Document Number: P1505R1
Date: 2019-06-16
Reply to: Daniel Sunderland
Sandia National Laboratories
dsunder@sandia.gov

Mandating the Standard Library: Clause 31 - Atomic operations library

With the adoption of P0788R3, we have a new way of specifying requirements for the library clauses of the standard. This is one of a series of papers reformulating the requirements into the new format. This effort was strongly influenced by the informational paper P1369R0.

The changes in this series of papers fall into four broad categories.

— Change "participate in overload resolution" wording into "Constraints" elements

— Change "Requires" elements into either "Mandates" or "Expects", depending (mostly) on whether or not they can be checked at compile time.

— Drive-by fixes (hopefully very few)

This paper covers Clause 31 (Atomic operations library)

The entire clause is reproduced here, but the changes are confined to a few sections:

— atomics.ref.operations 31.6.1

— atomics.ref.pointer 31.6.4

— atomics.types.operations 31.7.1

— atomics.types.pointer 31.7.4

— atomics.flag 31.9

Changes from R0:

— Update Clause number from 30 to 31.

— Replaced use of `&ptr` with `std::addressof(ptr);`

Help for the editors: The changes here can be viewed as latex sources with the following commands

```
git clone git@github.com:dsunder/draft.git dsunder-draft
cd dsunder-draft
git diff master..P1505 -- source/atomics.tex
```

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.

31 Atomic operations library [atomics]

31.1 General [atomics.general]

1 This Clause describes components for fine-grained atomic access. This access is provided via operations on atomic objects.

2 The following subclauses describe atomics requirements and components for types and operations, as summarized below.

Table 131 — Atomics library summary

	Subclause	Header(s)
31.3	Type aliases	
31.4	Order and consistency	
31.5	Lock-free property	
31.6	Class template `atomic_ref`	`<atomic>`
31.7	Class template `atomic`	`<atomic>`
31.8	Non-member functions	`<atomic>`
31.9	Flag type and operations	`<atomic>`
31.10	Fences	`<atomic>`

