Acquire and Release Fences

This lesson gives an overview of acquire and release fences used in C++ as memory barriers.

WE'LL COVER THE FOLLOWING

The most obvious difference between *acquire* and *release* fences and atomics with acquire-release semantics is that fences need no atomics. There is also a more subtle difference: the acquire and release fences are more heavyweight.

Atomic Operations vs Fences

For the sake of simplicity, I will now refer to acquire operations when I use fences or atomic operations with **acquire semantic**. The same will hold for release operations.

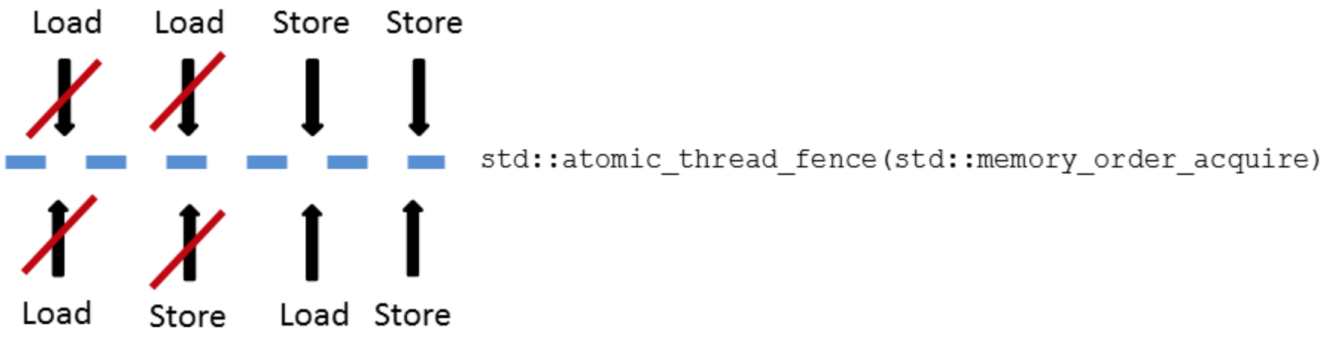
The main idea of an acquire and a release operation is that it establishes synchronization and ordering constraints between threads. These synchronization and ordering constraints also hold for atomic operations with relaxed semantic or non-atomic operations. Note that acquire and release operations come in pairs. In addition, operations on atomic variables with acquire-release semantic must act on the same atomic variable. Having said that, I will now look at these operations in isolation. Let's start with the acquire operation.

Acquire Operation:

A load (read) operation on an atomic variable with the memory model set to `std::memory_order_acquire` is an acquire operation.

```
int ready= var.load(std::memory_order_acquire);
// load and store operations
```

`std::atomic_thread_fence` with the memory order set to `std::memory_order_acquire` imposes stricter constraints on memory access reordering:



This comparison emphasizes two points:

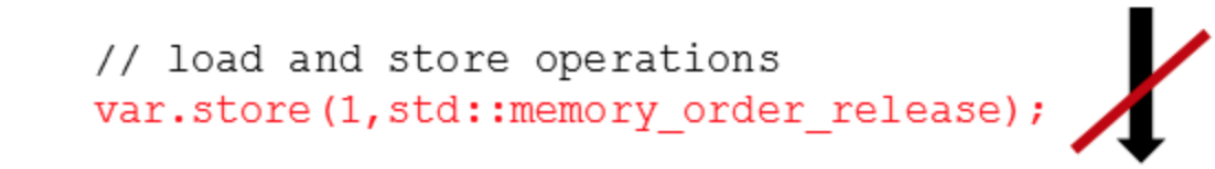
1. A fence with *acquire-semantic* establishes stronger ordering constraints. Although the acquire operation on an atomic and a fence requires that no read or write operation can be moved before the acquire operation, there is an additional guarantee with the acquire fence. No read operation can be moved after the acquire fence.
2. The *relaxed-semantic* is sufficient for the reading of the atomic variable `var`. Thanks to `std::atomc_thread_fence(std::memory_order_acquire)`, this operation cannot be moved after the acquire fence.

Similar observations can be made for the release fence.

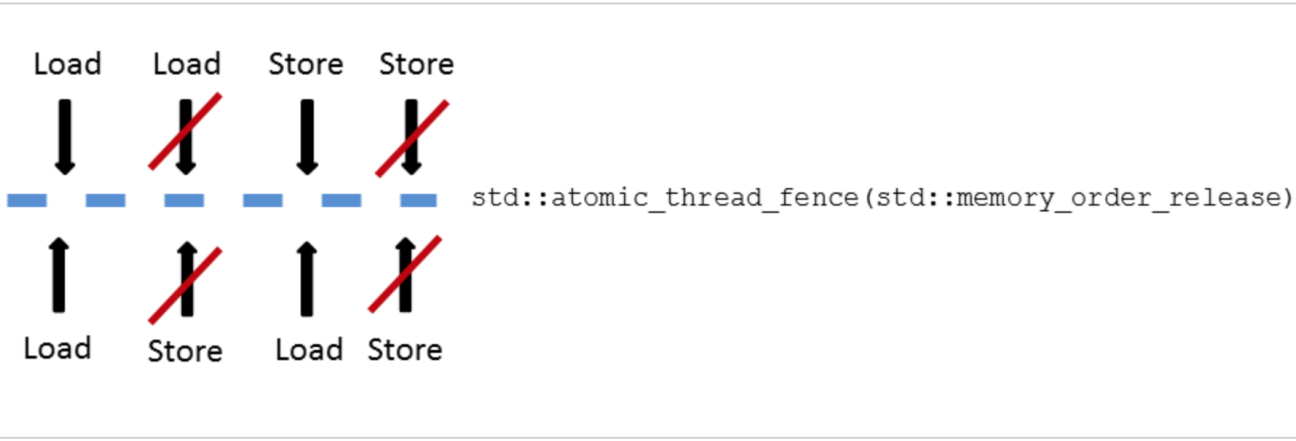
Release Operation:

The store (write) operation on an atomic variable attached with the memory model set to `std::memory_order_release` is a release operation.

```
// load and store operations
var.store(1,std::memory_order_release);
```



Here is the corresponding release fence.



In addition to the constraints imposed by the release operation on an atomic variable `var`, the release fence guarantees two properties:

1. Store operations can't be moved before the fence.
2. It's sufficient for the variable `var` to have relaxed semantic.

But now, it's time to go one step further and build a program in the next lesson that will use fences.