DOTNEXT

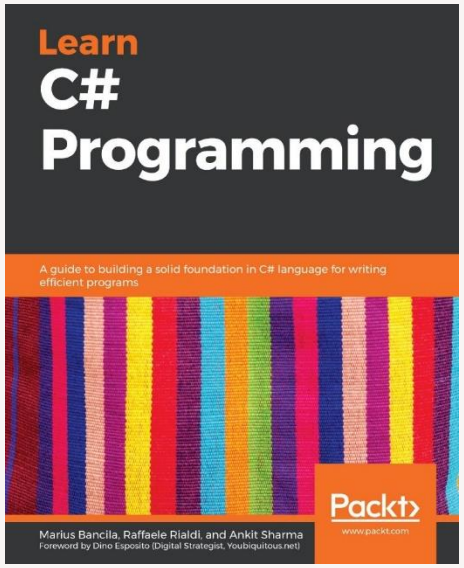
The new inter-language interoperability in .NET 5 and .NET 6

Raffaele Rialdi - Senior Software Architect

@raffaeler
raffaeler@vevy.com

Who am I?

- Raffaele Rialdi, Senior Software Architect in Vevy Europe – Italy
  - @raffaeler also known as "Raf"
- Consultant in many industries
  - Manufacturing, racing, healthcare, financial, …
- Speaker and Trainer around the globe
  - Italy, Romania, Bulgaria, Russia, USA, …
- Proud member of the great Microsoft MVP family since 2003



in·ter·op·er·a·bil·i·ty

« the ability of computer systems or software
to exchange and make use of information »

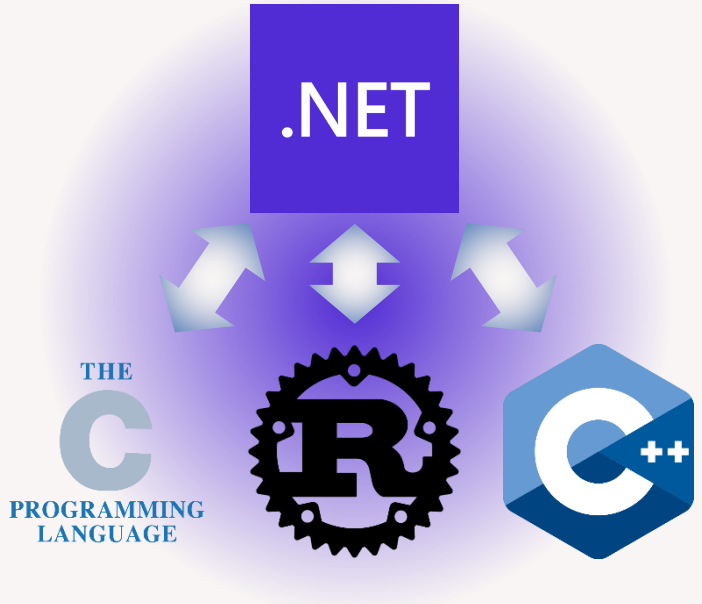
ability to make a call

marshalling data

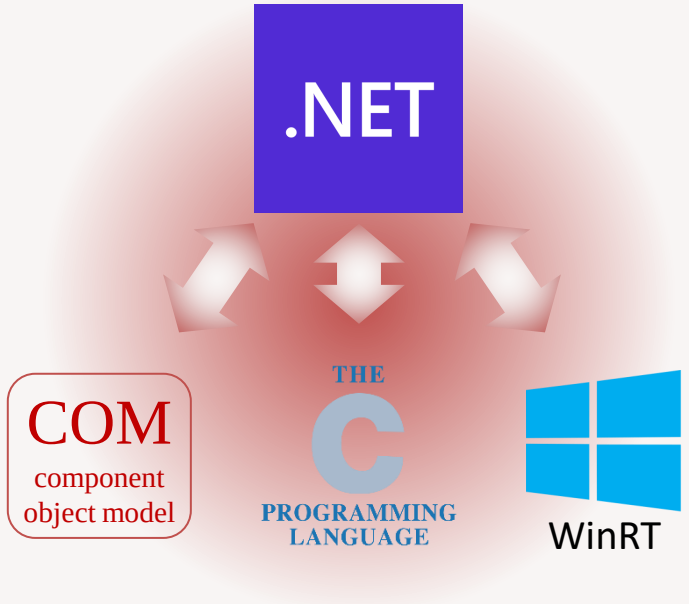
definition by Oxford Languages

Variety of OS'es

Variety of Languages



Three ABIs



@raffaeler

Agenda

- Many interoperability mechanisms
  - Manged to Native
  - Native to managed
- Discuss the marshalling machinery
- Hosting managed code from C++ and Rust
- Code generators

