Document Number: DXXXXR0
Date: 2019-03-22
Reply to: Daniel Sunderland
Sandia National Laboratories
dsunder@sandia.gov

Alternate `atomic_ref` for Non-Lockfree Types

```
git clone git@github.com:dsunder/draft.git dsunder-draft
cd dsunder-draft
git checkout atomic_ref_alt
```

Motivation

The current specification of `atomic_ref` implicitly requires that implementations use a lock array to implement `atomic_ref` when `is_lock_free()` is `false`. This is undesirable for at least two reasons. First, it forces implementations have a universally available lock array. Second, the length of the lock array and how locks are assigned to `atomic_ref` objects can have significant performance effects on the underlying code.

The following proposal should allow for more implementation freedom are improved control of the performance of `atomic_ref` on non lock-free types.

Proposed Wording

This font is used to provide guidance to the editors.

Make the following changes in [atomics.ref.generic].

```
namespace std {
  template<class T, class Lock = mutex> struct atomic_ref {
  private:
    T* ptr;       // exposition only
    Lock* lock; // exposition only
  public:
    using value_type = T;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr size_t required_alignment = implementation-defined;

    atomic_ref& operator=(const atomic_ref&) = delete;

    explicit atomic_ref(T&, Lock* = nullptr);
    atomic_ref(const atomic_ref&) noexcept;

    T operator=(T) const noexcept;
    operator T() const noexcept;

    bool is_lock_free() const noexcept;
    void store(T, memory_order = memory_order_seq_cst) const noexcept;
    T load(memory_order = memory_order_seq_cst) const noexcept;
    T exchange(T, memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_weak(T&, T,
                               memory_order, memory_order) const noexcept;
    bool compare_exchange_strong(T&, T,
                                 memory_order, memory_order) const noexcept;
    bool compare_exchange_weak(T&, T,
                               memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_strong(T&, T,
                                 memory_order = memory_order_seq_cst) const noexcept;
  };
}
```

Make the following changes in [atomics.ref.operations].

`atomic_ref(T& obj\added{, Lock* lk = nullptr});`

1 *Constraints:* The `Lock` type meets the *Cpp17BasicLockable* requirements.

2 ~~*Requires:*~~ *Expects:* The referenced object shall be aligned to `required_alignment` and type `Lock` meets the *Cpp17BasicLockable* requirements. If `is_lock_free()` is `false` then `lk` is not equivalent to `nullptr`. Furthermore, if `is_lock_free()` is `false`, then all `atomic_ref` objects which exist concurrently whose `*ptr` values are equivalent are constructed such that `*lock` refers to the same `Lock` object.

3 *Effects:* ~~Constructs an atomic reference that references the object.~~
Equivalent to:

```
ptr = &obj;
lock = is_lock_free() ? nullptr : lk;
```

4 *Throws:* Nothing.

```
atomic_ref(const atomic_ref& ref) noexcept;
```

5 *Effects:* ~~Constructs an atomic reference that references the object referenced by `ref`.~~ Equivalent to:

```
ptr = ref.obj;
lock = ref.lock;
```

```
void store(T desired, memory_order order = memory_order_seq_cst) const noexcept;
```

6 ~~*Requires:*~~ *Expects:* The `order` argument shall not be `memory_order_consume`, `memory_order_acquire`, nor `memory_order_acq_rel`.

7 *Effects:* If `is_lock_free()` is `true` ~~A~~atomically replaces the value referenced by `*ptr` with the value of `desired`. Otherwise, equivalent to:

```
lock->lock();
memcpy(ptr, &desired, sizeof(T));
lock->unlock();
```

Memory is affected according to the value of `order`.

```
T load(memory_order order = memory_order_seq_cst) const noexcept;
```

8 ~~*Requires:*~~ *Expects:* The `order` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

9 *Effects:* If `is_lock_free()` is `true` atomically returns the value referenced by `*ptr`. Otherwise, equivalent to:

```
T result;
lock->lock();
memcpy(&result, ptr, sizeof(T));
lock->unlock();
return result;
```

Memory is affected according to the value of `order`.

