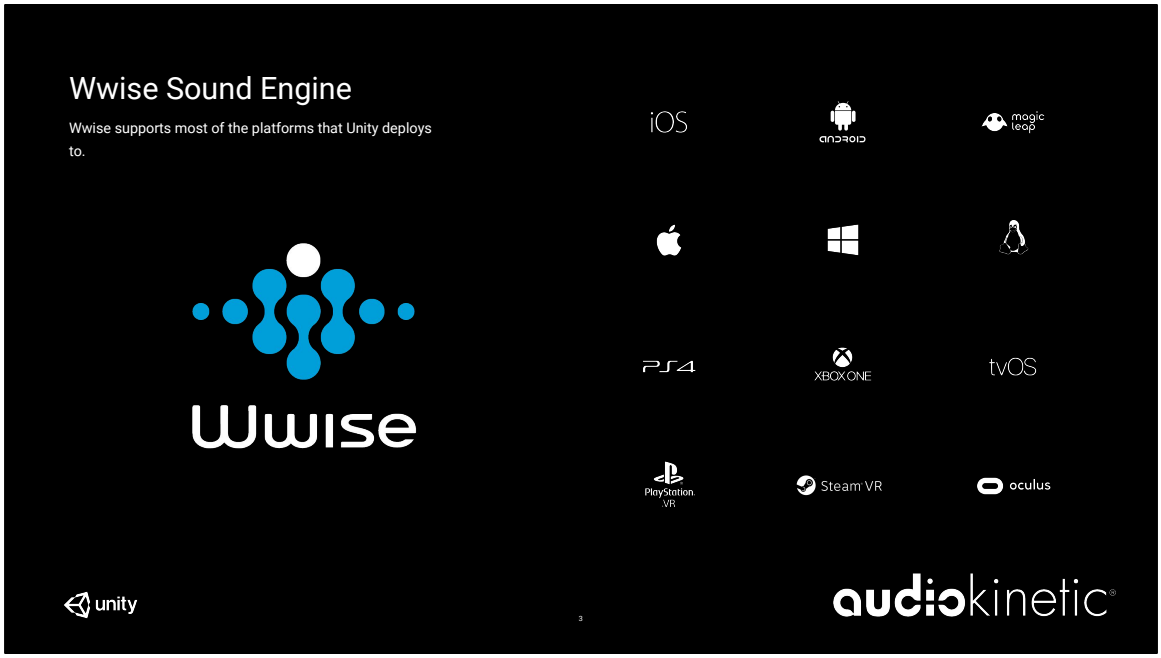
Unite Los Angeles 2018

Adding Love to an API
(or How I Learned to Stop Worrying and Expose C++ to Unity)

Shawn Laptiste
Software Developer - Game Integrations, Audiokinetic

Hi! My name is Shawn Laptiste, and today I'd like to present to you: Adding Love to an API: (or How I Learned to Stop Worrying and Expose C++ to Unity).
I am currently a Software developer at Audiokinetic - the company that makes the audio middleware, Wwise.



Wwise is a sound engine integrated into many of your favorite games and some of your favorite game engines.
Wwise is also an authoring tool that offers sound designers a platform for integrating their sound assets into games.

