Resolving `atomic<T>` named base class inconsistencies

ISO/IEC JTC1 SC22 WG21 P0558R0
Billy Robert O'Neal III bion@microsoft.com
Jens Maurer Jens.Maurer@gmx.net
Target audience: SG1, LWG
2017-02-03

Introduction

This paper resolves inconsistencies allowed by C++11-17's Clause 29 atomic specification, which allow programs to compile under some implementations but not others. It also removes overspecification where this is not detectable to users, and actively harmful to standard library implementations. Other parts of Clause 29, such as `kill_dependency`, `atomic_flag`, and fences are unaffected.

Inconsistencies caused by "named base classes"

Consider the following function:

```
long example(atomic<long> * a) {
    return atomic_fetch_add(a, 42);
}
```

The C++ standard is inconsistent about whether this program should compile. N4618 29.5[atomics.types.generic]/7 says:

> There shall be named types corresponding to the integral specializations of atomic, as specified in Table 138, and a named type `atomic_bool` corresponding to the specified `atomic<bool>`. Each named type is either a typedef to the corresponding specialization or a base class of the corresponding specialization. [...]

And later in 29.6.3 [atomics.types.operations.arith]/2 says when describing integral nonmembers (like `atomic_fetch_add`) that:

> In the declarations of these functions and function template specializations, the name integral refers to an integral type and the name atomic-integral refers to either `atomic<`*`integral`*`>` or to a named base class for integral from Table 138 or inferred from Table 139.

This means that our example function above has 2 different answers. If an implementation does not choose to have named base classes, then this program does not compile. The signature

```
template <class T>
T atomic_fetch_add(atomic<T> * a, T x);
```

tries to deduce `T = long` for the parameter `a`, and `T = int` for parameter `x`, and type deduction fails. If an implementation uses the named base class option, then the above type deduction failure still occurs, but an additional signature is present which would be viable:

```
long atomic_fetch_add(atomic_long * a, long);
```

That is, the atomic's type "wins" and does a promotion on the second parameter from int to long.

We propose to make this program compile on all implementations, by disabling type deduction on the second parameter of the atomic_xxx nonmember functions. We can disable type deduction by

providing `value_type` and `difference_type` typedefs in std::atomic (which may be useful for other uses), and changing the second parameter to use this typedef:

```
template <class T>
T atomic_fetch_add(volatile atomic<T>*, typename atomic<T>::difference_type) noexcept;
```

`atomic<T>::value_type` is always `T`, and `atomic<T>::difference_type` is the type of the second parameter for add and subtract functions; `T` for integral types and `ptrdiff_t` for pointers.
This change has a number of positive characteristics:

- The named base class and typedef based implementations of std::atomic become equivalent, and the C compatibility nonmember functions can be specified as "Effects: Equivalent to member function call"
- LWG 2715 is resolved, as the class definition in the synopsis now serves to indicate that list initialization is possible.
- Other kinds of operations where the user expects to convert to the atomic's type now compile. For example, the following function trips the same type deduction failure:

```
void example() {
    struct X{};
    atomic<X *> ptr{};
    atomic_store(&ptr, 0); // is T X * or int?
}
```

`atomic<T *>` providing operations not available on `T *`

Today, atomic is depicted as having a partial specialization for `T *`, and unconditionally provides operations like `++` and `--`for pointer types. However, not all pointer types themselves provide `++` and `--`. For example, that partial specialization is matched for `void *`, and for function pointer types like `int (*)(int)`, for which `++` and `--` make no semantic sense and would be ill-formed.
Because the core language doesn't say what happens with such pointers, and Clause 29 did not specify, there has been implementation divergence in this area. MSVC++ and libstdc++ implement these operations as if the `void *` or `int (*)(int)` were a `char *` (producing invalid function pointers!), and for libc++ it fails to compile.

```
#include <stdio.h>
#include <atomic>

int main() {
    std::atomic<int (*)(int)> x{0};
    ++x;
    printf("%p\n", x.load());
}
```

This seems like a specification oversight and we should not be allowing such programs to compile.

