Username: Pralay Patoria **Book:** C++ Concurrency in Action: Practical Multithreading. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

D.3. <atomic> header

The <atomic> header provides the set of basic atomic types and operations on those types and a class template for constructing an atomic version of a user-defined type that meets certain criteria.

Header contents

```
#define ATOMIC_BOOL_LOCK_FREE  see description
#define ATOMIC_CHAR_LOCK_FREE  see description
#define ATOMIC_SHORT_LOCK_FREE  see description
#define ATOMIC_INT_LOCK_FREE  see description
#define ATOMIC_LONG_LOCK_FREE  see description
#define ATOMIC_LLONG_LOCK_FREE  see description
#define ATOMIC_CHAR16_T_LOCK_FREE  see description
#define ATOMIC_CHAR32_T_LOCK_FREE  see description
#define ATOMIC_WCHAR_T_LOCK_FREE  see description
#define ATOMIC_POINTER_LOCK_FREE  see description

#define ATOMIC_VAR_INIT(value)  see description

namespace std
{
    enum memory_order;

    struct atomic_flag;
    typedef see description  atomic_bool;
    typedef see description  atomic_char;
    typedef see description  atomic_char16_t;
    typedef see description  atomic_char32_t;
    typedef see description  atomic_schar;
    typedef see description  atomic_uchar;
    typedef see description  atomic_short;
    typedef see description  atomic_ushort;
    typedef see description  atomic_int;
    typedef see description  atomic_uint;
    typedef see description  atomic_long;
    typedef see description  atomic_ulong;
    typedef see description  atomic_llong;
    typedef see description  atomic_ullong;
    typedef see description  atomic_wchar_t;

    typedef see description  atomic_int_least8_t;
    typedef see description  atomic_uint_least8_t;
    typedef see description  atomic_int_least16_t;
    typedef see description  atomic_uint_least16_t;
    typedef see description  atomic_int_least32_t;
    typedef see description  atomic_uint_least32_t;
    typedef see description  atomic_int_least64_t;
    typedef see description  atomic_uint_least64_t;
    typedef see description  atomic_int_fast8_t;
    typedef see description  atomic_uint_fast8_t;
    typedef see description  atomic_int_fast16_t;
    typedef see description  atomic_uint_fast16_t;
    typedef see description  atomic_int_fast32_t;
    typedef see description  atomic_uint_fast32_t;
    typedef see description  atomic_int_fast64_t;
    typedef see description  atomic_uint_fast64_t;
    typedef see description  atomic_int8_t;
    typedef see description  atomic_uint8_t;
    typedef see description  atomic_int16_t;
    typedef see description  atomic_uint16_t;
    typedef see description  atomic_int32_t;
    typedef see description  atomic_uint32_t;
    typedef see description  atomic_int64_t;
    typedef see description  atomic_uint64_t;
    typedef see description  atomic_intptr_t;
    typedef see description  atomic_uintptr_t;
    typedef see description  atomic_size_t;
    typedef see description  atomic_ssize_t;
    typedef see description  atomic_ptrdiff_t;
    typedef see description  atomic_intmax_t;
    typedef see description  atomic_uintmax_t;

    template<typename T>
    struct atomic;

    extern "C" void atomic_thread_fence(memory_order order);
    extern "C" void atomic_signal_fence(memory_order order);

    template<typename T>
    T kill_dependency(T);
}
```

D.3.1. std::atomic_xxx typedefs

For compatibility with the forthcoming C Standard, typedefs for the atomic integral types are provided. These are either typedefs to the corresponding `std::atomic<T>` specialization or a base class of that specialization with the same interface.

Table D.1. Atomic typedefs and their corresponding `std::atomic<>` specializations

`std::atomic_` *`itype`*	`std::atomic<>` specialization
`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_wchar_t`	`std::atomic<wchar_t>`
`std::atomic_char16_t`	`std::atomic<char16_t>`
`std::atomic_char32_t`	`std::atomic<char32_t>`

D.3.2. ATOMIC_xxx_LOCK_FREE macros

These macros specify whether the atomic types corresponding to particular built-in types are lock-free or not.

Macro declarations

```
#define ATOMIC_BOOL_LOCK_FREE  see description
#define ATOMIC_CHAR_LOCK_FREE  see description
#define ATOMIC_SHORT_LOCK_FREE  see description
#define ATOMIC_INT_LOCK_FREE see description
#define ATOMIC_LONG_LOCK_FREE  see description
#define ATOMIC_LLONG_LOCK_FREE  see description
#define ATOMIC_CHAR16_T_LOCK_FREE  see description
#define ATOMIC_CHAR32_T_LOCK_FREE  see description
#define ATOMIC_WCHAR_T_LOCK_FREE  see description
#define ATOMIC_POINTER_LOCK_FREE  see description
```

The value of `ATOMIC_xxx_LOCK_FREE` is either 0, 1, or 2. A value of 0 means that operations on both the signed and unsigned atomic types corresponding to the named type are never lock-free, a value of 1 means that the operations may be lock-free for particular instances of those types and not for others, and a value of 2 means that the operations are always lock-free. For example, if `ATOMIC_INT_LOCK_FREE` is 2, operations on instances of `std::atomic<int>` and `std::atomic<unsigned>` are always lock-free.

The macro `ATOMIC_POINTER_LOCK_FREE` describes the lock-free property of operations on the atomic pointer specializations `std::atomic<T*>`.

D.3.3. ATOMIC_VAR_INIT macro

The `ATOMIC_VAR_INIT` macro provides a means of initializing an atomic variable to a particular value.

Declaration

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

The macro expands to a token sequence that can be used to initialize one of the standard atomic types with the specified value in an expression of the following form:

```
std::atomic<type> x = ATOMIC_VAR_INIT(val);
```

The specified value must be compatible with the nonatomic type corresponding to the atomic variable, for example:

```
std::atomic<int> i = ATOMIC_VAR_INIT(42);
std::string s;
std::atomic<std::string*> p = ATOMIC_VAR_INIT(&s);
```

Such initialization is not atomic, and any access by another thread to the variable being initialized where the initialization doesn't happen-before that access is a data race and thus undefined behavior.

D.3.4. std::memory_order enumeration

The `std::memory_order` enumeration is used to specify the ordering constraints of atomic operations.

Declaration

```
typedef enum memory_order
{
    memory_order_relaxed,memory_order_consume,
    memory_order_acquire,memory_order_release,
    memory_order_acq_rel,memory_order_seq_cst
} memory_order;
```

Operations tagged with the various memory order values behave as follows (see chapter 5 for detailed descriptions of the ordering constraints).