Outline

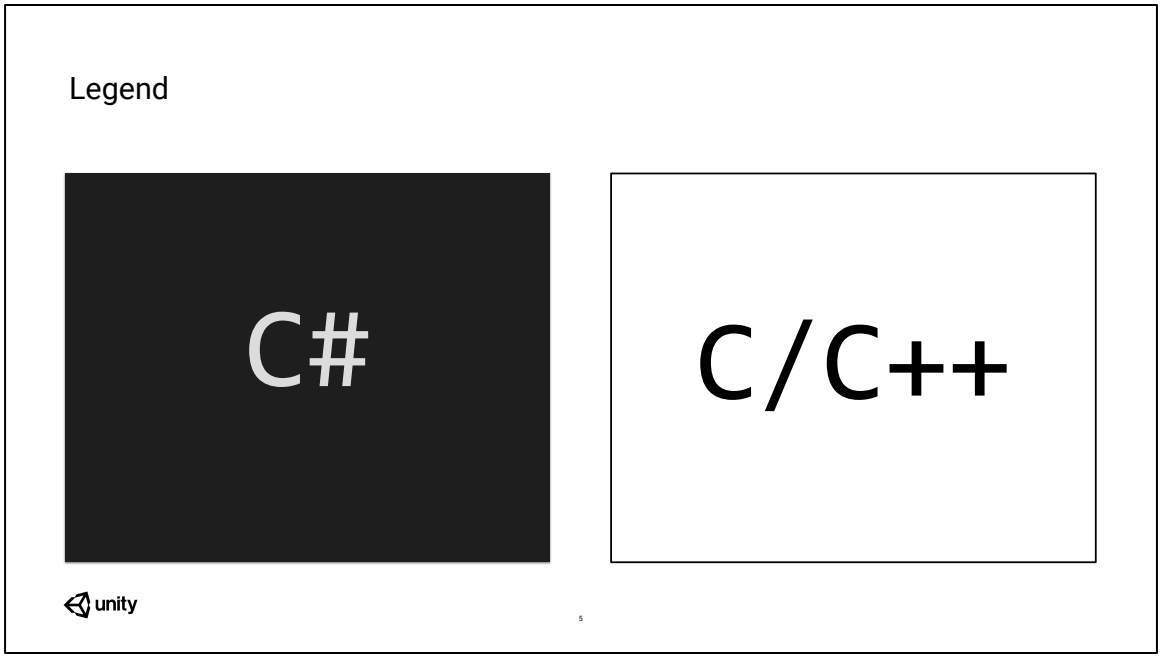
https://github.com/lazerfalcon/Unite2018_Native

- Calling Functions
- Passing Data
  - Primitive Types
  - One-Dimensional Arrays
  - Enums
  - Structs and Classes
- Advanced Topics
  - Marshalling Strings
  - Marshalling Delegates
  - Automation

unity

audiokinetic

Non-scripted: I have a lot of information to cover but don’t worry it’s all on GitHub…
blah blah reach me afterwards if you have questions.



The intended audience for this talk are developers with C/C++ experience.
To keep things as concise as possible, the code samples leave out a few details.
I hope they’re still clear enough for you to get the main idea.

C# examples will be listed on a black background, and C/C++ examples will be listed on a white background.

Calling Functions

Calling Functions
Pseudo-code

```
// Unity C# - Pseudo Code!!!
public class NormalUnityComponent :
UnityEngine.MonoBehaviour
{
  private void Start()
  {
    AmazingCppFunctionality();
  }
}
```

```
// C/C++ - Pseudo code!!!
void AmazingCppFunctionality()
{
  do
  {
    AmazingStuff();
  }
  while (!finished);
}
```

unity

7

So... Let's get into it.
You've identified a C/C++ library that has all of the functionality that you've wished for in the editor or in your game.
[Click]
All you want to do is plug this functionality into Unity and then get paid millions!
Well, first things first, you have to expose the functions to C#.

Exporting C/C++ Functions

Interoperability

```
#if IS_A_MICROSOFT_PLATFORM
  #define DLL_EXPORT extern "C" __declspec(dllexport)
  #define CALL_CONV __stdcall
#else
  #define DLL_EXPORT extern "C"
  #define CALL_CONV
#endif

DLL_EXPORT void CALL_CONV AmazingCppFunctionality();

// Generic Function declaration
DLL_EXPORT ReturnType CALL_CONV GenericFunction(ArgType1 arg1, ArgType2, arg2);
```

unity

8

Here, I have the C/C++ function declaration of the function from the previous slide, but this time decorated with 2 macros.

DLL_EXPORT is used to specify symbols that will be accessible outside of the library.
CALL_CONV is used to specify the calling convention, if required.

This is all used to make your native functions accessible via Unity C# scripts!

[Click]
Here is how these macros are defined for Microsoft platforms - including XboxOne; and
[Click]

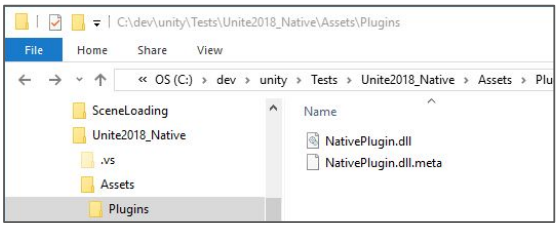
This is how they are defined for the rest.

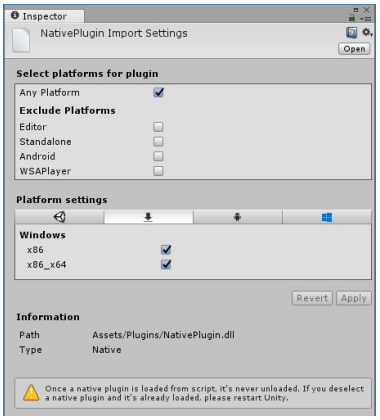
I must note that since extern "C" is used, function overloading is not available.
However, you will see that this should not be of major concern since the name of the function exposed to C# does not need to match its C/C++ counterpart.