31.2 Header <atomic> synopsis [atomics.syn]

```
namespace std {
  // 31.4, order and consistency
  enum class memory_order : unspecified;
  template<class T>
    T kill_dependency(T y) noexcept;

  // 31.5, lock-free property
  #define ATOMIC_BOOL_LOCK_FREE unspecified
  #define ATOMIC_CHAR_LOCK_FREE unspecified
  #define ATOMIC_CHAR8_T_LOCK_FREE unspecified
  #define ATOMIC_CHAR16_T_LOCK_FREE unspecified
  #define ATOMIC_CHAR32_T_LOCK_FREE unspecified
  #define ATOMIC_WCHAR_T_LOCK_FREE unspecified
  #define ATOMIC_SHORT_LOCK_FREE unspecified
  #define ATOMIC_INT_LOCK_FREE unspecified
  #define ATOMIC_LONG_LOCK_FREE unspecified
  #define ATOMIC_LLONG_LOCK_FREE unspecified
  #define ATOMIC_POINTER_LOCK_FREE unspecified

  // 31.6, class template atomic_ref
  template<class T> struct atomic_ref;
  // 31.6.4, partial specialization for pointers
  template<class T> struct atomic_ref<T*>;

  // 31.7, class template atomic
  template<class T> struct atomic;
  // 31.7.4, partial specialization for pointers
  template<class T> struct atomic<T*>;

  // 31.8, non-member functions
  template<class T>
    bool atomic_is_lock_free(const volatile atomic<T>*) noexcept;
  template<class T>
    bool atomic_is_lock_free(const atomic<T>*) noexcept;
```

```
template<class T>
  void atomic_init(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  void atomic_init(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  void atomic_store(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  void atomic_store(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  void atomic_store_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
                             memory_order) noexcept;
template<class T>
  void atomic_store_explicit(atomic<T>*, typename atomic<T>::value_type,
                             memory_order) noexcept;
template<class T>
  T atomic_load(const volatile atomic<T>*) noexcept;
template<class T>
  T atomic_load(const atomic<T>*) noexcept;
template<class T>
  T atomic_load_explicit(const volatile atomic<T>*, memory_order) noexcept;
template<class T>
  T atomic_load_explicit(const atomic<T>*, memory_order) noexcept;
template<class T>
  T atomic_exchange(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_exchange(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_exchange_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
                             memory_order) noexcept;
template<class T>
  T atomic_exchange_explicit(atomic<T>*, typename atomic<T>::value_type,
                             memory_order) noexcept;
template<class T>
  bool atomic_compare_exchange_weak(volatile atomic<T>*,
                                    typename atomic<T>::value_type*,
                                    typename atomic<T>::value_type) noexcept;
template<class T>
  bool atomic_compare_exchange_weak(atomic<T>*,
                                    typename atomic<T>::value_type*,
                                    typename atomic<T>::value_type) noexcept;
template<class T>
  bool atomic_compare_exchange_strong(volatile atomic<T>*,
                                      typename atomic<T>::value_type*,
                                      typename atomic<T>::value_type) noexcept;
template<class T>
  bool atomic_compare_exchange_strong(atomic<T>*,
                                      typename atomic<T>::value_type*,
                                      typename atomic<T>::value_type) noexcept;
template<class T>
  bool atomic_compare_exchange_weak_explicit(volatile atomic<T>*,
                                             typename atomic<T>::value_type*,
                                             typename atomic<T>::value_type,
                                             memory_order, memory_order) noexcept;
template<class T>
  bool atomic_compare_exchange_weak_explicit(atomic<T>*,
                                             typename atomic<T>::value_type*,
                                             typename atomic<T>::value_type,
                                             memory_order, memory_order) noexcept;
template<class T>
  bool atomic_compare_exchange_strong_explicit(volatile atomic<T>*,
                                               typename atomic<T>::value_type*,
                                               typename atomic<T>::value_type,
                                               memory_order, memory_order) noexcept;
```

```
template<class T>
  bool atomic_compare_exchange_strong_explicit(atomic<T>*,
                                               typename atomic<T>::value_type*,
                                               typename atomic<T>::value_type,
                                               memory_order, memory_order) noexcept;

template<class T>
  T atomic_fetch_add(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
  T atomic_fetch_add(atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
  T atomic_fetch_add_explicit(volatile atomic<T>*, typename atomic<T>::difference_type,
                              memory_order) noexcept;
template<class T>
  T atomic_fetch_add_explicit(atomic<T>*, typename atomic<T>::difference_type,
                              memory_order) noexcept;
template<class T>
  T atomic_fetch_sub(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
  T atomic_fetch_sub(atomic<T>*, typename atomic<T>::difference_type) noexcept;
template<class T>
  T atomic_fetch_sub_explicit(volatile atomic<T>*, typename atomic<T>::difference_type,
                              memory_order) noexcept;
template<class T>
  T atomic_fetch_sub_explicit(atomic<T>*, typename atomic<T>::difference_type,
                              memory_order) noexcept;
template<class T>
  T atomic_fetch_and(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_fetch_and(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_fetch_and_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
                              memory_order) noexcept;
template<class T>
  T atomic_fetch_and_explicit(atomic<T>*, typename atomic<T>::value_type,
                              memory_order) noexcept;
template<class T>
  T atomic_fetch_or(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_fetch_or(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_fetch_or_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
                             memory_order) noexcept;
template<class T>
  T atomic_fetch_or_explicit(atomic<T>*, typename atomic<T>::value_type,
                             memory_order) noexcept;
template<class T>
  T atomic_fetch_xor(volatile atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_fetch_xor(atomic<T>*, typename atomic<T>::value_type) noexcept;
template<class T>
  T atomic_fetch_xor_explicit(volatile atomic<T>*, typename atomic<T>::value_type,
                              memory_order) noexcept;
template<class T>
  T atomic_fetch_xor_explicit(atomic<T>*, typename atomic<T>::value_type,
                              memory_order) noexcept;

// 31.7.1, initialization
#define ATOMIC_VAR_INIT(value) see below

// 31.3, type aliases
using atomic_bool            = atomic<bool>;
using atomic_char            = atomic<char>;
using atomic_schar           = atomic<signed char>;
```

```
  using atomic_uchar          = atomic<unsigned char>;
  using atomic_short          = atomic<short>;
  using atomic_ushort         = atomic<unsigned short>;
  using atomic_int            = atomic<int>;
  using atomic_uint           = atomic<unsigned int>;
  using atomic_long           = atomic<long>;
  using atomic_ulong          = atomic<unsigned long>;
  using atomic_llong          = atomic<long long>;
  using atomic_ullong         = atomic<unsigned long long>;
  using atomic_char8_t        = atomic<char8_t>;
  using atomic_char16_t       = atomic<char16_t>;
  using atomic_char32_t       = atomic<char32_t>;
  using atomic_wchar_t        = atomic<wchar_t>;

  using atomic_int8_t         = atomic<int8_t>;
  using atomic_uint8_t        = atomic<uint8_t>;
  using atomic_int16_t        = atomic<int16_t>;
  using atomic_uint16_t       = atomic<uint16_t>;
  using atomic_int32_t        = atomic<int32_t>;
  using atomic_uint32_t       = atomic<uint32_t>;
  using atomic_int64_t        = atomic<int64_t>;
  using atomic_uint64_t       = atomic<uint64_t>;

  using atomic_int_least8_t   = atomic<int_least8_t>;
  using atomic_uint_least8_t  = atomic<uint_least8_t>;
  using atomic_int_least16_t  = atomic<int_least16_t>;
  using atomic_uint_least16_t = atomic<uint_least16_t>;
  using atomic_int_least32_t  = atomic<int_least32_t>;
  using atomic_uint_least32_t = atomic<uint_least32_t>;
  using atomic_int_least64_t  = atomic<int_least64_t>;
  using atomic_uint_least64_t = atomic<uint_least64_t>;

  using atomic_int_fast8_t    = atomic<int_fast8_t>;
  using atomic_uint_fast8_t   = atomic<uint_fast8_t>;
  using atomic_int_fast16_t   = atomic<int_fast16_t>;
  using atomic_uint_fast16_t  = atomic<uint_fast16_t>;
  using atomic_int_fast32_t   = atomic<int_fast32_t>;
  using atomic_uint_fast32_t  = atomic<uint_fast32_t>;
  using atomic_int_fast64_t   = atomic<int_fast64_t>;
  using atomic_uint_fast64_t  = atomic<uint_fast64_t>;

  using atomic_intptr_t       = atomic<intptr_t>;
  using atomic_uintptr_t      = atomic<uintptr_t>;
  using atomic_size_t         = atomic<size_t>;
  using atomic_ptrdiff_t      = atomic<ptrdiff_t>;
  using atomic_intmax_t       = atomic<intmax_t>;
  using atomic_uintmax_t      = atomic<uintmax_t>;

  // 31.9, flag type and operations
  struct atomic_flag;
  bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
  bool atomic_flag_test_and_set(atomic_flag*) noexcept;
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear(volatile atomic_flag*) noexcept;
  void atomic_flag_clear(atomic_flag*) noexcept;
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;
  #define ATOMIC_FLAG_INIT see below

  // 31.10, fences
  extern "C" void atomic_thread_fence(memory_order) noexcept;
  extern "C" void atomic_signal_fence(memory_order) noexcept;
}
```

31.3 Type aliases [atomics.alias]

1 The type aliases `atomic_int`N`_t`, `atomic_uint`N`_t`, `atomic_intptr_t`, and `atomic_uintptr_t` are defined if and only if `int`N`_t`, `uint`N`_t`, `intptr_t`, and `uintptr_t` are defined, respectively.

31.4 Order and consistency [atomics.order]

```
namespace std {
  enum class memory_order : unspecified {
    relaxed, consume, acquire, release, acq_rel, seq_cst
  };
  inline constexpr memory_order memory_order_relaxed = memory_order::relaxed;
  inline constexpr memory_order memory_order_consume = memory_order::consume;
  inline constexpr memory_order memory_order_acquire = memory_order::acquire;
  inline constexpr memory_order memory_order_release = memory_order::release;
  inline constexpr memory_order memory_order_acq_rel = memory_order::acq_rel;
  inline constexpr memory_order memory_order_seq_cst = memory_order::seq_cst;
}
```

1 The enumeration `memory_order` specifies the detailed regular (non-atomic) memory synchronization order as defined in **??** and may provide for operation ordering. Its enumerated values and their meanings are as follows:

(1.1) — `memory_order::relaxed`: no operation orders memory.

(1.2) — `memory_order::release`, `memory_order::acq_rel`, and `memory_order::seq_cst`: a store operation performs a release operation on the affected memory location.

(1.3) — `memory_order::consume`: a load operation performs a consume operation on the affected memory location. [*Note*: Prefer `memory_order::acquire`, which provides stronger guarantees than `memory_order::consume`. Implementations have found it infeasible to provide performance better than that of `memory_order::acquire`. Specification revisions are under consideration. — *end note*]

(1.4) — `memory_order::acquire`, `memory_order::acq_rel`, and `memory_order::seq_cst`: a load operation performs an acquire operation on the affected memory location.

