# std::memory_order

C++

Atomic operations library

Defined in header <atomic>

```
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;
```

(since C++11) (until C++20)

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

(since C++20)

`std::memory_order` specifies how memory accesses, including regular, non-atomic memory accesses, are to be ordered around an atomic operation. Absent any constraints on a multi-core system, when multiple threads simultaneously read and write to several variables, one thread can observe the values change in an order different from the order another thread wrote them. Indeed, the apparent order of changes can even differ among multiple reader threads. Some similar effects can occur even on uniprocessor systems due to compiler transformations allowed by the memory model.

The default behavior of all atomic operations in the library provides for *sequentially consistent ordering* (see discussion below). That default can hurt performance, but the library's atomic operations can be given an additional `std::memory_order` argument to specify the exact constraints, beyond atomicity, that the compiler and processor must enforce for that operation.

## Constants

Defined in header <atomic>

| Value | Explanation |
|---|---|

| | |
|---|---|
| `memory_order_relaxed` | Relaxed operation: there are no synchronization or ordering constraints imposed on other reads or writes, only this operation's atomicity is guaranteed (see Relaxed ordering below) |
| `memory_order_consume` | A load operation with this memory order performs a *consume operation* on the affected memory location: no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load. Writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimizations only (see Release-Consume ordering below) |
| `memory_order_acquire` | A load operation with this memory order performs the *acquire operation* on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread (see Release-Acquire ordering below) |
| `memory_order_release` | A store operation with this memory order performs the *release operation*: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable (see Release-Acquire ordering below) and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic (see Release-Consume ordering below). |
| `memory_order_acq_rel` | A read-modify-write operation with this memory order is both an *acquire operation* and a *release operation*. No memory reads or writes in the current thread can be reordered before or after this store. All writes in other threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable. |
| `memory_order_seq_cst` | A load operation with this memory order performs an *acquire operation*, a store performs a *release operation*, and read-modify-write performs both an *acquire operation* and a *release operation*, plus a single total order exists in which all threads observe all modifications in the same order (see Sequentially-consistent ordering below) |

## Formal description

Inter-thread synchronization and memory ordering determine how *evaluations* and *side effects* of expressions are ordered between different threads of execution. They are defined in the following terms:

### Sequenced-before

Within the same thread, evaluation A may be *sequenced-before* evaluation B, as described in evaluation order.

### Carries dependency

Within the same thread, evaluation A that is *sequenced-before* evaluation B may also carry a dependency into B (that is, B depends on A), if any of the following is true

1) The value of A is used as an operand of B, **except**
a) if B is a call to std::kill_dependency
b) if A is the left operand of the built-in `&&`, `||`, `?:`, or `,` operators.
2) A writes to a scalar object M, B reads from M
3) A carries dependency into another evaluation X, and X carries dependency into B

### Modification order

All modifications to any particular atomic variable occur in a total order that is specific to this one atomic variable.

The following four requirements are guaranteed for all atomic operations:

1) **Write-write coherence**: If evaluation A that modifies some atomic M (a write) *happens-before* evaluation B that modifies M, then A appears earlier than B in the *modification order* of M
2) **Read-read coherence**: if a value computation A of some atomic M (a read) *happens-before* a value computation B on M, and if the value of A comes from a write X on M, then the value of B is either the value stored by X, or the value stored by a side effect Y on M that appears later than X in the *modification order* of M.
3) **Read-write coherence**: if a value computation A of some atomic M (a read) *happens-before* an operation B on M (a write), then the value of A comes from a side-effect (a write) X that appears earlier than B in the *modification order* of M
4) **Write-read coherence**: if a side effect (a write) X on an atomic object M *happens-before* a value computation (a read) B of M, then the evaluation B shall take its value from X or from a side effect Y that follows X in the modification order of M

### Release sequence

After a *release operation* A is performed on an atomic object M, the longest continuous subsequence of the modification order of M that consists of

rmed by the same thread that performed A (until C++20)

2) Atomic read-modify-write operations made to M by any thread

is known as *release sequence headed by A*

### Dependency-ordered before

Between threads, evaluation A is *dependency-ordered before* evaluation B if any of the following is true

1) A performs a *release operation* on some atomic M, and, in a different thread, B performs a *consume operation* on the same atomic M, and B reads a value written by any part of the release sequence headed (until C++20) by A.
2) A is dependency-ordered before X and X carries a dependency into B.

