Document Number:	DXXXXR0
Date:	2019-03-26
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

Enable implementations of `atomic_ref` which do not require a global lock array while perserving current behavior as much as possible.

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_perfer_user_lock_t    {};

  inline constexpr atomic_ref_assume_lock_free_t  atomic_ref_assume_lock_free {};
  inline constexpr atomic_ref_perfer_user_lock_t  atomic_ref_perfer_user_lock {};

  struct atomic_ref_assume_no_user_locks {};
}
```

```
namespace std {

  template<class T, class Lock = mutex> struct atomic_ref {
  private:
    T* ptr;       // exposition only
    Lock* ulock; // exposition only
  public:
    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;

    explicit atomic_ref(T&) noexcept(!is_same_v<Lock,atomic_ref_assume_no_user_locks>
      || implementation-defined);

    atomic_ref(T&, atomic_ref_assume_lock_free_t) noexcept;

    atomic_ref(T&, Lock&) noexcept;
    atomic_ref(T&, Lock&, atomic_ref_perfer_user_lock_t) noexcept;

    atomic_ref(T&, ranges::RandomAccessRange<Lock>) noexcept;
    atomic_ref(T&, ranges::RandomAccessRange<Lock>, atomic_ref_perfer_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;
  };
}
```

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

`static constexpr bool is_always_lock_free;`

1 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;`

2 The alignment required for an object to be referenced lock-free by an atomic reference, which is at least `alignof(T)`.

3 [*Note*: Hardware could require an object referenced by an `atomic_ref` to have stricter alignment (**??**) than other objects of type `T`. Further, 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;`

4 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 is equivalent to `atomic_ref_assume_lock_free_t`.

`static requires_user_lock(T& obj); noexcept`

Returns: Returns `false` if the `Lock` is equivalent to `atomic_ref_assume_lock_free_t` or `Lock` is equivalent to `atomic_ref_assume_no_user_lock_t`. Otherwise, returns `true` if `atomic_ref` requires the user to provide a valid pointer to a `Lock` object when constructing an `atomic_ref` from `obj`.

```
\begin{removedblock}
explicit atomic_ref(T&) noexcept;
\end{removedblock}
\begin{addblock}
explicit atomic_ref(T&) noexcept(!is_same_v<Lock,atomic_ref_assume_no_user_locks>
  || implementation-defined);
\end{addblock}
```

5 ~~*Requires:* The referenced object shall be aligned to `required_lock_free_alignment`~~. *Expects:* `requires_user_lock(obj)` is `false`, and this constructor is used for all `atomic_ref` objects which reference the object referenced by `obj` concurrently.

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

```
ptr = &obj;
ulock = nullptr;
```

7 *Throws:* ~~Nothing~~ *implementation-defined.* If `requires_user_lockobj` could be `true`, an implementation throws `???_error` if `requires_user_lock` is `true`;

`atomic_ref(T& obj, atomic_ref_assume_lock_free_t);`

8 *Expects:* The object referenced by `obj` is be aligned to `required_lock_free_alignment` and this constructor is used for all `atomic_ref` objects which reference the object referenced by `obj` concurrently.

9 *Effects:* Equivalent to:

```
ptr = &obj;
ulock = nullptr;
```

10 *Throws:* Nothing.

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

11 *Expects:* The type `Lock` either meets the *Cpp17BasicLockable* requirements or is equivalent to `atomic_ref_assume_lock_free_t`. For all `atomic_ref` objects that concurrently reference the object referenced by `obj` either `requires_user_lock(obj)` is `false` or the following conditions are true:

(11.1) — `lk` references the same `Lock` object, and

(11.2) — this constructor is used for all the `atomic_ref` objects mentioned above.

12 *Effects:* Equivalent to:

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

13 *Throws:* Nothing.

`atomic_ref(T& obj, Lock& lk, atomic_ref_perfer_user_lock_t);`

14 *Expects:* The type `Lock` either meets the *Cpp17BasicLockable* requirements, is equivalent to `atomic_ref_assume_lock_free_t`, or `requires_user_lock(obj)` is `false`. For all `atomic_ref` objects that concurrently reference the object referenced by `obj` both of the following conditions are true:

(14.1) — `lk` references the same `Lock` object, and

(14.2) — this constructor is used for all the `atomic_ref` objects mentioned above.

15 *Effects:* Equivalent to:

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

16 *Throws:* Nothing.

`atomic_ref(T& obj, ranges::RandomAccessRange<Lock> lks);`

17 *Expects:* The type `Lock` either meets the *Cpp17BasicLockable* requirements, is equivalent to `atomic_ref_assume_lock_free_t`, or `requires_user_lock(obj)` is `false`. For all `atomic_ref` objects that concurrently reference the object referenced by `obj` either `requires_user_lock(obj)` is `false` or the following conditions are true:

(17.1) — `lks` is a non-empty range of `Lock` objects,

(17.2) — `lks` represents equivalent ranges of `Lock` objects, and

(17.3) — this constructor is used for all the `atomic_ref` objects mentioned above.

18 *Effects:* Equivalent to:

