Atomic Operations on std::shared_ptr

WE'LL COVER THE FOLLOWING

There are specializations for the atomic operations load, store, compare, and exchange for an `std::shared_ptr`. By using the explicit variant, you can even specify the memory model. Here are the free atomic operations for `std::shared_ptr`:

```
std::atomic_is_lock_free(std::shared_ptr)
std::atomic_load(std::shared_ptr)
std::atomic_load_explicit(std::shared_ptr)
std::atomic_store(std::shared_ptr)
std::atomic_store_explicit(std::shared_ptr)
std::atomic_exchange(std::shared_ptr)
std::atomic_exchange_explicit(std::shared_ptr)
std::atomic_compare_exchange_weak(std::shared_ptr)
std::atomic_compare_exchange_strong(std::shared_ptr)
std::atomic_compare_exchange_weak_explicit(std::shared_ptr)
std::atomic_compare_exchange_strong_explicit(std::shared_ptr)
```

For the details, have a look at cppreference.com. Now it is quite easy to modify a shared pointer that is bound by reference in a thread-safe way.

```
std::shared_ptr<int> ptr = std::make_shared<int>(2011);

for (auto i =0;i<10;i++){
   std::thread([&ptr]{
     auto localPtr= std::make_shared<int>(2014);
     std::atomic_store(&ptr, localPtr);
   }).detach();
}
```

The update of the `std::shared_ptr ptr` in the expression `auto localPtr= std::make_shared<int>(2014)` is thread-safe. All is well? NO! Finally, we need atomic smart pointers.

> ### Atomic Smart Pointers
>
> That is not the end of the story for atomic smart pointers. With C++20, there is a high probability that we can expect two new smart pointers: `std::atomic_shared_ptr` and `std::atomic_weak_ptr`. For the impatient reader, here are the details of the upcoming atomic smart pointers.

Atomics and their atomic operations are the basic building blocks for the memory model. They establish synchronization and ordering constraints that hold for both atomics and non-atomics. Let's have a deeper look into the synchronization and ordering constraints.