Cleaning up unobservable specification details

- The `atomic<integral>` description said that `atomic<integral>` had to be implemented as an explicit specialization, which is undetectable to users. An implementation could choose to provide the correct operations by selecting an appropriate base class.
- Fixed use of "shall" to indicate a requirement for standard libraries (rather than users).

29 Atomic operations library [atomics]

29.1 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 1 — Atomics library summary

	Subclause	Header(s)
29.3	Order and Consistency	
29.4	Lock-free Property	
29.5	Atomic Types	`<atomic>`
29.6	Operations on Atomic Types	
29.9	Flag Type and Operations	
29.10	Fences	

29.2 Header `<atomic>` synopsis [atomics.syn]

```
namespace std {
  // 29.3, order and consistency
  enum memory_order;
  template <class T>
    T kill_dependency(T y) noexcept;

  // 29.4, lock-free property
  #define ATOMIC_BOOL_LOCK_FREE unspecified
  #define ATOMIC_CHAR_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

  // 29.5, generic types atomic
  template<class T> struct atomic;
  template<> struct atomic<integral>;
  // 29.6.7, partial specialization for pointers
  template<class T> struct atomic<T*>;

  // 29.6.1, general operations on atomic types
  // In the following declarations, atomic-type is either atomic<T> or a named base class
  // for T from Table ?? or inferred from Table ?? or from bool. If it is atomic<T>,
  // then the declaration is a template declaration prefixed with template <class T>.
  // 29.7, non-member functions
```

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

```
    memory_order, memory_order) noexcept;
template<class T>
  bool atomic_compare_exchange_strong_explicit(volatile atomic-type 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-type 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>::difference_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>*, Ttypename atomic<T>::value_type,
    memory_order) noexcept;
template <class T>
  T atomic_fetch_xor_explicit(atomic<T>*, Ttypename atomic<T>::value_type,
    memory_order) noexcept;

// 29.6.3, arithmetic operations on atomic types
// In the following declarations, atomic-integral is either atomic<T> or a named base class
// for T from Table ?? or inferred from Table ??. If it is atomic<T>, then the declaration
// is a template specialization declaration prefixed with template <>.
integral atomic_fetch_add(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_add(atomic-integral*, integral) noexcept;
integral atomic_fetch_add_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_add_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_sub(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_sub(atomic-integral*, integral) noexcept;
integral atomic_fetch_sub_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_sub_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_and(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_and(atomic-integral*, integral) noexcept;
integral atomic_fetch_and_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_and_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_or(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_or(atomic-integral*, integral) noexcept;
integral atomic_fetch_or_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_or_explicit(atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_xor(volatile atomic-integral*, integral) noexcept;
integral atomic_fetch_xor(atomic-integral*, integral) noexcept;
integral atomic_fetch_xor_explicit(volatile atomic-integral*, integral, memory_order) noexcept;
integral atomic_fetch_xor_explicit(atomic-integral*, integral, memory_order) noexcept;

// 29.6.4, overloads for pointers
template <class T>
 T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
 T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
 T* atomic_fetch_add_explicit(volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
template <class T>
 T* atomic_fetch_add_explicit(atomic<T*>*, ptrdiff_t, memory_order) noexcept;
template <class T>
 T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
 T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
template <class T>
 T* atomic_fetch_sub_explicit(volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
template <class T>
 T* atomic_fetch_sub_explicit(atomic<T*>*, ptrdiff_t, memory_order) noexcept;

// 29.6.5, initialization
#define ATOMIC_VAR_INIT(value) see below

// 29.8, 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_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>;

// 29.9, 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

  // 29.10, fences
  extern "C" void atomic_thread_fence(memory_order) noexcept;
  extern "C" void atomic_signal_fence(memory_order) noexcept;
}
```

