The atomics are the base of the C++ memory model. Per default, sequential consistency is applied.

The strong C++ memory model

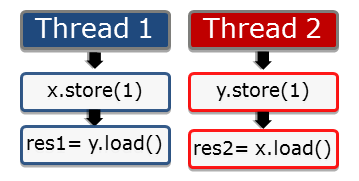
In 2004 Java 5.0 gets its current [memory model](https://en.wikipedia.org/wiki/Java_memory_model), in 2011 C++.  Before that, Java had an erroneous memory model, C++ had no memory model. Who thinks, that this is the endpoint of a long process, is totally wrong. The foundations of multithreading programming are 40 to 50 years old. So [Leslie Lamport](https://en.wikipedia.org/wiki/Leslie_Lamport)defined 1979 the concept of sequential consistency.

**Sequential consistency** provides two guarantees.

1. The instructions of a program are executed in source code order.
2. There is a global order of all operations on all threads.

Before I look deeper in these two guarantees, I explicitly emphasise. The statements only hold for **atomics** but influence **non-atomics.**

The simple graphic displays two threads. Each thread is storing its variable x or y, loads the other variable y and x and stores them in the variable res1 or res2.

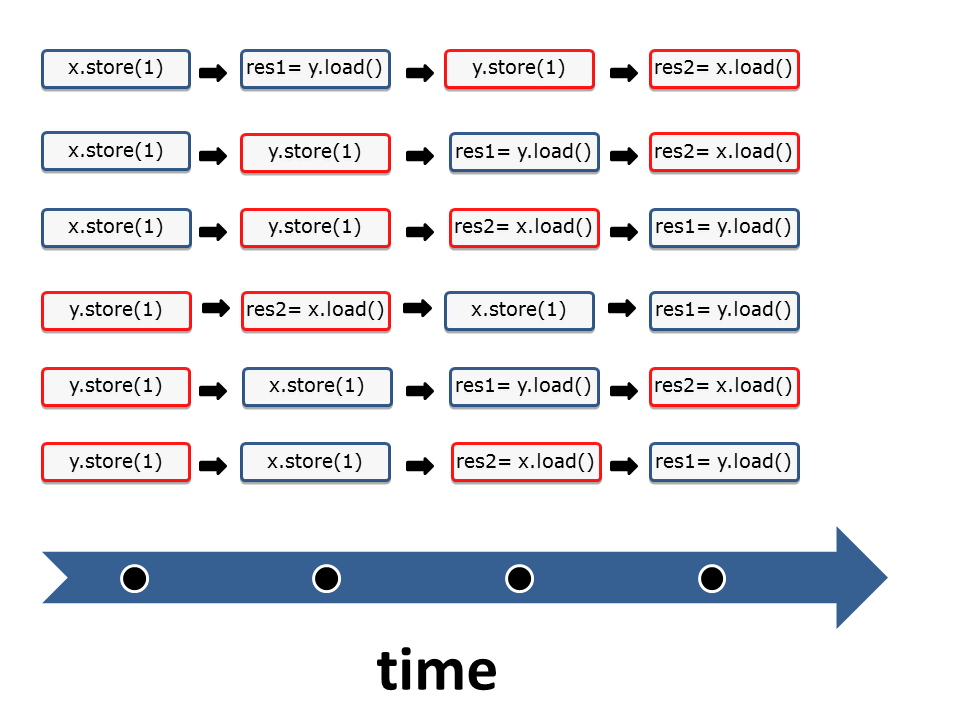


The operations take place on atomics and so, they are atomic. By default, sequential consistency applies. But the question is. In which order can the statements take place?

The first guarantee of the sequential consistency is that the instruction will be executed in the order of the source code. That is easy. No store operation can overtake a load operation.

The second guarantee of the sequential consistency is, that all instructions of all threads have to follow a global order. That means in that concrete case, that thread 2 sees the operations of thread 1 in the same order, in which thread 1 executes them. This is the key observation. **Thread 2 sees all operations of thread 1 in the source code order of thread 1.** The same holds from the perspective of thread 1. So you can think about characteristic 2 as a global counter, which all threads have to obey. The global counter is the global order.

We're not done with our riddle right now. What is still missing, is to look at the different interleaving executions of the two threads. So the following six interleavings of the two threads are possible.



That was easy. Or?

From the strong to the weak memory model

I want once more refer to the [picture](http://modernescpp.com/index.php/c-memory-model) of the contract between the programmer and the system.

The programmer uses atomics in this particular example. So he obeys his part of the contract by using them in the right way. The system guarantees him a well-defined program behaviour without data races. In addition to that, the system can execute the four operations in each combination. In case the programmer uses the relaxed semantic, the pillars of the contract dramatically changes. On one hand, it is a lot more difficult for the programmer to apply the contract in the right way. On the other hand, the system has a lot more optimisation possibilities. With the relaxed semantic - also called weak memory model - there are a lot more combinations of the four operations possible. The counter-intuitive behaviour is, that the thread 1 can see the operations of thread 2 in a different order. So there is no picture of a global counter. From the perspective of thread 1 it is possible, that the operation res= y.load() overtake x.store().

Between the sequential consistency and the relaxed-semantic, there are a few more models. The most important one is the acquire-release semantic. I think you already guess it. With acquire-release semantic, the programmer has to obey weaker rules than with sequential consistency. But the system has more optimisation possibilities. The acquire-release semantic is the key for a deeper understanding of the multithreading programming because the threads will be synchronised at specific synchronisation points in the code. Without these synchronisation points, there is no well-defined behaviour of threads, tasks or condition variables possible. More about that in the following post.