# General

This lesson provides a simple set of rules for writing well-defined and fast, concurrent programs in modern C++.

WE'LL COVER THE FOLLOWING



Multithreading, parallelism, and concurrency, in particular, are quite new topics in C++; therefore, more and more best practices will be discovered in the coming years. Consider the rules in this chapter not as a complete list, but rather as a necessary starting point that will evolve over time. This holds particularly true for the parallel STL. At the time of writing this course (08/2017), the new C++17 standard - including the parallel algorithms - hasn't been published yet; therefore, it is too early to formulate best practices for it.

Let's start with a few very general best practices that will apply to atomics and threads.

## Code Reviews #

Code reviews should be part of each professional software development process; this holds especially true when you deal with concurrency. Concurrency is inherently complicated and requires a lot of thoughtful analysis and experience.

To make the review most effective, send the code you want to discuss to the reviewers before the review. Explicitly state which invariants should apply to your code. The reviewers should have enough time to analyze the code before the official review starts.

### Minimize Data Sharing of Mutable Data #

You should minimize data sharing of mutable data for two reasons: performance and safety. Safety is mainly about data races. Let me focus on performance in this paragraph. I will deal with correctness in the following best practices section.

You may have heard of Amdahl's law. It predicts the theoretical maximum speedup you can get using multiple processors. The law is quite simple: If p is the proportion of your code that can run concurrently, you will get a maximum speedup of $ \frac{1}{1-p} $. So, if 90% of your code can run concurrently, you will get at most a 10 times speedup: $ \frac{1}{1-p} = \frac{1}{1-0.9} = \frac{1}{0.1} = 10 $.

From the opposite perspective: if 10% of your code has to run sequentially because you use a lock, you will get at most a 10 times speedup. Of course, I assumed that you have access to infinite processing resources.

## Minimize Waiting #

Waiting has at least two drawbacks. First, when a thread waits it cannot make any progress; therefore, your performance goes down. Even worse: if the waiting is busy, the underlying CPU will be fully utilized (In the memory model chapter, I compared the busy waiting of a spinlock with the non-busy waiting of a mutex). Second, the more waiting you have in your program in order to synchronize the threads, the more likely it will become that a bad interleaving of waiting periods causes a deadlock.

## Prefer Immutable Data

A data race is a situation in which at least two threads access a shared variable at the same time. In that case, at least one thread tries to modify the variable; the definition makes it quite obvious. A requirement for getting a data race is mutable data. If you have immutable data, no data race can happen. You only have to guarantee that the immutable data will be initialized in a thread-safe way.

Functional programming languages such as Haskell, having no mutable data, are very suitable for concurrent programming.

## Look for the Right Abstraction

There are various ways to initialize a Singleton in a multithreading environment. You can rely on the standard library using a `lock_guard` or `std::call_once`, rely on the core language using a static variable, or rely on atomics using acquire-release semantic. The acquire-release semantic is by far the most challenging one. It's a big challenge in various aspects: you have to implement it, maintain it, and explain it to your coworkers. In contrast to your effort, the well-known Meyers Singleton is a lot easier to implement and runs faster.

The story with the right abstractions goes on. Instead of implementing a parallel loop for summing up a container, use `std::reduce`. You can parametrise `std::reduce` with a binary callable and the parallel execution policy.

The more you go for the right abstraction, the less likely it will become that you shoot yourself in the foot.

## Use Static Code Analysis Tools

In the chapter on case studies, I introduced CppMem as an interactive tool for exploring the behavior of small code snippets using the C++ memory model. CppMem can help you in two aspects: First, you can verify the correctness of your code; Second, you get a deeper understanding of the memory model and

your code, Second, you get a deeper understanding of the memory model and, therefore, of the multithreading issues in general.

## Use Dynamic Enforcement Tools #

ThreadSanitizer is a data races detector for C/C++; it's also part of Clang 3.2 and GCC 4.8. To use ThreadSanitizer, you have to compile and link your program using the flag `-fsanitize=thread`, or more generally,

```
g++ -std=c++11 dataRace.cpp -fsanitize=thread pthread -g -o dataRace
```

The following program has a data race.

```
// dataRace.cpp

#include <thread>

int main(){

  int globalVar{};

  std::thread t1([&globalVar]{ ++globalVar; });
  std::thread t2([&globalVar]{ ++globalVar; });

  t1.join();
  t2.join();

}
```

`t1` and `t2` access `globalVar` at the same time, and both threads try to modify the `globalVar`. The bottom line is that there is a data race in line 10.