### Inter-thread happens-before

Between threads, evaluation A *inter-thread happens before* evaluation B if any of the following is true

1) A *synchronizes-with* B
2) A is *dependency-ordered before* B
3) A *synchronizes-with* some evaluation X, and X is *sequenced-before* B
4) A is *sequenced-before* some evaluation X, and X *inter-thread happens-before* B
5) A *inter-thread happens-before* some evaluation X, and X *inter-thread happens-before* B

### Happens-before

Regardless of threads, evaluation A *happens-before* evaluation B if any of the following is true:

1) A is *sequenced-before* B
2) A *inter-thread happens before* B

The implementation is required to ensure that the *happens-before* relation is acyclic, by introducing additional synchronization if necessary (it can only be necessary if a consume operation is involved, see Batty et al)

If one evaluation modifies a memory location, and the other reads or modifies the same memory location, and if at least one of the evaluations is not an atomic operation, the behavior of the program is undefined (the program has a data race) unless there exists a *happens-before* relationship between these two evaluations.

**Simply happens-before**

Regardless of threads, evaluation A simply happens-before evaluation B if any of the following is true:

1) A is sequenced-before B

2) A synchronizes-with B

3) A simply happens-before X, and X simply happens-before B

Note: without consume operations, simply happens-before and happens-before relations are the same.

(since C++20)

**Strongly happens-before**

Regardless of threads, evaluation A strongly happens-before evaluation B if any of the following is true:

1) A is sequenced-before B

2) A synchronizes-with B

3) A strongly happens-before X, and X strongly happens-before B

(until C++20)

1) A is sequenced-before B

2) A synchronizes with B, and both A and B are sequentially consistent atomic operations

3) A is sequenced-before X, X simply happens-before Y, and Y is sequenced-before B

4) A strongly happens-before X, and X strongly happens-before B

Note: informally, if A strongly happens-before B, then A appears to be evaluated before B in all contexts.

Note: strongly happens-before excludes consume operations.

(since C++20)

## Visible side-effects

The side-effect A on a scalar M (a write) is *visible* with respect to value computation B on M (a read) if both of the following are true:

1) A *happens-before* B
2) There is no other side effect X to M where A *happens-before* X and X *happens-before* B

If side-effect A is visible with respect to the value computation B, then the longest contiguous subset of the side-effects to M, in *modification order*, where B does not *happen-before* it is known as the *visible sequence of side-effects*. (the value of M, determined by B, will be the value stored by one of these side effects)

Note: inter-thread synchronization boils down to preventing data races (by establishing happens-before relationships) and defining which side effects become visible under what conditions

### Consume operation

Atomic load with `memory_order_consume` or stronger is a consume operation. Note that std::atomic_thread_fence imposes stronger synchronization requirements than a consume operation.

### Acquire operation

Atomic load with `memory_order_acquire` or stronger is an acquire operation. The lock() operation on a *Mutex* is also an acquire operation. Note that std::atomic_thread_fence imposes stronger synchronization requirements than an acquire operation.

### Release operation

Atomic store with `memory_order_release` or stronger is a release operation. The unlock() operation on a *Mutex* is also a release operation. Note that std::atomic_thread_fence imposes stronger synchronization requirements than a release operation.

## Explanation

### Relaxed ordering

Atomic operations tagged `memory_order_relaxed` are not synchronization operations; they do not impose an order among concurrent memory accesses. They only guarantee atomicity and modification order consistency.

For example, with `x` and `y` initially zero,

```
// Thread 1:
r1 = y.load(std::memory_order_relaxed); // A
x.store(r1, std::memory_order_relaxed); // B
// Thread 2:
r2 = x.load(std::memory_order_relaxed); // C
y.store(42, std::memory_order_relaxed); // D
```

is allowed to produce `r1 == r2 == 42` because, although A is *sequenced-before* B within thread 1 and C is *sequenced before* D within thread 2, nothing prevents D from appearing before A in the modification order of y, and B from appearing before C in the modification order of x. The side-effect of

D on y could be visible to the load A in thread 1 while the side effect of B on x could be visible to the load C in thread 2. In particular, this may occur if D is completed before C in thread 2, either due to compiler reordering or at runtime.

