# The C++/WinRT “capture” function helps you interoperate with the COM ABI world

devblogs.microsoft.com/oldnewthing/20200917-00

September 17, 2020

Raymond Chen

C++/WinRT is a freestanding library that is not dependent upon any Windows operating system header files. You are welcome to stay completely in the C++/WinRT world, but it’s not uncommon to have to interact with objects outside that world from time to time.

It is a common pattern for COM methods to return objects through a pair of parameters `REFIID` and `void**`. The first parameter describes how you would like to receive the object, and the second parameter is where the object is returned. (Some functions which don’t follow this pattern have come to regret the decision.)

If you are calling a COM ABI method that follows this pattern, you can put the result into a C++/WinRT smart pointer object with the help of the `capture` function.

There are actually four flavors of `capture`, and they fall neatly into a table.

| | Functor | Method pointer |
|---|---|---|
| **Instance method** | `winrt::com_ptr<I> p; p.capture(Functor, args...);` | `winrt::com_ptr<I> p; p.capture(q, &I2::Method, args...);` |
| **Free function** | `auto p = winrt::capture<I>(Functor, args...);` | `auto p = winrt::capture<I>(q, &I2::Method, args...);` |

In all cases, `capture` throws a `winrt::hresult_error` exception if the call fails.

Today, we’ll look at the first column: The functor.

*Functor* is C++-speak for “anything you can call as if it were a function.” The functor could be an actual function, like `CoGetObjectContext`, or it could be something that has an overloaded `operator()`, such as a lambda.

After the functor comes an optional list of additional parameters.

In the functor form, `capture` invokes the functor by passing the additional parameters, followed by a `REFIID` and `void**`.

The functor is expected to return in the `void**` parameter a pointer to object whose type is specified by the `REFIID` parameter, and that pointer shall have it reference count incremented by one, indicating ownership. Fortunately, this convention is standard in COM, so you're already in good shape on that front.

Here's an example of the most basic case:

```
// Capture into an existing variable.
winrt::com_ptr<IContextCallback> context;
context.capture(CoGetObjectContext);

// Create a variable and capture in one step.
auto p = winrt::capture<IContextCallback>(CoGetObjectContext);
```

This calls the `CoGetObjectContext` function and puts the result into an existing variable `p` or returns the result directly, which we place into a new variable `p`.

If you pass additional parameters, those are passed to the functor as well, and the `REFIID` and `void**` parameters go at the end.

```
// Capture into existing object.
winrt::com_ptr<IWidget> widget;
widget.capture(CoUnmarshalInterface, stream.get());

// Capture into new object.
auto widget = winrt::capture<IWidget>(CoUnmarshalInterface, stream.get());
```

This will perform a

```
CoUnmarshalInterface(stream.get(), riid, ppv)
```

where the `riid` and `ppv` receive the object.

If you use `capture` as a free function, you aren't required to store the result into a variable. You could just use the result right away:

```
HRESULT hr = winrt::capture<IWidget>(CoUnmarshalInterface, stream.get())->Toggle();
if (FAILED(hr)) throw_hresult(hr);

// Or equivalently
winrt::check_hresult(winrt::capture<IWidget>(CoUnmarshalInterface, stream.get())-
>Toggle());
```

You can pass a lambda or other callable object if you want to do something that isn't expressible as a simple function. For example, the `OleCreate` function does not place both the `riid` and `void**` parameters at the end of the parameter list. It doesn't fit the pattern

required by `capture`, but you can fix that with a lambda that rearranges the parameters.[2]

```
o.capture(
    [](auto&& a, auto&& b, auto&& c, auto&& d, auto&& e, REFIID riid, void** ppv)
    { return OleCreate(a, riid, b, c, d, e, ppv); },
    rclsid, renderopt, pFormatEtc, pClientSite, pStg);
```

If you're going to be doing this a lot, you may want to factor the lambda into a helper function.

```
inline HRESULT CapturableOleCreate(
  IN REFCLSID        rclsid,
  IN DWORD           renderopt,
  IN LPFORMATETC     pFormatEtc,
  IN LPOLECLIENTSITE pClientSite,
  IN LPSTORAGE       pStg,
  IN REFIID          riid,
  OUT LPVOID         *ppvObj)
{
    return OleCreate(rclsid, riid, renderopt, pFormatEtc, pClientSite, pStg, riid,
ppv);
}

o.capture(CapturableOleCreate, rclsid, renderopt, pFormatEtc, pClientSite, pStg);
```

After all this discussion, the second column is going to sound anticlimactic.