From ABI to Metadata

- The Application Binary Interface is the lower level boundary
  - It is the contract defining how two binaries could call each other

- For historical reasons, the "C exports" are popular
  - Weak contract, just defining how to pass the parameters on the stack
  - This is why PInvokes are difficult to write

- This is why we need metadata: for example IDL (now version 3)
  - **buffer size**, **array length**, **in/out/ref, …**

Which metadata should we use?

- None is an option
  - You just write the PInvoke declarations by hand
- COM is still an option
  - Work-in-progress to use its ABI on Linux/Mac for interoperability purposes
  - Uses the older version of IDL (version 1)
- WinRT currently is Windows only
  - A new version of IDL (version 3) is compiled into ".winmd"
    - Winmd are ECMA-335 metadata which is the official standard defining the .NET CLI
    - You can inspect winmd files with ILSpy (no implementation of course)
- Win32 metadata definition (produce winmd)
  - https://github.com/microsoft/win32metadata (you won't use this directly)
  - Metadata for all the Win32 API (will be announced at #build2021)

Metadata to C-Language ABI Projections

- The goal of the projections is generating code to provide access to the boundary in the most natural way for each language.

- Projections using the C-Language / PInvoke ABI
  - C# (source generator) https://github.com/microsoft/cswin32
  - C# (pre-generated) https://github.com/dotnet/pinvoke
  - C++ https://github.com/microsoft/cppwin32
  - Rust https://github.com/retep998/winapi-rs
  - Dart https://github.com/timsneath/win32

Pre-generated
Win32 PInvoke declarations
on NuGet

Library	Package name	NuGet	Description
advapi32.dll	PInvoke.AdvApi32	nuget v0.7.104 ▾ 80.5k	Windows Advanced Services
bcrypt.dll	PInvoke.BCrypt	nuget v0.7.104 ▾ 3.89m	Windows Cryptography API: Next Generation
cabinet.dll	PInvoke.Cabinet	nuget v0.7.104 ▾ 1.4k	Cabinet API Functions
cfgmgr32.dll	PInvoke.CfgMgr32	nuget v0.7.104 ▾ 1.1k	Device and Driver Installation
crypt32.dll	PInvoke.Crypt32	nuget v0.7.104 ▾ 19.3k	Windows Cryptography API
DwmApi.dll	PInvoke.DwmApi	nuget v0.7.104 ▾ 14.6k	Desktop Window Manager
fusion.dll	PInvoke.Fusion	nuget v0.7.104 ▾ 12.2k	.NET Framework Fusion
gdi32.dll	PInvoke.Gdi32	nuget v0.7.104 ▾ 21.6k	Windows Graphics Device Interface
hid.dll	PInvoke.Hid	nuget v0.7.104 ▾ 16.8k	Windows Human Interface Devices
iphlpapi.dll	PInvoke.IPHlpApi	nuget v0.7.104 ▾ 548	IP Helper
kernel32.dll	PInvoke.Kernel32	nuget v0.7.104 ▾ 4.04m	Windows Kernel API
magnification.dll	PInvoke.Magnification	nuget v0.7.104 ▾ 6.5k	Windows Magnification API
mscoree.dll	PInvoke.MSCorEE	nuget v0.7.104 ▾ 14.5k	.NET Framework CLR host
msi.dll	PInvoke.Msi	nuget v0.7.104 ▾ 13.3k	Microsoft Installer
ncrypt.dll	PInvoke.NCrypt	nuget v0.7.104 ▾ 3.99m	Windows Cryptography API: Next Generation
netapi32.dll	PInvoke.NetApi32	nuget v0.7.104 ▾ 10.8k	Network Management
newdev.dll	PInvoke.NewDev	nuget v0.7.104 ▾ 1.1k	Device and Driver Installation
ntdll.dll	PInvoke.NTDll	nuget v0.7.104 ▾ 16.2k	Windows NTDll
psapi.dll	PInvoke.Psapi	nuget v0.7.104 ▾ 15.1k	Windows Process Status API
setupapi.dll	PInvoke.SetupApi	nuget v0.7.104 ▾ 18.8k	Windows setup API
SHCore.dll	PInvoke.SHCore	nuget v0.7.104 ▾ 10.6k	Windows Shell
shell32.dll	PInvoke.Shell32	nuget v0.7.104 ▾ 25.2k	Windows Shell
user32.dll	PInvoke.User32	nuget v0.7.104 ▾ 74.9k	Windows User Interface
userenv.dll	PInvoke.Userenv	nuget v0.7.104 ▾ 14.0k	Windows User Environment
uxtheme.dll	PInvoke.UxTheme	nuget v0.7.104 ▾ 14.8k	Windows Visual Styles
winusb.dll	PInvoke.WinUsb	nuget v0.7.104 ▾ 1.1k	USB Driver
WtsApi32.dll	PInvoke.WtsApi32	nuget v0.7.104 ▾ 9.4k	Windows Remote Desktop Services