10 ~~*Returns:* Atomically returns the value referenced by `*ptr`.~~

```
T exchange(T desired, memory_order order = memory_order_seq_cst) const noexcept;
```

11 *Effects:* If `is_lock_free()` is `true` ~~A~~atomically replaces the value referenced by `*ptr` with `desired` and return the value previously referenced by `*ptr`. Otherwise, equivalent to:

```
T result;
lock->lock();
memcpy(&result, ptr, sizeof(T));
memcpy(ptr, &desired, sizeof(T));
lock->unlock();
return result;
```

Memory is affected according to the value of `order`. This operation is an atomic read-modify-write operation (**??**).

12 *Returns:* Atomically returns the value referenced by `*ptr` immediately before the effects.

```
bool compare_exchange_weak(T& expected, T desired,
                           memory_order success, memory_order failure) const noexcept;

bool compare_exchange_strong(T& expected, T desired,
                             memory_order success, memory_order failure) const noexcept;

bool compare_exchange_weak(T& expected, T desired,
                           memory_order order = memory_order_seq_cst) const noexcept;
```

```
bool compare_exchange_strong(T& expected, T desired,
                             memory_order order = memory_order_seq_cst) const noexcept;
```

13 *Requires:* The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

14 *Effects:* When only one `memory_order` argument is supplied, the value of `success` is `order`, and the value of `failure` is `order` except that a value of `memory_order_acq_rel` shall be replaced by the value `memory_order_acquire` and a value of `memory_order_release` shall be replaced by the value `memory_order_relaxed`.

If `is_lock_free()` is `true` ~~R~~retrieves the value in `expected`. It then atomically compares the value representation of the value referenced by `*ptr` for equality with that previously retrieved from `expected`, and if `true`, replaces the value referenced by `*ptr` with that in `desired`. If and only if the comparison is `true`, memory is affected according to the value of `success`, and if the comparison is `false`, memory is affected according to the value of `failure`.

When only one `memory_order` argument is supplied, the value of `success` is `order`, and the value of `failure` is `order` except that a value of `memory_order_acq_rel` shall be replaced by the value `memory_order_acquire` and a value of `memory_order_release` shall be replaced by the value `memory_order_relaxed`. If and only if the comparison is `false` then, after the atomic operation, the value in `expected` is replaced by the value read from the value referenced by `*ptr` during the atomic comparison. If the operation returns `true`, these operations are atomic read-modify-write operations (**??**) on the value referenced by `*ptr`. Otherwise, these operations are atomic load operations on that memory.

15 If `is_lock_free()` is `false` equivalent to:

```
T old;
bool result;
lock->lock();
memcpy(&old, ptr, sizeof(T));
result = 0 == memcmp(ptr, &old, sizeof(T);
if (result)
  memcpy(ptr, &desired, sizeof(T));
else
  memcpy(&expected, &old, sizeof(T));
lock->unlock();
return result;
```

16 *Returns:* The result of the comparison.

17 *Remarks:* A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `ptr` are equal, it may return `false` and store back to `expected` the same memory contents that were originally there. [*Note*: This spurious failure enables implementation of compare-and-exchange on a broader class of machines, e.g., load-locked store-conditional machines. A consequence of spurious failure is that nearly all uses of weak compare-and-exchange will be in a loop. When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable. — *end note*]

Make the following changes in [atomics.ref.int].

```
namespace std {
  template<Lock> struct atomic_ref<integral, Lock> {
  private:
    integral* ptr;        // exposition only
    Lock* lock; // exposition only
  public:
    using value_type = integral;
    using difference_type = value_type;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr size_t required_alignment = implementation-defined;

    atomic_ref& operator=(const atomic_ref&) = delete;
```

```
    explicit atomic_ref(integral&, Lock* = nullptr);
    atomic_ref(const atomic_ref&) noexcept;

    integral operator=(integral) const noexcept;
    operator integral() const noexcept;

    bool is_lock_free() const noexcept;
    void store(integral, memory_order = memory_order_seq_cst) const noexcept;
    integral load(memory_order = memory_order_seq_cst) const noexcept;
    integral exchange(integral,
                      memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_weak(integral&, integral,
                               memory_order, memory_order) const noexcept;
    bool compare_exchange_strong(integral&, integral,
                                 memory_order, memory_order) const noexcept;
    bool compare_exchange_weak(integral&, integral,
                               memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_strong(integral&, integral,
                                 memory_order = memory_order_seq_cst) const noexcept;

    integral fetch_add(integral,
                       memory_order = memory_order_seq_cst) const noexcept;
    integral fetch_sub(integral,
                       memory_order = memory_order_seq_cst) const noexcept;
    integral fetch_and(integral,
                       memory_order = memory_order_seq_cst) const noexcept;
    integral fetch_or(integral,
                      memory_order = memory_order_seq_cst) const noexcept;
    integral fetch_xor(integral,
                       memory_order = memory_order_seq_cst) const noexcept;

    integral operator++(int) const noexcept;
    integral operator--(int) const noexcept;
    integral operator++() const noexcept;
    integral operator--() const noexcept;
    integral operator+=(integral) const noexcept;
    integral operator-=(integral) const noexcept;
    integral operator&=(integral) const noexcept;
    integral operator|=(integral) const noexcept;
    integral operator^=(integral) const noexcept;
  };
}
```