29.3 Order and consistency [atomics.order]

```
namespace std {
  enum memory_order {
    memory_order_relaxed, memory_order_consume, memory_order_acquire,
    memory_order_release, memory_order_acq_rel, 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 There shall be a single total order S on all `memory_order_seq_cst` operations, consistent with the "happens before" order and modification orders for all affected locations, such that each `memory_order_seq_cst` operation B that loads a value from an atomic object M observes one of the following values:

(3.1) — the result of the last modification A of M that precedes B in S, if it exists, or

(3.2) — if A exists, the result of some modification of M that is not `memory_order_seq_cst` and that does not happen before A, or

(3.3) — if A does not exist, the result of some modification of M that is not `memory_order_seq_cst`.

[*Note:* Although it is not explicitly required that S include locks, it can always be extended to an order that does include lock and unlock operations, since the ordering between those is already included in the "happens before" ordering. — *end note*]

4 For an atomic operation B that reads the value of an atomic object M, if there is a `memory_order_seq_cst` fence X sequenced before B, then B observes either the last `memory_order_seq_cst` modification of M

preceding X in the total order S or a later modification of M in its modification order.

5 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there is a `memory_order_seq_cst` fence X such that A is sequenced before X and B follows X in S, then B observes either the effects of A or a later modification of M in its modification order.

6 For atomic operations A and B on an atomic object M, where A modifies M and B takes its value, if there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S, then B observes either the effects of A or a later modification of M in its modification order.

7 For atomic modifications A and B of an atomic object M, B occurs later than A in the modification order of M if:

(7.1) — there is a `memory_order_seq_cst` fence X such that A is sequenced before X, and X precedes B in S, or

(7.2) — there is a `memory_order_seq_cst` fence Y such that Y is sequenced before B, and A precedes Y in S, or

(7.3) — there are `memory_order_seq_cst` fences X and Y such that A is sequenced before X, Y is sequenced before B, and X precedes Y in S.

8 [*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` operations. Any use of weaker ordering will invalidate this guarantee unless extreme care is used. In particular, `memory_order_seq_cst` fences ensure a total order only for the fences themselves. Fences cannot, in general, be used to restore sequential consistency for atomic operations with weaker ordering specifications. — *end note*]

9 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*]

10 [*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*]

11 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.

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

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

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

14 *Returns:* `y`.

29.4 Lock-free property [atomics.lockfree]

```
#define ATOMIC_BOOL_LOCK_FREE unspecified
#define ATOMIC_CHAR_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` (29.6) 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*]

29.5 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 ~~There is a generic class template `atomic<T>`.~~ The ~~type of 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 semantics of the operations on specializations of `atomic` are defined in 29.6.~~

3 ~~Specializations and instantiations of the `atomic` template shall have a deleted copy constructor, a deleted copy assignment operator, and a constexpr value constructor.~~

4 [*Note:* The representation of an atomic specialization need not have the same size as its corresponding argument type. Specializations should have the same size whenever possible, as this reduces the effort required to port existing code. — *end note*]

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

[Editor's note: Remove all of the following subsections and the heading preceding this note.]

29.6.1 General operations on atomic types [atomics.types.operations.general]

1 ~~The implementation shall provide the functions and function templates identified as "general operations on atomic types" in 29.2.~~

2 ~~In the declarations of these functions and function templates, the name *`atomic-type`* refers to either `atomic<T>` or to a named base class for `T` from Table **??** or inferred from Table **??**.~~

29.6.2 Templated operations on atomic types [atomics.types.operations.templ]

1 ~~The implementation shall declare but not define the function templates identified as "templated operations on atomic types" in 29.2.~~

29.6.3 Arithmetic operations on atomic types [atomics.types.operations.arith]

1 ~~The implementation shall provide the functions and function template specializations identified as "arithmetic operations on atomic types" in 29.2.~~

2 ~~In the declarations of these functions and function template specializations, the name *`integral`* refers to an integral type and the name *`atomic-integral`* refers to either `atomic<`*`integral`*`>` or to a named base class for *`integral`* from Table **??** or inferred from Table **??**.~~

29.6.4 Operations on atomic pointer types [atomics.types.operations.pointer]

1 ~~The implementation shall provide the function template specializations identified as "partial specializations for pointers" in 29.2.~~

29.6.5 Requirements for operations on atomic types [atomics.types.operations.req]

1 ~~There are only a few kinds of operations on atomic types, though there are many instances on those kinds. This section specifies each general kind. The specific instances are defined in 29.5, 29.6.1, 29.6.3, and 29.6.4.~~

2 ~~In the following operation definitions:~~

(2.1) — ~~an *A* refers to one of the atomic types.~~

(2.2) — ~~a *C* refers to its corresponding non-atomic type.~~

(2.3) — ~~an *M* refers to type of the other argument for arithmetic operations. For integral atomic types, *M* is *C*. For atomic address types, *M* is `ptrdiff_t`.~~

