Document Number: DXXXXR0
Date: 2019-03-23

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 {
  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 bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_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&, 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].

`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_alignment`, and `false` otherwise.

`static constexpr bool never_requires_user_lock;`

2 The static data member `never_requires_user_lock` is `true` if the `atomic_ref` type's operations never require the user to provide a valid pointer to a `Lock` object.

`static is_lock_free(T& obj); noexcept`

Returns: Returns `true` if atomic operations on the object referenced by `obj` can be lock-free.

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

Returns: Returns `true` if `atomic_ref` requires the user to provide a valid pointer to a `Lock` object when constructing an `atomic_ref` from `obj`. [*Note*: Implementations could provide ways for `atomic_ref` to use implementation defined locking mechanisms when `is_lock_free(obj)` is `false`. — *end note*]

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

3 ~~*Requires:* The referenced object shall be aligned to `required_alignment`~~ *Expects:* The type `Lock` meets the *Cpp17BasicLockable* requirements and `lk` points to the the same address for all `atomic_ref` which reference the object referenced by `obj`. If `is_lock_free(obj)` is `false` and `requires_user_lock(obj)` is `false` then `lk` points to either a valid `Lock` object or `nullptr`. If `requires_user_lock` is `true` then `lk` points to valid `Lock` object.

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

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

5 *Throws:* Nothing.

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

6 *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;
```

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

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

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

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;
```

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

10 *Effects:* If `lock` 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;
lock->lock();
memcpy(&result, ptr, sizeof(T));
lock->unlock();
return result;
```

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

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

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

12 ~~Atomically replaces the value referenced by `*ptr` with `desired`.~~ *Effects:* If `lock` 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;
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 (??).

13 *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;
```

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

15 *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 `lock` 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;
lock->lock()
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));
lock->unlock();
return result;
```

16 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.

17 *Returns:* The result of the comparison.

18 *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 bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_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&, 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 bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_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&, 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 bool never_requires_user_lock = implementation-defined;
    static constexpr size_t required_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*&, 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;
  };
}
```