[*Note*: Atomic operations specifying `memory_order::relaxed` are relaxed with respect to memory ordering. Implementations must still guarantee that any given atomic access to a particular atomic object be indivisible with respect to all other atomic accesses to that object. — *end note*]

2 An atomic operation A that performs a release operation on an atomic object M synchronizes with an atomic operation B that performs an acquire operation on M and takes its value from any side effect in the release sequence headed by A.

3 An atomic operation A on some atomic object M is *coherence-ordered before* another atomic operation B on M if

(3.1) — A is a modification, and B reads the value stored by A, or

(3.2) — A precedes B in the modification order of M, or

(3.3) — A and B are not the same atomic read-modify-write operation, and there exists an atomic modification X of M such that A reads the value stored by X and X precedes B in the modification order of M, or

(3.4) — there exists an atomic modification X of M such that A is coherence-ordered before X and X is coherence-ordered before B.

4 There is a single total order S on all `memory_order::seq_cst` operations, including fences, that satisfies the following constraints. First, if A and B are `memory_order::seq_cst` operations and A strongly happens before B, then A precedes B in S. Second, for every pair of atomic operations A and B on an object M, where A is coherence-ordered before B, the following four conditions are required to be satisfied by S:

(4.1) — if A and B are both `memory_order::seq_cst` operations, then A precedes B in S; and

(4.2) — if A is a `memory_order::seq_cst` operation and B happens before a `memory_order::seq_cst` fence Y, then A precedes Y in S; and

(4.3) — if a `memory_order::seq_cst` fence X happens before A and B is a `memory_order::seq_cst` operation, then X precedes B in S; and

(4.4) — if a `memory_order::seq_cst` fence X happens before A and B happens before a `memory_order::seq_cst` fence Y, then X precedes Y in S.

5 [*Note*: This definition ensures that S is consistent with the modification order of any atomic object M. It also ensures that a `memory_order::seq_cst` load A of M gets its value either from the last modification of M that precedes A in S or from some non-`memory_order::seq_cst` modification of M that does not happen before any modification of M that precedes A in S. — *end note*]

6 [*Note*: We do not require that S be consistent with "happens before" (**??**). This allows more efficient implementation of `memory_order::acquire` and `memory_order::release` on some machine architectures. It can produce surprising results when these are mixed with `memory_order::seq_cst` accesses. — *end note*]

7 [*Note*: `memory_order::seq_cst` ensures sequential consistency only for a program that is free of data races and uses exclusively `memory_order::seq_cst` atomic operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In many cases, `memory_order::seq_cst` atomic operations are reorderable with respect to other atomic operations performed by the same thread. — *end note*]

8 Implementations should ensure that no "out-of-thin-air" values are computed that circularly depend on their own computation.

[*Note*: For example, with `x` and `y` initially zero,

```
// Thread 1:
r1 = y.load(memory_order::relaxed);
x.store(r1, memory_order::relaxed);
// Thread 2:
r2 = x.load(memory_order::relaxed);
y.store(r2, memory_order::relaxed);
```

should not produce `r1 == r2 == 42`, since the store of 42 to `y` is only possible if the store to `x` stores `42`, which circularly depends on the store to `y` storing `42`. Note that without this restriction, such an execution is possible. — *end note*]

9 [*Note*: The recommendation similarly disallows `r1 == r2 == 42` in the following example, with `x` and `y` again initially zero:

```
// Thread 1:
r1 = x.load(memory_order::relaxed);
if (r1 == 42) y.store(42, memory_order::relaxed);
// Thread 2:
r2 = y.load(memory_order::relaxed);
if (r2 == 42) x.store(42, memory_order::relaxed);
```

— *end note*]

10 Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation.

11 Implementations should make atomic stores visible to atomic loads within a reasonable amount of time.

```
template<class T>
  T kill_dependency(T y) noexcept;
```

12 *Effects:* The argument does not carry a dependency to the return value (**??**).

13 *Returns:* `y`.

31.5 Lock-free property [atomics.lockfree]

```
#define ATOMIC_BOOL_LOCK_FREE unspecified
#define ATOMIC_CHAR_LOCK_FREE unspecified
#define ATOMIC_CHAR8_T_LOCK_FREE unspecified
#define ATOMIC_CHAR16_T_LOCK_FREE unspecified
#define ATOMIC_CHAR32_T_LOCK_FREE unspecified
#define ATOMIC_WCHAR_T_LOCK_FREE unspecified
#define ATOMIC_SHORT_LOCK_FREE unspecified
#define ATOMIC_INT_LOCK_FREE unspecified
#define ATOMIC_LONG_LOCK_FREE unspecified
#define ATOMIC_LLONG_LOCK_FREE unspecified
#define ATOMIC_POINTER_LOCK_FREE unspecified
```

1 The `ATOMIC_..._LOCK_FREE` macros indicate the lock-free property of the corresponding atomic types, with the signed and unsigned variants grouped together. The properties also apply to the corresponding (partial)

specializations of the `atomic` template. A value of 0 indicates that the types are never lock-free. A value of 1 indicates that the types are sometimes lock-free. A value of 2 indicates that the types are always lock-free.

2 The function `atomic_is_lock_free` (31.7.1) indicates whether the object is lock-free. In any given program execution, the result of the lock-free query shall be consistent for all pointers of the same type.

3 Atomic operations that are not lock-free are considered to potentially block (**??**).

4 [*Note*: Operations that are lock-free should also be address-free. That is, atomic operations on the same memory location via two different addresses will communicate atomically. The implementation should not depend on any per-process state. This restriction enables communication by memory that is mapped into a process more than once and by memory that is shared between two processes. — *end note*]

31.6 Class template atomic_ref [atomics.ref.generic]

```
namespace std {
  template<class T> struct atomic_ref {
  private:
    T* ptr;              // 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&);
    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;
  };
}
```

1 An `atomic_ref` object applies atomic operations (31.1) to the object referenced by `*ptr` such that, for the lifetime (**??**) of the `atomic_ref` object, the object referenced by `*ptr` is an atomic object (**??**).

2 The template argument for `T` shall be trivially copyable (**??**).

3 The lifetime (**??**) of an object referenced by `*ptr` shall exceed the lifetime of all `atomic_ref`s that reference the object. While any `atomic_ref` instances exist that reference the `*ptr` object, all accesses to that object shall exclusively occur through those `atomic_ref` instances. No subobject of the object referenced by `atomic_ref` shall be concurrently referenced by any other `atomic_ref` object.