[Click]
This is an example of how a new GenericFunction (with a return value and arguments) would be declared so that it can be exposed to C#.

Advanced:
https://docs.microsoft.com/en-us/cpp/cpp/extern-cpp?view=vs-2017
https://docs.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=vs-2017
https://docs.microsoft.com/en-us/cpp/cpp/stdcall?view=vs-2017
IS_A_MICROSOFT_PLATFORM = defined(_WIN32) || defined(_WIN64) || defined(WINAPI_FAMILY) || defined(_XBOX_ONE)

Exposing the Library Code
Dynamically Linked Library (DLL)



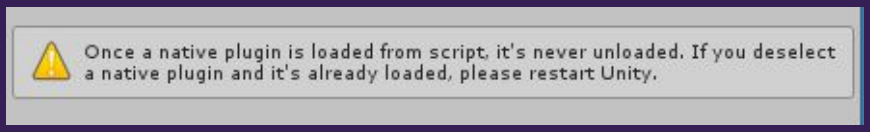


unity

9

Now that the functions are decorated so that they can be accessed outside of the library, we'll have to compile our code.
In this example, I've placed the built DLL in the Plugins folder, but it simply needs to be located somewhere within the Assets folder hierarchy so that Unity can pick it up.
Within Unity, you will have to ensure that the appropriate libraries are setup to be loaded for their respective platforms.

Native Plug-ins Are NEVER Unloaded
Gotcha



This can present problems while developing and iterating on your native code.

Consider testing your native code natively!

unity

audiokinetic

This brings us to our first GOTCHA! Once loaded, native plug-ins are never unloaded in the editor.
When iterating on your native code, this can slow you down considerably!
Be sure to test you native code natively.

Calling C/C++ Functions From C#

Platform Invocation Services (P/Invoke)

```
using System.Runtime.InteropServices;
public class NormalUnityComponent : UnityEngine.MonoBehaviour
{
#if STATIC_LINKING_WITH_PLUGIN
  public const string PluginName = "__Internal";
#else
  public const string PluginName = "NativePlugin";
#endif

  [DllImport(PluginName, EntryPoint = "AmazingCppFunctionality")]
  private static extern void Functionality();

  private void Start() { Functionality(); }
}
```

unity

12

P/Invoke is the .NET feature that enables managed code to call native code.
[Click]
It can be found in the System.Runtime.InteropServices namespace.
[Click]
For native functions to be called from managed code, a function prototype is required.
[Click]
The function prototype must be decorated by the DllImport attribute which specifies the plug-in name as its first argument and the function name via the EntryPoint field.
The EntryPoint field defaults to the name of the C# function prototype, but as seen here, can be explicitly specified. Setup this way, the C# method named “Functionality” will invoke the native

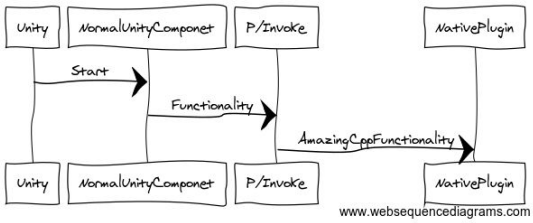
function “AmazingCppFunctionality”.
[Click]
The first argument to the DllImport attribute should be "__Internal" when the plug-in is statically linked, and should be the library name without the .DLL suffix when dynamically linked.

Now that we can call a function, let’s have a word or two about C/C++ and C# exceptions...

Advanced:
STATIC_LINKING_WITH_PLUGIN = (UNITY_IOS || UNITY_TVOS) && !UNITY_EDITOR

Calling C/C++ Functions From C#

Interaction Diagram



unity

Here’s the interaction diagram for the previous example.

Exceptions
Gotcha

Exceptions must not cross the managed/unmanaged boundary.

