C++ Component Extension for WinRT

Ale Contenti

Development Manager | Visual C++ | Microsoft

C++ now | may 2012

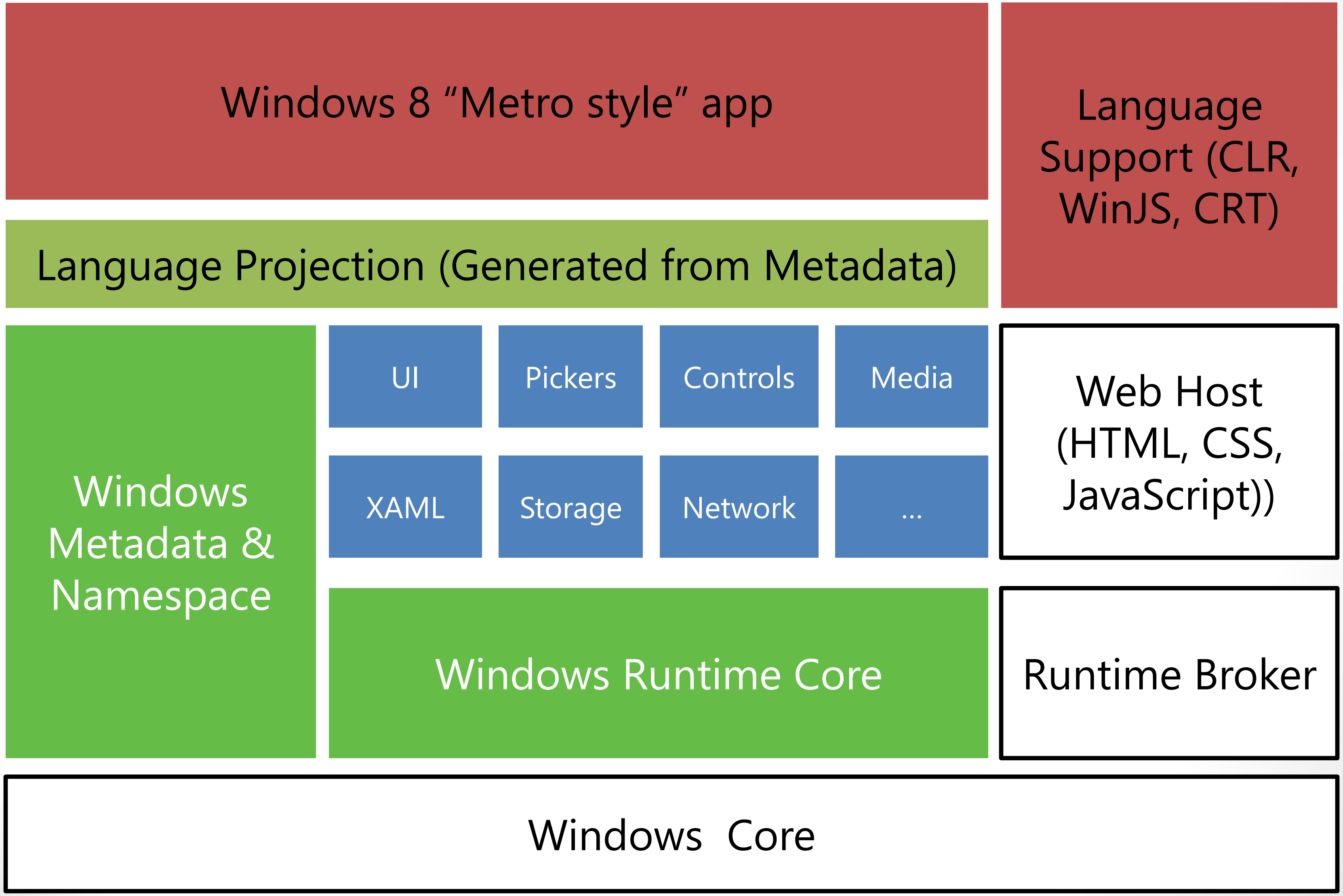
Agenda

- The Windows Runtime (aka WinRT)
  - What is WinRT?
  - Design principles (and a bit of history ☺)
  - Language “bindings” or “projections”

- WRL and C++/CX
  - C++ has two language projections for WinRT
  - Differences and goals
  - Why two projections?