Make the following changes in [atomics.ref.float].

```
namespace std {
  template<Lock> struct atomic_ref<floating-point, Lock> {
  private:
    floating-point* ptr;  // exposition only
    Lock* lock; // exposition only
  public:
    using value_type = floating-point;
    using difference_type = value_type;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr size_t required_alignment = implementation-defined;

    atomic_ref& operator=(const atomic_ref&) = delete;

    explicit atomic_ref(floating-point&, Lock* = nullptr);
    atomic_ref(const atomic_ref&) noexcept;

    floating-point operator=(floating-point) noexcept;
    operator floating-point() const noexcept;
```

```
    bool is_lock_free() const noexcept;
    void store(floating-point, memory_order = memory_order_seq_cst) const noexcept;
    floating-point load(memory_order = memory_order_seq_cst) const noexcept;
    floating-point exchange(floating-point,
                            memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_weak(floating-point&, floating-point,
                               memory_order, memory_order) const noexcept;
    bool compare_exchange_strong(floating-point&, floating-point,
                                 memory_order, memory_order) const noexcept;
    bool compare_exchange_weak(floating-point&, floating-point,
                               memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_strong(floating-point&, floating-point,
                                 memory_order = memory_order_seq_cst) const noexcept;

    floating-point fetch_add(floating-point,
                             memory_order = memory_order_seq_cst) const noexcept;
    floating-point fetch_sub(floating-point,
                             memory_order = memory_order_seq_cst) const noexcept;

    floating-point operator+=(floating-point) const noexcept;
    floating-point operator-=(floating-point) const noexcept;
  };
}
```

Make the following changes in [atomics.ref.pointer].

```
namespace std {
  template<class T, Lock> struct atomic_ref<T*, Lock> {
  private:
    T** ptr;                // exposition only
    Lock* lock; // exposition only
  public:
    using value_type = T*;
    using difference_type = ptrdiff_t;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr size_t required_alignment = implementation-defined;

    atomic_ref& operator=(const atomic_ref&) = delete;

    explicit atomic_ref(T*&, Lock* = nullptr);
    atomic_ref(const atomic_ref&) noexcept;

    T* operator=(T*) const noexcept;
    operator T*() const noexcept;

    bool is_lock_free() const noexcept;
    void store(T*, memory_order = memory_order_seq_cst) const noexcept;
    T* load(memory_order = memory_order_seq_cst) const noexcept;
    T* exchange(T*, memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_weak(T*&, T*,
                               memory_order, memory_order) const noexcept;
    bool compare_exchange_strong(T*&, T*,
                                 memory_order, memory_order) const noexcept;
    bool compare_exchange_weak(T*&, T*,
                               memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_strong(T*&, T*,
                                 memory_order = memory_order_seq_cst) const noexcept;

    T* fetch_add(difference_type, memory_order = memory_order_seq_cst) const noexcept;
    T* fetch_sub(difference_type, memory_order = memory_order_seq_cst) const noexcept;

    T* operator++(int) const noexcept;
    T* operator--(int) const noexcept;
    T* operator++() const noexcept;
```

```
    T* operator--() const noexcept;
    T* operator+=(difference_type) const noexcept;
    T* operator-=(difference_type) const noexcept;
  };
}
```