Inside C++/WinRT: How does C++/WinRT represent ABI types?

devblogs.microsoft.com/oldnewthing/20200924-00

September 24, 2020

Raymond Chen

C++/WinRT offers a high-level interface to the the low-level Windows Runtime ABI (application binary interface). It does this without any dependency on the Windows header files, which means that it needs some way to talk about the ABI types without actually using the ABI types. How does this work?

C++/WinRT sets up a collection of types which run parallel to the ABI types defined in the system header files. The types are not the same, but they are equivalent at the ABI level, meaning that they have identical binary representations.

When you work in C++/WinRT, there are three (sometimes four) versions of every type, listed here in decreasing order of popularity:

- C++/WinRT projected types.
- C++/WinRT implementation types.
- C++/WinRT ABI-equivalent types.
- System-defined ABI types. (Not used by C++/WinRT.)

In practice, you will be spending nearly all of your time with C++/WinRT projected types. If you are implementing C++/WinRT classes, then you will also have to deal with C++/WinRT implementation types.

But you will rarely have to deal with C++/WinRT ABI-equivalent types or the underlying system-defined ABI types. Those come into play only when you are interoperating at the ABI layer, and that's typically something you let the C++/WinRT library do for you.

But I'm going to discuss it anyway, because you may on occasion find yourself having to work at the ABI layer.

Here's how it works for scalar types:

System	C++/WinRT

ABI	ABI	Projection
`BYTE`	`uint8_t`	
`INT16`	`int16_t`	
`UINT16`	`uint16_t`	
`INT32`	`int32_t`	
`UINT32`	`uint32_t`	
`INT64`	`int64_t`	
`UINT64`	`uint64_t`	
`FLOAT`	`float`	
`DOUBLE`	`double`	
`boolean`	`bool`	
`WCHAR`	`char16_t`	
`GUID`	`winrt::guid`	
`enum`	`int32_t` `uint32_t`	`enum`
`HSTRING`	`void*`	`winrt::hstring`
`HRESULT`	`int32_t`	`winrt::hresult`

For enumerations, the C++/WinRT ABI type is `int32_t`, unless the enumeration is a flags enumeration, in which case the C++/WinRT ABI type is `uint32_t`.

The C++/WinRT ABI structures take the form of structures where each member has its corresponding C++/WinRT ABI type. For example,

System	**ABI**	`struct { INT16 Value1; HSTRING Value2; SomeEnum Value3; };`
C++/WinRT	**ABI**	`struct { int16_t Value1; void* Value2; int32_t Value3; };`

	Projection	`struct { int16_t Value1; hstring Value2; SomeEnum Value3; };`

If the structure contains another structure, then the rule is applied recursively.

Finally, C++/WinRT interfaces are represented in the C++/WinRT ABI by a pure virtual class whose members are the interface methods, but with all parameters converted to their C++/WinRT ABI types. For example,

System	**ABI**	`struct ISomething : ::IInspectable { virtual HRESULT Method1(INT32 param1) = 0; virtual HRESULT Method2(HSTRING* result) = 0; };`
C++/WinRT	**ABI**	`struct ISomething : inspectable_abi { virtual int32_t Method1(int32_t param1) = 0; virtual int32_t Method2(void** result) = 0; };`
	Projection	`struct ISomething : winrt::IInspectable { void Method1(int32_t param1); winrt::hstring Method2(); };`

These different versions are placed in separate namespaces.

The System ABI puts metadata-defined types in the `ABI` namespace. For example, `Windows.Foundation.Point` is defined in the System ABI as `ABI::Windows::Foundation::Point`. (Metadata types are the types defined in the `.winmd` metadata files. Fundamental types like the basic integer types, `HSTRING`, `IUnknown`, and `IInspectable` are not defined in metadata and reside in the global namespace.)

The C++/WinRT ABI puts metadata-defined types in the `winrt::impl` namespace, often as anonymous types. You need to know that they exist, and what they look like, but you aren’t expected to be using them directly.

The C++/WinRT projection puts metadata-defined types in the `winrt` namespace. For example, `Windows.Foundation.Point` is defined in the C++/WinRT projection as `winrt::Windows::Foundation::Point`.

The `winrt::impl` namespace contains internal implementation details, and that's where the `abi` template type hangs out. Its job is to convert C++/WinRT types into their corresponding C++/WinRT ABI types. For any projected type `T`, the type `winrt::impl::abi<T>::type` is the corresponding C++/WinRT ABI type. You shouldn't be using this template directly, but I'm mentioning it so that when you find yourself single-stepping through the C++/WinRT library, you'll know what that weird `abi` template is.

Raymond Chen

Follow