- ABI, C++, modules and libraries
  - An open discussion about library packaging, best practices, problems

Windows Runtime Architecture


Windows 8 “Metro style” app
Language Support (CLR, WinJS, CRT)
Language Projection (Generated from Metadata)
Windows Metadata & Namespace
UI
Pickers
Controls
Media
XAML
Storage
Network
…
Web Host (HTML, CSS, JavaScript))
Windows Runtime Core
Runtime Broker
Windows Core

The Windows Runtime (aka WinRT)

Windows Runtime (WinRT)

- The Windows Runtime is the solid, efficient foundation for building Windows 8 Metro style apps
- A new API surface which replaces Win32
- Modern, object oriented, easier to use

Windows Runtime (WinRT)

- You’re in early 2010, and you want to revamp the developer experience for Windows
- What do you do?

Windows Runtime (WinRT)

- You throw away the old “C” style based Win32
  - You literally have tens of thousands of APIs with a lot of duplication. It’s time to cleanup!
- You think hard about the developer experience and the developer productivity
  - IntelliSense, tooling, etc.
- You create a solid, clear, consistent and modern API surface
  - Object oriented, namespace organization, async patterns
- You enable all major programming style to easily “bind” to this API surface
  - Native (C++), Managed (think C#, Java), Dynamic (think JavaScript, Python)

Windows Runtime (WinRT)

- You (Windows) also call up all your friends from Visual Studio…
- …and you end up putting a bunch of dudes from C++, C#, CLR, .NET Framework and JavaScript in a room for a couple of months
- …and, depending on many other factors, you might end up with something like WinRT ☺
- Ah, you also invent a new string type

