# std::atomic

| Defined in header <atomic> | | |
|---|---|---|
| `template< class T > struct atomic;` | (1) | (since C++11) |
| `template< class U > struct atomic<U*>;` | (2) | (since C++11) |
| Defined in header <memory> | | |
| `template< class U > struct atomic<std::shared_ptr<U>>;` | (3) | (since C++20) |
| `template< class U > struct atomic<std::weak_ptr<U>>;` | (4) | (since C++20) |

Each instantiation and full specialization of the `std::atomic` template defines an atomic type. If one thread writes to an atomic object while another thread reads from it, the behavior is well-defined (see memory model for details on data races).

In addition, accesses to atomic objects may establish inter-thread synchronization and order non-atomic memory accesses as specified by `std::memory_order`.

std::atomic is neither copyable nor movable.

## Specializations

### Primary template

The primary `std::atomic` template may be instantiated with any *TriviallyCopyable* type T satisfying both *CopyConstructible* and *CopyAssignable*. The program is ill-formed if any of following values is `false`:

- `std::is_trivially_copyable<T>::value`
- `std::is_copy_constructible<T>::value`
- `std::is_move_constructible<T>::value`
- `std::is_copy_assignable<T>::value`
- `std::is_move_assignable<T>::value`

```
struct Counters { int a; int b; }; // user-defined trivially-copyable type
std::atomic<Counters> cnt;         // specialization for the user-defined type
```

`std::atomic<bool>` uses the primary template. It is guaranteed to be a standard layout struct.

### Partial specializations

The standard library provides partial specializations of the `std::atomic` template for the following types with additional properties that the primary template does not have:

2) Partial specializations `std::atomic<U*>` for all pointer types. These specializations have standard layout, trivial default constructors, (until C++20) and trivial destructors. Besides the operations provided for all atomic types, these specializations additionally support atomic arithmetic operations appropriate to pointer types, such as `fetch_add`, `fetch_sub`.

3-4) Partial specializations `std::atomic<std::shared_ptr<U>>` and `std::atomic<std::weak_ptr<U>>` are provided for `std::shared_ptr` and `std::weak_ptr`. (since C++20)

See std::atomic<std::shared_ptr> and std::atomic<std::weak_ptr> for details.

### Specializations for integral types

When instantiated with one of the following integral types, `std::atomic` provides additional atomic operations appropriate to integral types such as `fetch_add`, `fetch_sub`, `fetch_and`, `fetch_or`, `fetch_xor`:

- The character types `char`, `char8_t` (since C++20), `char16_t`, `char32_t`, and `wchar_t`;
- The standard signed integer types: `signed char`, `short`, `int`, `long`, and `long long`;
- The standard unsigned integer types: `unsigned char`, `unsigned short`, `unsigned int`, `unsigned long`, and `unsigned long long`;
- Any additional integral types needed by the typedefs in the header `<cstdint>`.

Additionally, the resulting `std::atomic<`*`Integral`*`>` specialization has standard layout, a trivial default constructor, (until C++20) and a trivial destructor. Signed integer arithmetic is defined to use two's complement; there are no undefined results.

### Specializations for floating-point types

When instantiated with one of the floating-point types `float`, `double`, and `long double`, `std::atomic` provides additional atomic operations appropriate to floating-point types such as `fetch_add` and `fetch_sub`. (since C++20)

Additionally, the resulting `std::atomic<`*`Floating`*`>` specialization has standard layout and a trivial destructor.

No operations result in undefined behavior even if the result is not representable in the floating-point type. The floating-point environment in effect may be different from the calling thread's floating-point environment.

