# Atomic stores and object lifetimes

Doc. No: P3181R1
Authors: Hans Boehm, Dave Clausen, David Goldblatt
Contact: Hans Boehm (hboehm@google.com)
Audience: SG1
Date: Feb 22, 2026
Target: C++26

## Abstract

The current object lifetime rules insist that the last update to an object happens-before destruction of the object. This is arguably too conservative in ways that the authors found surprising and undesirable. We propose to relax the rule in a way that better aligns it with programmer intuition.

## Modification history

An initial draft version of this paper was discussed in Tokyo. There appeared to be general consensus to proceed, and that we need carefully worked out wording. There was no real consensus on direction for the wording. There is clearly work remaining to be done here.

R0 was briefly discussed again at the St. Louis meeting, with tentative agreement to

1. Attempt to remove memory_order_consume, which separately happened since then.
2. Limit the change to object lifetimes, for reasons given below.
3. Separately clean up the uses of different happens-before flavors in the standard. Still needs to be done, but was facilitated by the memory_order_consume removal.

R1
Typo fixes. Replace “Towards a solution” with a more concrete wording proposal. Expand the wording to include [ basic.life ] in general.

## Introduction

The intent of the standard is that all accesses to an object should happen during its lifetime, i.e. construction should happen before the access and the access should happen before destruction.[1]

Now consider the case in which the final access is in Thread A, and it is a store to an atomic<bool>. Thread B performs an acquire load on the same atomic and then destroys the object, after which it gets deallocated, or goes out of scope.

According to the standard, this is fine if the store in A is a release-store. It is not fine if the store to A is a release fence, followed by a relaxed store, since in that case it is the fence, not the store, that synchronizes with the acquire-load. In the release-store case, the store happens-before the deletion; in the fence case it does not.

For something like the standard message passing litmus test the difference between “fence; relaxed-store” is usually stronger than “release-store”, and the difference often doesn’t matter. Here it is the other way around.

## Detailed example

We can write the version with the release-fence as follows. Here we also include a third thread C that subsequently happens to allocate the same memory used for the previously described load and store. The third thread, though not very relevant from the standard’s perspective, is important for discussing possible resulting implementation issues.

Thread A:

```
fence(release);                 // irrelevant for present purposes.
a1: a->store(1, relaxed);
```

Thread B:

```
b1: r1 = a->load(acquire);      // sees 1.
b2: delete a;
```

Reallocating thread C:

```
c1: b = new atomic<int>(0);     // gets the same location as a.
c2: r2 = b->load(relaxed);      // Should not see the 1 from thread A.
```

[res.on.objects]p2 states:

[1] This paper will ignore accesses during construction and destruction, which is an important, but separable issue.

> “If an object of a standard library type is accessed, and … the access does not happen before the end of the object’s lifetime, the behavior is undefined unless otherwise specified.
> [Note 2 : This applies even to objects such as mutexes intended for thread synchronization. — end note]”

The allocator must ensure that b2 happens-before c1. We need a1 happens-before c2, which just doesn't follow. If we somehow promised that a1 did happen-before b1 would effectively turn `memory_order::relaxed` into `memory_order::release`, which is clearly unacceptable.

The current rules are backed by some possible intuition: [ Warning: This is a bit of a stretch. ] Consider a distributed shared-memory implementation that may nondeterministically send asynchronous updates, and has release operations send visible changes to acquire operations. (A release fence would ensure that all later acquire operations see changes.) b1 could happen to see a1's update, but there is no guarantee the reallocating thread sees it before the allocation, since there is no communicating release-acquire pair involving the child thread.

## Stronger semantics are probably still fine

On the other hand, this can’t really happen on implementations that compile the non-atomic initializing store in c1 in the same way as they would a relaxed atomic store. In that case, the modification order / coherence rules in the standard prohibit the above behavior.

We can roughly model the preceding example as:

Thread A:

```
a1: a.store(1, relaxed);
```

Thread B:

```
b1: r1 = a.load(relaxed);  // sees 1
b2: b.store(1, release);   // Somewhere inside the delete implementation.
```

Thread C: (reallocating thread)

```
c1: r2 = b.load(acquire);     // Somewhere inside allocator implementation.
c2: a.store(0, non-atomic);   //  initialization of new object, kind of
c3: a.load();                 // Sees 1?
```

For now, let's treat c2 as a relaxed store.