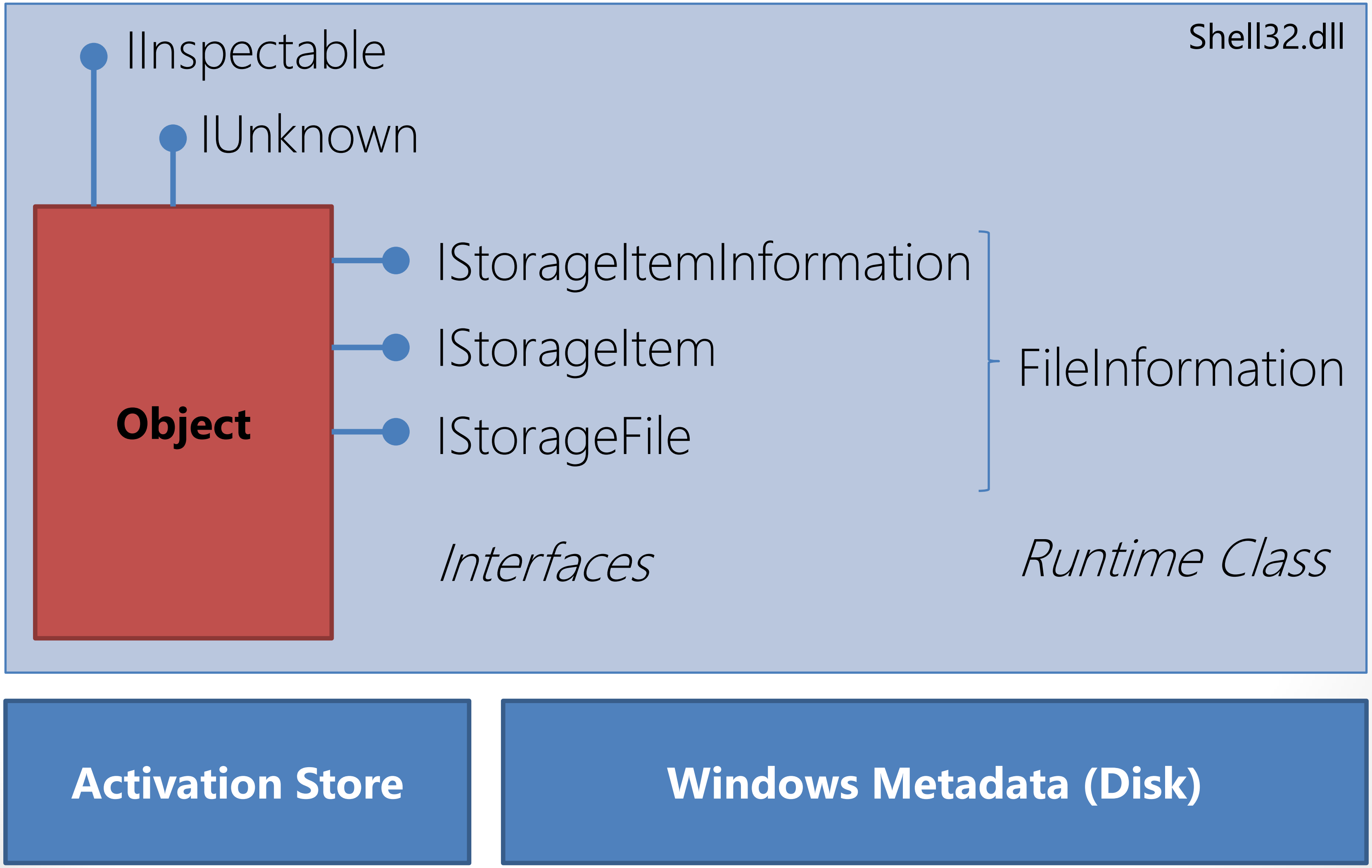
WinRT design principles

- Major improvement to developer experience
  - Great IntelliSense and tooling
- Native, Managed, Dynamic all first-class citizens
  - JavaScript, C#/VB and C++ initial targets
- Platform based Versioning
  - Apps keep running on future Windows versions
  - Simple low-level constructs; usability in projection
- Responsive and Fluid Apps
  - Async APIs where they are needed
- Well-designed, consistent objects
  - API surface is clear and consistent

WinRT implementation

- For each WinRT object:
  - Interfaces
  - No data members
  - Factory “construction” pattern
  - Described by metadata
- Each language projection can figure out the exact binary contract just looking at the metadata
- Basic types are well specified
- A very small number of patterns are perused across the API surface
  - Async, Collections, Enumerators/Iterators/Ranges

