Document Number: DXXXXR0
Date: 2019-04-01
Reply to: Daniel Sunderland
Sandia National Laboratories
dsunder@sandia.gov

Improving `atomic_ref` of Non Lock-free Types

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

Motivation

Many implementations cannot support the current specification of `atomic_ref` for non lock-free types. Also, using an `atomic_ref` of an insufficiently aligned object of a lock-free type can fail silently, leading to subtle and difficult to debug errors.

There are proposals *citation needed* to remove non lock-free `atomic_ref` from freestanding. However, since implementations are not required to support lock-free atomic operations, these proposals remove the ability of using `atomic_ref` in portable code.

The following proposal extends the `atomic_ref` specification to allow more implementations to fully support `atomic_ref` on objects which cannot be lock-free. This proposal preserves the existing behavior of `atomic_ref` in implementations which can support the current specification while enabling additional implementations.

Question to reviewers: Should `atomic_ref` also provide deduction guides?

Proposed Wording

This font is used to provide guidance to the editors.

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

```
namespace std {
  struct atomic_ref_assume_lock_free_t    {};
  struct atomic_ref_assume_no_user_lock_t {};
  struct atomic_ref_prefer_user_lock_t    {};

  inline constexpr atomic_ref_assume_lock_free_t  atomic_ref_assume_lock_free {};
  inline constexpr atomic_ref_prefer_user_lock_t  atomic_ref_prefer_user_lock {};
}
```

```
namespace std {

  template<class T, class LockT=unspecified> struct atomic_ref {
  private:
    T* ptr;         // exposition only
    lock_type* ulock; // exposition only
  public:
    using lock_type = LockT;
    using value_type = T;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_lock_free_alignment = implementation-defined;

    static bool is_lock_free(const T &) noexcept;
    static bool requires_user_lock(const T &) noexcept;

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

    explicit atomic_ref(T&) noexcept;
    atomic_ref(T&, atomic_ref_assume_lock_free_t) noexcept;

    atomic_ref(T&, lock_type&) noexcept;
    atomic_ref(T&, lock_type&, atomic_ref_prefer_user_lock_t) noexcept;

    template<class Select> atomic_ref(T&, ranges::RandomAccessRange<lock_type>,
     Select = unspecified) noexcept;
    template<class Select> atomic_ref(T&, ranges::RandomAccessRange<lock_type>,
     atomic_ref_prefer_user_lock_t, Select = unspecified) noexcept;

    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].

1 The type `lock_type` can be `atomic_ref_assume_lock_free_t`, `atomic_ref_assume_no_user_lock_t`, or the type `lock_type` meets the *Cpp17BasicLockable* requirements. If `lock_type` is equivalent to `atomic_ref_assume_lock_free_t` it is a diagnosable error if the implementation cannot provide lock-free atomic operations for objects of type `T` aligned to `required_lock_free_alignment`. If `lock_type` is equivalent to `atomic_ref_assume_no_user_lock_t` it is a diagnosable error if the implementation cannot provide atomic operations for objects of type `T` which do not require a user provided lock.

2 `atomic_ref` instances referencing the same value of `ptr` and `ulock` are called *equivalent*. Concurrent access to the same value through equivalent `atomic_ref` instances does not create a data race (**??**). [*Note*: Concurrent access to the value directly, or through a non-equivalent `atomic_ref` instance, can introduce a data race. — *end note*]

3 For all `atomic_ref` member functions excluding static methods, constructors, the destructor, and `is_lock_free()` one of the following statements is `true`:

(3.1) — If `ulock` points to a valid `lock_type` object which meets the *Cpp17BasicLockable* requirements then the implementation will use `ulock` to atomically perform these methods. If `lock_type` does not meet the *Cpp17BasicLockable* requirements then these methods can introduce a data race.

(3.2) — Otherwise, if `ulock` is equivalent to `nullptr` then the implementation will ensure that these methods happen atomically.

`static constexpr bool is_always_lock_free;`

4 The static data member `is_always_lock_free` is `true` if the `atomic_ref` type's operations are always lock-free on objects aligned to `required_lock_free_alignment`, and `false` otherwise.

`static constexpr size_t required_lock_free_alignment;`

5 The alignment required for an object to be referenced lock-free by an atomic reference, which is at least `alignof(T)`. If the implementation cannot support lock-free operations on objects of type `T` then `required_lock_free_alignment` is 0.

6 [*Note*: Hardware could require an object referenced by an `atomic_ref` to have stricter alignment (**??**) than other objects of type `T`. Furthermore, whether operations on an `atomic_ref` are lock-free could depend on the alignment of the referenced object. For example, lock-free operations on `std::complex<double>` could be supported only if aligned to `2*alignof(double)`. — *end note*]

`static constexpr bool never_requires_user_lock;`

7 Is `true` if an implementation never requires the user to provide a lock for objects of type `T` and `false` otherwise.

`static is_lock_free(T& obj) noexcept;`

Returns: Returns `true` if atomic operations on the object referenced by `obj` can be lock-free or if the `lock_type` type is equivalent to `atomic_ref_assume_lock_free_t`.