Even with relaxed memory model, out-of-thin-air values are not allowed to circularly depend on their own computations, for example, with `x` and `y` initially zero,

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

(since C++14)

is not allowed to 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 until C++14, this was technically allowed by the specification, but not recommended for implementors.

Typical use for relaxed memory ordering is incrementing counters, such as the reference counters of std::shared_ptr, since this only requires atomicity, but not ordering or synchronization (note that decrementing the shared_ptr counters requires acquire-release synchronization with the destructor)

```
#include <vector>
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<int> cnt = {0};

void f()
{
    for (int n = 0; n < 1000; ++n) {
        cnt.fetch_add(1, std::memory_order_relaxed);
    }
}

int main()
{
    std::vector<std::thread> v;
    for (int n = 0; n < 10; ++n) {
        v.emplace_back(f);
    }
    for (auto& t : v) {
        t.join();
    }
    std::cout << "Final counter value is " << cnt << '\n';
}
```

Output:

```
Final counter value is 10000
```

## Release-Acquire ordering

If an atomic store in thread A is tagged `memory_order_release` and an atomic load in thread B from the same variable is tagged `memory_order_acquire`, all memory writes (non-atomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A,

become *visible side-effects* in thread B. That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.

The synchronization is established only between the threads *releasing* and *acquiring* the same atomic variable. Other threads can see different order of memory accesses than either or both of the synchronized threads.

On strongly-ordered systems — x86, SPARC TSO, IBM mainframe, etc. — release-acquire ordering is automatic for the majority of operations. No additional CPU instructions are issued for this synchronization mode; only certain compiler optimizations are affected (e.g., the compiler is prohibited from moving non-atomic stores past the atomic store-release or performing non-atomic loads earlier than the atomic load-acquire). On weakly-ordered systems (ARM, Itanium, PowerPC), special CPU load or memory fence instructions are used.

Mutual exclusion locks, such as std::mutex or atomic spinlock, are an example of release-acquire synchronization: when the lock is released by thread A and acquired by thread B, everything that took place in the critical section (before the release) in the context of thread A has to be visible to thread B (after the acquire) which is executing the same critical section.

```
#include <thread>
#include <atomic>
#include <cassert>
#include <string>

std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // never fires
    assert(data == 42); // never fires
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

The following example demonstrates transitive release-acquire ordering across three threads

```
#include <thread>
#include <atomic>
#include <cassert>
```

```cpp
#include <vector>

std::vector<int> data;
std::atomic<int> flag = {0};

void thread_1()
{
    data.push_back(42);
    flag.store(1, std::memory_order_release);
}

void thread_2()
{
    int expected=1;
    while (!flag.compare_exchange_strong(expected, 2,
std::memory_order_acq_rel)) {
        expected = 1;
    }
}

void thread_3()
{
    while (flag.load(std::memory_order_acquire) < 2)
        ;
    assert(data.at(0) == 42); // will never fire
}

int main()
{
    std::thread a(thread_1);
    std::thread b(thread_2);
    std::thread c(thread_3);
    a.join(); b.join(); c.join();
}
```

### Release-Consume ordering

If an atomic store in thread A is tagged `memory_order_release` and an atomic load in thread B from the same variable that read the stored value is tagged `memory_order_consume`, all memory writes (non-atomic and relaxed atomic) that *happened-before* the atomic store from the point of view of thread A, become *visible side-effects* within those operations in thread B into which the load operation *carries dependency*, that is, once the atomic load is completed, those operators and functions in thread B that use the value obtained from the load are guaranteed to see what thread A wrote to memory.

The synchronization is established only between the threads *releasing* and *consuming* the same atomic variable. Other threads can see different order of memory accesses than either or both of the synchronized threads.

On all mainstream CPUs other than DEC Alpha, dependency ordering is automatic, no additional CPU instructions are issued for this synchronization mode, only certain compiler optimizations are affected (e.g. the compiler is prohibited from performing speculative loads on the objects that are involved in the dependency chain).

Typical use cases for this ordering involve read access to rarely written concurrent data structures (routing tables, configuration, security policies, firewall rules, etc) and publisher-subscriber situations with pointer-mediated publication, that is, when the producer publishes a pointer through which the consumer can access information: there is no need to make everything else the producer wrote to memory visible to the consumer (which may be an expensive operation on weakly-ordered architectures). An example of such scenario is rcu_dereference.

See also std::kill_dependency and `[[carries_dependency]]` for fine-grained dependency chain control.

Note that currently (2/2015) no known production compilers track dependency chains: consume operations are lifted to acquire operations.

> The specification of release-consume ordering is being revised, and the use of `memory_order_consume` is temporarily discouraged. (since C++17)

This example demonstrates dependency-ordered synchronization for pointer-mediated publication: the integer data is not related to the pointer to string by a data-dependency relationship, thus its value is undefined in the consumer.

