Programming with Native Languages in the 21st Century

Track: C++/CX and WinRT

Overview

In this track, you will learn about C++/CX, Microsoft's latest extensions to standard C++, and WinRT, the new programmatic interface for the next generation of Windows operating systems. You do not need to be an expert in C++ to complete this track, but some up-front familiarity with C++ will certainly grease the machinery.

Project

The source code we will be working with consists of examples that highlight various features of C++/CX and WinRT and also includes unit tests developed with the new C++ unit test framework shipping with Visual Studio 2012.

Preparation

1. You must have some version of Microsoft Visual Studio 2012 installed and running on a Windows 8 system in order to compile the example code. The Visual Studio 2012 Release Candidate and Windows 8 Release Preview should be sufficient. If you do not have access to that compiler, you can still follow along with the exercises and view the source code.

2. Download the workshop code at: https://github.com/michaelbprice/21st_century_native_programming

Exercises

Exercise 1 - What is C++/CX and WinRT

C++/CX is a Microsoft-specific vendor extension to the C++ standard. Among many of the things it adds are: externally managed types (not to be confused with "managed types"), interfaces, generics, partial classes, generated metadata, properties, delegates, and events.

The Windows Runtime (WinRT) is application programming interface (API) for the newest generation of Windows operating systems (Windows 8 / Windows Server 2012) and the underlying infrastructure on which the API is built. WinRT is fundamentally an evolution of Microsoft's Component Object Model (COM) that removes much of the complexity of using COM by using language projection onto the supported programming languages (C++/CX, C#, VB.NET, JavaScript).

The C++ language projection of WinRT relies on the new features in C++/CX. Microsoft representatives have repeatedly said that most of your C++ code should still be written to conform with the C++ standard, and that only a very thin layer that interacts with the operating system should be written with the C++/CX extension (although I've seen evidence that they are violating this principle themselves).

For language purists, there is an WinRT alternative to C++/CX called the Windows Runtime Library (WRL). The WRL is a template-based approach to interacting with WinRT (think the old ATL CComPtr) and is used primarily within Microsoft to achieve results that are not yet exposed via the C++/CX language projection. Most non-Microsoft programmers will never need to touch the WRL.

Step 1

In Visual Studio 2012, load the CppCx.sln solution file. It contains all of the example code as well as a unit test project that consumes the created WinRT types. Compile the solution.

Step 2

Visual Studio 2012 has introduced a new C++ unit testing framework that we are using to demonstrate the consumption of WinRT types in this part of the lab. Explore the unit tests (particularly the `Assert` class), compile, and run the tests using the integrated Test Explorer window.

Exercise 2 - Reference Types vs Value Types

There are two fundamental kinds of type in C++/CX, reference types and value types.

Value types are simple *plain-old-data (POD)* types and are very restricted. Value types

- have pass-by-copy behavior by default
- can only contain public member fields (no properties or methods)
- cannot contain standard C++ types (excluding fundamental types)
- cannot contain C++/CX reference types (except Platform::String)

Reference types are rich types. An instantiated object of reference type will remain exist until the last reference (^) to the object goes out of scope (in WinRT they are reference-counted shared objects). Reference types

- have (effectively) pass-by-reference behavior by default
- cannot have any standard C++ types emitted to metadata (see Exercise 3)
- are single implementation-inheritance and multiple interface-inheritance
- can be instantiated on the heap or the stack (just syntactic sugar)
- must be declared in a **`namespace`** (may be bugs here…)
- may be a **`template`**, but may not be a **`generic`** (it can implement a generic)
- are allocated using the **`ref new`** syntax

Step 1

Explore **`MyValueType`** and the corresponding unit tests.

Step 2

Explore **`MyReferenceType`** and the corresponding unit tests.

Exercise 3 - Namespace and Accessibility

Namespaces in C++/CX do more than their standard counterparts. In standard C++, namespaces simply partition the global scope in order to avoid naming collisions and to increase readability. In C++/CX, namespaces are also control the generation of WinRT metadata, which is critical to WinRT's ability to share a foreign type system between the different supporting languages.

Within a **`namespace`**, reference types are either **`public`** or **`private`** (default). A reference class marked **`private`** does not emit any metadata whatsoever and should not normally occur. A reference class marked **`public`** will emit metadata and is subject to further restrictions.

A public reference type can contain six distinct accessibility levels (as opposed to three for standard C++ types)

- **`public`** - emitted to public metadata, public C++ accessibility
- **`protected`** - emitted to protected metadata, protected C++ accessibility
- **`private`** - **not** emitted to metadata, private C++ accessibility (default)
- **`public protected`** - emitted to protected metadata, public C++ accessibility
- **`protected private`** - **not** emitted to metadata, protected C++ accessibility
- **`internal`** - **not** emitted to metadata, public C++ accessibility

Any accessibility level that will emit metadata cannot contain standard C++ types (except fundamental types), in other words, standard C++ types can only appear in **`private`**, **`protected private`**, or **`internal`** sections.

Step 1

Explore the **`MyAccessibility`** example..