4 Atomic operations applied to an object through a referencing `atomic_ref` are atomic with respect to atomic operations applied through any other `atomic_ref` referencing the same object. [*Note*: Atomic operations or the `atomic_ref` constructor could acquire a shared resource, such as a lock associated with the referenced object, to enable atomic operations to be applied to the referenced object. — *end note*]

31.6.1 Operations [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, and `false` otherwise.

```
static constexpr size_t required_alignment;
```

2 The alignment required for an object to be referenced 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*]

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

4 ~~*Requires:*~~ *Expects:* The ~~referenced~~ object referenced by `obj` ~~shall be~~is aligned to `required_alignment`.

5 *Effects:* ~~Constructs an atomic reference that references the object.~~ Equivalent to: `ptr = std::addressof(obj)`.

6 *Throws:* Nothing.

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

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

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

8 *Effects:* Equivalent to:

```
store(desired);
return desired;
```

```
operator T() const noexcept;
```

9 *Effects:* Equivalent to: `return load();`

```
bool is_lock_free() const noexcept;
```

10 *Returns:* `true` if the object's operations are lock-free, `false` otherwise.

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

11 ~~*Requires:*~~ *Expects:* The `order` argument ~~shall not be~~is neither `memory_order_consume`, nor `memory_order_acquire`, nor `memory_order_acq_rel`.

12 *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;
```

13 ~~*Requires:*~~ *Expects:* The `order` argument ~~shall not be~~is neither `memory_order_release` nor `memory_order_acq_rel`.

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

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

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

16 *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 (??).

17 *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;
```

18 ~~*Requires:*~~ *Expects:* The `failure` argument ~~shall not be~~is neither `memory_order_release` nor `memory_order_acq_rel`.

19 *Effects:* 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.

20 *Returns:* The result of the comparison.

21 *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*]

31.6.2 Specializations for integral types [atomics.ref.int]

1 There are specializations of the `atomic_ref` class template for the integral types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, and any other types needed by the typedefs in the header `<cstdint>`. For each such type *`integral`*, the specialization `atomic_ref<`*`integral`*`>` provides additional atomic operations appropriate to integral types. [*Note*: For the specialization `atomic_ref<bool>`, see 31.6. — *end note*]

```
namespace std {
  template<> struct atomic_ref<integral> {
  private:
    integral* ptr;        // 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&);
    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;
  };
}
```

2 Descriptions are provided below only for members that differ from the primary template.

3 The following operations perform arithmetic computations. The key, operator, and computation correspondence is identified in Table 132.

`integral fetch_key(integral operand, memory_order order = memory_order_seq_cst) const noexcept;`

4 *Effects:* Atomically replaces the value referenced by `*ptr` with the result of the computation applied to the value referenced by `*ptr` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (**??**).

5 *Returns:* Atomically, the value referenced by `*ptr` immediately before the effects.

6 *Remarks:* For signed integer types, the result is as if the object value and parameters were converted to their corresponding unsigned types, the computation performed on those types, and the result converted back to the signed type. [*Note*: There are no undefined results arising from the computation. — *end note*]

`integral operator op=(integral operand) const noexcept;`

7 *Effects:* Equivalent to: `return fetch_key(operand) op operand;`

31.6.3 Specializations for floating-point types [atomics.ref.float]

1 There are specializations of the `atomic_ref` class template for the floating-point types `float`, `double`, and `long double`. For each such type *`floating-point`*, the specialization `atomic_ref<floating-point>` provides additional atomic operations appropriate to floating-point types.

```
namespace std {
  template<> struct atomic_ref<floating-point> {
  private:
    floating-point* ptr;  // 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&);
    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;
  };
}
```

2 Descriptions are provided below only for members that differ from the primary template.

3 The following operations perform arithmetic computations. The key, operator, and computation correspondence are identified in Table 132.

```
floating-point fetch_key(floating-point operand,
                         memory_order order = memory_order_seq_cst) const noexcept;
```

4 *Effects:* Atomically replaces the value referenced by `*ptr` with the result of the computation applied to the value referenced by `*ptr` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (??).

5 *Returns:* Atomically, the value referenced by `*ptr` immediately before the effects.

6 *Remarks:* If the result is not a representable value for its type (??), the result is unspecified, but the operations otherwise have no undefined behavior. Atomic arithmetic operations on *`floating-point`* should conform to the `std::numeric_limits<floating-point>` traits associated with the floating-point type (??). The floating-point environment (??) for atomic arithmetic operations on *`floating-point`* may be different than the calling thread's floating-point environment.

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

Effects: Equivalent to: `return fetch_key(operand) op operand;`

31.6.4 Partial specialization for pointers [atomics.ref.pointer]

```
namespace std {
  template<class T> struct atomic_ref<T*> {
  private:
    T** ptr;            // 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*&);
    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;
  };
}
```

1 Descriptions are provided below only for members that differ from the primary template.

2 The following operations perform arithmetic computations. The key, operator, and computation correspondence is identified in Table 133.

```
T* fetch_key(difference_type operand, memory_order order = memory_order_seq_cst) const noexcept;
```

3 ~~*Requires:*~~ *Mandates:* `T` ~~shall be~~is a~~n~~ complete object type. ~~, otherwise the program is ill-formed.~~

4 *Effects:* Atomically replaces the value referenced by `*ptr` with the result of the computation applied to the value referenced by `*ptr` and the given operand. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (**??**).

5 *Returns:* Atomically, the value referenced by `*ptr` immediately before the effects.

6 *Remarks:* The result may be an undefined address, but the operations otherwise have no undefined behavior.

```
T* operator op=(difference_type operand) const noexcept;
```

7 *Effects:* Equivalent to: `return fetch_key(operand) op operand;`

31.6.5 Member operators common to integers and pointers to objects [atomics.ref.memop]

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

1 *Effects:* Equivalent to: `return fetch_add(1);`

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

2 *Effects:* Equivalent to: `return fetch_sub(1);`

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

3 *Effects:* Equivalent to: `return fetch_add(1) + 1;`

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

4 *Effects:* Equivalent to: `return fetch_sub(1) - 1;`

31.7 Class template atomic [atomics.types.generic]

```
namespace std {
  template<class T> struct atomic {
    using value_type = T;
    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T, memory_order = memory_order::seq_cst) volatile noexcept;
    void store(T, memory_order = memory_order::seq_cst) noexcept;
    T load(memory_order = memory_order::seq_cst) const volatile noexcept;
    T load(memory_order = memory_order::seq_cst) const noexcept;
    operator T() const volatile noexcept;
    operator T() const noexcept;
    T exchange(T, memory_order = memory_order::seq_cst) volatile noexcept;
    T exchange(T, memory_order = memory_order::seq_cst) noexcept;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_weak(T&, T, memory_order = memory_order::seq_cst) noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_strong(T&, T, memory_order = memory_order::seq_cst) noexcept;

    atomic() noexcept = default;
    constexpr atomic(T) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T operator=(T) volatile noexcept;
    T operator=(T) noexcept;
  };
}
```