If c2 follows a1 in the modification order, then, by [intro.races]p16, this can't happen, since b1 happens before c3, and thus c3 must see a later store.

So we would have to have a1 follow c2 in the modification order. But that would violate [intro.races]p17: "If a value computation A [b1] of an atomic object M [a] happens before an operation B [c2] that modifies M [a], then A[b1] takes its value from a side effect X on M, where X precedes B [c2]  in the modification order of M."

This seems to be a fairly convincing argument that the problematic outcome in which the load at c2 sees 1 isn't possible on common implementations.

The only small fly in the ointment here is the non-atomic store at c2. The question then is: What implementations are there that implement a non-atomic store differently from a relaxed one? Any sort of compiler reordering seems implausible here, since in reality b2 and c1 must communicate a's address. The only architecture I know of that by design required stronger ordering for relaxed stores was Itanium. There were some errata for other old processors that sort of had this effect, but we don't know of anything modern enough to be a serious constraint. And we're not convinced that even those old processors would actually pose a problem here; it's just that the above safety argument does not directly apply.

## Why this matters

An important property of synchronization primitives generally is that they be able to "synchronize their own destruction". As an example: if an object has an embedded mutex protecting it, as well as a refcount, it should be valid for a thread to acquire the mutex, decrement the refcount, see that it became zero, release the mutex, and destroy the object.

(C++ and posix mutexes work this way, but historically some mutex implementations did not -- if your platform implements mutexes by embedding an OS handle alongside an atomic word there are very difficult to manage races in between unlocking a mutex and waking any waiters).

If fences aren't strong enough to make this work, "relaxed atomic + fence" suddenly loses the ability to provide this guarantee. That hurts the memory model sanity generally: "inserting seq_cst fences before and after relaxed accesses gives seq_cst semantics" is very intuitive, and true except for this edge case (related "very obviously should be true" claims break too, like "it's correct to replace store(release) with fence(release); store(relaxed)"). Fences end up with a weird safety restriction that's not motivated by the underlying hardware.

For example, a reference count decrement operation might be written as

```
int count = x->refcount.fetch_sub(1, acq_rel);
if (count == 1) delete x;
```

This becomes subtly incorrect when rewritten as

```
atomic_thread_fence(release);
int count = x->refcount.fetch_sub(1, relaxed);
atomic_thread_fence(acquire);
if (count == 1) delete x;
```

When you're writing some bit of synchronization you don't necessarily know how your caller will use the synchronization you provide, so you can't assume it *won't* be used to synchronize destruction. When you're using someone else's primitive, you don't know how they implemented their synchronization guarantees, and shouldn't have to in order to be able to safely destroy it.

## Proposed wording

We follow the argument made in St. Louis that modifying the happens-before relationship itself is undesirable. It would involve adding a new kind of happens-before that does not compose with sequenced-before on the left, since otherwise relaxed stores would basically turn into release stores. And the additional guarantees we would get out of such an approach are basically applicable only to object destruction anyway, since other guarantees are either already provided by coherence or invalid.

Wording is relative to n5032.

Modify [ basic.life ] p1 as follows:

In this subclause, “before” and “after” refer to the “happens before” relation (6.10.2), with the following relaxation: An atomic store *X* to an object is also considered to be “before” the end of an object's lifetime, if a load both observes the value stored by *X* and happens-before the end of the object’s lifetime. [ Note: This allows a `memory_order_relaxed` store, possibly preceded by an `atomic_thread_fence()` to inform another thread that an object is ready for deletion. --end note ]

Modify [ res.on.objects ] as follows:

If an object of a standard library type is accessed, and the beginning of the object’s lifetime (6.8.4) does not happen before the access, or the access does not happen before the end of the object’s lifetime, the behavior is undefined unless otherwise specified.

If an object of a standard library type is accessed, and the beginning of the object’s lifetime (6.8.4) does not happen before the access, or the access ~~does not happen~~ is not “before” the end of the object’s lifetime, as defined in [ basic.life ], the behavior is undefined unless otherwise specified.