```
static requires_user_lock(T& obj) noexcept;
```

Returns: Returns `false` if `lock_type` is equivalent to either `atomic_ref_assume_lock_free_t` or `atomic_ref_assume_no_user_lock_t`. Otherwise, returns `true` if `atomic_ref` requires the user to provide a valid reference to a `lock_type` object or a non-empty `ranges::RandomAccessRange<lock_type>` when constructing an `atomic_ref` from `obj`.

```
explicit atomic_ref(T& obj) noexcept;
```

8 ~~*Requires:* The referenced object shall be aligned to `required_lock_free_alignment`~~. *Expects:* `requires_user_lock(obj)` is `false`.

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

```
ptr = std::addressof(obj);
ulock = nullptr;
```

10 *Throws:* Nothing.

11 *Remarks:* The implementation calls `terminate()` if `requires_user_lock(obj)` is `true`.

```
atomic_ref(T& obj, atomic_ref_assume_lock_free_t);
```

12 *Expects:* `is_lock_free(obj)` is `true`.

13 *Effects:* Equivalent to:

```
ptr = std::addressof(obj);
ulock = nullptr;
```

14 *Throws:* Nothing.

15 *Remarks:* The implementation can call `terminate` if `is_lock_free(obj)` is `false`.

```
atomic_ref(T& obj, lock_type& lk);
```

16 *Effects:* Equivalent to:

```
ptr = std::addressof(obj);
ulock = requires_user_lock(obj) ? std::addressof(lk) : nullptr;
```

17 *Throws:* Nothing.

```
atomic_ref(T& obj, lock_type& lk, atomic_ref_prefer_user_lock_t);
```

18 *Effects:* Equivalent to:

```
ptr = std::addressof(obj);
ulock = is_lock_free(obj) ? nullptr : std::addressof(lk);
```

19 *Throws:* Nothing.

```
template <class Select>
atomic_ref(T& obj, ranges::RandomAccessRange<lock_type> lks,
  Select sel = unspecified );
```

20 *Mandates:* `INVOKE(sel(const T&, ranges::RandomAccessRange<lock_type>))` returns a reference to a `lock_type` object.

21 *Expects:* If `requires_user_lock(obj)` is `true` then the range `lks` is non-empty and `INVOKE(sel(obj, lks))` returns a reference to the same `lock_type` object in the range `lks` for all `obj` which reference the same value.

22 *Effects:* Equivalent to:

```
ptr = std::addressof(obj);
ulock = requires_user_lock(obj) ?
  std::addressof(INVOKE(sel(obj, lks))) : nullptr;
```

23 *Throws:* Nothing.

24 *Remarks:* The implementation can call `terminate` if the range `lks` is empty and `requires_user_lock(obj)` is `true`.

```
template <class Select>
atomic_ref(T& obj, ranges::RandomAccessRange<lock_type> lks,
  atomic_ref_prefer_user_lock_t, Sel sel = unspecified );
```

25 *Mandates:* `INVOKE(sel(const T&, ranges::RandomAccessRange<lock_type>))` returns a reference to a `lock_type` object.

26 *Expects:* The range `lks` is non-empty and `INVOKE(sel(obj, lks))` returns a reference to the same `lock_type` object in the range `lks` for all `obj` which reference the same value.

27 *Effects:* Equivalent to:

```
ptr = std::addressof(obj);
ulock = is_lock_free(obj) ?
  nullptr : std::addressof(INVOKE(sel(obj, lks)));
```

28 *Throws:* Nothing.

29 *Remarks:* The implementation can call `terminate` if the range `lks` is empty.

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

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

```
ptr = ref.obj;
ulock = ref.ulock;
```

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

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

32 *Effects:* Atomically replaces the value referenced by `*ptr` with the value of `desired`. Memory is affected according to the value of `order`.

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

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

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

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

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

35 *Effects:* Atomically replaces the value referenced by `*ptr` with `desired`. Memory is affected according to the value of `order`. This operation is an atomic read-modify-write operation (**??**).

36 *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;
```

37 ~~*Requires:*~~ *Expects:* The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

38 *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`.