1 The template argument for `T` shall be trivially copyable (**??**). [*Note*: Type arguments that are not also statically initializable may be difficult to use. — *end note*]

2 The specialization `atomic<bool>` is a standard-layout struct.

3 [*Note*: The representation of an atomic specialization need not have the same size and alignment requirement as its corresponding argument type. — *end note*]

31.7.1 Operations on atomic types [atomics.types.operations]

1 [*Note*: Many operations are volatile-qualified. The "volatile as device register" semantics have not changed in the standard. This qualification means that volatility is preserved when applying these operations to volatile objects. It does not mean that operations on non-volatile objects become volatile. — *end note*]

`atomic() noexcept = default;`

2 *Effects:* Leaves the atomic object in an uninitialized state. [*Note*: These semantics ensure compatibility with C. — *end note*]

`constexpr atomic(T desired) noexcept;`

3 *Effects:* Initializes the object with the value `desired`. Initialization is not an atomic operation (**??**). [*Note*: It is possible to have an access to an atomic object `A` race with its construction, for example by communicating the address of the just-constructed object `A` to another thread via `memory_order::relaxed` operations on a suitable atomic pointer variable, and then immediately accessing `A` in the receiving thread. This results in undefined behavior. — *end note*]

```
#define ATOMIC_VAR_INIT(value) see below
```

4 The macro expands to a token sequence suitable for constant initialization of an atomic variable of static storage duration of a type that is initialization-compatible with `value`. [*Note*: This operation may need to initialize locks. — *end note*] Concurrent access to the variable being initialized, even via an atomic operation, constitutes a data race. [*Example*:

```
atomic<int> v = ATOMIC_VAR_INIT(5);
```

— *end example*]

```
static constexpr bool is_always_lock_free = implementation-defined;
```

5 The `static` data member `is_always_lock_free` is `true` if the atomic type's operations are always lock-free, and `false` otherwise. [*Note*: The value of `is_always_lock_free` is consistent with the value of the corresponding `ATOMIC_..._LOCK_FREE` macro, if defined. — *end note*]

```
bool is_lock_free() const volatile noexcept;
bool is_lock_free() const noexcept;
```

6 *Returns:* `true` if the object's operations are lock-free, `false` otherwise. [*Note*: The return value of the `is_lock_free` member function is consistent with the value of `is_always_lock_free` for the same type. — *end note*]

```
void store(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
void store(T desired, memory_order order = memory_order::seq_cst) noexcept;
```

7 ~~*Requires:*~~ *Expects:* The `order` argument ~~shall not be~~is neither `memory_order::consume`, `memory_order::acquire`, nor `memory_order::acq_rel`.

8 *Effects:* Atomically replaces the value pointed to by `this` with the value of `desired`. Memory is affected according to the value of `order`.

```
T operator=(T desired) volatile noexcept;
T operator=(T desired) noexcept;
```

9 *Effects:* Equivalent to `store(desired)`.

10 *Returns:* `desired`.

```
T load(memory_order order = memory_order::seq_cst) const volatile noexcept;
T load(memory_order order = memory_order::seq_cst) const noexcept;
```

11 ~~*Requires:*~~ *Expects:* The `order` argument ~~shall not be~~is neither `memory_order::release` nor `memory_order::acq_rel`.

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

13 *Returns:* Atomically returns the value pointed to by `this`.

```
operator T() const volatile noexcept;
operator T() const noexcept;
```

14 *Effects:* Equivalent to: `return load();`

```
T exchange(T desired, memory_order order = memory_order::seq_cst) volatile noexcept;
T exchange(T desired, memory_order order = memory_order::seq_cst) noexcept;
```

15 *Effects:* Atomically replaces the value pointed to by `this` with `desired`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (**??**).

16 *Returns:* Atomically returns the value pointed to by `this` immediately before the effects.

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

```
bool compare_exchange_weak(T& expected, T desired,
                           memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_weak(T& expected, T desired,
                           memory_order order = memory_order::seq_cst) noexcept;
bool compare_exchange_strong(T& expected, T desired,
                             memory_order order = memory_order::seq_cst) volatile noexcept;
bool compare_exchange_strong(T& expected, T desired,
                             memory_order order = memory_order::seq_cst) noexcept;
```

17 ~~*Requires:*~~ *Expects:* The `failure` argument ~~shall not be~~is neither `memory_order::release` nor `memory_order::acq_rel`.

18 *Effects:* Retrieves the value in `expected`. It then atomically compares the value representation of the value pointed to by `this` for equality with that previously retrieved from `expected`, and if true, replaces the value pointed to by `this` 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 pointed to by `this` during the atomic comparison. If the operation returns `true`, these operations are atomic read-modify-write operations (??) on the memory pointed to by `this`. Otherwise, these operations are atomic load operations on that memory.

19 *Returns:* The result of the comparison.

20 [*Note*: For example, the effect of `compare_exchange_strong` on objects without padding bits (??) is

```
if (memcmp(this, &expected, sizeof(*this)) == 0)
  memcpy(this, &desired, sizeof(*this));
else
  memcpy(expected, this, sizeof(*this));
```

— *end note*] [*Example*: The expected use of the compare-and-exchange operations is as follows. The compare-and-exchange operations will update `expected` when another iteration of the loop is needed.

```
expected = current.load();
do {
  desired = function(expected);
} while (!current.compare_exchange_weak(expected, desired));
```

— *end example*] [*Example*: Because the expected value is updated only on failure, code releasing the memory containing the `expected` value on success will work. For example, list head insertion will act atomically and would not introduce a data race in the following code:

```
do {
  p->next = head; // make new list node point to the current head
} while (!head.compare_exchange_weak(p->next, p)); // try to insert
```

— *end example*]

21 Implementations should ensure that weak compare-and-exchange operations do not consistently return `false` unless either the atomic object has value different from `expected` or there are concurrent modifications to the atomic object.

