const atomic_shared_ptr&) = delete;
    shared_ptr<T> operator=(shared_ptr<T>) noexcept;
};

template <class T> struct atomic_weak_ptr {
    bool is_lock_free() const noexcept;
    void store(weak_ptr<T>, memory_order = memory_order_seq_cst) noexcept;
    weak_ptr<T> load(memory_order = memory_order_seq_cst) const noexcept;
    operator weak_ptr<T>() const noexcept;
    weak_ptr<T> exchange(weak_ptr<T>, memory_order = memory_order_seq_cst) no-
    except;
```

```
    bool compare_exchange_weak(weak_ptr<T>&, weak_ptr<T>, memory_order,
    memory_order) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, weak_ptr<T>, memory_order,
    memory_order) noexcept;
    atomic_weak_ptr() noexcept = default;
    constexpr atomic_weak_ptr( weak_ptr<T> ) noexcept;
    atomic_weak_ptr(const atomic_weak_ptr&) = delete;
    atomic_weak_ptr& operator=(const atomic_weak_ptr&) = delete;
    weak_ptr<T> operator=(weak_ptr<T>) noexcept;
};
```

Change 29.6.5/4 as follows:

```
A::A() noexcept = default;
```

4 *Effects:* For `atomic_unique_ptr`, `atomic_shared_ptr`, and `atomic_weak_ptr`, initializes the atomic object to null. Otherwise, leaves the atomic object in an uninitialized state. [*Note:* These semantics ensure compatibility with C. —*end note*]

Insert the following in 29.6.5:

```
T* A::get(memory_order order = memory_order_seq_cst) const noexcept;
```

x *Requires:* The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

x *Effects:* Memory is affected according the value of `order`.

x *Returns:* Atomically returns the value pointed to by `this`.

4. Q&A

4.1. Q: Why not use the specialization syntax `atomic<*_ptr<T>>`? A: Because of SG1 direction and good technical arguments.

In Redmond (September 2014), SG1 expressed a strong preference for `atomic_*_ptr<T>` over `atomic<*_ptr<T>>` for several reasons. First, specializing `atomic<>` was considered by many to be strange and inconsistent, because `atomic<T>` requires T to be trivially copyable and this is not true in general for smart pointers. Second, specializing `atomic<>` seemed to give preference to the standard smart pointers over non-standard smart pointers, whereas just prepending `atomic_` sets a simple precedent for authors of non-standard smart pointers who want to provide atomic versions. Finally, specializing `atomic<>` makes it harder to put into the `std::experimental` namespace for a TS.

4.2. Q: What about user-defined smart pointers? A: Not in scope for this proposal.

Making user-defined smart pointers work with atomics generally requires the collaboration of the smart pointer author; see [3]. So this proposal is only about atomic use of the standard smart pointers, which already has partial support today and should be completed.

4.3. Q: Would allowing atomic<non-POD> enable atomic<any-smart-pointer>? A: Alas, no.

It seems the answer is no. Smart pointers are special; see [3].

4.4. Q: Well, could we still allow atomic<non-POD>, in addition to this proposal? A: Sure.

As a separate proposal, allowing atomic<non-POD> might still be interesting on its own merits, and not for smart pointers. The following notes capture some ideas that could help the author of such a separate proposal.

Here is one motivating example that Nat Goodspeed gave in [4]:

> Broadening the set of T for which atomic<T> is well-defined has the immediate effect of permitting better implementation decisions when you need atomic<> functionality.
>
> Case in point: N3877's violation_handler is defined this way:
>
> ```
> using violation_handler = void (*)(const assert_info&);
> ```
>
> In a language with lambdas and callable objects, do we really still want to restrict any standard interface to a classic C function pointer? Why wouldn't we choose std::function<void(const assert_info&)> instead?
>
> Ah: the reason surfaces in the reference implementation:
>
> ```
> std::atomic<violation_handler> handler{abort_handler};
> ```
>
> In this case, lack of support for std::atomic<std::function<>> has a direct and unfortunate impact on the library's interface.

However, there are also objections. As summarized by Jeffrey Yasskin:

> atomic<user-defined-non-POD> risks deadlocks because it involves calling user-defined code (copy constructors) under a lock that the user doesn't see.
>
> *[Anthony Williams' proposal in [5] for]* synchronized_value is probably a better way to do this, since it at least makes the fact of locking visible.

Lawrence Crowl responded to add:

> One of the reasons that shared_ptr locking is the way it is is to avoid a situation in which we weaken the precondition on the atomic template parameter that it be trivial, and hence have no risk of deadlock.
>
> That said, we could weaken the requirement so that the argument type only needs to be lock-free, or perhaps only non-recursively locking.
>
> However, while trivial makes for reasonably testable traits, I see no effective mechanism to test for the weaker property.

5. Appendix

I believe the following is a correct and ABA-safe implementation of a thread-safe singly linked list that supports insert/erase at the front only (like a stack) but also supports finding values in the list. It is written entirely in portable C++11, except only that it uses this paper's proposed `atomic<shared_ptr<Node>>`.

Note: This code can be written in C++11 as // commented to use the existing facilities, with the usability and performance drawbacks mentioned earlier in this paper.

```
template<typename T> class concurrent_stack {
    struct Node { T t; shared_ptr<Node> next; };
    atomic_shared_ptr<Node> head;
        // in C++11: remove "atomic_" and remember to use the special
        // functions every time you touch the variable
    concurrent_stack(concurrent_stack&) =delete;
    void operator=(concurrent_stack&) =delete;

public:
    concurrent_stack() =default;
    ~concurrent_stack() =default;
    class reference {
        shared_ptr<Node> p;
    public:
        reference(shared_ptr<Node> p_) : p{p_} { }
        T& operator* () { return  p->t; }
        T* operator->() { return &p->t; }
    };
    auto find( T t ) const {
        auto p = head.load(); // in C++11: atomic_load(&head)
        while( p && p->t != t )
            p = p->next;
        return reference(move(p));
    }
    auto front() const {
        return reference(head); // in C++11: atomic_load(&head)
    }
    void push_front( T t ) {
        auto p = make_shared<Node>();
        p->t = t;
        p->next = head; // in C++11: atomic_load(&head)
        while( !head.compare_exchange_weak(p->next, p) )
            { }
        // in C++11: atomic_compare_exchange_weak(&head, &p->next, p);
    }
```

```
    void pop_front() {
        auto p = head.load();
        while( p && !head.compare_exchange_weak(p, p->next) )
            { }
        // in C++11: atomic_compare_exchange_weak(&head, &p, p->next);
    }
};
```

6. Acknowledgments

Thanks to Hans Boehm, Lawrence Crowl, Peter Dimov, Gabriel Dos Reis, Olivier Giroux, Stephan T. Lavavej, Tony Van Eerd, Jonathan Wakely, Anthony Williams and Jeffrey Yasskin for their comments and feedback on this topic and/or on drafts of this paper.

7. References

[1] H. Boehm in c++std-lib-22167, LWG reflector thread "Shared Pointer Atomicity" (August 2008).

[2] SG1 reflector thread, "shared_ptr atomic access -> atomic<shared_ptr<>>" (March 2014). Starts at c++std-parallel-735, and see also etc.

[3] P. Dimov in c++std-parallel-754, "atomic<T> for non-PODs …" (March 2014). See also related nearby messages in that thread.

[4] N. Goodspeed in c++std-parallel-752, "atomic<T> for non-PODs …" (March 2014).

[5] A. Williams. N4033: "synchronized_value<T> for associating a mutex with a value" (May 2014).