```
ptr = &obj;
ulock = requires_user_lock(obj)) ? ranges::begin(lks)[std::hash(&obj)ranges::size(lks)] :  nullptr;
```

19 *Throws:* Nothing.

`atomic_ref(T& obj, ranges::RandomAccessRange<Lock> lks, atomic_ref_perfer_user_lock_t);`

20 *Expects:* The type `Lock` either meets the *Cpp17BasicLockable* requirements or is equivalent to `atomic_ref_assume_lock_free_t`. For all `atomic_ref` objects that concurrently reference the object referenced by `obj` all of the following conditions are true:

(20.1) — `lks` is a non-empty range of `Lock` objects,

(20.2) — `lks` represents equivalent ranges of `Lock` objects, and

(20.3) — this constructor is used for all the `atomic_ref` objects mentioned above.

21 *Effects:* Equivalent to:

```
ptr = &obj;
ulock = is_lock_free(obj) ? nullptr : ranges::begin(lks)[std::hash(&obj)ranges::size(lks)];
```

22 *Throws:* Nothing.

`atomic_ref(const atomic_ref& ref) noexcept;`

23 *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;
```

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

25 ~~Atomically replaces the value referenced by `*ptr` with the value of `desired`~~. *Effects:* If `ulock` is equivalent to `nullptr` then the operation is equivalent to atomically performing the following:

```
memcpy(ptr, &desired, sizeof(T));
```

Otherwise, equivalent to:

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

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

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

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

27 *Effects:* If `ulock` is equivalent to `nullptr` then the operation is equivalent to atomically performing the following:

```
T result;
memcpy(&result, ptr, sizeof(T));
return result;
```

Otherwise, equivalent to:

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

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

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

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

29 ~~Atomically replaces the value referenced by `*ptr` with `desired`.~~ *Effects:* If `ulock` is equivalent to `nullptr` then the operation is equivalent to atomically performing the following:

```
T result;
memcpy(&result, ptr, sizeof(T));
memcpy(ptr, &desired, sizeof(T));
return result;
```

Otherwise, equivalent to:

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

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

30 *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;
```

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

32 *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 `ulock` is equivalent to `nullptr` then the operation is equivalent to atomically performing the following:

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

Otherwise, equivalent to:

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

33 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.

34 *Returns:* The result of the comparison.

35 *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* ulock; // exposition only
  public:
    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(integral&) noexcept(implementation-defined);
    atomic_ref(integral&, atomic_ref_assume_lock_free_t) noexcept;

    atomic_ref(integral&, Lock&) noexcept;
    atomic_ref(integral&, Lock&, atomic_ref_perfer_user_lock_t) noexcept;

    atomic_ref(integral&, ranges::RandomAccessRange<Lock>) noexcept;
    atomic_ref(integral&, ranges::RandomAccessRange<Lock>, atomic_ref_perfer_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;
  };
}
```

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* ulock; // exposition only
  public:
    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(floating-point&) noexcept(implementation-defined);
    atomic_ref(floating-point&, atomic_ref_assume_lock_free_t) noexcept;

    atomic_ref(floating-point&, Lock&) noexcept;
    atomic_ref(floating-point&, Lock&, atomic_ref_perfer_user_lock_t) noexcept;

    atomic_ref(floating-point&, ranges::RandomAccessRange<Lock>) noexcept;
    atomic_ref(floating-point&, ranges::RandomAccessRange<Lock>, atomic_ref_perfer_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;
  };
}
```

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* ulock; // exposition only
  public:
    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(implementation-defined);
    atomic_ref(T*, atomic_ref_assume_lock_free_t) noexcept;

    atomic_ref(T*, Lock&) noexcept;
    atomic_ref(T*, Lock&, atomic_ref_perfer_user_lock_t) noexcept;

    atomic_ref(T*, ranges::RandomAccessRange<Lock>) noexcept;
    atomic_ref(T*, ranges::RandomAccessRange<Lock>, atomic_ref_perfer_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;
  };
}
```