Calling the Win32 APIs from .NET

~~http://pinvoke.net/~~

- Option 1: Add one of the NuGet packages created by: https://github.com/dotnet/pinvoke

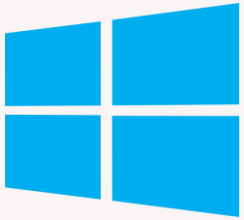
(many more)

Library	Package name	NuGet	Description
advapi32.dll	`PInvoke.AdvApi32`	nuget v0.7.104 50.5k	Windows Advanced Services
bcrypt.dll	`PInvoke.BCrypt`	nuget v0.7.104 3.89m	Windows Cryptography API: Next Generation
cabinet.dll	`PInvoke.Cabinet`	nuget v0.7.104 1.4k	Cabinet API Functions
cfgmgr32.dll	`PInvoke.CfgMgr32`	nuget v0.7.104 1.1k	Device and Driver Installation
crypt32.dll	`PInvoke.Crypt32`	nuget v0.7.104 19.3k	Windows Cryptography API
DwmApi.dll	`PInvoke.DwmApi`	nuget v0.7.104 14.6k	Desktop Window Manager

- Option 2 : Let a C# source generator create the PInvoke for you

```
<PackageReference Include="Microsoft.Windows.CsWin32" Version="0.1.378-beta">
  <PrivateAssets>all</PrivateAssets>
  <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
</PackageReference>
```

.NET projections accessing Windows WinRT API



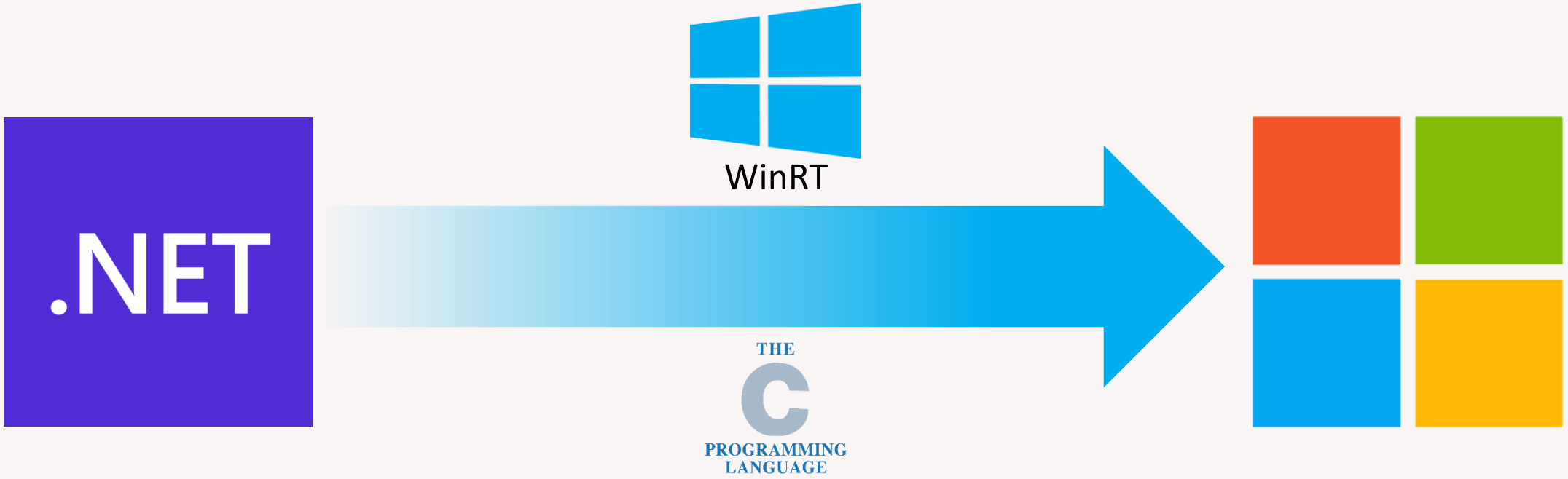
- We can natively reference a Windows SDK
  - All the required interoperability code is automatically available