Equivalent to atomically performing the following:

```
alignas(T) std::byte old[sizeof(T)];
```

```
memcpy(old, ptr, sizeof(T));
bool result = 0 == memcmp(std::addressof(expected), old, sizeof(T));
if (result) memcpy(ptr, std::addressof(desired), sizeof(T));
else memcpy(std::addressof(expected), old, sizeof(T));
return result;
```

39 Let `R` be the return value of the operation. If and only if `R` is `true`, memory is affected according to the value of `success`, and if `R` is `false`, memory is affected according to the value of `failure`.

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`. 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~~R is `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.

40 *Returns:* The result of the comparison.

41 *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<class LockT> struct atomic_ref<integral, LockT> {
  private:
    integral* ptr;          // exposition only
    lock_type* ulock; // exposition only
  public:
    using lock_type = LockT;
    using value_type = integral;
    using difference_type = value_type;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_lock_free_alignment = implementation-defined;

    static bool is_lock_free(const T &) noexcept;
    static bool requires_user_lock(const T &) noexcept;

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

    explicit atomic_ref(T&) noexcept;
    atomic_ref(integral&, atomic_ref_assume_lock_free_t) noexcept;

    atomic_ref(integral&, lock_type&) noexcept;
    atomic_ref(integral&, lock_type&, atomic_ref_prefer_user_lock_t) noexcept;

    template<class Select> atomic_ref(integral&, ranges::RandomAccessRange<lock_type>,
     Select = unspecified) noexcept;
    template<class Select> atomic_ref(integral&, ranges::RandomAccessRange<lock_type>,
     atomic_ref_prefer_user_lock_t, Select = unspecified) noexcept;
```

```
    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<class LockT> struct atomic_ref<floating-point, LockT> {
  private:
    floating-point* ptr;  // exposition only
    lock_type* ulock; // exposition only
  public:
    using lock_type = LockT;
    using value_type = floating-point;
    using difference_type = value_type;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_lock_free_alignment = implementation-defined;

    static bool is_lock_free(const T &) noexcept;
    static bool requires_user_lock(const T &) noexcept;

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

    explicit atomic_ref(T&) noexcept;
    atomic_ref(floating-point&, atomic_ref_assume_lock_free_t) noexcept;
```

```
    atomic_ref(floating-point&, lock_type&) noexcept;
    atomic_ref(floating-point&, lock_type&, atomic_ref_prefer_user_lock_t) noexcept;

    template<class Select> atomic_ref(floating-point&, ranges::RandomAccessRange<lock_type>,
     Select = unspecified) noexcept;
    template<class Select> atomic_ref(floating-point&, ranges::RandomAccessRange<lock_type>,
     atomic_ref_prefer_user_lock_t, Select = unspecified) noexcept;

    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,class LockT> struct atomic_ref<T*, LockT> {
  private:
    T** ptr;              // exposition only
    lock_type* ulock; // exposition only
  public:
    using lock_type = LockT;
    using value_type = T*;
    using difference_type = ptrdiff_t;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_lock_free_alignment = implementation-defined;

    static bool is_lock_free(const T &) noexcept;
    static bool requires_user_lock(const T &) noexcept;

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

    explicit atomic_ref(T&) noexcept;
    atomic_ref(T*, atomic_ref_assume_lock_free_t) noexcept;

    atomic_ref(T*, lock_type&) noexcept;
    atomic_ref(T*, lock_type&, atomic_ref_prefer_user_lock_t) noexcept;
```

```
    template<class Select> atomic_ref(T*, ranges::RandomAccessRange<lock_type>,
     Select = unspecified) noexcept;
    template<class Select> atomic_ref(T*, ranges::RandomAccessRange<lock_type>,
     atomic_ref_prefer_user_lock_t, Select = unspecified) noexcept;

    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;
  };
}
```