Std::Memory_Order_Relaxed

The operation doesn't provide any additional ordering constraints.

Std::Memory_Order_Release

The operation is a release operation on the specified memory location. This therefore synchronizes-with an acquire operation on the same memory location that reads the stored value.

Std::Memory_Order_Acquire

The operation is an acquire operation on the specified memory location. If the stored value was written by a release operation, that store synchronizes-with this operation.

Std::Memory_Order_Acq_Rel

The operation must be a read-modify-write operation, and it behaves as both `std::memory_order_acquire` and `std::memory_order_release` on the specified location.

Std::Memory_Order_Seq_Cst

The operation forms part of the single global total order of sequentially consistent operations. In addition, if it's a store, it behaves like a `std::memory_order_release` operation; if it's a load, it behaves like a `std::memory_order_acquire` operation; and if it's a read-modify-write operation, it behaves as both `std::memory_order_acquire` and `std::memory_order_release`. *This is the default for all operations.*

Std::Memory_Order_Consume

The operation is a consume operation on the specified memory location.

D.3.5. std::atomic_thread_fence function

The `std::atomic_thread_fence()` function inserts a "memory barrier" or "fence" in the code to force memory-ordering constraints between operations.

Declaration

```
extern "C" void atomic_thread_fence(std::memory_order order);
```

Effects

Inserts a fence with the required memory-ordering constraints.

A fence with an `order` of `std::memory_order_release`, `std::memory_order_acq_rel`, or `std::memory_order_seq_cst` synchronizes-with an acquire operation on the some memory location if that acquire operation reads a value stored by an atomic operation following the fence on the same thread as the fence.

A release operation synchronizes-with a fence with an `order` of `std::memory_order_acquire`, `std::memory_order_acq_rel`, or `std::memory_order_seq_cst` if that release operation stores a value that's read by an atomic operation prior to the fence on the same thread as the fence.

Throws

Nothing.

D.3.6. std::atomic_signal_fence function

The `std::atomic_signal_fence()` function inserts a memory barrier or fence in the code to force memory ordering constraints between operations on a thread and operations in a signal handler on that thread.

Declaration

```
extern "C" void atomic_signal_fence(std::memory_order order);
```

Effects

Inserts a fence with the required memory-ordering constraints. This is equivalent to `std::atomic_thread_fence(order)` except that the constraints apply only between a thread and a signal handler on the same thread.

Throws

Nothing.

D.3.7. std::atomic_flag class

The `std::atomic_flag` class provides a simple bare-bones atomic flag. It's the only data type that's *guaranteed* to be lock-free by the C++11 Standard (although many atomic types will be lock-free in most implementations).

An instance of std::atomic_flag is either *set* or *clear*.

Class definition

```
struct atomic_flag
{
    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 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;
};

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 unspecified
```

Std::Atomic_Flag Default Constructor

It's unspecified whether a default-constructed instance of std::atomic_flag is *clear* or *set*. For objects of static storage duration, initialization shall be static initialization.

Declaration

```
std::atomic_flag() noexcept = default;
```

Effects

Constructs a new std::atomic_flag object in an unspecified state.

Throws

Nothing.

Std::Atomic_Flag Initialization with Atomic_Flag_Init

An instance of std::atomic_flag may be initialized using the ATOMIC_FLAG_INIT macro, in which case it's initialized into the *clear* state. For objects of static storage duration, initialization shall be static initialization.

Declaration

```
#define ATOMIC_FLAG_INIT unspecified
```

Usage

```
std::atomic_flag flag=ATOMIC_FLAG_INIT;
```

Effects

Constructs a new std::atomic_flag object in the *clear* state.

Throws

Nothing.

Std::Atomic_Flag::Test_and_Set Member Function

Atomically sets the flag and checks whether or not it was set.

Declaration

```
bool test_and_set(memory_order order = memory_order_seq_cst) volatile
     noexcept;
bool test_and_set(memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically sets the flag.

Returns

true if the flag was set at the point of the call, false if the flag was clear.

Throws

Nothing.

> **Note**
>
> This is an atomic read-modify-write operation for the memory location comprising *this.

Std::Atomic_Flag_Test_and_Set Nonmember Function

Atomically sets the flag and checks whether or not it was set.

Declaration

```
bool atomic_flag_test_and_set(volatile atomic_flag* flag) noexcept;
bool atomic_flag_test_and_set(atomic_flag* flag) noexcept;
```

Effects

```
return flag->test_and_set();
```

Std::Atomic_Flag_Test_And_Set_Explicit Nonmember Function

Atomically sets the flag and checks whether or not it was set.

Declaration

```
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag* flag, memory_order order) noexcept;
bool atomic_flag_test_and_set_explicit(
    atomic_flag* flag, memory_order order) noexcept;
```

Effects

```
return flag->test_and_set(order);
```

Std::Atomic_Flag::Clear Member Function

Atomically clears the flag.

Declaration

```
void clear(memory_order order = memory_order_seq_cst) volatile noexcept;
void clear(memory_order order = memory_order_seq_cst) noexcept;
```

Preconditions

The supplied order must be one of `std::memory_order_relaxed`, `std::memory_order_release`, or `std::memory_order_seq_cst`.

Effects

Atomically clears the flag.

Throws

Nothing.

Note

This is an atomic store operation for the memory location comprising `*this`.

Std::Atomic_Flag_Clear Nonmember Function

Atomically clears the flag.

Declaration

```
void atomic_flag_clear(volatile atomic_flag* flag) noexcept;
void atomic_flag_clear(atomic_flag* flag) noexcept;
```

Effects

```
flag->clear();
```

Std::Atomic_Flag_Clear_Explicit Nonmember Function

Atomically clears the flag.

Declaration

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

Effects

```
return flag->clear(order);
```

D.3.8. std::atomic class template

The `std::atomic` class provides a wrapper with atomic operations for any type that satisfies the following requirements.

The template parameter `BaseType` must

- Have a trivial default constructor
- Have a trivial copy-assignment operator
- Have a trivial destructor
- Be bitwise-equality comparable

This basically means that `std::atomic<`*some-built-in-type*`>` is fine, as is `std::atomic<`*some-simple-struct*`>`, but things like `std::atomic<std::string>` are not.

In addition to the primary template, there are specializations for the built-in integral types and pointers to provide additional operations such as `x++`.

Instances of `std::atomic` are not `CopyConstructible` or `CopyAssignable`, because these operations can't be performed as a single atomic operation.

Class definition