If you want to obtain the object from a method call on an existing object, you can use the method pointer version of the `capture` function. For example, suppose you have a `IGlobalInterfaceTable` and you want to call `GetInterfaceFromGlobal` to obtain an object from the global interface table.

```
auto git = winrt::create_instance<
    IGlobalInterfaceTable>(CLSID_StdGlobalInterfaceTable);

// Capture into existing object.
winrt::com_ptr<IWidget> widget;
widget.capture(git, &IGlobalInterfaceTable::GetInterfaceFromGlobal, dwCookie);

// Capture into new object.
auto widget = winrt::capture<IWidget>(
    git, &IGlobalInterfaceTable::GetInterfaceFromGlobal, dwCookie);
```

**Bonus chatter**: C++/WinRT comes with a premade capture wrapper for `CoCreateInstance`:

```
template <typename Interface>
auto create_instance(guid const& clsid, uint32_t context = 0x1
/*CLSCTX_INPROC_SERVER*/, void* outer = nullptr);
{
    return capture(WINRT_IMPL_CoCreateInstance[1], clsid, outer, context);
}
```

The default context is “in-process server”, and there is no aggregation by default. You can use it like this:

```
auto shellWindows = winrt::create_instance<IShellWindows>(CLSID_ShellWindows,
CLSCTX_ALL);
```

**Bonus bonus chatter**: The definition of `capture` is quite simple:

```
template <typename T>
struct com_ptr
{
    ...

    template <typename F, typename...Args>
    void capture(F function, Args&&...args)
    {
        check_hresult(function(args..., guid_of<T>(), put_void()));
    }

    template <typename O, typename M, typename...Args>
    void capture(com_ptr<O> const& object, M method, Args&&...args)
    {
        check_hresult((object.get()->*(method))(args..., guid_of<T>(), put_void()));
    }
}

template <typename T, typename F, typename...Args>
auto capture(F function, Args&&...args)
{
    impl::com_ref<T> result{ nullptr };
    check_hresult(function(args..., guid_of<T>(), reinterpret_cast<void**>
(put_abi(result))));
    return result;
}

template <typename T, typename O, typename M, typename...Args>
auto capture(com_ptr<O> const& object, M method, Args&&...args)
{
    impl::com_ref<T> result{ nullptr };
    check_hresult((object.get()->*(method))(args..., guid_of<T>(),
reinterpret_cast<void**>(put_abi(result))));
    return result;
}
```

This entire series is based on me reverse-engineering these four functions.

[1] The C++/WinRT library is freestanding, so it contains its own private redeclaration of the `CoCreateInstance` function. This redeclared version is for internal use only. Don't use it from your own code. Basically, anything named `WINRT_IMPL` or in the `winrt::impl` namespace is reserved for internal use.

[2] I guess you could also have captured the parameters into the lambda:

```
o.capture(
    [&](REFIID riid, void** ppv)
    { return OleCreate(rclsid, riid, renderopt, pFormatEtc, pClientSite, pStg, ppv);
});
```

This could be handy if you are creating the same object many times with the same parameters. For example, if you were creating five objects from the stream, you could do this:

```
auto create = [&](REFIID riid, void** ppv)
    { return OleCreate(rclsid, riid, renderopt, pFormatEtc, pClientSite, pStg, ppv);
});

o1.capture(create);
o2.capture(create);
o3.capture(create);
o4.capture(create);
o5.capture(create);
```

Note that the lambda is passed *by value* to `capture`, so a copy is made each time you use it. This means that your lambda probably shouldn't have any mutable state, because the mutations are made to a copy of the original. It also means that your lambda probably shouldn't capture objects with nontrivial copy constructors or destructors, because they will be invoked each time you use it.

Raymond Chen

**Follow**