22 *Remarks:* A weak compare-and-exchange operation may fail spuriously. That is, even when the contents of memory referred to by `expected` and `this` 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*]

23 [*Note*: Under cases where the `memcpy` and `memcmp` semantics of the compare-and-exchange operations apply, the outcome might be failed comparisons for values that compare equal with `operator==` if the value representation has trap bits or alternate representations of the same value. Notably, on

implementations conforming to ISO/IEC/IEEE 60559, floating-point `-0.0` and `+0.0` will not compare equal with `memcmp` but will compare equal with `operator==`, and NaNs with the same payload will compare equal with `memcmp` but will not compare equal with `operator==`. — *end note*] [*Note*: Because compare-and-exchange acts on an object's value representation, padding bits that never participate in the object's value representation are ignored. As a consequence, the following code is guaranteed to avoid spurious failure:

```
struct padded {
  char clank = 0x42;
  // Padding here.
  unsigned biff = 0xC0DEFEFE;
};
atomic<padded> pad = ATOMIC_VAR_INIT({});

bool zap() {
  padded expected, desired{0, 0};
  return pad.compare_exchange_strong(expected, desired);
}
```

— *end note*] [*Note*: For a union with bits that participate in the value representation of some members but not others, compare-and-exchange might always fail. This is because such padding bits have an indeteminate value when they do not participate in the value representation of the active member. As a consequence, the following code is not guaranteed to ever succeed:

```
union pony {
  double celestia = 0.;
  short luna; // padded
};
atomic<pony> princesses = ATOMIC_VAR_INIT({});

bool party(pony desired) {
  pony expected;
  return princesses.compare_exchange_strong(expected, desired);
}
```

— *end note*]

31.7.2 Specializations for integers [atomics.types.int]

1 There are specializations of the `atomic` class template for the integral types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, and any other types needed by the typedefs in the header `<cstdint>`. For each such type *`integral`*, the specialization `atomic<`*`integral`*`>` provides additional atomic operations appropriate to integral types. [*Note*: For the specialization `atomic<bool>`, see 31.7. — *end note*]

```
namespace std {
  template<> struct atomic<integral> {
    using value_type = integral;
    using difference_type = value_type;
    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(integral, memory_order = memory_order::seq_cst) volatile noexcept;
    void store(integral, memory_order = memory_order::seq_cst) noexcept;
    integral load(memory_order = memory_order::seq_cst) const volatile noexcept;
    integral load(memory_order = memory_order::seq_cst) const noexcept;
    operator integral() const volatile noexcept;
    operator integral() const noexcept;
    integral exchange(integral, memory_order = memory_order::seq_cst) volatile noexcept;
    integral exchange(integral, memory_order = memory_order::seq_cst) noexcept;
    bool compare_exchange_weak(integral&, integral,
                               memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(integral&, integral,
                               memory_order, memory_order) noexcept;
    bool compare_exchange_strong(integral&, integral,
                                 memory_order, memory_order) volatile noexcept;
```

```
    bool compare_exchange_strong(integral&, integral,
                                 memory_order, memory_order) noexcept;
    bool compare_exchange_weak(integral&, integral,
                               memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_weak(integral&, integral,
                               memory_order = memory_order::seq_cst) noexcept;
    bool compare_exchange_strong(integral&, integral,
                                 memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_strong(integral&, integral,
                                 memory_order = memory_order::seq_cst) noexcept;
    integral fetch_add(integral, memory_order = memory_order::seq_cst) volatile noexcept;
    integral fetch_add(integral, memory_order = memory_order::seq_cst) noexcept;
    integral fetch_sub(integral, memory_order = memory_order::seq_cst) volatile noexcept;
    integral fetch_sub(integral, memory_order = memory_order::seq_cst) noexcept;
    integral fetch_and(integral, memory_order = memory_order::seq_cst) volatile noexcept;
    integral fetch_and(integral, memory_order = memory_order::seq_cst) noexcept;
    integral fetch_or(integral, memory_order = memory_order::seq_cst) volatile noexcept;
    integral fetch_or(integral, memory_order = memory_order::seq_cst) noexcept;
    integral fetch_xor(integral, memory_order = memory_order::seq_cst) volatile noexcept;
    integral fetch_xor(integral, memory_order = memory_order::seq_cst) noexcept;

    atomic() noexcept = default;
    constexpr atomic(integral) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    integral operator=(integral) volatile noexcept;
    integral operator=(integral) noexcept;

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

2 The atomic integral specializations are standard-layout structs. They each have a trivial default constructor and a trivial destructor.

3 Descriptions are provided below only for members that differ from the primary template.

4 The following operations perform arithmetic computations. The key, operator, and computation correspondence is:

Table 132 — Atomic arithmetic computations

key	**Op**	**Computation**	*key*	**Op**	**Computation**
`add`	`+`	addition	`sub`	`-`	subtraction
`or`	`|`	bitwise inclusive or	`xor`	`^`	bitwise exclusive or
`and`	`&`	bitwise and			

```
T fetch_key(T operand, memory_order order = memory_order::seq_cst) volatile noexcept;
T fetch_key(T operand, memory_order order = memory_order::seq_cst) noexcept;
```

5 *Effects:* Atomically replaces the value pointed to by `this` with the result of the computation applied to the value pointed to by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (**??**).

6 *Returns:* Atomically, the value pointed to by `this` immediately before the effects.

7 *Remarks:* For signed integer types, the result is as if the object value and parameters were converted to their corresponding unsigned types, the computation performed on those types, and the result converted back to the signed type. [*Note*: There are no undefined results arising from the computation. — *end note*]

```
T operator op=(T operand) volatile noexcept;
T operator op=(T operand) noexcept;
```

8 *Effects:* Equivalent to: `return fetch_key(operand) op operand;`

31.7.3 Specializations for floating-point types [atomics.types.float]

1 There are specializations of the `atomic` class template for the floating-point types `float`, `double`, and `long double`. For each such type *`floating-point`*, the specialization `atomic<floating-point>` provides additional atomic operations appropriate to floating-point types.