```
template<typename BaseType>
struct atomic
{
    atomic() noexcept = default;
    constexpr atomic(BaseType) noexcept;
    BaseType operator=(BaseType) volatile noexcept;
    BaseType operator=(BaseType) noexcept;

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;
    void store(BaseType,memory_order = memory_order_seq_cst)
        volatile noexcept;
    void store(BaseType,memory_order = memory_order_seq_cst) noexcept;
    BaseType load(memory_order = memory_order_seq_cst)
        const volatile noexcept;
    BaseType load(memory_order = memory_order_seq_cst) const noexcept;
    BaseType exchange(BaseType,memory_order = memory_order_seq_cst)
        volatile noexcept;
    BaseType exchange(BaseType,memory_order = memory_order_seq_cst)
        noexcept;

    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) volatile noexcept;
    bool compare_exchange_strong(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst)
        volatile noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) volatile noexcept;
    bool compare_exchange_weak(
        BaseType & old_value, BaseType new_value,
        memory_order success_order,
        memory_order failure_order) noexcept;
    operator BaseType () const volatile noexcept;
    operator BaseType () const noexcept;
};

template<typename BaseType>
bool atomic_is_lock_free(volatile const atomic<BaseType>*) noexcept;
template<typename BaseType>
bool atomic_is_lock_free(const atomic<BaseType>*) noexcept;
template<typename BaseType>
```

```
void atomic_init(volatile atomic<BaseType>*, void*) noexcept;
template<typename BaseType>
void atomic_init(atomic<BaseType>*, void*) noexcept;
template<typename BaseType>
BaseType atomic_exchange(volatile atomic<BaseType>*, memory_order)
    noexcept;
template<typename BaseType>
BaseType atomic_exchange(atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
    volatile atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_exchange_explicit(
    atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
void atomic_store(volatile atomic<BaseType>*, BaseType) noexcept;
template<typename BaseType>
void atomic_store(atomic<BaseType>*, BaseType) noexcept;
template<typename BaseType>
void atomic_store_explicit(
    volatile atomic<BaseType>*, BaseType, memory_order) noexcept;
template<typename BaseType>
void atomic_store_explicit(
    atomic<BaseType>*, BaseType, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>*) noexcept;
template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>*) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    volatile const atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    const atomic<BaseType>*, memory_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    volatile atomic<BaseType>*,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    atomic<BaseType>*,BaseType * old_value,BaseType new_value) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    atomic<BaseType>*,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

Note

Although the nonmember functions are specified as templates, they may be provided as an overloaded set of functions, and explicit specification of the template arguments shouldn't be used.

Std::Atomic Default Constructor

Constructs an instance of `std::atomic` with a default-initialized value.

Declaration

```
atomic() noexcept;
```

Effects

Constructs a new `std::atomic` object with a default-initialized value. For objects with static storage duration this is static initialization.

Note

Instances of `std::atomic` with nonstatic storage duration initialized with the default constructor can't be relied on to have a predictable value.

Throws

Nothing.

Std::Atomic_Init Nonmember Function

Nonatomically stores the supplied value in an instance of `std::atomic<BaseType>`.

Declaration

```
template<typename BaseType>
void atomic_init(atomic<BaseType> volatile* p, BaseType v) noexcept;
template<typename BaseType>
```

```
void atomic_init(atomic<BaseType>* p, BaseType v) noexcept;
```

Effects

Nonatomically stores the value of v in *p. Invoking atomic_init() on an instance of atomic<BaseType> that hasn't been default constructed, or that has had any operations performed on it since construction, is undefined behavior.

Note

Because this store is nonatomic, any concurrent access to the object pointed to by p from another thread (even with atomic operations) constitutes a data race.

Throws

Nothing.

Std::Atomic Conversion Constructor

Construct an instance of std::atomic with the supplied BaseType value.

Declaration

```
constexpr atomic(BaseType b) noexcept;
```

Effects

Constructs a new std::atomic object with a value of b. For objects with static storage duration this is static initialization.

Throws

Nothing.

Std::Atomic Conversion Assignment Operator

Stores a new value in *this.

Declaration

```
BaseType operator=(BaseType b) volatile noexcept;
BaseType operator=(BaseType b) noexcept;
```

Effects

```
return this->store(b);
```

Std::Atomic::Is_Lock_Free Member Function

Determines if operations on *this are lock-free.

Declaration

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

Returns

`true` if operations on `*this` are lock-free, `false` otherwise.

Throws

Nothing.

Std::Atomic_Is_Lock_Free Nonmember Function

Determine if operations on `*this` are lock-free.

Declaration

```
template<typename BaseType>
bool atomic_is_lock_free(volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
bool atomic_is_lock_free(const atomic<BaseType>* p) noexcept;
```

Effects

```
return p->is_lock_free();
```

Std::Atomic::Load Member Function

Atomically loads the current value of the `std::atomic` instance.

Declaration

```
BaseType load(memory_order order = memory_order_seq_cst)
    const volatile noexcept;
BaseType load(memory_order order = memory_order_seq_cst) const noexcept;
```

Preconditions

The supplied `order` must be one of `std::memory_order_relaxed, std::memory_order_acquire, std::memory_order_consume`, or `std::memory_order_seq_cst`.

Effects

Atomically loads the value stored in `*this`.

Returns

The value stored in `*this` at the point of the call.

Throws

Nothing.

> **Note**
>
> This is an atomic load operation for the memory location comprising `*this`.

Std::Atomic_Load Nonmember Function

Atomically loads the current value of the std::atomic instance.

Declaration

```
template<typename BaseType>
BaseType atomic_load(volatile const atomic<BaseType>* p) noexcept;
template<typename BaseType>
BaseType atomic_load(const atomic<BaseType>* p) noexcept;
```

Effects

```
return p->load();
```

Std::Atomic_Load_Explicit Nonmember Function

Atomically loads the current value of the std::atomic instance.

Declaration

```
template<typename BaseType>
BaseType atomic_load_explicit(
    volatile const atomic<BaseType>* p, memory_order order) noexcept;
template<typename BaseType>
BaseType atomic_load_explicit(
    const atomic<BaseType>* p, memory_order order) noexcept;
```

Effects

```
return p->load(order);
```

Std::Atomic::Operator Basetype Conversion Operator

Loads the value stored in *this.

Declaration

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

Effects

```
return this->load();
```

Std::Atomic::Store Member Function

Atomically store a new value in an atomic<BaseType> instance.