WinRT objects



Shell32.dll

IInspectable

IUnknown

Object

IStorageItemInformation

IStorageItem

IStorageFile

FileInformation

Interfaces

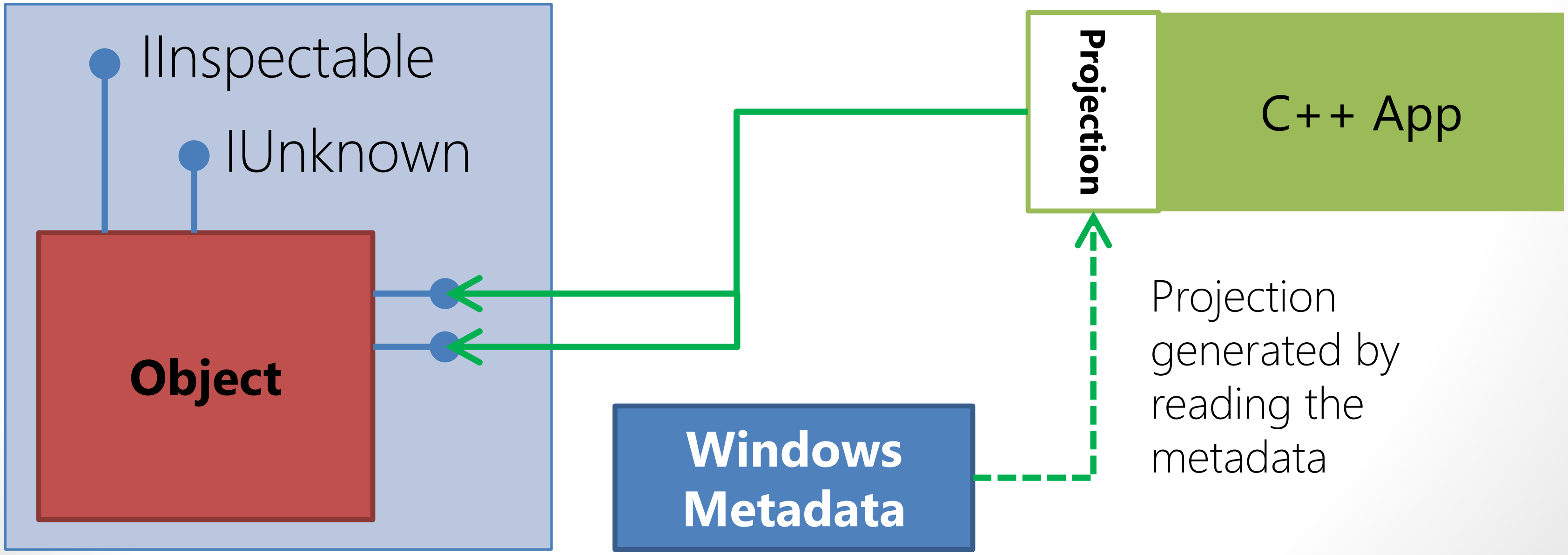
Runtime Class

Activation Store

Windows Metadata (Disk)

WinRT Metadata

- Efficient binary format derived CLI Metadata
  - Profile of ECMA 335, Partition II
  - Same structures, different meanings
  - Readable by existing tools
- Rich enough to allow multi-language projection generation
- Full IntelliSense on statically known information



Projection generated by reading the metadata

WinRT Basic Types

Basic Types	INT32, UINT64, etc.	Our usual friends
Strings	HSTRING	Avoids copying in multiple languages
Enumerations	enum AsyncStatus	Flag or non-flag styles
Structures	struct Rect;	Can contain strings, but not interfaces
Simple Arrays	INT32 []	For very basic collections
Interfaces	IInspectable	All methods are defined as part of interfaces
Generic Interfaces	IVector<T>	Type-generic interface. Not extensible
Runtime Class	Windows::Storage::StorageFile	Binds interfaces to make a class

WinRT Patterns

Collections	IVector<T>, IVectorView<T>, IMap<T>, IObservableVector<T>	Treat them like STL collections (begin()/end()/for()/etc.)
Delegates	delegate AsyncActionCompletedHandler	Encapsulate the context to call back to an object
Events	IApplicationLayout:: LayoutChanged	Lists of callback recipients
PropertySet	interface IPropertySet	Collection of items with varying types
Async Operation	ReceivePropertiesOperation	A way to get a delayed result without blocking