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net5.0-windows10.0.19041.0</TargetFramework>
  <SupportedOSPlatform>windows7</SupportedOSPlatform>
</PropertyGroup>
```

.NET + Windows SDK versions

Minimum OS Version at runtime

- This is possible from **<u>any</u>** Windows application
  - Windows Forms, WPF or Console included
- The same technology is used by WinUI and the new WebView (Edge based on Chromium)



Demo: WindowsAPI

.NET exe application accessing a Win32 and WinRT APIs on Windows

Accessing custom components

- PInvoke
  - Currently required for code running cross-platform
    - Declarations written by hand or using 3rd party generators
  - Microsoft is working to a generate these declarations as well

- WinRT: currently available projection generators:
  - C# (producer and consumer) https://github.com/microsoft/cswinrt
  - C++ (producer and consumer) https://github.com/microsoft/cppwinrt
  - Rust (consumer) https://github.com/microsoft/windows-rs
  - Python (very experimental, only official WinRT API)
    - https://github.com/microsoft/xlang/tree/master/src/tool/python

Consuming WinRT C++ components

- The best replacement for C++/CLI
- On the C++ side CppWinRT
  - Use only ISO standard C++ language
  - VS Extension: https://marketplace.visualstudio.com/items?itemName=CppWinRTTeam.cppwinrt101804264
  - Use the C++/WinRT component template

- On the .NET side CsWinRT
  - Create a zero-code project and ad a reference to the C++ component
    - CsWinRT will generate the projection code for you https://github.com/microsoft/CsWinRT/
  - Rich interop: objects, methods, properties, events, async, …
  - Performant: leverage the latest .NET "calli" opcode and function pointers

Demo: ManagedWinRT

.NET exe application accessing a C++ custom component using WinRT on Windows

Inverting the actors: C++ calling .NET using WinRT

- CsWinRT allows exposing a class library as WinRT component
- C++ starts the process and automatically **host** the CLR
- Must be packaged as nuget to include three Microsoft libraries
- The C++ client must:
  1. Ship a json file with the .NET runtime version
  2. Add a manifest with the list of activatable classes

```
{ "runtimeOptions": {
    "tfm": "net5.0",
    "rollForward": "LatestMinor",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "5.0.0" } } }
```