Declaration

```
void store(BaseType new_value,memory_order order = memory_order_seq_cst)
    volatile noexcept;
```

```
void store(BaseType new_value,memory_order order = memory_order_seq_cst)
    noexcept;
```

Preconditions

The supplied order must be one of `std::memory_order_relaxed`, `std::memory_order_release`, or `std::memory_order_seq_cst`.

Effects

Atomically stores `new_value` in `*this`.

Throws

Nothing.

> **Note**
>
> This is an atomic store operation for the memory location comprising `*this`.

Std::Atomic_Store Nonmember Function

Atomically stores a new value in an `atomic<BaseType>` instance.

Declaration

```
template<typename BaseType>
void atomic_store(volatile atomic<BaseType>* p, BaseType new_value)
    noexcept;
template<typename BaseType>
void atomic_store(atomic<BaseType>* p, BaseType new_value) noexcept;
```

Effects

```
p->store(new_value);
```

Std::Atomic_Store_Explicit Nonmember Function

Atomically stores a new value in an `atomic<BaseType>` instance.

Declaration

```
template<typename BaseType>
void atomic_store_explicit(
    volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
    noexcept;
template<typename BaseType>
void atomic_store_explicit(
    atomic<BaseType>* p, BaseType new_value, memory_order order) noexcept;
```

Effects

```
p->store(new_value,order);
```

Std::Atomic::Exchange Member Function

Atomically stores a new value and reads the old one.

Declaration

```
BaseType exchange(
    BaseType new_value,
    memory_order order = memory_order_seq_cst)
    volatile noexcept;
```

Effects

Atomically stores new_value in *this and retrieves the existing value of *this.

Returns

The value of *this immediately prior to the store.

Throws

Nothing.

Note

This is an atomic read-modify-write operation for the memory location comprising *this.

Std::Atomic_Exchange Nonmember Function

Atomically stores a new value in an atomic<BaseType> instance and reads the prior value.

Declaration

```
template<typename BaseType>
BaseType atomic_exchange(volatile atomic<BaseType>* p, BaseType new_value)
    noexcept;
template<typename BaseType>
BaseType atomic_exchange(atomic<BaseType>* p, BaseType new_value) noexcept;
```

Effects

```
return p->exchange(new_value);
```

Std::Atomic_Exchange_Explicit Nonmember Function

Atomically stores a new value in an atomic<BaseType> instance and reads the prior value.

Declaration

```
template<typename BaseType>
BaseType atomic_exchange_explicit(
    volatile atomic<BaseType>* p, BaseType new_value, memory_order order)
    noexcept;
```

```
template<typename BaseType>
BaseType atomic_exchange_explicit(
    atomic<BaseType>* p, BaseType new_value, memory_order order) noexcept;
```

Effects

```
return p->exchange(new_value,order);
```

Std::Atomic::Compare_Exchange_Strong Member Function

Atomically compares the value to an expected value and stores a new value if the values are equal. If the values aren't equal, updates the expected value with the value read.

Declaration

```
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) volatile noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order)
    volatile noexcept;
bool compare_exchange_strong(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order) noexcept;
```

Preconditions

failure_order shall not be std::memory_order_release or std::memory_order_acq_rel.

Effects

Atomically compares expected to the value stored in *this using bitwise comparison and stores new_value in *this if equal; otherwise updates expected to the value read.

Returns

true if the existing value of *this was equal to expected, false otherwise.

Throws

Nothing.

Note

The three-parameter overload is equivalent to the four-parameter overload with success_order==order and failure_order==order, except that if order is std::memory_order_acq_rel, then failure_order is std::memory_order_acquire, and if order is std::memory_order_release, then failure_order is std::memory_order_relaxed.

Note

This is an atomic read-modify-write operation for the memory location comprising `*this` if the result is `true`, with memory ordering `success_order`; otherwise, it's an atomic load operation for the memory location comprising `*this` with memory ordering `failure_order`.

Std::Atomic_Compare_Exchange_Strong Nonmember Function

Atomically compares the value to an expected value and stores a new value if the values are equal. If the values aren't equal, updates the expected value with the value read.

Declaration

```
template<typename BaseType>
bool atomic_compare_exchange_strong(
    volatile atomic<BaseType>* p,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong(
    atomic<BaseType>* p,BaseType * old_value,BaseType new_value) noexcept;
```

Effects

```
return p->compare_exchange_strong(*old_value,new_value);
```

Std::Atomic_Compare_Exchange_Strong_Explicit Nonmember Function

Atomically compares the value to an expected value and stores a new value if the values are equal. If the values aren't equal, updates the expected value with the value read.

Declaration

```
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_strong_explicit(
    atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

Effects

```
return p->compare_exchange_strong(
    *old_value,new_value,success_order,failure_order) noexcept;
```

Std::Atomic::Compare_Exchange_Weak Member Function

Atomically compares the value to an expected value and stores a new value if the values are equal and the update can be done atomically. If the values aren't equal or the update can't be done atomically, updates the expected value with the value read.

Declaration

```
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) volatile noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order order = std::memory_order_seq_cst) noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order)
    volatile noexcept;
bool compare_exchange_weak(
    BaseType& expected,BaseType new_value,
    memory_order success_order,memory_order failure_order) noexcept;
```

Preconditions

`failure_order` shall not be `std::memory_order_release` or `std::memory_order_acq_rel`.

Effects

Atomically compares `expected` to the value stored in `*this` using bitwise comparison and stores `new_value` in `*this` if equal. If the values aren't equal or the update can't be done atomically, updates `expected` to the value read.

Returns

`true` if the existing value of `*this` was equal to `expected` and `new_value` was successfully stored in `*this, false` otherwise.

Throws

Nothing.

> **Note**
>
> The three-parameter overload is equivalent to the four-parameter overload with `success_order==order` and `failure_order==order`, except that if `order` is `std::memory_order_acq_rel`, then `failure_order` is `std::memory_order_acquire`, and if `order` is `std::memory_order_release`, then `failure_order` is `std::memory_order_relaxed`.

> **Note**
>
> This is an atomic read-modify-write operation for the memory location comprising `*this` if the result is `true`, with memory ordering `success_order`; otherwise, it's an atomic load operation for the memory location comprising `*this` with memory ordering `failure_order`.

Std::Atomic_Compare_Exchange_Weak Nonmember Function

Atomically compares the value to an expected value and stores a new value if the values are equal and the update can be done atomically. If the values aren't equal or the update can't be done atomically, updates the expected value with the value read.

Declaration