Windows 8

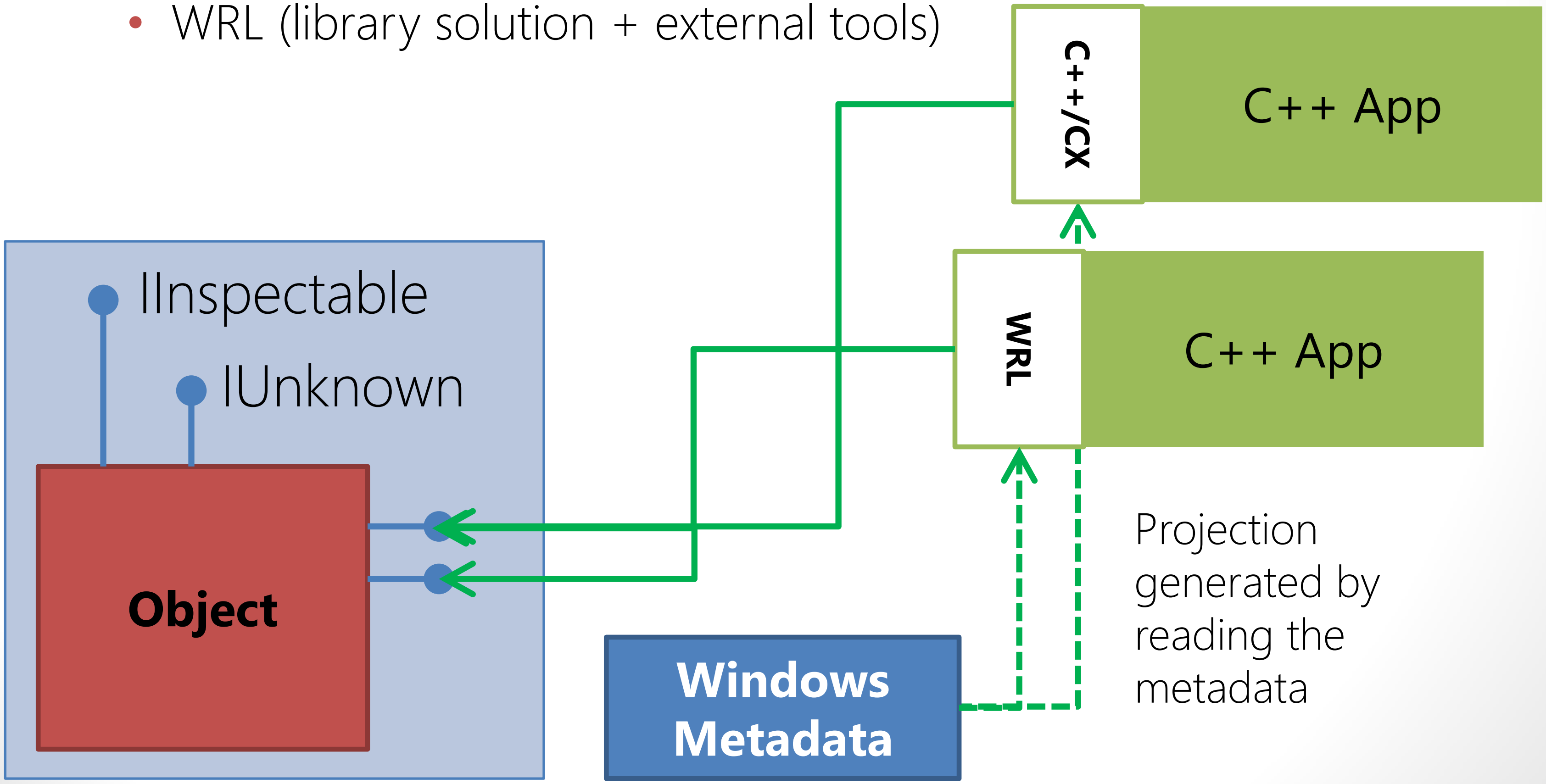
Face Recon demo

DEMO

WRL and C++/CX

C++ projection(s)

- VC++ has two different ways to “project” WinRT metadata, and thus consume WinRT constructs
  - C++/CX (language component extensions)
  - WRL (library solution + external tools)



Projection generated by reading the metadata

WRL – first look

```cpp
#include <wrl.h>
#include <wrl\wrappers\corewrappers.h>
#include <windows.storage.pickers.h>

using namespace ABI::Windows::Storage::Pickers;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

ComPtr<IFileOpenPicker> openPicker;
HString classid;
classid.Set(L"Windows.Storage.Pickers.FileOpenPicker");
CHECKHR(ActivateInstance(classid, &openPicker));
CHECKHR(openPicker->put_SuggestedStartLocation(
   PickerLocationId::PickerLocationId_PicturesLibrary));
CHECKHR(openPicker->put_ViewMode(
   PickerViewMode::PickerViewMode_Thumbnail));
```

WRL

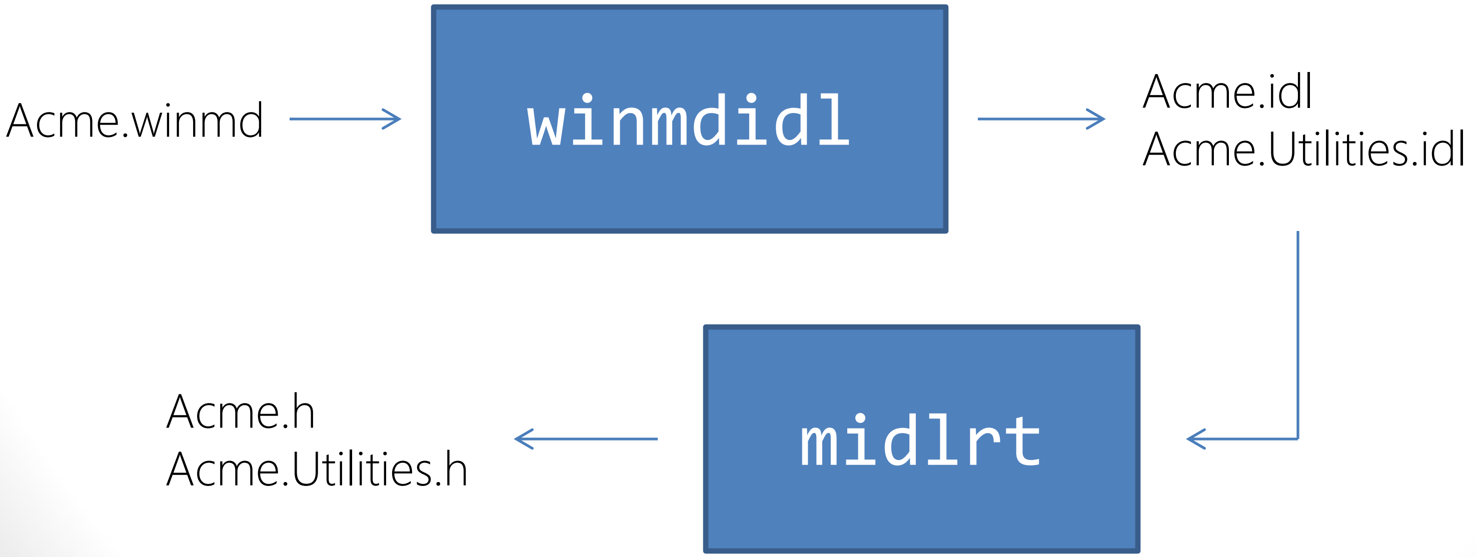
- WRL stands for Windows Runtime Library
- Developed by VC++
- Part of the Windows SDK
- Used by Windows to build basically every WinRT object offered by Windows 8
- Predates C++/CX and WinRT
- WRL was originally designed as a prototype (called nCOM)
  - Modern way to create and consume light-COM objects
  - Solve the ABI problem across C++ modules (.dll)