## Type aliases

Type aliases are provided for `bool` and all integral types listed above, as follows:

| Type alias | Definition |
|---|---|
| `std::atomic_bool` | `std::atomic<bool>` |
| `std::atomic_char` | `std::atomic<char>` |
| `std::atomic_schar` | `std::atomic<signed char>` |
| `std::atomic_uchar` | `std::atomic<unsigned char>` |
| `std::atomic_short` | `std::atomic<short>` |
| `std::atomic_ushort` | `std::atomic<unsigned short>` |
| `std::atomic_int` | `std::atomic<int>` |
| `std::atomic_uint` | `std::atomic<unsigned int>` |
| `std::atomic_long` | `std::atomic<long>` |
| `std::atomic_ulong` | `std::atomic<unsigned long>` |
| `std::atomic_llong` | `std::atomic<long long>` |
| `std::atomic_ullong` | `std::atomic<unsigned long long>` |
| `std::atomic_char8_t` (C++20) | `std::atomic<char8_t>` |
| `std::atomic_char16_t` | `std::atomic<char16_t>` |
| `std::atomic_char32_t` | `std::atomic<char32_t>` |
| `std::atomic_wchar_t` | `std::atomic<wchar_t>` |
| `std::atomic_int8_t` | `std::atomic<std::int8_t>` |
| `std::atomic_uint8_t` | `std::atomic<std::uint8_t>` |
| `std::atomic_int16_t` | `std::atomic<std::int16_t>` |
| `std::atomic_uint16_t` | `std::atomic<std::uint16_t>` |
| `std::atomic_int32_t` | `std::atomic<std::int32_t>` |
| `std::atomic_uint32_t` | `std::atomic<std::uint32_t>` |
| `std::atomic_int64_t` | `std::atomic<std::int64_t>` |
| `std::atomic_uint64_t` | `std::atomic<std::uint64_t>` |
| `std::atomic_int_least8_t` | `std::atomic<std::int_least8_t>` |
| `std::atomic_uint_least8_t` | `std::atomic<std::uint_least8_t>` |
| `std::atomic_int_least16_t` | `std::atomic<std::int_least16_t>` |
| `std::atomic_uint_least16_t` | `std::atomic<std::uint_least16_t>` |
| `std::atomic_int_least32_t` | `std::atomic<std::int_least32_t>` |
| `std::atomic_uint_least32_t` | `std::atomic<std::uint_least32_t>` |
| `std::atomic_int_least64_t` | `std::atomic<std::int_least64_t>` |
| `std::atomic_uint_least64_t` | `std::atomic<std::uint_least64_t>` |
| `std::atomic_int_fast8_t` | `std::atomic<std::int_fast8_t>` |
| `std::atomic_uint_fast8_t` | `std::atomic<std::uint_fast8_t>` |
| `std::atomic_int_fast16_t` | `std::atomic<std::int_fast16_t>` |
| `std::atomic_uint_fast16_t` | `std::atomic<std::uint_fast16_t>` |
| `std::atomic_int_fast32_t` | `std::atomic<std::int_fast32_t>` |
| `std::atomic_uint_fast32_t` | `std::atomic<std::uint_fast32_t>` |
| `std::atomic_int_fast64_t` | `std::atomic<std::int_fast64_t>` |
| `std::atomic_uint_fast64_t` | `std::atomic<std::uint_fast64_t>` |
| `std::atomic_intptr_t` | `std::atomic<std::intptr_t>` |
| `std::atomic_uintptr_t` | `std::atomic<std::uintptr_t>` |
| `std::atomic_size_t` | `std::atomic<`*`std::size_t`*`>` |
| `std::atomic_ptrdiff_t` | `std::atomic<std::ptrdiff_t>` |
| `std::atomic_intmax_t` | `std::atomic<std::intmax_t>` |
| `std::atomic_uintmax_t` | `std::atomic<std::uintmax_t>` |

Note: `std::atomic_int`*N*`_t`, `std::atomic_uint`*N*`_t`, `std::atomic_intptr_t`, and `atomic_uintptr_t` are defined if and only if `std::int`*N*`_t`, `std::uint`*N*`_t`, `std::intptr_t`, and `std::uintptr_t` are defined, respectively.