```
template<typename BaseType>
bool atomic_compare_exchange_weak(
    volatile atomic<BaseType>* p,BaseType * old_value,BaseType new_value)
    noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak(
    atomic<BaseType>* p,BaseType * old_value,BaseType new_value) noexcept;
```

Effects

```
return p->compare_exchange_weak(*old_value,new_value);
```

Std::Atomic_Compare_Exchange_Weak_Explicit Nonmember Function

Atomically compares the value to an expected value and stores a new value if the values are equal and the update can be done atomically. If the values aren't equal or the update can't be done atomically, updates the expected value with the value read.

Declaration

```
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
template<typename BaseType>
bool atomic_compare_exchange_weak_explicit(
    atomic<BaseType>* p,BaseType * old_value,
    BaseType new_value, memory_order success_order,
    memory_order failure_order) noexcept;
```

Effects

```
return p->compare_exchange_weak(
    *old_value,new_value,success_order,failure_order);
```

D.3.9. Specializations of the std::atomic template

Specializations of the `std::atomic` class template are provided for the integral types and pointer types. For the integral types, these specializations provide atomic addition, subtraction, and bitwise operations in addition to the operations provided by the primary template. For pointer types, the specializations provide atomic pointer arithmetic in addition to the operations provided by the primary template.

Specializations are provided for the following integral types:

```
std::atomic<bool>
std::atomic<char>
```

```
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

and for `std::atomic<T*>` for all types `T`.

D.3.10. std::atomic<integral-type> specializations

The `std::atomic<`*integral-type*`>` specializations of the `std::atomic` class template provide an atomic integral data type for each fundamental integer type, with a comprehensive set of operations.

The following description applies to these specializations of the `std::atomic<>` class template:

```
std::atomic<char>
std::atomic<signed char>
std::atomic<unsigned char>
std::atomic<short>
std::atomic<unsigned short>
std::atomic<int>
std::atomic<unsigned>
std::atomic<long>
std::atomic<unsigned long>
std::atomic<long long>
std::atomic<unsigned long long>
std::atomic<wchar_t>
std::atomic<char16_t>
std::atomic<char32_t>
```

Instances of these specializations are not `CopyConstructible` or `CopyAssignable`, because these operations can’t be performed as a single atomic operation.

Class definition

```
template<>
struct atomic<integral-type>
{
    atomic() noexcept = default;
    constexpr atomic(integral-type) noexcept;
    bool operator=(integral-type) volatile noexcept;

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;
    bool is_lock_free() const volatile noexcept;
    bool is_lock_free() const noexcept;

    void store(integral-type,memory_order = memory_order_seq_cst)
         volatile noexcept;
    void store(integral-type, memory_order = memory_order_seq_cst) noexcept;
    integral-type load(memory_order = memory_order_seq_cst)
         const volatile noexcept;
    integral-type  load(memory_order = memory_order_seq_cst) const noexcept;
    integral-type  exchange(
        integral-type, memory_order = memory_order_seq_cst)
         volatile noexcept;
    integral-type  exchange(
        integral-type,memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_strong(
        integral-type & old_value,integral-type  new_value,
         memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        integral-type & old_value,integral-type  new_value,
          memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        integral-type & old_value,integral-type  new_value,
         memory_order success_order,memory_order failure_order)
         volatile noexcept;
    bool compare_exchange_strong(
        integral-type & old_value,integral-type  new_value,
         memory_order success_order,memory_order failure_order) noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type  new_value,
          memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type  new_value,
          memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type  new_value,
         memory_order success_order,memory_order failure_order)
         volatile noexcept;
    bool compare_exchange_weak(
        integral-type & old_value,integral-type  new_value,
         memory_order success_order,memory_order failure_order) noexcept;
    operator integral-type () const volatile noexcept;
    operator integral-type() const noexcept;

    integral-type  fetch_add(
        integral-type, memory_order = memory_order_seq_cst)
         volatile noexcept;
    integral-type  fetch_add(
        integral-type, memory_order = memory_order_seq_cst) noexcept;
    integral-type  fetch_sub(
        integral-type, memory_order = memory_order_seq_cst)
         volatile noexcept;
    integral-type  fetch_sub(
        integral-type, memory_order = memory_order_seq_cst) noexcept;
    integral-type  fetch_and(
        integral-type, memory_order = memory_order_seq_cst)
         volatile noexcept;
    integral-type  fetch_and(
        integral-type, memory_order = memory_order_seq_cst) noexcept;
    integral-type  fetch_or(
        integral-type, memory_order = memory_order_seq_cst)
         volatile noexcept;
    integral-type  fetch_or(
        integral-type, memory_order = memory_order_seq_cst) noexcept;
    integral-type  fetch_xor(
        integral-type, memory_order = memory_order_seq_cst)
         volatile noexcept;
    integral-type  fetch_xor(
        integral-type, memory_order = memory_order_seq_cst) noexcept;

    integral-type  operator++() volatile noexcept;
    integral-type  operator++() noexcept;
    integral-type  operator++(int) volatile noexcept;
    integral-type  operator++(int) noexcept;
    integral-type  operator--() volatile noexcept;
    integral-type  operator--() noexcept;
    integral-type  operator--(int) volatile noexcept;
    integral-type  operator--(int) noexcept;

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

bool atomic_is_lock_free(volatile const atomic<integral-type >*) noexcept;
bool atomic_is_lock_free(const atomic<integral-type >*) noexcept;
void atomic_init(volatile atomic<integral-type>*,integral-type ) noexcept;
void atomic_init(atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_exchange(
    volatile atomic<integral-type>*,integral-type) noexcept;
integral-type  atomic_exchange(
    atomic<integral-type>*,integral-type) noexcept;
integral-type  atomic_exchange_explicit(
    volatile atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_exchange_explicit(
    atomic<integral-type>*,integral-type , memory_order) noexcept;
void atomic_store(volatile atomic<integral-type>*,integral-type) noexcept;
void atomic_store(atomic<integral-type>*,integral-type) noexcept;
void atomic_store_explicit(
    volatile atomic<integral-type>*,integral-type , memory_order) noexcept;
void atomic_store_explicit(
    atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type atomic_load(volatile const atomic<integral-type >*) noexcept;
integral-type atomic_load(const atomic<integral-type >*) noexcept;
integral-type  atomic_load_explicit(
    volatile const atomic<integral-type >*,memory_order) noexcept;
integral-type  atomic_load_explicit(
    const atomic<integral-type>*,memory_order) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value) noexcept;
bool atomic_compare_exchange_strong(
    atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value,
     memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value) noexcept;
bool atomic_compare_exchange_weak(
    atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
    volatile atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value,
     memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
    atomic<integral-type>*,
    integral-type * old_value,integral-type  new_value,
     memory_order success_order,memory_order failure_order) noexcept;

integral-type  atomic_fetch_add(
    volatile atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_add(
    atomic<integral-type>*,integral-type) noexcept;
integral-type  atomic_fetch_add_explicit(
    volatile atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_add_explicit(
    atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_sub(
    volatile atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_sub(
    atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_sub_explicit(
    volatile atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_sub_explicit(
    atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_and(
    volatile atomic<integral-type>*,integral-type ) noexcept;
```

```
integral-type  atomic_fetch_and(
    atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_and_explicit(
    volatile atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_and_explicit(
    atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_or(
    volatile atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_or(
    atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_or_explicit(
    volatile atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_or_explicit(
    atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_xor(
    volatile atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_xor(
    atomic<integral-type>*,integral-type ) noexcept;
integral-type  atomic_fetch_xor_explicit(
    volatile atomic<integral-type>*,integral-type , memory_order) noexcept;
integral-type  atomic_fetch_xor_explicit(
    atomic<integral-type>*,integral-type , memory_order) noexcept;
```