WRL – key characteristics

- No exceptions
  - WRL constructs will not throw any exception
  - Error codes (HRESULT) are used to return error codes
- Low level library
  - Gives developer full control over the WinRT architecture (e.g. out-of-proc servers, etc.)
- Library solution
  - Does not require any extension to the C++ language
  - Easier to re-target to a different C++ compiler
- Can be used to mix WinRT components and COM components
- Does not hide WinRT and COM complexity
- Heavily templated library (error messages are a beauty… ☺)

WRL – toolchain

- You need to build the “projection” from the metadata
- The compiler is a normal C++ compiler, so it cannot read or interpret metadata files (.winmd)
- Use winmdidl + midlrt:



WRL – toolchain

```
#include <wrl.h>
#include <wrl\wrappers\corewrappers.h>
#include <acme.h>

using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;

ComPtr<Acme::IWidget> w;
HString classid;
classid.Set(L“Acme.Widget");
CHECKHR(ActivateInstance(classid.Get(), &w));
CHECKHR(w->DoSomething());
```

WRL – under the hood

- Microsoft::WRL::ComPtr is a “modern” COM ref counted smart pointer
- Classical COM smart pointers are not very safe
    - operator& is usually very dangerous

WRL – under the hood

- For example, ATL::CComPtr looks like this:

```
template <class T>
class CComPtr
{
  // …
  T** operator&() throw()
  {
    ATLASSERT(ptr_ == NULL);
    return &ptr_;
  }
protected:
  T* ptr_;
};
```

- Returning the address to the bare pointer breaks the encapsulation of the smart pointer class
- Also, it's hard to get the address of the real CComPtr

WRL – under the hood

- In Microsoft::WRL::ComPtr, operator& returns the helper class ComPtrRef

```
template <class T>
class ComPtr
{
  // …
  Details::ComPtrRef<ComPtr<T>> operator&() throw()
  {
    return Details::ComPtrRef<ComPtr<T>>(this);
  }
protected:
  T* ptr_;
};
```

- ComPtrRef<ComPtr<T>> can convert to both ComPtr<T>* and the classic T**

WRL – under the hood

- This way we maintain the usability of the classic T** COM pattern like:

```
HRESULT get_FileTypeFilter(
  __FIVector_1_HSTRING **value);
ComPtr<__FIVector_1_HSTRING> filter;
CHECKHR(openPicker->get_FileTypeFilter(&filter));
```

- While enabling the “safer” version:

```
template<typename T>
HRESULT ActivateInstance(
  HSTRING activatableClassId,
  WRL::Details::ComPtrRef<T> instance) throw();
ComPtr<IFileOpenPicker> openPicker;
ActivateInstance(classid, &openPicker)
```

