C++/WinRT implementation extension points: abi_guard, abi_enter, abi_exit, and final_release

devblogs.microsoft.com/oldnewthing/20191018-00

October 18, 2019

Raymond Chen

C++/WinRT provides a few extension points for implementations to customize default behavior of inspectable objects.

When the last reference to an object is released, the object is destroyed. However, you may need to do some special cleanup while the object is still alive. The classic example of this is COM objects suffering double-destruction due to a temporary refcount, and the standard solution is to artificially bump the reference count during destruction.

The C++/WinRT library takes the standard solution and goes a step further: An implementation class can optionally implement a `final_release` method. If you provide such a method, then instead of destructing the object immediately upon the release of the final client reference, the C++/WinRT library calls your `final_release` method with the last remaining reference to the object, in the form of a `unique_ptr`. The object is still alive (it has not started destructing), so you can do normal things with it, like pass it to another method that may temporarily bump its reference count. You can even `co_await` in your `final_release` if you need to do some asynchronous work before letting the object finally disappear.

Normally, the object will destruct when the `unique_ptr` destructs, but you can hasten its death by calling `unique_ptr.reset()`, or you can postpone the inevitable by saving the `unique_ptr` somewhere. You can read Kenny Kerr's discussion of `final_release` for more details.

The less commonly-used extension point is the `abi_guard` and its close friends `abi_enter` and `abi_exit`.

If your implementation defines a method named `abi_enter`, then it will be called at the entry to every projected interface method (not counting the methods of `IInspectable`). Similarly, if you define a method named `abi_exit`, it will be called at the exit from every such method, but will not be called if `abi_enter` throws an exception. (It will be called if an exception is thrown by the method itself.)

The calls to `abi_enter` and `abi_exit` are made with no parameters, and the return value is discarded.

You might use `abi_enter` to, say, throw an `invalid_state_error` exception if a client tries to use an object after it has been put into an unusable state, say, after a `ShutDown` or `Disconnect` method. The C++/WinRT iterator classes use this feature to throw a `invalid_state_error` exception in the `abi_enter` method if the underlying collection has changed.

If the simple `abi_enter` and `abi_exit` methods aren't fancy enough for you, you can define a nested class named `abi_guard`, in which case an instance of the `abi_guard` will be created on entry to each non-`IInspectable` projected interface method with a reference to the object as its constructor parameter. The `abi_guard` is destructed on exit from the method. You can put whatever extra state you like into the `abi_guard` class.

Basically, the deal is that the default `abi_guard` calls `abi_enter` at construction and calls `abi_exit` at destruction. And the default `abi_enter` methods do nothing. You can therefore plug in either at the `abi_enter` / `abi_exit` level, or at the `abi_guard` level.

Note that these guards are used only if you invoke the methods via the projected interface. If you invoke the methods directly on the implementation object, then those calls go straight to the implementation without any guards.

```
struct Thing : ThingT<Thing, IClosable>
{
  void abi_enter();
  void abi_exit();

  void Close();
};

void example1()
{
  auto thing = make<Thing>();
  thing.Close(); // calls abi_enter and abi_exit
}

void example2()
{
  auto thing = make_self<Thing>();
  thing->Close(); // does not call abi_enter or abi_exit
}
```

Note also that the guards are used only for the duration of the method call. If the method is a coroutine, the guard applies only until the `IAsyncXxx` is returned, not until the coroutine completes.

```
IAsyncAction CloseAsync()
{
  // guard is active here
  DoSomething();

  // guard becomes inactive once co_await starts,
  // at which point CloseAsync returns an IAsyncAction.
  co_await DoSomethingElseAsync();

  // guard is not active here
}
```

Guards are useful for specific cases like the "object that is no longer usable", but their applicability in general is somewhat limited because they don't know what method is being invoked. So you can't do things like "If the object is not connected, then reject all method calls except for `Connect`."

Raymond Chen

Follow