Under most circumstances, this will lead to a crash!

unity

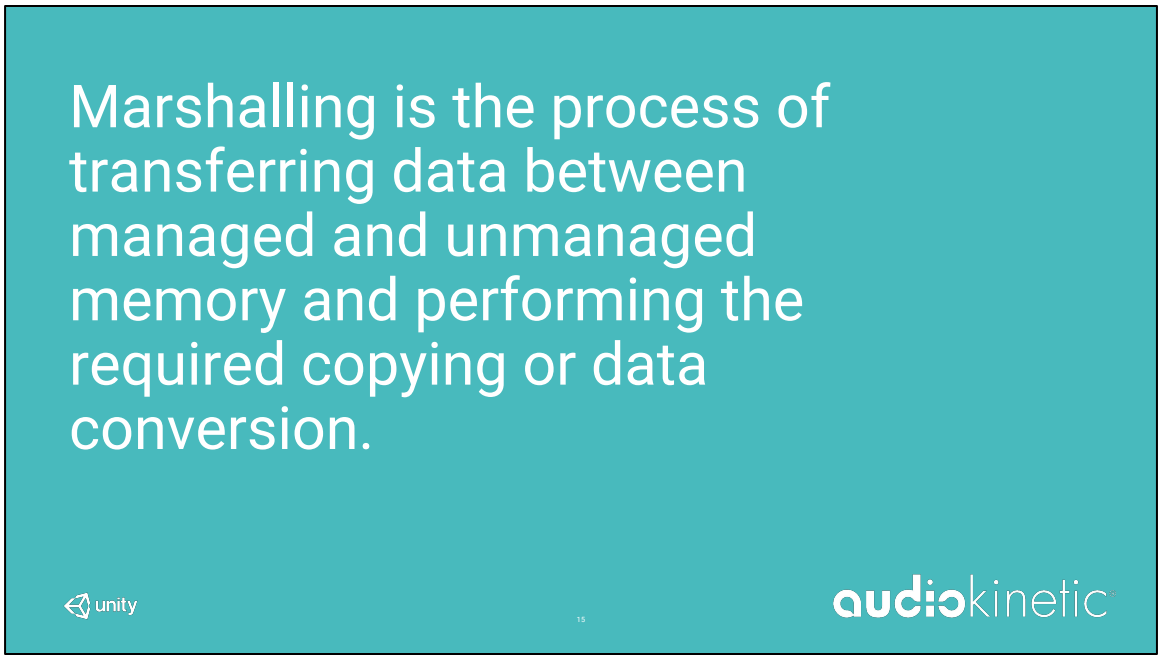
audiokinetic

Exceptions must not cross the managed/unmanaged boundary as this will likely lead to a crash that you will not be able to avoid by scoping the problematic code with a simple try-catch statement.

Advanced:
Exceptions are rarely used in games. Continue using best practices!

Calling functions is only half of the fun. Now, we're going to get into passing data to and from native functions.

Advanced:
https://docs.microsoft.com/en-us/dotnet/framework/interop/marshaling-data-with-platform-invoke



Marshalling is the process of transferring data between managed and unmanaged memory and performing the required copying or data conversion.

Primitive Data Types
Blittable Types

C# Primitives	Equivalent System Types	C/C++ Primitives	C++11 Fixed-Width Typedefs
byte	System.SByte	signed char	std::int8_t
ubyte	System.Byte	unsigned char	std::uint8_t
short	System.Int16	short	std::int16_t
ushort	System.UInt16	unsigned short	std::uint16_t
int	System.Int32	int	std::int32_t
uint	System.UInt32	unsigned int	std::uint32_t
long	System.Int64	long long	std::int64_t
ulong	System.UInt64	unsigned long long	std::uint64_t
	System.IntPtr	void*	std::intptr_t
	System.UIntPtr	void*	std::uintptr_t
float	System.Single	float	
double	System.Double	double	

unity

18

In the .NET framework, blittable types are data types that have an identical representation in memory for both native and managed land. They are essentially marshalled for free since data is simply copied.

Note that the managed types: bool (System.Boolean), string (System.String), and System.Char, are NOT listed here!

Also note that on certain platforms, "unsigned long" is 64-bit.
The cleanest way around the inconsistencies of the sizes of native integer types is to use typedefs in C/C++.

Advanced:
https://en.wikipedia.org/wiki/Blittable_types

https://docs.microsoft.com/en-us/dotnet/framework/interop/blittable-and-non-blittable-types

Complex Blittable Types
Blittable Types

- One-Dimensional Arrays of Blittable Types
- Structs Containing Only Blittable Types

unity

There are also complex blittable types which are simply composites of blittable types. These complex blittable types include one-dimensional arrays of blittable types and structs containing blittable types.

Non-Blittable Types
~~Blittable Types~~

- All Other Types!!!

unity

All other types are considered non-blittable.
For non-blittable types, P/Invoke provides special attributes that can be used to aid in data conversion.

