The COM static store, part 1: Introduction

devblogs.microsoft.com/oldnewthing/20210208-00

February 8, 2021

Raymond Chen

Storing COM pointers in global variables has been a consistent source of problems, because the globals are destructed when the DLL receives it `DLL_PROCESS_DETACH`, which happens after COM has already shut down. The destructor then calls into an object that no longer exists, and that makes people sad.

But what if I told you there's a place to put your global COM pointers, and that place has the property that the global COM pointers will be released as part of COM teardown? In other words, the global COM pointers are released *just in time*: The global COM pointer has its lifetime tied to the lifetime of COM, rather than the lifetime of the process.

That place is informally known as the COM static store. Formally, it's the property set on the `CoreApplication` object. Even though the `CoreApplication` is a Windows Runtime object, it is available in any process that uses COM, not just UWP apps.

The idea is that you move the storage of your global variable out of your data segment and into the COM static store. When you want to update the value, you write it to the COM static store, and when you want to read the value, you read it from the COM static store.

For concreteness, I'll write the code in C++/WinRT, but the same algorithm applies to other projections.

```
Thing GetTheThing()
{
    auto props = CoreApplication::Properties();
    if (auto found = props.TryLookup(L"Thing")) {
        return found.as<Thing>();
    }
    return nullptr;
}

void SetTheThing(Thing const& thing)
{
    auto props = CoreApplication::Properties();
    props.Insert(L"Thing", thing);
}
```

Here, we used the incredibly uncreative key name `"Thing"`. In practice, you should pick a name that is not going to collide with others who are sharing the COM static store. Since the `CoreApplication` is a Windows Runtime object, its initial clients were mostly other Windows Runtime objects, and the convention developed that the key is the fully-qualified Windows Runtime name of the object being added, like `Contoso.Deluxe.Widget`. Of course, if your object is not a Windows Runtime object, then that pattern doesn't apply to you, so come up with some other unique name, like maybe a stringified GUID. For demonstration purposes, I'm just going to use the string `"Thing"`.

Since the COM static store is a per-process store, it is available to any thread, so you would be best served if the objects you put in it are free-threaded,[1] because any thread can retrieve an object from it.[2]

The most common use for the COM static store is providing a place to keep a singleton which is created on demand and remains alive until COM is torn down. The basic idea goes something like this:

```
// Don't use this code yet
Thing GetSingletonThing()
{
    auto props = CoreApplication::Properties();
    if (auto found = props.TryLookup(L"Thing")) {
        return found.as<Thing>();
    }
    auto thing = MakeAThing();
    props.Insert(L"Thing", thing);
    return thing;
}
```

First we look to see if there is a `Thing` already. If so, then we return it. Otherwise, we make a new `Thing`, insert the newly-created `Thing` into the property set, and return it.

That's the basic idea. The rest is filling in the holes. We'll start doing that next time.

Bonus chatter: Just for example, here's the translation of `GetTheThing` into C++/CX:

```
Thing^ GetTheThing()
{
    auto props = CoreApplication::Properties;
    if (auto found = props->TryLookup(L"Thing")) {
        return safe_cast<Thing^>(found);
    }
    return nullptr;
}
```

It's a fair direct line-for-line translation. Translating into C++/WRL is much more work because WRL works at the ABI layer.

```
HRESULT GetTheThing(Thing** result)
{
  *result = nullptr;

  // auto props = CoreApplication::Properties;
  WRL::ComPtr<ICoreApplication> app;
  RETURN_IF_FAILED(RoGetActivationFactory(
    WRL::Wrappers::HStringReference(
      RuntimeClass_Windows_ApplicationModel_Core_CoreApplication).Get(),
      IID_PPV_ARGS(&app)));

  WRL::ComPtr<IPropertySet> props;
  RETURN_IF_FAILED(app->get_Properties(&props));

  // auto found = props->TryLookup(L"Thing");
  WRL::ComPtr<IMap<HSTRING, IInspectable*>> map;
  RETURN_IF_FAILED(props.As(&map));

  WRL::ComPtr<IInspectable> found;
  HRESULT hr = map->Lookup(
    WRL::Wrappers::HStringReference(L"Thing").Get(),
    &found);
  RETURN_HR_IF(hr, FAILED(hr) && hr != E_BOUNDS);

  // if (found) return safe_cast<Thing^>(found);
  // return nullptr;
  if (hr == S_OK) {
    RETURN_IF_FAILED(found.CopyTo(result));
  }
  return S_OK;
}
```

I don't anticipate giving many examples in WRL in this series of articles. It's far too verbose.

That's the basic idea behind the COM static store, but of course you have to be careful how you use it. We'll dig in next time.

[1] You can create a free-threaded wrapper for any COM object with `RoGetAgileReference`. WRL, C++/CX, and C++/WinRT all provide helpers for managing agile references.

Projection	Wrapper
WRL	`Microsoft::WRL::AgileRef`
C++/CX	`Platform::Agile<T>`
C++/WinRT	`winrt::agile_ref`

[2] If your program is single-threaded,[3] then I guess you don’t have to worry about the “wrong” thread retrieving the value, since you have only one thread anyway.

[3] Looking at you, JavaScript.

Raymond Chen

Follow