```
#include <thread>
#include <atomic>
#include <cassert>
#include <string>

std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)))
        ;
    assert(*p2 == "Hello"); // never fires: *p2 carries dependency from ptr
    assert(data == 42); // may or may not fire: data does not carry
dependency from ptr
}

int main()
{
    std::thread t1(producer);
    std::thread t2(consumer);
    t1.join(); t2.join();
}
```

### Sequentially-consistent ordering

Atomic operations tagged `memory_order_seq_cst` not only order memory the same way as release/acquire ordering (everything that *happened-before* a store in one thread becomes a *visible side effect* in the thread that did a load), but also establish a *single total modification order* of all atomic operations that are so tagged.

Formally,

Each `memory_order_seq_cst` operation B that loads from atomic variable M, observes one of the following:

- the result of the last operation A that modified M, which appears before B in the single total order
- OR, if there was such an A, B may observe the result of some modification on M that is not `memory_order_seq_cst` and does not *happen-before* A
- OR, if there wasn't such an A, B may observe the result of some unrelated modification of M that is not `memory_order_seq_cst`

If there was a `memory_order_seq_cst` std::atomic_thread_fence operation X *sequenced-before* B, then B observes one of the following:

- the last `memory_order_seq_cst` modification of M that appears before X in the single total order
- some unrelated modification of M that appears later in M's modification order

For a pair of atomic operations on M called A and B, where A writes and B reads M's value, if there are two `memory_order_seq_cst` std::atomic_thread_fences X and Y, and if A is *sequenced-before* X, Y is *sequenced-before* B, and X appears before Y in the Single Total Order, then B observes either: (until C++20)

- the effect of A
- some unrelated modification of M that appears after A in M's modification order

For a pair of atomic modifications of M called A and B, B occurs after A in M's modification order if

- there is a `memory_order_seq_cst` std::atomic_thread_fence X such that A is *sequenced-before* X and X appears before B in the Single Total Order
- or, there is a `memory_order_seq_cst` std::atomic_thread_fence Y such that Y is *sequenced-before* B and A appears before Y in the Single Total Order
- or, there are `memory_order_seq_cst` std::atomic_thread_fences X and Y such that A is *sequenced-before* X, Y is *sequenced-before* B, and X appears before Y in the Single Total Order.

Note that this means that:

omic operations that are not tagged `memory_order_seq_cst` enter the picture, the sequential consistency is lost

ally-consistent fences are only establishing total ordering for the fences themselves, not for the atomic operations in the general case (*sequenced-before* is not a cross-thread relationship, unlike *happens-before*)

Formally,

An atomic operation A on some atomic object M is *coherence-ordered-before* another atomic operation B on M if any of the following is true: (since C++20)

1) A is a modification, and B reads the value stored by A
2) A precedes B in the *modification order* of M

e value stored by an atomic modification X, X precedes B in the *modification order*, and A and B are not the same atomic read-modify-write operation

4) A is *coherence-ordered-before* X, and X is *coherence-ordered-before* B

There is a single total order S on all `memory_order_seq_cst` operations, including fences, that satisfies the following constraints:

d B are `memory_order_seq_cst` operations, and A *strongly happens-before* B, then A precedes B in S
for every pair of atomic operations A and B on an object M, where A is *coherence-ordered-before* B:
a) if A and B are both `memory_order_seq_cst` operations, then A precedes B in S
`emory_order_seq_cst` operation, and B *happens-before* a `memory_order_seq_cst` fence Y, then A precedes Y in S
`y_order_seq_cst` fence X *happens-before* A, and B is a `memory_order_seq_cst` operation, then X precedes B in S
`y_order_seq_cst` fence X *happens-before* A, and B *happens-before* a `memory_order_seq_cst` fence Y, then X precedes Y in S

The formal definition ensures that:

1) the single total order is consistent with the *modification order* of any atomic object
`_order_seq_cst` load gets its value either from the last `memory_order_seq_cst` modification, or from some non-`memory_order_seq_cst` modification that does not *happen-before* preceding `memory_order_seq_cst` modifications

The single total order might not be consistent with *happens-before*. This allows more efficient implementation of `memory_order_acquire` and `memory_order_release` on some CPUs. It can produce surprising results when `memory_order_acquire` and `memory_order_release` are mixed with `memory_order_seq_cst`.

For example, with `x` and `y` initially zero,