The only non-blittable type that I will discuss today, at length, is string.

To reiterate, when the bitwise representation of data is the same in managed and native land, it can be VERY easy to work with the data.
The syntax becomes concise and uncluttered, and few if any extra P/Invoke attributes are required.

Marshalling Primitive Data Types

unity

audiokinetic®

Marshalling Primitive Data Types
Actual code!

```
using System.Runtime.InteropServices;
public partial class NativePlugin
{
  [DllImport(PluginName)]
  public static extern
  float GetDataByReturnValue();

  [DllImport(PluginName)]
  public static extern
  void GetDataByReference(ref float data);

  [DllImport(PluginName)]
  public static extern
  void SetData(float data);
}
```

```
static float Data = 90000.0f;

DLL_EXPORT
float CALL_CONV GetDataByReturnValue()
{ return Data; }

DLL_EXPORT
void CALL_CONV GetDataByReference(float& data)
{ data = Data; }

DLL_EXPORT
void CALL_CONV SetData(float data)
{ Data = data; }
```

unity

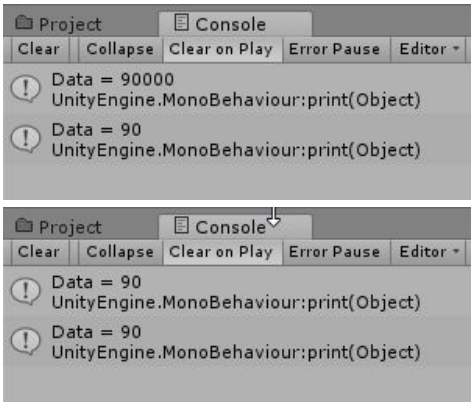
Here, we have some native data that is statically initialized to 90,000 and exposed via a few functions.
[Click]
Here is the corresponding managed code that grants access to this data.

Please observe that the functions exposed are simply accessors to our native data.

Advanced:
Take note that, for the function that is returning a value by reference, we could replace "ref" with "out" - possibly making for more efficient code.

Accessing Primitive Data Types in Unity
M04R C0D3!!!

```
public class MarshallingPrimitiveDataTypesTest :
UnityEngine.MonoBehaviour
{
  private void Start()
  {
    float d = NativePlugin.GetDataByValue();
    print("Data = " + d); // Data = ???

    NativePlugin.SetData(90);
    NativePlugin.GetDataByReference(ref d);
    print("Data = " + d); // Data = 90
  }
}
```



unity

24

The previous code can be used like this...

I retrieve the value of our native variable “Data” in the managed variable “d” using the GetDataByValue function.
After printing the value to the console, I then set the data to 90.
I retrieve the value in “d” a final time using GetDataByReference and then print it out again.

[Click]
On first run, it results in this console output.

[Click]
However, all subsequent runs would result in this.

This is due to static initialization.

Static Initialization
Gotcha

Static initialization of C/C++ variables occurs only once, when the plug-in is loaded.

Prepare for this or it can lead to unexpected behaviour!

unity

audiokinetic

Static initialization occurs only once and that is when the plug-in is loaded.
You must keep this in mind and ensure that data is reset or initialized adequately.

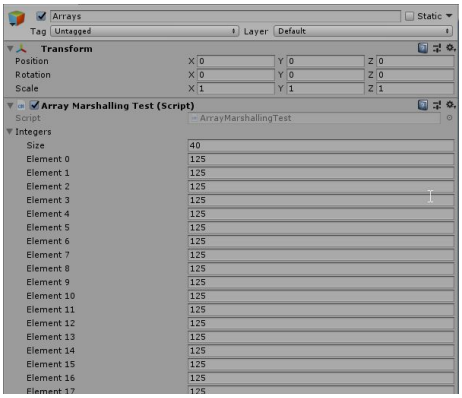
Marshalling
One-Dimensional Arrays

unity

audiokinetic®

Marshalling One-Dimensional Arrays
Pseudo-code

```
public class NormalUnityComponent :
UnityEngine.MonoBehaviour
{
  public int[] Integers = new int[40];

  private void Start()
  {
    FillArray(Integers, 125);
  }
}
```



unity

Let's say that we have a wicked large array of data, like this managed array of 40!!! ints, that we want to fill with the value 125, natively.