Those operations that are also provided by the primary template (see D.3.8) have the same semantics.

Std::Atomic<Integral-Type>::Fetch_Add Member Function

Atomically loads a value and replaces it with the sum of that value and the supplied value i.

Declaration

```
integral-type  fetch_add(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type  fetch_add(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically retrieves the existing value of *this and stores *old-value* + i in *this.

Returns

The value of *this immediately prior to the store.

Throws

Nothing.

Note

This is an atomic read-modify-write operation for the memory location comprising *this.

Std::Atomic_Fetch_Add Nonmember Function

Atomically reads the value from an atomic<*integral-type*> instance and replaces it with that value plus the supplied value i.

Declaration

```
integral-type  atomic_fetch_add(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type  atomic_fetch_add(
    atomic<integral-type>* p, integral-type i) noexcept;
```

Effects

```
return p->fetch_add(i);
```

Std::Atomic_Fetch_Add_Explicit Nonmember Function

Atomically reads the value from an atomic<*integral-type*> instance and replaces it with that value plus the supplied value i.

Declaration

```
integral-type  atomic_fetch_add_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type  atomic_fetch_add_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

Effects

```
return p->fetch_add(i,order);
```

Std::Atomic<Integral-Type>::Fetch_Sub Member Function

Atomically loads a value and replaces it with the sum of that value and the supplied value i.

Declaration

```
integral-type  fetch_sub(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type  fetch_sub(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically retrieves the existing value of *this and stores *old-value* - i in *this.

Returns

The value of *this immediately prior to the store.

Throws

Nothing.

Note

This is an atomic read-modify-write operation for the memory location comprising `*this`.

Std::Atomic_Fetch_Sub Nonmember Function

Atomically reads the value from an `atomic<`*integral-type*`>` instance and replaces it with that value minus the supplied value `i`.

Declaration

```
integral-type  atomic_fetch_sub(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type  atomic_fetch_sub(
    atomic<integral-type>* p, integral-type i) noexcept;
```

Effects

```
return p->fetch_sub(i);
```

Std::Atomic_Fetch_Sub_Explicit Nonmember Function

Atomically reads the value from an `atomic<`*integral-type*`>` instance and replaces it with that value minus the supplied value `i`.

Declaration

```
integral-type  atomic_fetch_sub_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type  atomic_fetch_sub_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

Effects

```
return p->fetch_sub(i,order);
```

Std::Atomic<Integral-Type>::Fetch_Sub Member Function

Atomically loads a value and replaces it with the bitwise-and of that value and the supplied value `i`.

Declaration

```
integral-type  fetch_and(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type  fetch_and(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically retrieves the existing value of `*this` and stores *old-value* `& i` in `*this`.

Returns

The value of `*this` immediately prior to the store.

Throws

Nothing.

Note

This is an atomic read-modify-write operation for the memory location comprising `*this`.

Std::Atomic_Fetch_And Nonmember Function

Atomically reads the value from an `atomic<`*integral-type*`>` instance and replaces it with the bitwise-and of that value and the supplied value `i`.

Declaration

```
integral-type  atomic_fetch_and(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type  atomic_fetch_and(
    atomic<integral-type>* p, integral-type i) noexcept;
```

Effects

```
return p->fetch_and(i);
```

Std::Atomic_Fetch_And_Explicit Nonmember Function

Atomically reads the value from an `atomic<`*integral-type*`>` instance and replaces it with the bitwise-and of that value and the supplied value `i`.

Declaration

```
integral-type  atomic_fetch_and_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type  atomic_fetch_and_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

Effects

```
return p->fetch_and(i,order);
```

Std::Atomic<Integral-Type>::Fetch_Or Member Function

Atomically loads a value and replaces it with the bitwise-or of that value and the supplied value i.

Declaration

```
integral-type  fetch_or(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type  fetch_or(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically retrieves the existing value of *this and stores *old-value* | i in *this.

Returns

The value of *this immediately prior to the store.

Throws

Nothing.

Note

This is an atomic read-modify-write operation for the memory location comprising *this.

Std::Atomic_Fetch_Or Nonmember Function

Atomically reads the value from an atomic<*integral-type*> instance and replaces it with the bitwise-or of that value and the supplied value i.

Declaration

```
integral-type  atomic_fetch_or(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type  atomic_fetch_or(
    atomic<integral-type>* p, integral-type i) noexcept;
```

Effects

```
return p->fetch_or(i);
```

Std::Atomic_Fetch_Or_Explicit Nonmember Function

Atomically reads the value from an atomic<*integral-type*> instance and replaces it with the bitwise-or of that value and the supplied value i.

Declaration

```
integral-type  atomic_fetch_or_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type  atomic_fetch_or_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

Effects

```
return p->fetch_or(i,order);
```

Std::Atomic<Integral-Type>::Fetch_or Member Function

Atomically loads a value and replaces it with the bitwise-xor of that value and the supplied value `i`.

