Wording for Atomic Smart Pointers

Herb Sutter

Document #:	N4260
Date:	2014-11-08
Reply to:	Herb Sutter (hsutter@microsoft.com)

This is a revision of N4162 to apply SG1 feedback in Redmond and Urbana and add the directed proposed wording.

In 29.2, add the following synopsis:

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

```
template <class T, class D = default_delete<T>> struct atomic_unique_ptr;
template <class T> struct atomic_shared_ptr;
template <class T> struct atomic_weak_ptr;
```

In 29.5 at the end, add the following:

x There are class templates atomic_unique_ptr<T,D>, atomic_shared_ptr<T>, and atomic_weak_ptr<T>, with the corresponding non-atomic types unique_ptr<T,D>, shared_ptr<T>, and weak_ptr<T>. The template parameter T of these class templates may be an incomplete type.

Add the following subclause 29.6.x:

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

```
template <class T, class D = default_delete<T>> struct atomic_unique_ptr {
   bool is_lock_free() const noexcept;
   void store(unique_ptr<T,D>&&, memory_order = memory_order_seq_cst) noexcept;
   unique_ptr<T,D> load(memory_order = memory_order_seq_cst) noexcept;
   operator unique_ptr<T,D>() && noexcept;
   unique_ptr<T,D> exchange(unique_ptr<T,D>&&, memory_order = memory_order_seq_cst) noexcept;
   bool compare_exchange_weak(unique_ptr<T,D>&, unique_ptr<T,D>&&, memory_order, memory_order) noexcept;
   bool compare_exchange_weak(unique_ptr<T,D>&, unique_ptr<T,D>&&, memory_order = memory_order_seq_cst) noexcept;
   bool compare_exchange_strong(unique_ptr<T,D>&, unique_ptr<T,D>&&,
   memory_order, memory_order) noexcept;
```

```
    bool compare_exchange_strong(unique_ptr<T,D>&, unique_ptr<T,D>&&,
    memory_order = memory_order_seq_cst) noexcept;
    atomic_unique_ptr() noexcept = default;
    constexpr atomic_unique_ptr(unique_ptr<T,D>&&) noexcept;
    atomic_unique_ptr(const atomic_unique_ptr&) = delete;
    atomic_unique_ptr& operator=(const atomic_unique_ptr&) = delete;
    atomic_unique_ptr& operator=(unique_ptr<T,D>&&) 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>&, const shared_ptr<T>&, memory_order,
    memory_order) noexcept;
    bool compare_exchange_weak(shared_ptr<T>&, shared_ptr<T>&&, memory_order,
    memory_order) noexcept;
    bool compare_exchange_weak(shared_ptr<T>&, const shared_ptr<T>&, memory_order
    = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(shared_ptr<T>&, shared_ptr<T>&&, memory_order =
    memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, const shared_ptr<T>&,
    memory_order, memory_order) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, shared_ptr<T>&&, memory_order,
    memory_order) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, const shared_ptr<T>&,
    memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(shared_ptr<T>&, shared_ptr<T>&&, memory_order
    = memory_order_seq_cst) 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;
    atomic_shared_ptr& 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) noexcept;
    bool compare_exchange_weak(weak_ptr<T>&, const weak_ptr<T>&, memory_order,
    memory_order) noexcept;
    bool compare_exchange_weak(weak_ptr<T>&, weak_ptr<T>&&, memory_order,
    memory_order) noexcept;
    bool compare_exchange_weak(weak_ptr<T>&, const weak_ptr<T>&, memory_order =
    memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(weak_ptr<T>&, weak_ptr<T>&&, memory_order =
    memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, const weak_ptr<T>&, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, weak_ptr<T>&&, memory_order,
    memory_order) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, const weak_ptr<T>&, memory_order
    = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(weak_ptr<T>&, weak_ptr<T>&&, memory_order =
    memory_order_seq_cst) 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;
    atomic_weak_ptr& operator=(weak_ptr<T>) noexcept;
};
```

1 When any operation on an atomic_unique_ptr, atomic_shared_ptr, or atomic_weak_ptr causes an object to be destroyed or memory to be deallocated, that destruction or deallocation shall be sequenced after the changes to the atomic object's state. [*Note:* This prevents potential deadlock if the atomic smart pointer operation is not lock-free, such as by including a spinlock as part of the atomic object's state, and the destruction or the deallocation may attempt to acquire a lock. —*end note*] [*Note:* These types replace all known uses of the functions in [util.smartptr.shared.atomic]. – *end note*.] [*Note:* It is not known whether the atomic_unique_ptr<T,D>::compare_exchange_* functions have use cases. – *end note*.]

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 an empty value. Otherwise, leaves the atomic object in an uninitialized state. [*Note:* These semantics ensure compatibility with C. —*end note*]