Document Number: P1632R0
Date: 2019-04-24
Reply to: Dan Sunderland
Sandia National Laboratories
dsunder@sandia.gov
Authors:

Dan Sunderland	dsunder@sandia.gov
Billy O'Neal	bion@microsoft.com
Michał Dominiak	griwes@griwes.info
Nevin Liber	nliber@anl.gov

Improving `atomic_ref` for Non Lock-free Types

The latex for this proposal is available at:

```
git clone https://github.com/ORNL/cpp-proposals-pub.git
cd cpp-proposals-pub
git checkout alt-atomic-ref
cd P1632
```

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Motivation

Many implementations do not 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 are not lock-free. This proposal preserves the existing behavior of `atomic_ref` in implementations which can support the current specification while enabling additional implementations.

Proposed Wording

[Editor's note: Make the following changes in [atomics.ref.generic].]

```
namespace std {
  struct atomic_ref_assume_lock_free_t {
    explicit constexpr atomic_ref_assume_lock_free_t() noexcept = default;
  };
  inline constexpr atomic_ref_assume_lock_free_t atomic_ref_assume_lock_free{};

  struct atomic_ref_prefer_user_lock_t {
    explicit constexpr atomic_ref_prefer_user_lock_t() noexcept = default;
  };
  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
    LockT* ulock; // exposition only
  public:
    using lock_type = LockT;
    using value_type = T;
    static constexpr bool is_always_lock_free = implementation-defined;
    static constexpr size_t required_lock_free_alignment = implementation-defined;
    static constexpr bool can_be_lock_free = required_lock_free_alignment > 0u;

    static constexpr bool never_requires_user_lock = 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;

    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;
  };

  template<class T>
  atomic_ref(T&) -> atomic_ref<T>;

  template<class T>
  atomic_ref(T&, atomic_ref_assume_lock_free_t) ->
    atomic_ref<T, atomic_ref_assume_lock_free_t>;

  template<class T, class Lock>
  atomic_ref(T&, Lock&) -> atomic_ref<T, Lock>;

  template<class T, class Lock>
  atomic_ref(T&, Lock&, atomic_ref_prefer_user_lock_t) ->
    atomic_ref<T, Lock>;
}
```

[Editor's note: Make the following changes in [atomics.ref.operations].]

1 The type `lock_type` can either:

- (1.1) — be equivalent to `atomic_ref_assume_lock_free_t`, or
- (1.2) — the type `lock_type` meets the *Cpp17Lockable* requirements.

Dianostics required if `lock_type` is equivalent to `atomic_ref_assume_lock_free_t` and the implementation does not provide lock-free atomic operations for objects of type `T` aligned to `required_lock_free_alignment`.

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()` the following conditional is `true`:

- (3.1) — If `ulock` points to a valid `lock_type` object which meets the *Cpp17Lockable* requirements then the implementation will use `ulock` to atomically perform these methods.
- (3.2) — If `ulock` is equivalent to `nullptr` and `requires_user_lock(obj)` is `false` then the implementation ensures that these methods happen atomically.
- (3.3) — Otherwise, the use of any of these methods can introduce a data race.

`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, 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 does not 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 `atomic_ref_assume_lock_free_t` or does not require the user to provide a valid reference to a `lock_type` object. Otherwise, returns `true` if `atomic_ref` requires the user to provide a valid reference to a `lock_type` object 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`~~.

9 *Effects:* ~~Constructs an atomic reference that references the object.~~
If `requires_user_lock(obj)` is `true` calls `std::terminate()`. Otherwise, equivalent to:

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

10 *Throws:* Nothing.

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

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

12 *Effects:* Equivalent to:

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

13 *Throws:* Nothing.

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

14 *Effects:* Equivalent to:

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

15 *Throws:* Nothing.

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

16 *Effects:* Equivalent to:

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

17 *Throws:* Nothing.

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

18 *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;
```

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

20 *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;
```

21 ~~*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`.

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

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

23 *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 (??).

24 *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;
```

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

26 *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(std::max(sizeof(T), required_lock_free_alignment)) 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;
```

27 If return value of the operation is `true`, memory is affected according to the value of `success` and this operation is an atomic read-modify-write operation (??) on the value referenced by `*ptr`. Otherwise memory is affected according to the value of `failure` and this operation is an atomic load operation on `*ptr`.

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 `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.

28 *Returns:* The result of the comparison.

29 *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*]

[Editor's note: Make the following changes in [atomics.ref.int].]

```
namespace std {
  template<class LockT> struct atomic_ref<integral, LockT> {
  private:
    integral* ptr; // exposition only
    LockT* 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 size_t required_lock_free_alignment = implementation-defined;
    static constexpr bool can_be_lock_free = required_lock_free_alignment > 0u;

    static constexpr bool never_requires_user_lock = 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;

    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;
  };
}
```

[Editor's note: 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
    LockT* 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 size_t required_lock_free_alignment = implementation-defined;
    static constexpr bool can_be_lock_free = required_lock_free_alignment > 0u;

    static constexpr bool never_requires_user_lock = 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;

    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;
  };
}
```

[Editor's note: 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
    LockT* 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 size_t required_lock_free_alignment = implementation-defined;
    static constexpr bool can_be_lock_free = required_lock_free_alignment > 0u;

    static constexpr bool never_requires_user_lock = 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;

    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;
  };
}
```