Additional special-purpose type aliases are provided:

| | | |
|---|---|---|
| `std::atomic_signed_lock_free` | a signed integral atomic type that is lock-free and for which waiting/notifying is most efficient | (since C++20) |
| `std::atomic_unsigned_lock_free` | an unsigned integral atomic type that is lock-free and for which waiting/notifying is most efficient | |

### Member types

| Member type | Definition |
|---|---|
| | T (regardless of whether specialized or not) |

| | |
|---|---|
| `value_type` | |
| `difference_type` | `value_type` (only for `atomic<`*`Integral`*`>` and `atomic<`*`Floating`*`>` (since C++20) specializations)<br>`std::ptrdiff_t` (only for `atomic<U*>` specializations) |

`difference_type` is not defined in the primary `atomic` template or in the partial specializations for `std::shared_ptr` and `std::weak_ptr`.

### Member functions

| | |
|---|---|
| (constructor) | constructs an atomic object<br>(public member function) |
| **operator=** | stores a value into an atomic object<br>(public member function) |
| **is_lock_free** | checks if the atomic object is lock-free<br>(public member function) |
| **store** | atomically replaces the value of the atomic object with a non-atomic argument<br>(public member function) |
| **load** | atomically obtains the value of the atomic object<br>(public member function) |
| **operator T** | loads a value from an atomic object<br>(public member function) |
| **exchange** | atomically replaces the value of the atomic object and obtains the value held previously<br>(public member function) |
| **compare_exchange_weak**<br>**compare_exchange_strong** | atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not<br>(public member function) |
| **wait** (C++20) | blocks the thread until notified and the atomic value changes<br>(public member function) |
| **notify_one** (C++20) | notifies at least one thread waiting on the atomic object<br>(public member function) |
| **notify_all** (C++20) | notifies all threads blocked waiting on the atomic object<br>(public member function) |

### Constants

| | |
|---|---|
| **is_always_lock_free** [static](C++17) | indicates that the type is always lock-free<br>(public static member constant) |

### Specialized member functions

| | |
|---|---|
| **fetch_add** | atomically adds the argument to the value stored in the atomic object and obtains the value held previously<br>(public member function) |
| **fetch_sub** | atomically subtracts the argument from the value stored in the atomic object and obtains the value held previously<br>(public member function) |
| **fetch_and** | atomically performs bitwise AND between the argument and the value of the atomic object and obtains the value held previously<br>(public member function) |
| **fetch_or** | atomically performs bitwise OR between the argument and the value of the atomic object and obtains the value held previously<br>(public member function) |
| **fetch_xor** | atomically performs bitwise XOR between the argument and the value of the atomic object and obtains the value held previously<br>(public member function) |
| **operator++**<br>**operator++**`(int)`<br>**operator--**<br>**operator--**`(int)` | increments or decrements the atomic value by one<br>(public member function) |
| **operator+=**<br>**operator-=**<br>**operator&=**<br>**operator\|=**<br>**operator^=** | adds, subtracts, or performs bitwise AND, OR, XOR with the atomic value<br>(public member function) |

### Notes

There are non-member function template equivalents for all member functions of `std::atomic`. Those non-member functions may be additionally overloaded for types that are not specializations of `std::atomic`, but are able to guarantee atomicity. The only such type in the standard library is `std::shared_ptr<U>`.

On gcc and clang, some of the functionality described here requires linking against `-latomic`.

### Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

| DR | Applied to | Behavior as published | Correct behavior |
|---|---|---|---|
| LWG 2441 (https://cplusplus.github.io/LWG/issue2441) | C++11 | | added specializations for the (optional) fixed width integer types |
| P0558R1 (https://wg21.link/P0558R1) | C++11 | | specification was substantially rewritten to resolve numerous issues<br>in particular, member typedefs value_type and difference_type are added |
| LWG 3012 (https://cplusplus.github.io/LWG/issue3012) | C++11 | `std::atomic<T>` was permitted for<br>any T that is trivially copyable but not copyable | such specializations are forbidden |

### See also

| | |
|---|---|
| **atomic_flag** (C++11) | the lock-free boolean atomic type<br>(class) |
| **std::atomic**<std::shared_ptr> (C++20) | atomic shared pointer<br>(class template specialization) |
| **std::atomic**<std::weak_ptr> (C++20) | atomic weak pointer<br>(class template specialization) |