```
<activatableClass
    name="ManagedComponent.QueryCatalog"
    threadingModel="both"
    xmlns="urn:schemas-microsoft-com:winrt.v1" />
```

Demo: WinRTNativeHosting

C++ exe application accessing a .NET component using WinRT on Windows

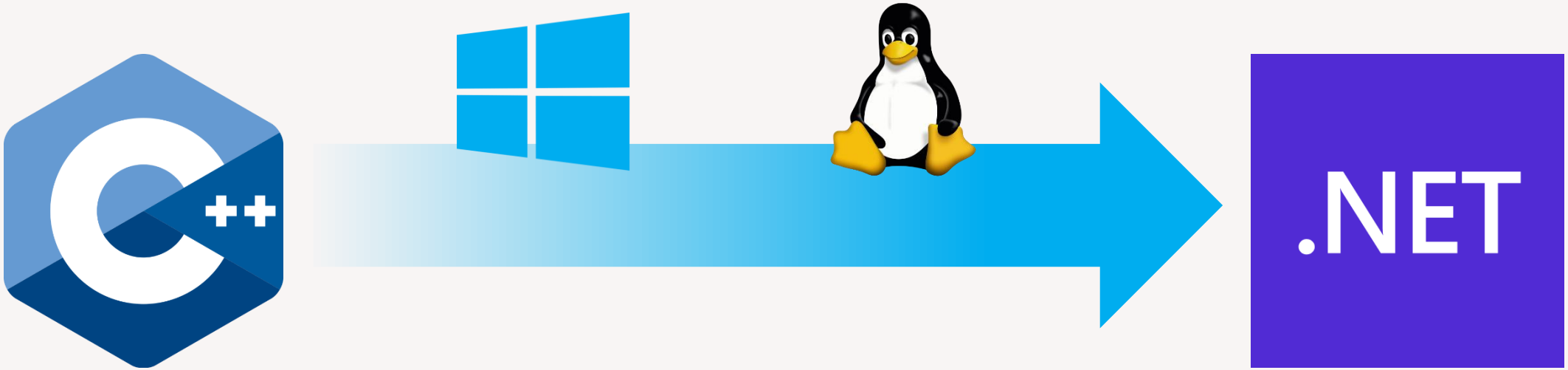
Demo: WinRTNativeHosting

Rust exe application accessing a .NET component using WinRT on Windows

What about going cross-platform?

- The only current solution is PInvoke and Reverse PInvoke
- Marshaling option 1 (less burden)
  - A static .NET method is exposed to the native world
  - We use Marshal attributes to obtain automatic Marshaling

- Marshaling option 2 (more perf)
  - `[UnmanagedCallersOnly(CallConvs = new[] { typeof(CallConvCdecl) })]`
  - Use only blittable types and **manually** Marshal the parameters
    - Span<T>, Memory<T>, MemoryMarshal, Unsafe are your friends



Demo: NativeHosting

C++ exe application accessing .NET methods using Reverse PInvoke cross-platform

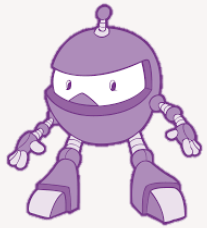
Conclusion

- Trivial cases of accessing Win32 and WinRT APIs work great!
- PInvoke generation for custom libraries is coming from MS

- Complex cases should be addressed case by case
  - Clang compiler provides a C++ parser library
    - You can create your own metadata and use Roslyn to generate the interop code

- Damn complex use-case: NodeJS hosting/calling .NET
  - https://github.com/raffaeler/xcore (my own project, presentation only)

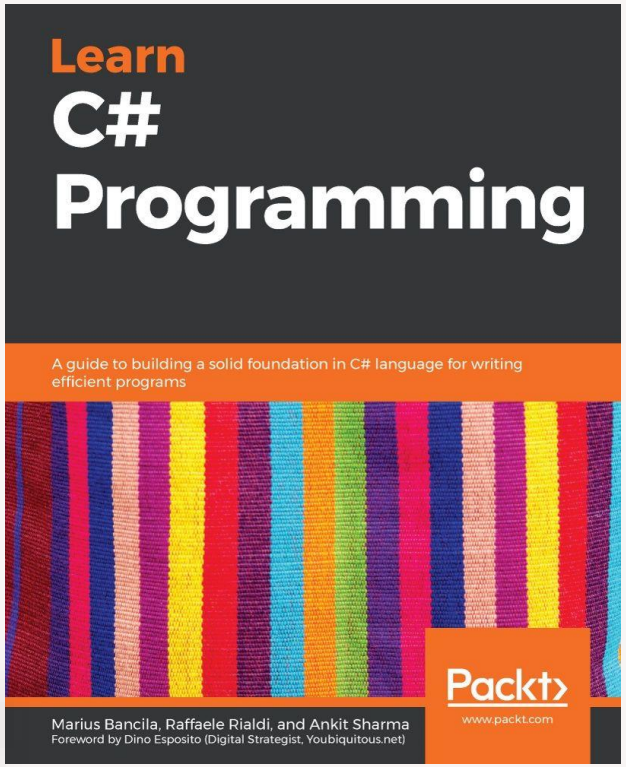
Questions?



Thank you!

@raffaeler

raffaeler@vevy.com



MVP Microsoft Most Valuable Professional