On the various ways of creating Windows Runtime delegates in C++/WinRT and C++/CX

devblogs.microsoft.com/oldnewthing/20230726-00

July 26, 2023

Raymond Chen

Windows Runtime delegates are a language-independent way of representing callbacks. We'll start with C++/WinRT. Let's say that `DelegateType` is a delegate type.

```
template<typename L>
DelegateType(L handler) :
    delegate_base(make(std::forward<L>(handler)))
{}

template<typename F>
DelegateType(F* handler) :
    delegate_base([=](auto&&... args)
    { return handler(args...); })
{}

template<typename O, typename M>
DelegateType(O* o, M method) :
    delegate_base([=](auto&&... args)
    { return (o->*method)(args...); })
{}

template<typename O, typename M>
DelegateType(com_ptr<O>&& o, M method) :
    delegate_base([=](auto&&... args)
    { return (o->*method)(args...); })
{}

template<typename O, typename M>
DelegateType(weak_ref<O>&& o, M method) :
    delegate_base([o = std::move(o), method](auto&&... args)
    { if (auto s = o.get()) (s->*method)(args...); })
{}
```

In words:

- Construct from a lambda. The delegate parameters are passed to the lambda.
- Construct from a function pointer. The delegate parameters are passed to the function.

- Construct from a raw pointer and a pointer to member function. The member function is invoked on the object with the delegate parameters.
- Construct from a `com_ptr` and a pointer to member function. The member function is invoked on the object with the delegate parameters.
- Construct from a `weak_ptr` and a pointer to member function. When the delegate is invoked, try to promote the weak pointer to strong, and if successful, the member function is invoked on the object with the delegate parameters.

All C++/WinRT delegates are deemed free-threaded. The delegate runs on whatever thread the invoker chooses.

C++/WinRT provides three different "pointer to member function" flavors.

- Raw pointer: It is your responsibility to ensure that the pointed-to object is not destroyed before the delegate is invoked.
- `com_ptr`: This retains a strong reference to the object, so be mindful of potential circular references.
- `weak_ref`: Best of both worlds.

I hope you're not still using C++/CX, but if you are, here are the ways of constructing a C++/CX delegate.

```
template<typename L>
DelegateType(L handler, CallbackContext context = CallbackContext::Any);

template<typename O>
DelegateType(O^ o, M method, CallbackContext context = CallbackContext::Any,
    bool strong = false);
```

In words:

- Construct from anything that you can use `operator()` with. This includes lambdas, `std::function`, and function pointers. The callable is invoked with the delegate parameters.
- Construct from a ref pointer to an object and a pointer to member function. The member function is invoked on the object with the delegate parameters.

The `context` parameter lets you customize the COM apartment context in which the callback is made. It defaults to `Any`, which means that the delegate is free-threaded and runs on whatever thread the invoker chooses. If you specify `Same`, then the delegate is non-agile, and the invoke will be marshalled into the apartment in which in which it was created.

In the case where you pass a pointer to member function, you also have the option of choosing whether the `o` is captured by weak reference (default) or strong reference (`strong = true`).

There is no option for passing a raw pointer. If you want the handler to be a plain C++ object, you can use a custom lambda.

Here is a table.

Scenario	C++/WinRT	C++/CX

Assumptions	`// IDL`
	`runtimeclass S { /* ... */ } /* sender */`
	`runtimeclass EA { /* ... */ } /* event args */`
	
	`delegate void D(S sender, EA e);`
	
	`// Code-behind`
	`/* Plain C++ class */`
	`struct C {`
	`  void OnEvent(S const&, EA const&);`
	`  void CreateDelegate();`
	`};`
	
	`/* C++/WinRT class */`
	`struct RC : winrt::implements<RC, ...>`
	`{`
	`  void OnEvent(S const&, EA const&);`
	`  void CreateDelegate();`
	`};`
	
	`auto lambda = [/* captures */]`
	`  (auto&& s, auto&& e)`
	`  {`
	`  /* ... */`
	`  };`
	
	`std::function<void(S, EA)> sf;`
	
	`void f(S const&, EA const&);`
		`ref class S { /* ... */ }; /* sender */`
		`ref class EA { /* ... */ }; /* event args */`
		
		`delegate void D(S^ sender, EA^ e);`
		
		`/* Plain C++ class */`
		`struct C {`
		`  void OnEvent(S^, EventArgs^);`
		`  void CreateDelegate();`
		`};`
		
		`/* ref class */`
		`ref class RC`
		`{`
		`  void OnEvent(S^, EventArgs^);`
		`  void CreateDelegate();`
		`};`
		
		`auto lambda = [/* captures */]`
		`  (S^ sender, EA^ e)`
		`  {`
		`  /* ... */`
		`  };`
		
		`std::function<void(S^, EA^)> sf;`
		
		`void f(Sender^ sender, EventArgs^ e);`

Lambda	`D(lambda)`
	`/* always agile */`
	(Lambda must be movable)
		`ref new D(lambda) // agile`
		`ref new D(lambda, Same) // non-agile`
		(Lambda must be copyable)

`std::function`	`D(sf)` `/* always agile */`	`ref new D(sf) // agile` `ref new D(sf, Same) // non-agile`

Free function pointer	`D(f)` `/* always agile */`	`ref new D(f) // agile` `ref new D(f, Same) // non-agile`

Raw pointer + method	`void C::CreateDelegate()`
	`{`
	`  auto d = D(this, &C::OnEvent);`
	`  /* always agile */`
	`}`
		Not available (see below)

Strong pointer + method	`void RC::CreateDelegate() {`
	`  auto d = D(get_strong(), &C::OnEvent);`
	`  /* always agile */`
	`}`
		`void RC::CreateDelegate() {`
		`  auto d = D(this, &C::OnEvent, Any, true); // agile`
		`  auto d = D(this, &C::OnEvent, Same, true); // non-agile`
		`}`

Weak pointer + method	`void RC::CreateDelegate() {`
	`  auto d = D(get_weak(), &C::OnEvent);`
	`  /* always agile */`
	`}`
		`void RC::CreateDelegate() {`
		`  auto d = D(this, &C::OnEvent); // agile`
		`  auto d = D(this, &C::OnEvent, Same); // non-agile`
		`}`

If weak pointer fails to resolve	Ignore	Throw `DisconnectedException`

Neither C++/WinRT delegates nor C++/CX delegates have native support for C++ weak and shared pointers, and C++/CX delegates lack support for raw pointers. Fortunately, you can make up any missing features by using a lambda.

```
// C++/WinRT

struct SC : std::enable_shared_from_this<SC>
{
  void OnEvent(S const&, EA const&);
  void CreateDelegate() {
    // with weak_ptr
    auto d = D([weak = weak_from_this()]
               (auto&& s, auto&& e) {
      if (auto shared = weak.lock()) shared->OnEvent(s, e);
    });

    // with shared_ptr
    auto d = D([shared = shared_from_this()]
               (auto&& s, auto&& e) {
      shared->OnEvent(s, e);
    });
  }
};

// C++/CX

struct SC : std::enable_shared_from_this<SC>
{
  void OnEvent(S^, EA^);
  void CreateDelegate() {
    // with weak_ptr
    auto d = D([weak = weak_from_this()]
               (S^ s, EA^ e) {
      if (auto shared = weak.lock()) shared->OnEvent(s, e);
    });

    // with shared_ptr
    auto d = D([shared = shared_from_this()]
               (S^ s, EA^ e) {
      shared->OnEvent(s, e);
    });

    // with raw pointer
    auto d = D([this]
               (S^ s, EA^ e) {
      this->OnEvent(s, e);
    });
  }
};
```