(2.4) — ~~the non-member functions not ending in `_explicit` have the semantics of their corresponding `_explicit` functions with `memory_order` arguments of `memory_order_seq_cst`.~~

3 [*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. ~~Thus, volatile qualified operations on non-volatile objects may be merged under some conditions.~~ — *end note*]

```
~~A::A~~atomic() noexcept = default;
```

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

```
constexpr ~~A::A~~atomic(~~C~~T desired) noexcept;
```

5 *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
```

6 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;
```

7 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 atomic_is_lock_free(const volatile A* object) noexcept;~~
~~bool atomic_is_lock_free(const A* object) noexcept;~~
bool ~~A::~~is_lock_free() const volatile noexcept;
bool ~~A::~~is_lock_free() const noexcept;
```

8 *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 atomic_store(volatile A* object, C desired) noexcept;~~
~~void atomic_store(A* object, C desired) noexcept;~~
~~void atomic_store_explicit(volatile A* object, C desired, memory_order order) noexcept;~~
~~void atomic_store_explicit(A* object, C desired, memory_order order) noexcept;~~
void ~~A::~~store(~~C~~T desired, memory_order order = memory_order_seq_cst) volatile noexcept;
void ~~A::~~store(~~C~~T desired, memory_order order = memory_order_seq_cst) noexcept;
```

9 *Requires:* The `order` argument shall not be `memory_order_consume`, `memory_order_acquire`, nor

`memory_order_acq_rel`.

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

```
~~C~~ T ~~A::~~operator=(~~C~~ T desired) volatile noexcept;
~~C~~ T ~~A::~~operator=(~~C~~ T desired) noexcept;
```

11 *Effects:* ~~As if by~~ `store(desired)`.

12 *Returns:* `desired`.

```
~~C atomic_load(const volatile A* object) noexcept;~~
~~C atomic_load(const A* object) noexcept;~~
~~C atomic_load_explicit(const volatile A* object, memory_order) noexcept;~~
~~C atomic_load_explicit(const A* object, memory_order) noexcept;~~
~~C~~ T ~~A::~~load(memory_order order = memory_order_seq_cst) const volatile noexcept;
~~C~~ T ~~A::~~load(memory_order order = memory_order_seq_cst) const noexcept;
```

13 *Requires:* The `order` argument shall not be `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 pointed to ~~by object or~~ by `this`.

```
~~A::~~operator ~~C~~T() const volatile noexcept;
~~A::~~operator ~~C~~T() const noexcept;
```

16 *Effects:* ~~As if by `load()`~~ Equivalent to: `return load();`

Returns: ~~The result of `load()`.~~

```
~~C atomic_exchange(volatile A* object, C desired) noexcept;~~
~~C atomic_exchange(A* object, C desired) noexcept;~~
~~C atomic_exchange_explicit(volatile A* object, C desired, memory_order) noexcept;~~
~~C atomic_exchange_explicit(A* object, C desired, memory_order) noexcept;~~
~~C~~ T ~~A::~~exchange(~~C~~ T desired, memory_order order = memory_order_seq_cst) volatile noexcept;
~~C~~ T ~~A::~~exchange(~~C~~ T desired, memory_order order = memory_order_seq_cst) noexcept;
```

