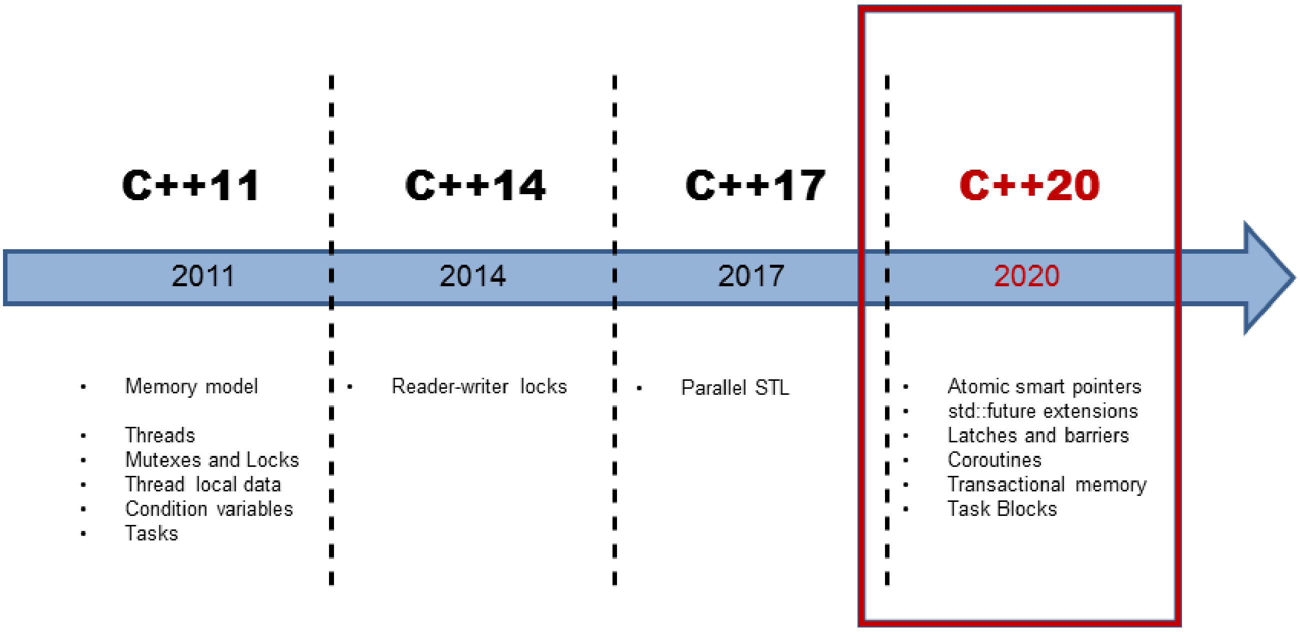
Atomic Smart Pointers

This lesson gives an overview of the atomic smart pointers, predicted to be introduced in C++20.

WE'LL COVER THE FOLLOWING ^

- Atomic Smart Pointers

This chapter is about the future of C++. In this chapter, my intent is not to be as precise as I was in the other chapters in this course. That's for two reasons: First, not all of the presented features will make it into the C++20 standard; second, if a feature makes it into the C++20 standard, the interface of that feature will very likely change. My aim in this chapter is just to give you an idea about the upcoming concurrency features in C++.



Atomic Smart Pointers

A `std::shared_ptr` consists of a control block and its resource. The control block is thread-safe, but the access to the resource is not. This means modifying the reference counter is an atomic operation and you have the guarantee that the resource will be deleted exactly once. These are the guarantees `std::shared_ptr` gives you.

> ### The importance of being thread-safe
>
> Before I start, I want to make a short detour. This detour should only emphasize how important it is that the `std::shared_ptr` has well-defined multithreading semantic. At first glance, use of an `std::shared_ptr` does not appear to be a sensible choice for multithreaded code. It is by definition shared and mutable, and it's the ideal candidate for *data races* and, hence, for undefined behavior. On the other hand, there is a guideline in modern *C++: Don't touch memory*. This means use smart pointers in multithreading programs.

The proposal N4162 for atomic smart pointers directly addresses the deficiencies of the current implementation. The deficiencies boil down to these three points: consistency, correctness, and performance. I will provide an overview of these three points. See the proposal N4162 for details.

- *Consistency*: the atomic operations for `std::shared_ptr` are the only atomic operations for a non-atomic data type.
- *Correctness*: the usage of the global atomic operations is quite error-prone because the right usage is based on discipline. It is easy to forget to use an atomic operation - such as using `ptr = localPtr` instead of `std::atomic_store(&ptr, localPtr)`. The result is undefined behavior because of a data race. If we used an atomic smart pointer instead, the type-system would not allow it.
- *Performance*: the `std::atomic_shared_ptr` and `std::atomic_weak_ptr` have a big advantage compared to the free `atomic_`* functions. The atomic versions are designed for the special use case and can internally have a `std::atomic_flag` as a kind of cheap ***spinlock***. Designing the non-atomic versions of the pointer functions to be thread safe would be overkill if they are used in a single-threaded scenario; they would have a

they are used in a single-threaded scenario, they would have a performance penalty.

For me, the correctness argument is the most important one. Why? The answer lies in the proposal. The proposal presents a thread-safe singly linked list that supports insertion, deletion, and searching of elements. This singly linked list is implemented in a lock-free way.