Introduction

This lesson introduces the concepts of synchronization and ordering constraints in C++.

WE'LL COVER THE FOLLOWING

You cannot configure the atomicity of an atomic data type, but you can accurately adjust the synchronization and ordering constraints of atomic operations. This possibility is unique to C++, as it's not possible in C#'s or Java's memory model.

There are six different variants of the memory model in C++. The key question is what are their characteristics?

Variants of the Memory Model

We already know C++ has six variants of the memory models. The default for atomic operations is `std::memory_order_seq_cst`; this expression stands for sequential consistency. In addition, you can explicitly specify one of the other five. So what does C++ have to offer?

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

To classify these six memory models, it helps to answer two questions:

1. Which kind of atomic operations should use which memory model?

2. Which synchronization and ordering constraints are defined by the six variants?

My plan is quite simple: I will answer both questions.

Kind of Atomic Operation

There are three different kinds of operations:

- Read operation: `memory_order_acquire` and `memory_order_consume`
- Write operation: `memory_order_release`
- Read-modify-write operation: `memory_order_acq_rel` and `memory_order_seq_cst`

`memory_order_relaxed` defines no synchronization and ordering constraints; therefore, it does not fit in this taxonomy. The following table orders the atomic operations based on their reading and/or writing characteristics.

Operation	read	write	read-modify-write
`test_and_set`			yes
`clear`		yes	
`is_lock_free`	yes		
`load`	yes		
`store`		yes	
`exchange`			yes

`compare_exchange_strong` `compare_exchange_weak`			yes

`fetch_add`, `+=`
`fetch_sub`, `-=`
			yes

`fetch_or`, `|=`
`fetch_and`, `&=`
`fetch_xor`, `^=`
			yes

`++`, `--`			yes

If you use an atomic operation `atomVar.load()` with a memory model that is designed for a write or read-modify-write operation, the write part has no effect. The result is that operation `atomVar.load(std::memory_order_acq_rel)` is equivalent to operation `atomVar.load(std::memory_order_acquire)`;

operation `atomVar.load(std::memory_order_release)` is equivalent to `atomVar.load(std::memory_order_relaxed)`.