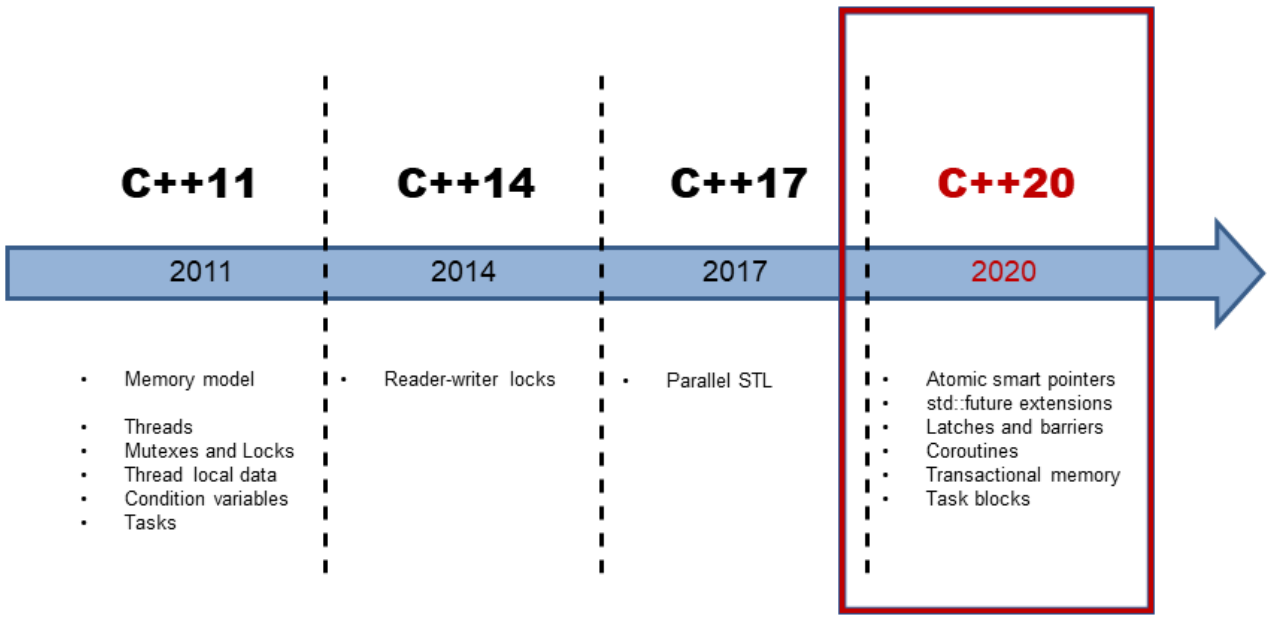
C++20: The Concurrent Future

A short introduction to concurrent future and all the new libraries and techniques which are predicted to be launched with C++20.

WE'LL COVER THE FOLLOWING

- Atomic Smart Pointers
- Extended Futures
- Latches and Barriers
- Coroutines
- Transactional Memory
- Task Blocks



It is difficult to make predictions, especially about the future (Niels Bohr). Therefore, I will make statements about the concurrency features of C++20.

Atomic Smart Pointers

The smart pointers `std::shared_ptr` and `std::weak_ptr` have a conceptional issue in concurrent programs: they share an intrinsically mutable state.

Therefore, they are prone to data races and, thus, lead to undefined behavior. `std::shared_ptr` and `std::weak_ptr` guarantee that the incrementing or decrementing of the reference counter is an atomic operation and the resource will be deleted exactly once, but neither of them can guarantee that the access to its resource is atomic. The new atomic smart pointers `std::atomic_shared_ptr` and `std::atomic_weak_ptr` solve this issue.

Extended Futures

Tasks called promises and futures, introduced in C++11, have a lot to offer. However, they also have a drawback: tasks are not composable into powerful workflows. That limitation will not hold for the extended futures in C++20. Therefore, an extended future becomes ready when its predecessor (`then`) becomes ready, `when_any` one of its predecessors becomes ready, or `when_all` of its predecessors become ready.

Latches and Barriers

C++14 has no semaphores, i.e. the variables used to control access to a limited number of resources. Worry no longer, because C++20 proposes latches and barriers. You can use latches and barriers for waiting at a synchronization point until the counter becomes zero. The difference between latches and barriers is that an `std::latch` can only be used once, while an `std::barrier` and `std::flex_barrier` can be used more than once. In contrast to a `std::barrier`, a `std::flex_barrier` can adjust its counter after each iteration.

Coroutines

Coroutines are functions that can suspend and resume their execution while maintaining their state. Coroutines are often the preferred approach to implement cooperative multitasking in operating systems, event loops, infinite lists, or pipelines.

Transactional Memory

The transactional memory is based on the ideas underlying transactions in database theory. A transaction is an action that provides the first three properties of ACID database transactions: **A**tomicity, **C**onsistency, and **I**solation. The durability that is characteristic for databases will not hold for the proposed transactional memory in C++. The new standard will have transactional memory in two flavors: synchronized blocks and atomic blocks. Both will be executed in total order and behave as if they were protected by a global lock. In contrast to synchronized blocks, atomic blocks cannot execute transaction-unsafe code.

Task Blocks

Task Blocks implement the fork-join paradigm in C++. The following graph illustrates the key idea of a task block: you have a fork phase in which you launch tasks and a join phase in which you synchronize them.