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

18 *Returns:* Atomically returns the value pointed to ~~by object or~~ by `this` immediately before the effects.

```
~~bool atomic_compare_exchange_weak(volatile A* object, C* expected, C desired) noexcept;~~
~~bool atomic_compare_exchange_weak(A* object, C* expected, C desired) noexcept;~~
~~bool atomic_compare_exchange_strong(volatile A* object, C* expected, C desired) noexcept;~~
~~bool atomic_compare_exchange_strong(A* object, C* expected, C desired) noexcept;~~
~~bool atomic_compare_exchange_weak_explicit(volatile A* object, C* expected, C desired,~~
    ~~memory_order success, memory_order failure) noexcept;~~
~~bool atomic_compare_exchange_weak_explicit(A* object, C* expected, C desired,~~
    ~~memory_order success, memory_order failure) noexcept;~~
~~bool atomic_compare_exchange_strong_explicit(volatile A* object, C* expected, C desired,~~
    ~~memory_order success, memory_order failure) noexcept;~~
~~bool atomic_compare_exchange_strong_explicit(A* object, C* expected, C desired,~~
    ~~memory_order success, memory_order failure) noexcept;~~
bool ~~A::~~compare_exchange_weak(~~C~~T& expected, ~~C~~T desired,
    memory_order success, memory_order failure) volatile noexcept;
bool ~~A::~~compare_exchange_weak(~~C~~T& expected, ~~C~~T desired,
    memory_order success, memory_order failure) noexcept;
bool ~~A::~~compare_exchange_strong(~~C~~T& expected, ~~C~~T desired,
```

```
    memory_order success, memory_order failure) volatile noexcept;
bool A::compare_exchange_strong(CT& expected, CT desired,
    memory_order success, memory_order failure) noexcept;
bool A::compare_exchange_weak(CT& expected, CT desired,
    memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_weak(CT& expected, CT desired,
    memory_order order = memory_order_seq_cst) noexcept;
bool A::compare_exchange_strong(CT& expected, CT desired,
    memory_order order = memory_order_seq_cst) volatile noexcept;
bool A::compare_exchange_strong(CT& expected, CT desired,
    memory_order order = memory_order_seq_cst) noexcept;
```

19 *Requires:* The `failure` argument shall not be `memory_order_release` nor `memory_order_acq_rel`.

20 *Effects:* Retrieves the value in `expected`. It then atomically compares the contents of the memory pointed to ~~by object or~~ by `this` for equality with that previously retrieved from `expected`, and if true, replaces the contents of the memory pointed to ~~by object or~~ 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 contents of the memory in `expected` are replaced by the value read ~~from object or~~ 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` ~~or object~~. Otherwise, these operations are atomic load operations on that memory.

21 *Returns:* The result of the comparison.

22 [*Note:* For example, the effect of `atomic_compare_exchange_strong` is

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

— *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. E.g. 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*]

23 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.

24 *Remarks:* A weak compare-and-exchange operation may fail spuriously. That is, even when the

contents of memory referred to by `expected` and ~~`object`~~`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*]

25 [*Note:* The `memcpy` and `memcmp` semantics of the compare-and-exchange operations may result in failed comparisons for values that compare equal with `operator==` if the underlying type has padding bits, trap bits, or alternate representations of the same value. Thus, `compare_exchange_strong` should be used with extreme care. On the other hand, `compare_exchange_weak` should converge rapidly. —*end note*]

29.6.6 Specializations for integers [atomics.types.int]

1 There ~~shall be explicit~~ are specializations of the `atomic` template for the integral types `char`, `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `char16_t`, `char32_t`, `wchar_t`, and any other types needed by the typedefs in section **??** ~~the header <cstdint>~~. For each such integral type *integral*, the specialization `atomic<`*integral*`>` provides additional atomic operations appropriate to integral types. ~~There shall be a specialization `atomic<bool>` which provides the general atomic operations as specified in 29.6.1.~~

```
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 and the specialization `atomic<bool>` ~~shall be~~ are standard-layout structs. They ~~shall~~ each have a trivial default constructor and a trivial destructor. ~~They shall each support aggregate initialization syntax.~~

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 2 — Atomic arithmetic computations

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

~~*C* `atomic_fetch_`*key*`(volatile` *A*`* object,` *M* `operand) noexcept;`~~

~~*C* atomic_fetch_*key*(*A** object, *M* operand) noexcept;~~
~~*C* atomic_fetch_*key*_explicit(volatile *A** object, *M* operand, memory_order order) noexcept;~~
~~*C* atomic_fetch_*key*_explicit(*A** object, *M* operand, memory_order order) noexcept;~~
~~*C*~~ *T* ~~*A*::~~fetch_*key*(*M* operand, memory_order order = memory_order_seq_cst) volatile noexcept;
~~*C*~~ *T* ~~*A*::~~fetch_*key*(*M* operand, memory_order order = memory_order_seq_cst) noexcept;

5 *Effects:* Atomically replaces the value pointed to ~~by object or~~ by `this` with the result of the computation applied to the value pointed to ~~by object or~~ 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 object or~~ by `this` immediately before the effects.

7 *Remarks:* For signed integer types, arithmetic is defined to use two's complement representation. There are no undefined results. For address types, the result may be an undefined address, but the operations otherwise have no undefined behavior.

~~*C*~~ *T* ~~*A*::~~operator *op*=(*M* operand) volatile noexcept;
~~*C*~~ *T* ~~*A*::~~operator *op*=(*M* operand) noexcept;

8 *Effects:* ~~As if by fetch_*key*(operand).~~ Equivalent to: `return fetch_`*key*`(operand)` *op* operand;

~~*Returns:* fetch_*key*(operand) op operand.~~

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