WRL

DEMO

C++/CX – first look

```
#using <Windows.winmd>

using namespace Windows::Storage::Pickers;

auto openPicker = ref new FileOpenPicker();
openPicker->SuggestedStartLocation =
   PickerLocationId::PicturesLibrary;
openPicker->ViewMode = PickerViewMode::Thumbnail;
```

C++/CX

- C++/CX stands for C++ Component Extensions
- Part of the VC++ compiler (in Dev11)
- Reuse the syntax of ECMA Standard C++/CLI

- Set of **language extensions** and **libraries** to allow direct **consumption** and **authoring** of Windows Runtime types
  - Strongly-typed system for Windows Runtime
  - Automatically reference counted
  - Exception-based
  - Deep integration with STL
  - Well defined binary contract across module boundaries

C++/CX

- No need for external tools
- The compiler can read and understand the metadata:

```
#using <Windows.winmd>
```

- The metadata in imported “on-demand”
  - As the compiler needs definitions for types and constructs from the metadata, it queries for more data
  - This model is superior to processing the entire .h file: The metadata is easier and faster to query
- The strong reference “^” (read “hat”) is basically a ComPtr
  - But the compiler knows the semantics of ^
  - And can optimize away redundant AddRef/Release and QueryInterface

C++/CX

DEMO

Mix it up

- You can use WRL and C++/CX in the same TU
- Most useful when you need to reference some classic COM components
  - e.g. DirectX is still light-COM in Windows 8
- A Platform::Object^ reference is just a pointer to a WinRT IInspectable interface

```
Windows::Storage::Pickers::IFileOpenPicker^ GetOpenPickerWithWRL()
{
  using namespace ABI::Windows::Storage::Pickers;
  using namespace Microsoft::WRL;

  ComPtr<IFileOpenPicker> openPicker;
  // ...
  return
    dynamic_cast<Windows::Storage::Pickers::IFileOpenPicker^>(
      reinterpret_cast<::Platform::Object^>(
        openPicker.Get()));
}
```

Why WRL *and* C++/CX?

- WRL was initially considered just for internal development in Windows
- Just before the //build/ conference (in Sept 2011), we decided to add WRL to the VC++ libraries
- Why? We wanted to target a small set of the C++ dev population, which might have specific needs (e.g. no exceptions)
  - With the WRL offering, we have a "no compromise" (but also not that pretty) option for coding against WinRT

