The COM static store, part 6: Using C++ weak references

devblogs.microsoft.com/oldnewthing/20210215-00

February 15, 2021

Raymond Chen

Last time, we looked at using a COM weak reference to access our singleton quickly. so you can access them all at once. That improves the efficiency of accessing objects in the COM static store, but you can do even better.

Use a C++ weak reference.

The idea here is that you create a `shared_ptr` to your state and put that `shared_ptr` inside an `IInspectable`, and put that `IInspectable` in the COM static store. The lifetime of the data is therefore controlled by the COM static store, and it will be destructed when COM shuts down. Meanwhile, you keep a `weak_ptr` to the state in an easily-accessed global variable. When you want to access the data, you just `lock()` the weak pointer. If the lock fails, then it means that the shared data doesn't exist (either it was never created, or it was destroyed when COM was shut down), so you get to make a new one.

Exercise: What's the scenario where we need to recreate the data after COM was shut down?

```
// C++/WinRT
struct SharedState
{
    int some_value = 0;
    winrt::com_ptr<IStream> stream;
    std::vector<winrt::com_ptr<IStorage>> storages;
};

struct SharedStateHolder :
    winrt::implements<SharedStateHolder,
                      winrt::Windows::Foundation::IInspectable>
{
    std::shared_ptr<SharedState> shared =
       std::make_shared<SharedState>();
};

std::shared_ptr<SharedState>
GetSingletonSharedState()
{
    static std::weak_ptr<SharedState> weak;
    static winrt::slim_mutex lock;
    {
        std::shared_lock guard{ lock };
        if (auto shared = weak.lock()) return shared;
    }

    auto value = winrt::make_self<SharedStateHolder>();

    winrt::slim_lock_guard guard{ lock };
    if (auto shared = weak.lock()) return shared;

    CoreApplication::Properties().Insert(L"SharedState", *value);
    weak = value->shared;
    return value->shared;
}
```

The idea here is that our shared state is stored in a `SharedState` structure, which is kept alive by a `shared_ptr` in the `SharedStateHolder`, which is in turn kept alive by the COM static store. Therefore, it is the COM static store that ultimately controls the lifetime of the shared state.

When we create the shared state, we also keep a weak reference to it in a static variable. That weak reference gives us quick access to the shared state, much faster than getting the `CoreApplication`'s `Properties` and then hunting around inside it. Resolving a C++ weak reference is just chasing a few pointers and doing an atomic increment.

Bonus reading: Advanced STL, part 1: `shared_ptr` by Stephan T. Lavavej.

If the weak pointer fails to resolve, then we need to go make a new shared state object. After entering the exclusive lock, we check again, in case somebody beat us to it.

Assuming we didn't hit the race condition, we put the object into the COM static store and update our weak pointer. (The order here is important in case the insertion operation fails.)

When COM shuts down, it will release all the objects in the COM static store, which will ultimately lead to the destruction of the shared state. The weak pointer, however, continues to point to an expired control block. When the weak pointer is destructed, even that control block gets freed.

The trick here is that the C++ weak pointer is not a COM object and therefore you don't run into the problem of using a COM object after COM has shut down. It's a weak pointer to an object in this same module, so there is no external code involved. (Indeed, there are no virtual methods at all! That's part of what makes C++ weak pointers so fast.)

We can now generalize this pattern into a helper:

```
template<typename D, typename Type>
struct ComSingleton
{
    std::weak_ptr<Type> weak;
    winrt::slim_mutex lock;

    std::shared_ptr<Type> Get()
    {
        {
            std::shared_lock guard{ lock };
            if (auto shared = weak.lock()) return shared;
        }

        struct Holder : winrt::implements<
            Holder, winrt::Windows::Foundation::IInspectable>
        {
            std::shared_ptr<Type> shared =
                std::make_shared<Type>();
        };
        auto value = winrt::make_self<Holder>();

        winrt::slim_lock_guard guard{ lock };
        if (auto shared = weak.lock()) return shared;

        winrt::Windows::ApplicationModel::Core::
        CoreApplication::Properties().Insert(
            static_cast<D*>(this)->name(), *value);
        weak = value->shared;
        return value->shared;
    }

    void Reset() try
    {
        winrt::Windows::ApplicationModel::Core::
        CoreApplication::Properties().Remove(
            static_cast<D*>(this)->name());
    }
    catch (winrt::hresult_out_of_bounds const&) {}
};
```

For each singleton thing you want to create, you create a corresponding `ComSingleton` specialization, providing a type that provides the name under which the item should be recorded.

```
struct SharedState
{
    int some_value = 0;
    winrt::com_ptr<IStream> stream;
    std::vector<winrt::com_ptr<IStorage>> storages;
};

struct SharedStateSingleton :
    ComSingleton<SharedStateSingleton, SharedState>
{
    static constexpr decltype(auto) name()
        { return L"SharedState"; };
};

SharedStateSingleton singleton;

void Something()
{
    auto state = singleton.Get();
}
```

The name of the key in the COM static store is provided in the form of a function because that lets you generate multiple instances of the `SharedState` under keys generated at runtime. And also because string literals aren't valid non-type template parameters (until C++20).

For completeness, I also added a `Reset` method which destroys the singleton.

One thing you may have noticed is that this version creates an extra allocation, since we have both a `shared_ptr` and a `IInspectable`. However, notice that the `IInspectable` exists solely to manage the lifetime of the shared object. Nobody actually accesses the object via the `IInspectable`. In fact, once the `IInspectable` has been put into the COM static store, it is completely forgotten! The memory and code for the `IInspectable` will not be used again until the static store is torn down.

And since the entire purpose is to manage a singleton object, it's not like you're going to be allocating a lot of these little babysitter `IInspectable` objects. There's going to be only one per singleton. The extra memory cost is not likely to be significant, and it leads to the removal of virtual calls from the common code path where you resolve the C++ reference to a live C++ object.

And then if you think about it some more, you realize that the C++ weak reference didn't add any allocations at all, because the version with COM weak references also allocated two objects anyway: One for the `IInspectable` and one for the COM weak reference. All you did was trade one type of weak reference for another. (Assuming you use `make_shared` to put the control block and payload in the same allocation.)

That completes our whirlwind tour of the COM static store. It's a great place to keep your stuff, if your stuff needs to vanish at the same time COM does.

Answer to exercise: The scenario where we need to recreate the data after COM was shut down is when the application shuts down COM, and then starts it back up again. In that case, the COM shutdown will destroy all of the shared state. When COM starts back up, the app can call back into your COM object, and it will then need to create a new set of data.

Raymond Chen

Follow