Step 2

Notice that the unit test source files never mention the header files in the **CppCx** project. Instead, there is a project reference that creates a dependency on **CppCx.winmd**. This is the WinRT metadata file generated by the **CppCx** project. Locate the **CppCx.winmd** file (look in the top-level build folder).

Step 3

Launch **ildasm.exe** (TOOLS → ILDasm). You will probably receive an error, just right click the **ildasm** icon in the taskbar and pin it. Close **ildasm** and launch it again via the taskbar. From **ildasm**, open the **CppCx.winmd** file we located in Step 2. Browse through the metadata. Feel free to revisit this step often, just be sure to close the .winmd file in **ildasm** before attempting a build.

Exercise 4 - Inheritance and Interfaces

C++/CX types support *single-implementation-inheritance*, meaning a type can only have a single concrete type as a base type. It also supports *multiple-interface-inheritance*, meaning a class can have any number of interfaces as base types. This is very similar to both C# and Java.

Furthermore, the only supported inheritance accessibility modifier on C++/CX base types is **public**, and it is the default.

```
class A { };
class B : A { }; // C++ defaults to private inheritance

ref class Y { };
ref class Z : Y { }; // C++/CX defaults to public inheritance
```

I recommend that you always explicitly declare the inheritance accessibility to reduce confusion that others (including the future-you) will experience were they to be missing.

```
class A { };
class B : private A { }; // But of course, private!

ref class Y { };
ref class Z : public Y { }; // I wouldn't have it any other way!
```

Interfaces in C++/CX can be declared as public or private, and this controls whether they are emitted to metadata. You may declare methods, properties, and events within an interface, and all members are inherently public and virtual (no need to mark them as such). You may declare properties as read-write, read-only, or write-only (see Exercise 5). Interfaces may also derive from other interfaces, and there is no need to restate the inherited members in the interface declaration.

Step 1

Explore the **MyInterfaces** example and the corresponding unit tests.

Exercise 5 - Properties

Properties are intended to resolve some of the confusion surrounding the idea of encapsulation in object-oriented languages. The problem is more explicitly laid out in the lab ***Modern C++, Exercise 4***, and I encourage you to go read that if you have not already.

Properties solve this problem by providing simple accessor methods that are backed by a hidden private member field. If accessibility needs to be something other than simple read-write (simple read, checked write for instance), the property syntax allows the programmer to easily state their intentions.

Furthermore, there are indexed properties that allow consumers to write

```
int x = MyObject.Element[1, 2]; // instead of

int x = MyObject.Element[1][2]; // as this allows access
                                // to an entire row
```

Step 1

Explore the **`MyProperties`** example and corresponding unit tests.

Exercise 6 - Partial Classes

In C++, one of the fundamental restrictions on the language is that type declarations must be completed in a single block. After a declaration block has been closed, the type cannot have any other data. This is so that parts of code that must know the size of the type can reasonably know that detail. However, this limitation has historically proven to be a major obstacle in automated code-generation tools that take some other representation as input and produce the corresponding C++ code.

For WinRT development, Microsoft wished to allow C++ code to hook into the *extensible application markup langauge (XAML)*, which is such a tool. In order to resolve this, they've introduced partial classes to C++ (similar to partial classes in C#, which were introduced for the same reasons).

The modifier **`partial`** designates that the following type information is just a fragment of a type. The information in that fragment is then available to the compiler. Once the compiler sees a declaration of the same type that is not marked as **`partial`**, it closes out the type declaration and considers it complete.

This functionality is primarily for tools, and you should not expect to create partial types yourself, although, you should be aware of their existence and how they affect your type.

Step 1

Explore the **`MyPartialType`** examples.

Exercise 7 - Delegates and Events

Events are a mechanism that allow the consumers of a type to register a callback function, or **`delegate`**, that will be called when a particular event or state change happens to said type. A delegate is very much like a function pointer type alias, and the type of an **`event`** is always a **`delegate`** type. There's not much more to say about them, other than using lambdas as delegates could result in circular references unless you are very careful.

Step 1

Explore the **`MyDelegateAndEvent`** example and corresponding unit tests.

Exercise 8 - Type Casting and Exceptions

Traditional COM (technically WinRT as well) uses a Windows opaque type called an **`HRESULT`** as the return type of methods to indicate whether a function call was successful or had some type of error. All of the language projects that support WinRT transform these **`HRESULT`** return values into exceptions if there was a failure and grab the "correct" parameter to use as the return value on success.

`Platform::COMException` is the exception type that all WinRT exceptions inherit from (or in the case of a custom error **`HRESULT`**, just a **`COMException^`** will be thrown).

All of the standard C++ type cast operators are supported, and C++/CX adds one more, **`safe_cast<T>`**. The **`safe_cast<T>`** operator will throw an **`InvalidCastException^`** if the cast was invalid (as opposed to **`dynamic_cast<T>`** which returns **`nullptr`**).

Step 1

Explore the **`MyCasting`** example and corresponding unit tests.

Wrapping Up

There are so many exciting features of the newest C++ standard, that it would quite literally take several semesters-worth of college courses to cover it all. I hope that you continue on the path of learning as much of it as you can. It is truly an exciting time to write applications with C++, and I'm looking forward to coming years and all of the great code that will be written utilizing all of these great advancement forward.