```
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 ~~shall be pointer~~ is a partial specialization~~s~~ of the `atomic` class template for pointers. ~~These s~~Specializations ~~shall be~~ of this partial specialization are standard-layout structs. They ~~shall~~ each have a trivial default constructor and a trivial destructor. ~~They shall each support aggregate initialization syntax.~~

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

[Editor's note: The following provisions are cloned from the integer arithmetic section to avoid the *M* placeholders.]

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

Table 3 — 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:* T shall be an object type. [*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.

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

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

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

```
~~C~~T ~~A::~~operator++(int) volatile noexcept;
~~C~~T ~~A::~~operator++(int) noexcept;
```

1 ~~*Returns:* fetch_add(1).~~

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

~~C~~T ~~A::~~operator--(int) volatile noexcept;
~~C~~T ~~A::~~operator--(int) noexcept;

2 ~~*Returns:* fetch_sub(1).~~

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

~~C~~T ~~A::~~operator++() volatile noexcept;
~~C~~T ~~A::~~operator++() noexcept;

3 *Effects:* ~~As if by `fetch_add(1)`.~~ Equivalent to: `return fetch_add(1) + 1;`

~~*Returns:* fetch_add(1) + 1.~~

~~C~~T ~~A::~~operator--() volatile noexcept;
~~C~~T ~~A::~~operator--() noexcept;

4 *Effects:* ~~As if by `fetch_sub(1)`.~~ Equivalent to: `return fetch_sub(1) - 1;`

~~*Returns:* fetch_sub(1) - 1.~~

29.7 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. [*Note:* If no such member function exists, the program is ill-formed. — *end note*]

```
template<class T>
  void atomic_init(volatile Aatomic<T>* object, Ctypename atomic<T>::value_type desired) noexcept;
template<class T>
  void atomic_init(Aatomic<T>* object, Ctypename 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. [*Example:*

```
atomic_int use_count;
void init() {
  atomic_init(&use_count, 0);
}
void add_use() {
  atomic_fetch_add(&use_count, 1);
}
bool remove_use {       // returns true if last use was removed
  return atomic_fetch_sub(&use_count, 1) == 1;
}
```

— *end example*] — *end note*]

29.8 Type aliases [atomics.alias]

1 ~~There shall be named types corresponding to the integral specializations of `atomic`, as specified in Table ??, and a named type `atomic_bool` corresponding to the specified `atomic<bool>`. Each named type is either a typedef to the corresponding specialization or a base class of the corresponding specialization. If it is a base class, it shall support the same member functions as the corresponding specialization.~~

2 ~~There shall be atomic typedefs corresponding to non-atomic typedefs as specified in Table ??.~~ The type

aliases `atomic_int`N`_t`, `atomic_uint`N`_t`, `atomic_intptr_t`, and `atomic_uintptr_t` ~~shall be~~ are defined if and only if `int`N`_t`, `uint`N`_t`, `intptr_t`, and `uintptr_t` are defined, respectively.

29.9 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. No other type requires lock-free operations, so the `atomic_flag` type is the minimum hardware-implemented type needed to conform to this International Standard. The remaining types can be emulated with `atomic_flag`, though with less than ideal properties. — *end note*]

3 The `atomic_flag` type shall be a standard-layout struct. It shall have a trivial default constructor, a deleted copy constructor, a deleted copy assignment operator, 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:* The `order` argument shall not be `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`.

29.10 Fences [atomics.fences]

1 This section 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 || 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.

8 *Note:* 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.