```
namespace std {
  template<> struct atomic<floating-point> {
    using value_type = floating-point;
    using difference_type = value_type;
    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(floating-point, memory_order = memory_order_seq_cst) volatile noexcept;
    void store(floating-point, memory_order = memory_order_seq_cst) noexcept;
    floating-point load(memory_order = memory_order_seq_cst) volatile noexcept;
    floating-point load(memory_order = memory_order_seq_cst) noexcept;
    operator floating-point() volatile noexcept;
    operator floating-point() noexcept;
    floating-point exchange(floating-point,
                            memory_order = memory_order_seq_cst) volatile noexcept;
    floating-point exchange(floating-point,
                            memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(floating-point&, floating-point,
                               memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(floating-point&, floating-point,
                               memory_order, memory_order) noexcept;
    bool compare_exchange_strong(floating-point&, floating-point,
                                 memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(floating-point&, floating-point,
                                 memory_order, memory_order) noexcept;
    bool compare_exchange_weak(floating-point&, floating-point,
                               memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(floating-point&, floating-point,
                               memory_order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(floating-point&, floating-point,
                                 memory_order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(floating-point&, floating-point,
                                 memory_order = memory_order_seq_cst) noexcept;

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

```
    atomic() noexcept = default;
    constexpr atomic(floating-point) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    floating-point operator=(floating-point) volatile noexcept;
    floating-point operator=(floating-point) noexcept;

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

2 The atomic floating-point specializations are standard-layout structs. They each have a trivial default constructor and a trivial destructor.

3 Descriptions are provided below only for members that differ from the primary template.

4 The following operations perform arithmetic addition and subtraction computations. The key, operator, and computation correspondence are identified in Table 132.

```
T A::fetch_key(T operand, memory_order order = memory_order_seq_cst) volatile noexcept;
T A::fetch_key(T operand, memory_order order = memory_order_seq_cst) noexcept;
```

5 *Effects:* Atomically replaces the value pointed to by `this` with the result of the computation applied to the value pointed to by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (**??**).

6 *Returns:* Atomically, the value pointed to by `this` immediately before the effects.

7 *Remarks:* If the result is not a representable value for its type (**??**) the result is unspecified, but the operations otherwise have no undefined behavior. Atomic arithmetic operations on *`floating-point`* should conform to the `std::numeric_limits<`*`floating-point`*`>` traits associated with the floating-point type (**??**). The floating-point environment (**??**) for atomic arithmetic operations on *`floating-point`* may be different than the calling thread's floating-point environment.

```
T operator op=(T operand) volatile noexcept;
T operator op=(T operand) noexcept;
```

8 *Effects:* Equivalent to: `return fetch_`*`key`*`(operand)` *`op`* `operand;`

9 *Remarks:* If the result is not a representable value for its type (**??**) the result is unspecified, but the operations otherwise have no undefined behavior. Atomic arithmetic operations on *`floating-point`* should conform to the `std::numeric_limits<`*`floating-point`*`>` traits associated with the floating-point type (**??**). The floating-point environment (**??**) for atomic arithmetic operations on *`floating-point`* may be different than the calling thread's floating-point environment.

31.7.4 Partial specialization for pointers [atomics.types.pointer]

```
namespace std {
  template<class T> struct atomic<T*> {
    using value_type = T*;
    using difference_type = ptrdiff_t;
    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(T*, memory_order = memory_order::seq_cst) volatile noexcept;
    void store(T*, memory_order = memory_order::seq_cst) noexcept;
    T* load(memory_order = memory_order::seq_cst) const volatile noexcept;
    T* load(memory_order = memory_order::seq_cst) const noexcept;
    operator T*() const volatile noexcept;
    operator T*() const noexcept;
    T* exchange(T*, memory_order = memory_order::seq_cst) volatile noexcept;
    T* exchange(T*, memory_order = memory_order::seq_cst) noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_weak(T*&, T*, memory_order, memory_order) noexcept;
```

```
    bool compare_exchange_strong(T*&, T*, memory_order, memory_order) volatile noexcept;
    bool compare_exchange_strong(T*&, T*, memory_order, memory_order) noexcept;
    bool compare_exchange_weak(T*&, T*,
                               memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_weak(T*&, T*,
                               memory_order = memory_order::seq_cst) noexcept;
    bool compare_exchange_strong(T*&, T*,
                                 memory_order = memory_order::seq_cst) volatile noexcept;
    bool compare_exchange_strong(T*&, T*,
                                 memory_order = memory_order::seq_cst) noexcept;
    T* fetch_add(ptrdiff_t, memory_order = memory_order::seq_cst) volatile noexcept;
    T* fetch_add(ptrdiff_t, memory_order = memory_order::seq_cst) noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order::seq_cst) volatile noexcept;
    T* fetch_sub(ptrdiff_t, memory_order = memory_order::seq_cst) noexcept;

    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    T* operator=(T*) volatile noexcept;
    T* operator=(T*) noexcept;

    T* operator++(int) volatile noexcept;
    T* operator++(int) noexcept;
    T* operator--(int) volatile noexcept;
    T* operator--(int) noexcept;
    T* operator++() volatile noexcept;
    T* operator++() noexcept;
    T* operator--() volatile noexcept;
    T* operator--() noexcept;
    T* operator+=(ptrdiff_t) volatile noexcept;
    T* operator+=(ptrdiff_t) noexcept;
    T* operator-=(ptrdiff_t) volatile noexcept;
    T* operator-=(ptrdiff_t) noexcept;
  };
}
```

1 There is a partial specialization of the `atomic` class template for pointers. Specializations of this partial specialization are standard-layout structs. They each have a trivial default constructor and a trivial destructor.

2 Descriptions are provided below only for members that differ from the primary template.

3 The following operations perform pointer arithmetic. The key, operator, and computation correspondence is:

Table 133 — Atomic pointer computations

Key	Op	Computation	Key	Op	Computation
`add`	`+`	addition	`sub`	`-`	subtraction

```
T* fetch_key(ptrdiff_t operand, memory_order order = memory_order::seq_cst) volatile noexcept;
T* fetch_key(ptrdiff_t operand, memory_order order = memory_order::seq_cst) noexcept;
```

4 ~~*Requires:*~~ *Mandates:* T ~~shall be~~is ~~an~~ complete object type. ~~, otherwise the program is ill-formed. [*Note:* Pointer arithmetic on `void*` or function pointers is ill-formed. — *end note*]~~

5 *Effects:* Atomically replaces the value pointed to by `this` with the result of the computation applied to the value pointed to by `this` and the given `operand`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (??).

6 *Returns:* Atomically, the value pointed to by `this` immediately before the effects.

7 *Remarks:* The result may be an undefined address, but the operations otherwise have no undefined behavior.

```
T* operator op=(ptrdiff_t operand) volatile noexcept;
T* operator op=(ptrdiff_t operand) noexcept;
```

8 *Effects:* Equivalent to: `return fetch_key(operand) op operand;`

31.7.5 Member operators common to integers and pointers to objects [atomics.types.memop]

```
T operator++(int) volatile noexcept;
T operator++(int) noexcept;
```

1 *Effects:* Equivalent to: `return fetch_add(1);`

```
T operator--(int) volatile noexcept;
T operator--(int) noexcept;
```

2 *Effects:* Equivalent to: `return fetch_sub(1);`

```
T operator++() volatile noexcept;
T operator++() noexcept;
```

3 *Effects:* Equivalent to: `return fetch_add(1) + 1;`

```
T operator--() volatile noexcept;
T operator--() noexcept;
```

4 *Effects:* Equivalent to: `return fetch_sub(1) - 1;`

31.8 Non-member functions [atomics.nonmembers]

1 A non-member function template whose name matches the pattern `atomic_f` or the pattern `atomic_f_explicit` invokes the member function `f`, with the value of the first parameter as the object expression and the values of the remaining parameters (if any) as the arguments of the member function call, in order. An argument for a parameter of type `atomic<T>::value_type*` is dereferenced when passed to the member function call. If no such member function exists, the program is ill-formed.

