Fences as Memory Barriers

This lesson introduces a concept of fences as memory barriers in C++.

An `std::atomic_thread_fence` prevents specific operations from crossing a fence, and it doesn't need an atomic variable; they are frequently just referred to as fences or memory barriers. You quickly get an idea of what an `std::atomic_thread_fence` is all about.

What does it mean by *Fences as Memory Barriers*? Specific operations cannot cross a memory barrier. What kind of operations? From a bird's-eye view, we have two kinds of operations: read and write or load and store operations. The expression `if(resultRead) return result` is a load, followed by a store operation. There are four different ways to combine load and store operations:

Combination	Meaning
LoadLoad	A load followed by a load
LoadStore	A load followed by a store
StoreLoad	A store followed by a load
StoreStore	A store followed by a store

Of course, there are more complex operations consisting of multiple load and stores (count++), and these operations fall into my general classification.

What about memory barriers? If you place memory barriers between two operations like LoadLoad, LoadStore, StoreLoad or StoreStore, you have the guarantee that specific LoadLoad, LoadStore, StoreLoad or StoreStore operations will not be reordered. The risk of reordering is always present if non-atomics or atomic operations with relaxed semantic are used.