Declaration

```
integral-type  fetch_xor(
    integral-type i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
integral-type  fetch_xor(
    integral-type i,memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically retrieves the existing value of `*this` and stores *old-value* `^ i` in `*this`.

Returns

The value of `*this` immediately prior to the store.

Throws

Nothing.

Note

This is an atomic read-modify-write operation for the memory location comprising `*this`.

Std::Atomic_Fetch_Xor Nonmember Function

Atomically reads the value from an `atomic<`*integral-type*`>` instance and replaces it with the bitwise-xor of that value and the supplied value `i`.

Declaration

```
integral-type  atomic_fetch_xor(
    volatile atomic<integral-type>* p, integral-type i) noexcept;
integral-type  atomic_fetch_xor(
    atomic<integral-type>* p, integral-type i) noexcept;
```

Effects

```
return p->fetch_xor(i);
```

Std::Atomic_Fetch_Xor_Explicit Nonmember Function

Atomically reads the value from an atomic<*integral-type*> instance and replaces it with the bitwise-xor of that value and the supplied value i.

Declaration

```
integral-type  atomic_fetch_xor_explicit(
    volatile atomic<integral-type>* p, integral-type i,
    memory_order order) noexcept;
integral-type  atomic_fetch_xor_explicit(
    atomic<integral-type>* p, integral-type i, memory_order order)
    noexcept;
```

Effects

```
return p->fetch_xor(i,order);
```

Std::Atomic<Integral-Type>::Operator++ Preincrement Operator

Atomically increments the value stored in *this and returns the new value.

Declaration

```
integral-type operator++() volatile noexcept;
integral-type operator++() noexcept;
```

Effects

```
return this->fetch_add(1) + 1;
```

Std::Atomic<Integral-Type>::Operator++ Postincrement Operator

Atomically increments the value stored in *this and returns the old value.

Declaration

```
integral-type operator++(int) volatile noexcept;
integral-type operator++(int) noexcept;
```

Effects

```
return this->fetch_add(1);
```

Std::Atomic<Integral-Type>::Operator-- Predecrement Operator

Atomically decrements the value stored in *this and returns the new value.

Declaration

```
integral-type operator--() volatile noexcept;
integral-type operator--() noexcept;
```

Effects

```
return this->fetch_sub(1) - 1;
```

Std::Atomic<Integral-Type>::Operator-- Postdecrement Operator

Atomically decrements the value stored in `*this` and returns the old value.

Declaration

```
integral-type operator--(int) volatile noexcept;
integral-type operator--(int) noexcept;
```

Effects

```
return this->fetch_sub(1);
```

Std::Atomic<Integral-Type>::Operator+= Compound Assignment Operator

Atomically adds the supplied value to the value stored in `*this` and returns the new value.

Declaration

```
integral-type operator+=(integral-type i) volatile noexcept;
integral-type operator+=(integral-type i) noexcept;
```

Effects

```
return this->fetch_add(i) + i;
```

Std::Atomic<Integral-Type>::Operator-= Compound Assignment Operator

Atomically subtracts the supplied value from the value stored in `*this` and returns the new value.

Declaration

```
integral-type operator-=(integral-type i) volatile noexcept;
integral-type operator-=(integral-type i) noexcept;
```

Effects

```
return this->fetch_sub(i,std::memory_order_seq_cst) - i;
```

Std::Atomic<Integral-Type>::Operator&= Compound Assignment Operator

Atomically replaces the value stored in `*this` with the bitwise-and of the supplied value and the value stored in `*this` and returns the new value.

Declaration

```
integral-type operator&=(integral-type i) volatile noexcept;
integral-type operator&=(integral-type i) noexcept;
```

Effects

```
return this->fetch_and(i) & i;
```

Std::Atomic<Integral-Type>::Operator|= Compound Assignment Operator

Atomically replaces the value stored in *this with the bitwise-or of the supplied value and the value stored in *this and returns the new value.

Declaration

```
integral-type operator|=(integral-type i) volatile noexcept;
integral-type operator|=(integral-type i) noexcept;
```

Effects

```
return this->fetch_or(i,std::memory_order_seq_cst) | i;
```

Std::Atomic<Integral-Type>::Operator-= Compound Assignment Operator

Atomically replaces the value stored in *this with the bitwise-or of the supplied value and the value stored in *this and returns the new value.

Declaration

```
integral-type operator^=(integral-type i) volatile noexcept;
integral-type operator^=(integral-type i) noexcept;
```

Effects

```
return this->fetch_xor(i,std::memory_order_seq_cst) ^ i;
```

Std::Atomic<T> Partial Specialization*

The std::atomic<T*> partial specialization of the std::atomic class template provides an atomic data type for each pointer type, with a comprehensive set of operations.

Instances of these std::atomic<T*> are not CopyConstructible or CopyAssignable, because these operations can't be performed as a single atomic operation.

Class definition

```
template<typename T>
struct atomic<T*>
{
    atomic() noexcept = default;
    constexpr atomic(T*) noexcept;
    bool operator=(T*) volatile;
    bool operator=(T*);

    atomic(const atomic&) = delete;
    atomic& operator=(const atomic&) = delete;
    atomic& operator=(const atomic&) volatile = delete;

    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;
    T* exchange(T*,memory_order = memory_order_seq_cst) volatile noexcept;
    T* exchange(T*,memory_order = memory_order_seq_cst) noexcept;

    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order)
        volatile noexcept;
    bool compare_exchange_strong(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order) noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) volatile noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order order = memory_order_seq_cst) noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order)
        volatile noexcept;
    bool compare_exchange_weak(
        T* & old_value, T* new_value,
        memory_order success_order,memory_order failure_order) noexcept;

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

    T* operator++() volatile noexcept;
    T* operator++() noexcept;
    T* operator++(int) volatile noexcept;
    T* operator++(int) noexcept;
    T* operator--() volatile noexcept;
    T* operator--() noexcept;
    T* operator--(int) volatile noexcept;
    T* operator--(int) noexcept;