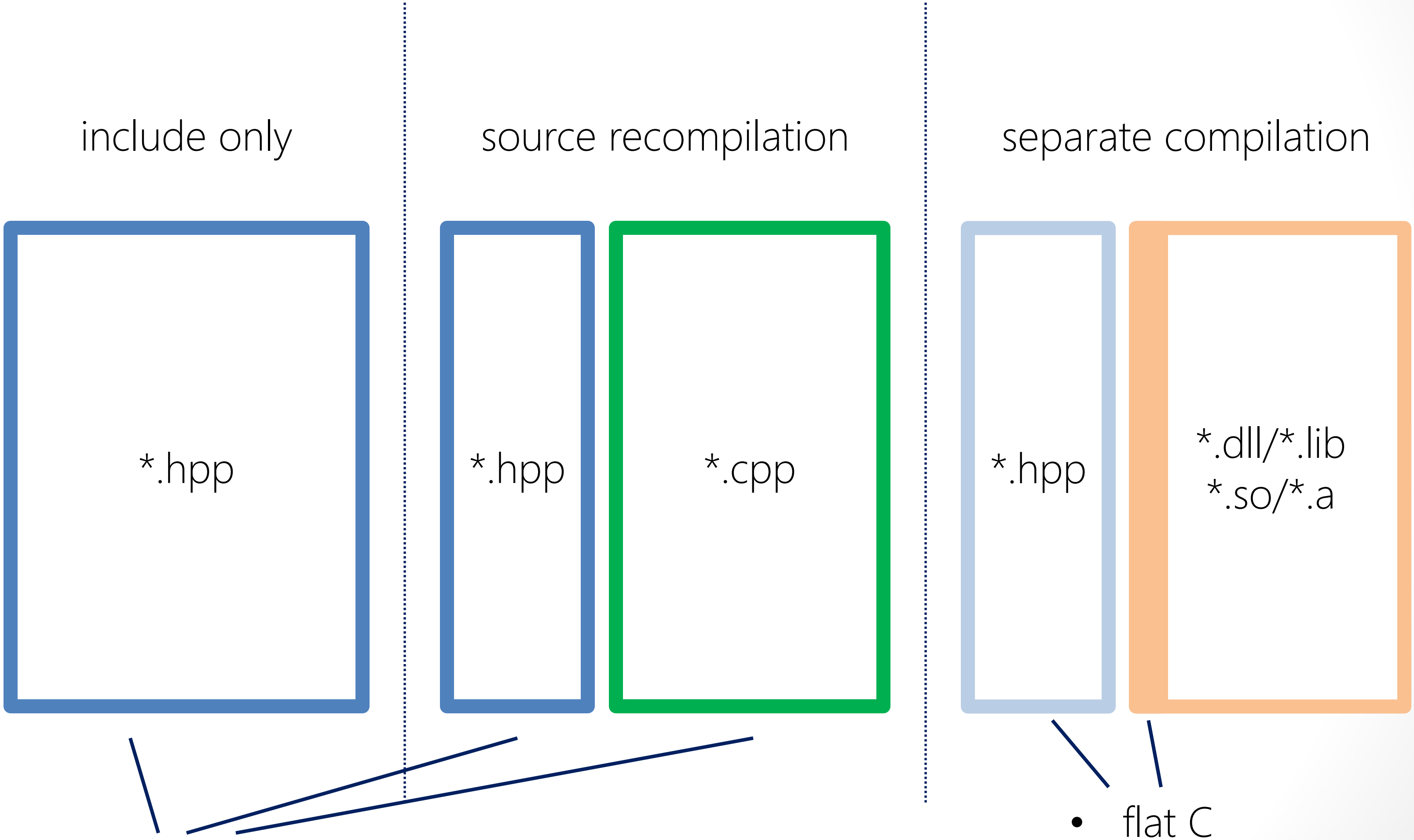
C++/CX and WRL comparison

C++/CX	WRL
Exception based	No exception; HRESULT based
Return value is used in a natural way	Return value is reserved for HRESULT
Extensions to C++ language	Pure library solution
Reference counted	Ref count via smart pointer
Can access low-level pointer	Can access low-level pointer
Compact	Verbose and complex
No need for external tools	Need external tools
Hides COM complexity	COM wiring is exposed

- General recommendation is to use C++/CX unless you are in an exception-free environment.
- WRL can be useful to mix classic COM components and WinRT components.

ABI, C++, modules and libraries

Libraries packaging



include only

source recompilation

separate compilation

*.hpp

*.hpp

*.cpp

*.hpp

.dll/.lib
.so/.a

full C++ power

- flat C
- COM?
- metadata?

Libraries packaging

- Include only model
  - Full C++ power
  - Easy distribution and packaging
  - Slower code compilation
  - Complexity with large libraries (central state, etc.)
  - No code obfuscation
  - User can modify the code
  - ODR violation problems

Libraries packaging

- Source recompilation model
  - Full C++ power
  - More complex distribution and packaging (needs to add build scripts, etc.)
  - Faster code compilation (still need to compile at least once though)
  - Easier to maintain large libraries
  - No code obfuscation
  - User can modify the code
  - ODR violation problems

Libraries packaging

- Separate compilation model
  - Full C++ power in the separately compiled module “guts”, but must to downgrade to flat “C” for interop
  - Medium complexity in distribution and packaging
    - Need to have import libs (for .dll)
    - Need to redist the .dll/.so
  - Faster code compilation
  - Easier to maintain large libraries
  - Code obfuscation, if needed
  - User cannot modify the code
  - ODR violation problems are minimized

  - We should avoid C++ construct in the .hpp interface: prone to errors, packing mis-alignements, etc.

What do you think?

- Which model do you like most?
- Which model do you use?
- Problems?

- What about the metadata?
- Aren’t we tired of .h/.hpp files? ☺

Questions?

Contacts

- alecont@microsoft.com
- http://blogs.msdn.com/b/vcblog/
- http://channel9.msdn.com/tags/C++/
- http://www.buildwindows.com/

MICROSOFT C++

2012

PARTICIPATE IN C++ DEVELOPMENT USER RESEARCH

MICROSOFT
DEVELOPER
DIVISION
DESIGN
RESEARCH

TALK TO GEORGE

GRAB A CARD

FILL IT ONLINE AT
http://bit.ly/VSUxResearch