```
template<class T>
  void atomic_init(volatile atomic<T>* object, typename atomic<T>::value_type desired) noexcept;
template<class T>
  void atomic_init(atomic<T>* object, typename atomic<T>::value_type desired) noexcept;
```

2 *Effects:* Non-atomically initializes `*object` with value `desired`. This function shall only be applied to objects that have been default constructed, and then only once. [*Note*: These semantics ensure compatibility with C. — *end note*] [*Note*: Concurrent access from another thread, even via an atomic operation, constitutes a data race. — *end note*]

3 [*Note*: The non-member functions enable programmers to write code that can be compiled as either C or C++, for example in a shared header file. — *end note*]

31.9 Flag type and operations [atomics.flag]

```
namespace std {
  struct atomic_flag {
    bool test_and_set(memory_order = memory_order::seq_cst) volatile noexcept;
    bool test_and_set(memory_order = memory_order::seq_cst) noexcept;
    void clear(memory_order = memory_order::seq_cst) volatile noexcept;
    void clear(memory_order = memory_order::seq_cst) noexcept;

    atomic_flag() noexcept = default;
    atomic_flag(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) = delete;
    atomic_flag& operator=(const atomic_flag&) volatile = delete;
  };

  bool atomic_flag_test_and_set(volatile atomic_flag*) noexcept;
  bool atomic_flag_test_and_set(atomic_flag*) noexcept;
  bool atomic_flag_test_and_set_explicit(volatile atomic_flag*, memory_order) noexcept;
  bool atomic_flag_test_and_set_explicit(atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear(volatile atomic_flag*) noexcept;
```

```
  void atomic_flag_clear(atomic_flag*) noexcept;
  void atomic_flag_clear_explicit(volatile atomic_flag*, memory_order) noexcept;
  void atomic_flag_clear_explicit(atomic_flag*, memory_order) noexcept;

  #define ATOMIC_FLAG_INIT see below
}
```

1 The `atomic_flag` type provides the classic test-and-set functionality. It has two states, set and clear.

2 Operations on an object of type `atomic_flag` shall be lock-free. [*Note*: Hence the operations should also be address-free. — *end note*]

3 The `atomic_flag` type is a standard-layout struct. It has a trivial default constructor and a trivial destructor.

4 The macro `ATOMIC_FLAG_INIT` shall be defined in such a way that it can be used to initialize an object of type `atomic_flag` to the clear state. The macro can be used in the form:

```
atomic_flag guard = ATOMIC_FLAG_INIT;
```

It is unspecified whether the macro can be used in other initialization contexts. For a complete static-duration object, that initialization shall be static. Unless initialized with `ATOMIC_FLAG_INIT`, it is unspecified whether an `atomic_flag` object has an initial state of set or clear.

```
bool atomic_flag_test_and_set(volatile atomic_flag* object) noexcept;
bool atomic_flag_test_and_set(atomic_flag* object) noexcept;
bool atomic_flag_test_and_set_explicit(volatile atomic_flag* object, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(atomic_flag* object, memory_order order) noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order::seq_cst) volatile noexcept;
bool atomic_flag::test_and_set(memory_order order = memory_order::seq_cst) noexcept;
```

5 *Effects:* Atomically sets the value pointed to by `object` or by `this` to `true`. Memory is affected according to the value of `order`. These operations are atomic read-modify-write operations (**??**).

6 *Returns:* Atomically, the value of the object immediately before the effects.

```
void atomic_flag_clear(volatile atomic_flag* object) noexcept;
void atomic_flag_clear(atomic_flag* object) noexcept;
void atomic_flag_clear_explicit(volatile atomic_flag* object, memory_order order) noexcept;
void atomic_flag_clear_explicit(atomic_flag* object, memory_order order) noexcept;
void atomic_flag::clear(memory_order order = memory_order::seq_cst) volatile noexcept;
void atomic_flag::clear(memory_order order = memory_order::seq_cst) noexcept;
```

7 ~~*Requires:*~~ *Expects:* The `order` argument ~~shall not be~~is neither `memory_order::consume`, `memory_order::acquire`, nor `memory_order::acq_rel`.

8 *Effects:* Atomically sets the value pointed to by `object` or by `this` to `false`. Memory is affected according to the value of `order`.

31.10 Fences [atomics.fences]

1 This subclause introduces synchronization primitives called *fences*. Fences can have acquire semantics, release semantics, or both. A fence with acquire semantics is called an *acquire fence*. A fence with release semantics is called a *release fence*.

2 A release fence A synchronizes with an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

3 A release fence A synchronizes with an atomic operation B that performs an acquire operation on an atomic object M if there exists an atomic operation X such that A is sequenced before X, X modifies M, and B reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

4 An atomic operation A that is a release operation on an atomic object M synchronizes with an acquire fence B if there exists some atomic operation X on M such that X is sequenced before B and reads the value written by A or a value written by any side effect in the release sequence headed by A.

```
extern "C" void atomic_thread_fence(memory_order order) noexcept;
```

5 *Effects:* Depending on the value of `order`, this operation:

(5.1) — has no effects, if `order == memory_order::relaxed`;

(5.2) — is an acquire fence, if `order == memory_order::acquire` or `order == memory_order::consume`;

(5.3) — is a release fence, if `order == memory_order::release`;

(5.4) — is both an acquire fence and a release fence, if `order == memory_order::acq_rel`;

(5.5) — is a sequentially consistent acquire and release fence, if `order == memory_order::seq_cst`.

```
extern "C" void atomic_signal_fence(memory_order order) noexcept;
```

6 *Effects:* Equivalent to `atomic_thread_fence(order)`, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.

7 [*Note*: `atomic_signal_fence` can be used to specify the order in which actions performed by the thread become visible to the signal handler. Compiler optimizations and reorderings of loads and stores are inhibited in the same way as with `atomic_thread_fence`, but the hardware fence instructions that `atomic_thread_fence` would have inserted are not emitted. — *end note*]