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

bool atomic_is_lock_free(volatile const atomic<T*>*) noexcept;
bool atomic_is_lock_free(const atomic<T*>*) noexcept;
void atomic_init(volatile atomic<T*>*, T*) noexcept;
void atomic_init(atomic<T*>*, T*) noexcept;
T* atomic_exchange(volatile atomic<T*>*, T*) noexcept;
T* atomic_exchange(atomic<T*>*, T*) noexcept;
T* atomic_exchange_explicit(volatile atomic<T*>*, T*, memory_order)
    noexcept;
T* atomic_exchange_explicit(atomic<T*>*, T*, memory_order) noexcept;
void atomic_store(volatile atomic<T*>*, T*) noexcept;
void atomic_store(atomic<T*>*, T*) noexcept;
void atomic_store_explicit(volatile atomic<T*>*, T*, memory_order)
    noexcept;
void atomic_store_explicit(atomic<T*>*, T*, memory_order) noexcept;
T* atomic_load(volatile const atomic<T*>*) noexcept;
T* atomic_load(const atomic<T*>*) noexcept;
T* atomic_load_explicit(volatile const atomic<T*>*, memory_order) noexcept;
T* atomic_load_explicit(const atomic<T*>*, memory_order) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_strong(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<T*>*,T* * old_value,T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_strong_explicit(
    atomic<T*>*,T* * old_value,T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak(
    volatile atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_weak(
    atomic<T*>*,T* * old_value,T* new_value) noexcept;
bool atomic_compare_exchange_weak_explicit(
   volatile atomic<T*>*,T* * old_value, T* new_value,
   memory_order success_order,memory_order failure_order) noexcept;
bool atomic_compare_exchange_weak_explicit(
    atomic<T*>*,T* * old_value, T* new_value,
    memory_order success_order,memory_order failure_order) noexcept;

T* atomic_fetch_add(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_add_explicit(
    volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_add_explicit(
    atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub(volatile atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub(atomic<T*>*, ptrdiff_t) noexcept;
T* atomic_fetch_sub_explicit(
    volatile atomic<T*>*, ptrdiff_t, memory_order) noexcept;
T* atomic_fetch_sub_explicit(
    atomic<T*>*, ptrdiff_t, memory_order) noexcept;
```

Those operations that are also provided by the primary template (see 11.3.8) have the same semantics.

Std::Atomic<T>::Fetch_Add Member Function*

Atomically loads a value and replaces it with the sum of that value and the supplied value `i` using standard pointer arithmetic rules, and returns the old value.

Declaration

```
T* fetch_add(
    ptrdiff_t i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
T* fetch_add(
    ptrdiff_t i,memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically retrieves the existing value of `*this` and stores *old-value* `+ i` in `*this`.

Returns

The value of `*this` immediately prior to the store.

Throws

Nothing.

> **Note**
>
> This is an atomic read-modify-write operation for the memory location comprising `*this`.

Std::Atomic_Fetch_Add Nonmember Function

Atomically reads the value from an `atomic<T*>` instance and replaces it with that value plus the supplied value `i` using standard pointer arithmetic rules.

Declaration

```
T* atomic_fetch_add(volatile atomic<T*>* p, ptrdiff_t i) noexcept;
T* atomic_fetch_add(atomic<T*>* p, ptrdiff_t i) noexcept;
```

Effects

```
return p->fetch_add(i);
```

Std::Atomic_Fetch_Add_Explicit Nonmember Function

Atomically reads the value from an atomic<T*> instance and replaces it with that value plus the supplied value i using standard pointer arithmetic rules.

Declaration

```
T* atomic_fetch_add_explicit(
    volatile atomic<T*>* p, ptrdiff_t i,memory_order order) noexcept;
T* atomic_fetch_add_explicit(
    atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;
```

Effects

```
return p->fetch_add(i,order);
```

Std::Atomic<T>::Fetch_Sub Member Function*

Atomically loads a value and replaces it with that value minus the supplied value i using standard pointer arithmetic rules, and returns the old value.

Declaration

```
T* fetch_sub(
    ptrdiff_t i,memory_order order = memory_order_seq_cst)
    volatile noexcept;
T* fetch_sub(
    ptrdiff_t i,memory_order order = memory_order_seq_cst) noexcept;
```

Effects

Atomically retrieves the existing value of *this and stores *old-value* - i in *this.

Returns

The value of *this immediately prior to the store.

Throws

Nothing.

> **Note**
>
> This is an atomic read-modify-write operation for the memory location comprising *this.

Std::Atomic_Fetch_Sub Nonmember Function

Atomically reads the value from an atomic<T*> instance and replaces it with that value minus the supplied value i using standard pointer arithmetic rules.

Declaration

```
T* atomic_fetch_sub(volatile atomic<T*>* p, ptrdiff_t i) noexcept;
T* atomic_fetch_sub(atomic<T*>* p, ptrdiff_t i) noexcept;
```

Effects

```
return p->fetch_sub(i);
```

Std::Atomic_Fetch_Sub_Explicit Nonmember Function

Atomically reads the value from an atomic<T*> instance and replaces it with that value minus the supplied value i using standard pointer arithmetic rules.

Declaration

```
T* atomic_fetch_sub_explicit(
    volatile atomic<T*>* p, ptrdiff_t i,memory_order order) noexcept;
T* atomic_fetch_sub_explicit(
    atomic<T*>* p, ptrdiff_t i, memory_order order) noexcept;
```

Effects

```
return p->fetch_sub(i,order);
```

Std::Atomic<T>::Operator++ Preincrement Operator*

Atomically increments the value stored in *this using standard pointer arithmetic rules and returns the new value.

Declaration

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

Effects

```
return this->fetch_add(1) + 1;
```

Std::Atomic<T>::Operator++ Postincrement Operator*

Atomically increments the value stored in *this and returns the old value.

Declaration

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

Effects

```
return this->fetch_add(1);
```

Std::Atomic<T>::Operator-- Predecrement Operator*

Atomically decrements the value stored in `*this` using standard pointer arithmetic rules and returns the new value.

Declaration

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

Effects

```
return this->fetch_sub(1) - 1;
```

Std::Atomic<T>::Operator-- Postdecrement Operator*

Atomically decrements the value stored in `*this` using standard pointer arithmetic rules and returns the old value.

Declaration

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

Effects

```
return this->fetch_sub(1);
```

Std::Atomic<T>::Operator+= Compound Assignment Operator*

Atomically adds the supplied value to the value stored in `*this` using standard pointer arithmetic rules and returns the new value.

Declaration

```
T* operator+=(ptrdiff_t i) volatile noexcept;
T* operator+=(ptrdiff_t i) noexcept;
```

Effects

```
return this->fetch_add(i) + i;
```

Std::Atomic<T>::Operator-= Compound Assignment Operator*

Atomically subtracts the supplied value from the value stored in `*this` using standard pointer arithmetic rules and returns the new value.

Declaration

```
T* operator-=(ptrdiff_t i) volatile noexcept;
T* operator-=(ptrdiff_t i) noexcept;
```

Effects

```
return this->fetch_sub(i) - i;
```