The Contract

This lesson goes over the three levels of contracts in modern C++.

WE'LL COVER THE FOLLOWING

The foundation of multithreading is a *well-defined* memory model. From the reader's perspective, it consists of two aspects. First, there is the enormous complexity of it, which often contradicts our intuition. Second, it helps to get a deeper insight into the multithreading challenges. In the first approach, we want to give you a mental model. That being said, the C++ memory model defines a contract.

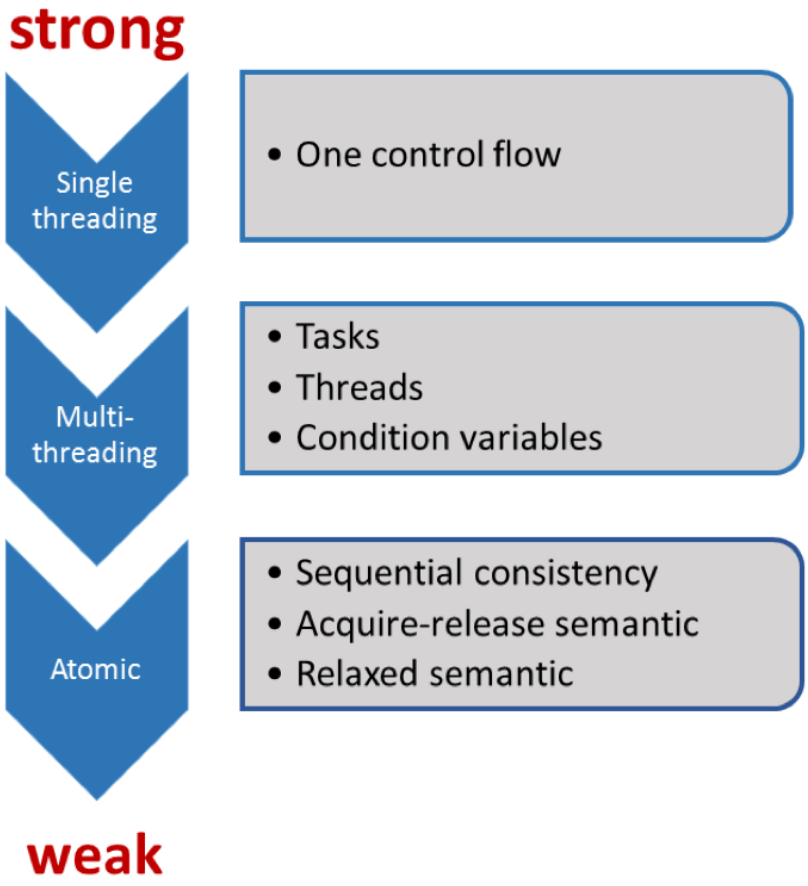
The Contract

This contract exists between the programmer and the system. The system consists of the compiler that generates machine code and the processor that executes the machine code. The system includes the different caches that store the state of the program. The result is a well-defined executable that is fully optimized for the hardware platform. There is not only one contract, but rather a fine-grained set of contracts; i.e., the weaker the rules are that the programmer has to follow, the more potential there is for the system to generate a highly optimized executable.

There is one general rule important to follow: the stronger the contract, the fewer liberties for the system to generate an optimized executable. When the programmer uses a weak contract or memory model, there are too many optimization choices, meaning that the program is only manageable by a

handful of worldwide recognized experts.

Roughly speaking, there are three contract levels in C++11. Let's discuss those three levels.



First Level

Before C++11, there was only one contract. The C++ language specification did not include multithreading or atomics. The system could only address one control flow and, therefore, there were restricted opportunities to optimize the executable. The key point of the system was to guarantee that the observed behavior of the program corresponds to the sequence of the instructions in the source code. Of course, this means that there was no memory model. Instead, there was only the concept of a sequence point.

Sequence points are points in the program, at which the effects of all instructions preceding it must be observable. The start or end of the execution are sequence points. When you invoke a function with two arguments, the C++ standard makes no guarantee as to which arguments will be evaluated first, so the behavior is unspecified. This is due to the fact that the comma operator is not a sequence point. This will not change in C++.

Second Level

With C++11, everything has changed. C++11 is the first standard that can address multiple threads. The C++ memory model was heavily inspired by the Java memory model, and it is the reason for the well-defined behavior of threads. However, the C++ memory model goes a few steps further. The programmer must obey a few rules in order to deal with shared variables and to get a well-defined program. The program is undefined if there is at lease one data race. You must be aware of data races if your threads share mutable data. Tasks are easier to use than threads or condition variables.

Third Level

With atomics, we enter the domain of experts. This will become more evident as we weaken the C++ memory model. We often talk about lock-free programming when we use atomics. Earlier, we discussed the weak and strong rules. Indeed, the sequential consistency is the strong memory model, and the relaxed semantic is the weak memory model.

In the next lesson, let’s take a look at atomics and atomic operations.