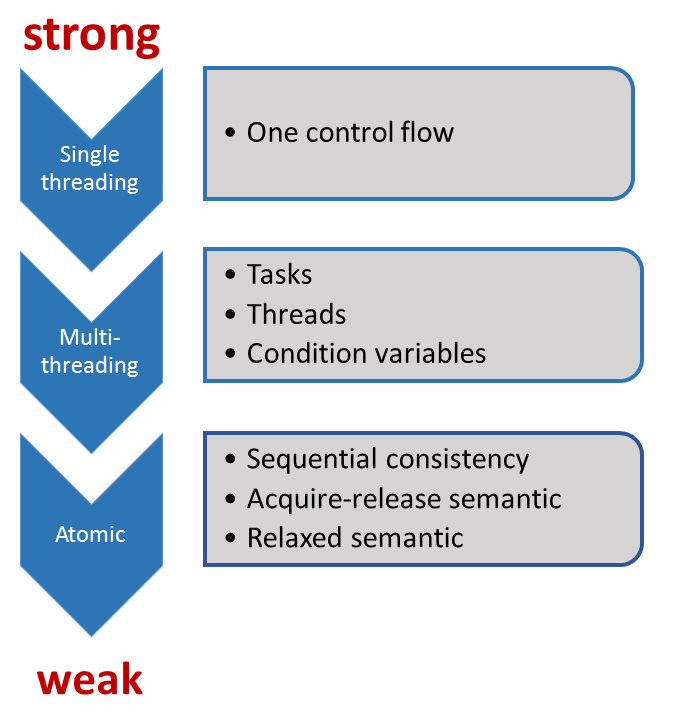
Since C++11, C++ has a memory model. It is the foundation for multithreading. Without it, multithreading is not well defined.

The C++ memory model consists of two aspects. On one hand, there is the enormous complexity of the memory model, which often contradicts our intuition. On the other hand, the memory model helps a lot to get a deeper insight into the multithreading challenges.

In the first approach, the C++ memory model defines a contract. This contract is established between the programmer and the system. The system consists of the compiler, which compiles the program into assembler instructions, the processor, which performs the assembler instructions and the different caches, which stores the state of the program. The contract requires from the programmer to obey certain rules and gives the system the full power to optimise the program as far as no rules are broken. The result is - in the good case - a well-defined program, that is maximal optimised. Precisely spoken, there is not only a single contract, but a fine-grained set of contracts. Or to say it differently. The weaker the rules are the programmer has to follow, the more potential is there for the system to generate a highly optimised executable.

The rule of thumb is quite easy. The stronger the contract, the fewer liberties for the system to generate an optimised executable. Sadly, the other way around will not work. In case the programmer uses an extremely weak contract or memory model, there are a lot of optimisation choices. But the program is only manageable by a few worldwide known experts.

There are three levels of the contract in C++11.



Before C++11 there was only one contract. C++ was not aware of the existence of multithreading or atomics. The system only knows about one control flow and therefore there were only restricted opportunities to optimise the executable. The key point of the system was it, to keep the illusion for the programmer, that the observed behaviour of the program corresponds to the sequence of the instructions in the source code. Of course, there was no memory model. Instead of that, there was the concept of a sequence point. Sequence points are points in the program, at which the effects of all instructions before must be observable. The start or the end of the execution of a function are sequence points. But in case you invoke a function with two arguments, the C++ standard makes no guarantee, which arguments will be evaluated at first. So the behaviour is unspecified. The reason is straightforward. The comma operator is no sequence point. That will not change in C++11.

But with C++ all will change. C++11 is the first time aware of multiple threads. The reason for the well-defined behaviour of threads is the C++ memory model. The C++ memory model is inspired by the Java memory model, but the C++ one goes  - as ever - a few steps further. But that will be a topic of the next posts. So the programmer has to obey to a few rules in dealing with shared variables to get a well-defined program. The program is undefined if there exists at least one data race. As I already mentioned, you have to be aware of data races, if your threads share mutable data. So tasks are a lot easier to use than threads or condition variables.

With atomics, we enter the domain of the experts. This will become more evident, the further we weaken the C++ memory model. Often, we speak about lock-free programming, when we use atomics. I spoke in the posts about the weak and strong rules. Indeed, the sequential consistency is called strong memory model, the relaxed semantic weak memory model.

The meat of the contract

The contract between the programmer and the system consists of three parts:

* **Atomic operations**: Operations, which will be executed without interruption.
* **The partial order of operations**: Sequence of operations, which can not be changed.
* **Visible effects of operations**: Guarantees, when an operation on shared variables will be visible in another thread.

The foundation of the contract are operations on atomics. These operations have two characteristics. They are atomic and they create synchronisation and order constraints on the program execution. These synchronisations and order constraints will often also hold for not atomic operations. At one hand an atomic operation is always atomic, but on the other hand, you can tailor the synchronisations and order constraints to your needs.

Back to the big picture

The more we weaken the memory model, the more our focus will change.

* More optimisation potential for the system
* The number of control flows of the program increases exponential
* Domain for the experts
* Break of the intuition
* Area for micro optimisation

To make multithreading, we should be an expert. In case we want to deal with atomics (sequential consistency), we should open the door to the next expertise level. And you know, what will happen when we talk about the acquire-release or relaxed semantic? We'll go each time one step higher to the next expertise level.

