# Memory Model

This lesson provides a simple set of rules for efficiently using C++ memory models.

WE'LL COVER THE FOLLOWING



The foundation of multithreading is a *well-defined* memory model. Having a basic understanding of the memory helps a lot to get a deeper insight into the multithreading challenges.

### Don’t use Volatile for Synchronization #

In C++, in contrast to C# or Java, `volatile` has no multithreading semantic . In C# or Java, `volatile` declares an atomic such as `std::atomic` declares an atomic in C++, and it is typically used for objects which can change independently of the regular program flow. Due to this characteristic, no optimized storing in caches takes place.

### Don’t Program Lock-Free #

This advice sounds ridiculous after writing a course about concurrency and having an entire chapter dedicated to the memory model. However, the reason for this advice is quite simple. Lock-free programming is very error-prone and requires an expert level of knowledge. In particular, if you want to implement a lock-free data structure, be aware of the ABA problem.

### If you program Lock-Free, use well-established patterns #

If you have identified a bottleneck that could benefit from a lock-free solution, apply established patterns.

1. Sharing an atomic boolean or an atomic counter is straightforward.
2. Use a thread-safe or even lock-free container to support consumer/producer scenario. If your container is thread-safe, you can put and get values from the container without worrying about synchronization; you simply shift the application challenges to the infrastructure.

### Don’t build your own abstraction, use guarantees of the language #

*Thread-safe initialization* of a shared variable can be done in various ways: you can rely on guarantees of the C++ runtime such as constant expressions, static variables with block scope, or use the function `std::call_once` in combination with the flag `std::once_flag`. We program in C++; therefore, you can build your own abstraction based on atomics using even the highly sophisticated acquire-release semantic. Don’t do this in the first place, unless you have to do it – i.e. if you have identified a bottleneck by measuring the performance of a critical code path. Only make the change if you know that your handcrafted version will outperform the default guarantees of the language.