Using the Windows::Globalization::Calendar object from a Win32 app

devblogs.microsoft.com/oldnewthing/20160622-00

June 22, 2016

Raymond Chen

The `Windows.Globalization.Calendar` object is documented as supported for use by both Windows Store apps and desktop apps. The UWP samples repo has a Calendar sample that demonstrates how to use it from a Windows Store app, but how do you use it from a desktop app?

That's our motivation for looking at the Windows Runtime ABI. Normally, these objects are consumed via a mechanism known as *projection*, whereby the low-level ABI guts of the Windows Runtime are adapted to the usage pattern of a target language, so that the objects feel more like language native objects.[1] That's great if you're using a projected language like C++/CX, C#, or JavaScript, but if you want to use raw C++, then you're going to be talking to the ABI.

Crash course in projection:

	Create an object
ABI	`IWidget* widget;` `ActivateInstance(L"Widget", &widget);`
C++/CX	`auto widget = ref new Widget();`
C#	`var widget = new Widget();`
JavaScript	`var widget = new Widget();`

At the ABI level, you use `ActivateInstance` to create an object; this projects as a parameterless constructor. We'll look at parameterized constructors some other time.

Method calls at the ABI level look pretty much like method calls in projection, except that output parameters are passed explicitly as [out] parameters rather than as the return value of the method. (The return value of ABI methods is always `HRESULT`. We'll look at projected `HRESULT`s later.)

	Invoke a method	
	no return value	with return value
ABI	`widget->Method();`	`widget->Method(&result);`
C++/CX	`widget->Method();`	`result = widget->Method();`
C#	`widget.Method();`	`result = widget.Method();`
JavaScript	`widget.method();`	`result = widget.method();`

Notice that the first character of the method name is converted to lowercase by the JavaScript projection, so that it matches existing JavaScript convention.

And the last piece of projection for today: Properties.

	Read property	Write property
ABI	`widget->get_Foo(&v);`	`widget->put_Foo(v);`
C++/CX	`v = widget->Foo;`	`widget->Foo = v;`
C#	`v = widget.Foo;`	`widget.Foo = v;`
JavaScript	`v = widget.foo;`	`widget.foo = v;`

Okay, we now know just enough to be dangerous. We'll write a little console program to get the current date, get the name of the day of the week in a form intended to be displayed on its own, get the last day of the current month, then skip forward six months and check whether daylight saving time will be in effect. Remember, Little Programs do little to no error checking.

```
#include <windows.h>
#include <wrl/client.h>                // WRL::ComPtr
#include <wrl/wrappers/corewrappers.h> // WRL::Wrappers
#include <windows.globalization.h>     // Windows::Globalization
#include <stdio.h>

namespace WRL = Microsoft::WRL;

int main(int argc, wchar_t** argv)
{
  CCoInitialize init;

  // C++/CX: auto cal = ref new Windows::Globalization::Calendar();
  WRL::ComPtr<ABI::Windows::Globalization::ICalendar> cal;
  Windows::Foundation::ActivateInstance(
    WRL::Wrappers::HStringReference(
      RuntimeClass_Windows_Globalization_Calendar).Get(), &cal);

  // C++/CX: auto dayOfWeek = cal->DayOfWeekAsFullSoloString();
  WRL::Wrappers::HString dayOfWeek;
  cal->DayOfWeekAsFullSoloString(dayOfWeek.GetAddressOf());

  wprintf(L"%ls\n", dayOfWeek.GetRawBuffer(nullptr));

  // C++/CX: lastDayThisMonth = cal->LastDayInThisMonth;
  INT32 lastDayThisMonth;
  cal->get_LastDayInThisMonth(&lastDayThisMonth);
  wprintf(L"Last day in this month is %d\n", lastDayThisMonth);

  // C++/CX: cal->AddMonths(6);
  cal->AddMonths(6);

  // C++/CX: isDST = cal->IsDaylightSavingTime;
  boolean isDST;
  cal->get_IsDaylightSavingTime(&isDST);

  wprintf(L"DST six months from now? %ls\n", isDST ? L"Yes" : L"No");

  return 0;
}
```

The raw C++ code is a straightforward translation of the corresponding C++/CX code. One thing to note is that we used the symbol `RuntimeClass_Windows_Globalization_Calendar` rather than hard-coding the string `L"Windows.Globalization.Calendar"`.

Another thing to note is that Windows Runtime strings are handled in the form of `HSTRING`s, which we discussed a little while ago.

[1]Another approach is Modern C++, which is effectively a projection of the Windows Runtime into native C++.

Raymond Chen

Follow