```
                    // Thread 1:
         x.store(1, std::memory_order_seq_cst); // A
         y.store(1, std::memory_order_release); // B
                    // Thread 2:
r1 = y.fetch_add(1, std::memory_order_seq_cst); // C
     r2 = y.load(std::memory_order_relaxed); // D
                    // Thread 3:
         y.store(3, std::memory_order_seq_cst); // E
         r3 = x.load(std::memory_order_seq_cst); // F
```

is allowed to produce `r1 == 1 && r2 == 3 && r3 == 0`, where A *happens-before* C, but C precedes A in the single total order C-E-F-A of `memory_order_seq_cst` (see Lahav et al).

Note that:

1) as soon as atomic operations that are not tagged memory_order_seq_cst enter the picture, the sequential consistency guarantee for the program is lost

2) in many cases, memory_order_seq_cst atomic operations are reorderable with respect to other atomic operations performed by the same thread

Sequential ordering may be necessary for multiple producer-multiple consumer situations where all consumers must observe the actions of all producers occurring in the same order.

Total sequential ordering requires a full memory fence CPU instruction on all multi-core systems. This may become a performance bottleneck since it forces the affected memory accesses to propagate to every core.

This example demonstrates a situation where sequential ordering is necessary. Any other ordering may trigger the assert because it would be possible for the threads `c` and `d` to observe changes to the atomics `x` and `y` in opposite order.

```
#include <thread>
#include <atomic>
#include <cassert>

std::atomic<bool> x = {false};
std::atomic<bool> y = {false};
std::atomic<int> z = {0};

void write_x()
{
    x.store(true, std::memory_order_seq_cst);
}

void write_y()
{
    y.store(true, std::memory_order_seq_cst);
}

void read_x_then_y()
{
    while (!x.load(std::memory_order_seq_cst))
        ;
    if (y.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

void read_y_then_x()
{
    while (!y.load(std::memory_order_seq_cst))
        ;
    if (x.load(std::memory_order_seq_cst)) {
        ++z;
    }
}

int main()
{
    std::thread a(write_x);
    std::thread b(write_y);
    std::thread c(read_x_then_y);
    std::thread d(read_y_then_x);
    a.join(); b.join(); c.join(); d.join();
    assert(z.load() != 0);  // will never happen
}
```

## Relationship with `volatile`

Within a thread of execution, accesses (reads and writes) through volatile glvalues cannot be reordered past observable side-effects (including other volatile accesses) that are *sequenced-before* or *sequenced-after* within the same thread, but this order is not guaranteed to be observed by another thread, since volatile access does not establish inter-thread synchronization.

In addition, volatile accesses are not atomic (concurrent read and write is a data race) and do not order memory (non-volatile memory accesses may be freely reordered around the volatile access).

One notable exception is Visual Studio, where, with default settings, every volatile write has release semantics and every volatile read has acquire semantics (MSDN), and thus volatiles may be used for inter-thread synchronization. Standard `volatile` semantics are not applicable to multithreaded programming, although they are sufficient for e.g. communication with a std::signal handler that runs in the same thread when applied to `sig_atomic_t` variables.