Marshalling One-Dimensional Arrays
Interoperability

```
using System.Runtime.InteropServices;

public partial class NativePlugin
{
  [DllImport(PluginName, EntryPoint = "FillArray"))]
  private static extern int FillArrayPrivate(int[] integers, int size, int v);
  public static int FillArray(int[] integers, int v)
  { return FillArrayPrivate(integers, integers.Length, v); }
}
```

```
DLL_EXPORT int CALL_CONV FillArray(int* integers, int size, int v)
{
  if (!integers || size < 1)
    return 0;
  // ...
}
```

unity

29

To obtain functionality like this, we'll create and expose a native function. To be safe, this native function must do its best to verify its arguments by, for instance, performing null pointer checks.
[Click]
Here, I have exposed the function to C# under a different name, [Click] so that I can then provide a function with the same name that is more suited to my needs.

Special care is needed to ensure that the contracts between callee and caller are respected.
If the C# FillArray function provides the only access point to the C/C++ FillArray function then we can carefully craft interactions between managed and native code, reducing or even eliminating the range of arguments to be verified.

Advanced:
https://docs.microsoft.com/en-us/dotnet/framework/interop/default-marshaling-for-arrays

Access Violations
Gotcha

Buffer overruns and dereferenced null-pointers can lead to serious problems.

Exceptions that could have been caught in C# may cause Unity and your application to crash.

Bounds checks and null-pointer checks are your friends!

In general, don't hold onto pointers received from managed land!

unity

audiokinetic

In native code, access violations, such as buffer overruns, buffer underruns and dereferencing invalid pointers, will crash your application.
Be sure to add argument verification code where appropriate.

In general, don't hold onto pointers received from managed code since the managed representation is not always guaranteed to be "pinned" down to a specific memory location.

Marshalling Enums

Marshalling Enums
Pseudo-code

```
// C# Pseudo-code
public class NormalUnityComponent :
UnityEngine.MonoBehaviour
{
  private void Start()
  {
    print(GetDayOfTheWeek());
  }
}
```

```
// Valid C/C++ code
enum WeekDay
{
  Sunday,
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday
};

WeekDay GetDayOfTheWeek();
```

unity

On the right, in C++, we have an enum that represents the days of the week and a function that returns the current day of the week, and we want to expose this functionality to C#.

Marshalling Enums
Bit representation

```
// Production ready Unity C# code
public enum WeekDay : int
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
}

public partial class NativePlugin
{
   [System.Runtime.InteropServices.DllImport(PluginName)]
   public static extern WeekDay GetDayOfTheWeek();
}
```

```
// Production ready C/C++ code
enum WeekDay : int
{
   Sunday,
   Monday,
   Tuesday,
   Wednesday,
   Thursday,
   Friday,
   Saturday
};

DLL_EXPORT WeekDay CALL_CONV GetDayOfTheWeek();
```

unity

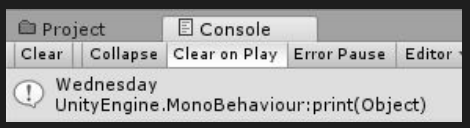
Marshalling enums is actually quite simple so long as you understand that:
1. enums are simply a fancy way of representing integers, and that
2. the bitwise representation needs to be the same in native and managed code.

[Click]
In both native and managed code, the enum is declared so that its underlying type is a 32-bit integer.
If I had not explicitly specified the underlying type on the native side, the native compiler could have been free to reduce the size of the enum to the smallest integer type that would fit all of its enumerators; whereas, on the managed side, a size of 32-bits is the default.

Advanced:
https://en.cppreference.com/w/cpp/language/enum

Marshalling Enums
Example code

```
using System.Runtime.InteropServices;
public class MarshallingEnumsTest : UnityEngine.MonoBehaviour
{
  [DllImport(NativePlugin.PluginName)]
  private static extern WeekDay GetDayOfTheWeek();

  private void Start()
  {
    WeekDay day = GetDayOfTheWeek();
    print(day);
  }
}
```




```
DLL_EXPORT WeekDay CALL_CONV GetDayOfTheWeek()
{ return Wednesday; }
```

unity

Here, we call the function resulting in the expected output. For brevity, the native implementation is rudimentary.

Marshalling Structs

Marshalling Structs
Pseudo-code

```
using UnityEngine;

public class NormalUnityComponent : MonoBehaviour
{
  private void Start()
  {
    Vector3 vec = GetVectorFromCppLand();
    // ...
    DoSomethingWithAVectorInCppLand(vec);
  }
}
```

unity

As with enums, marshalling of structs of blittable types can be easily dealt with. The default layout of a structure in C# is compatible with structs in C/C++.

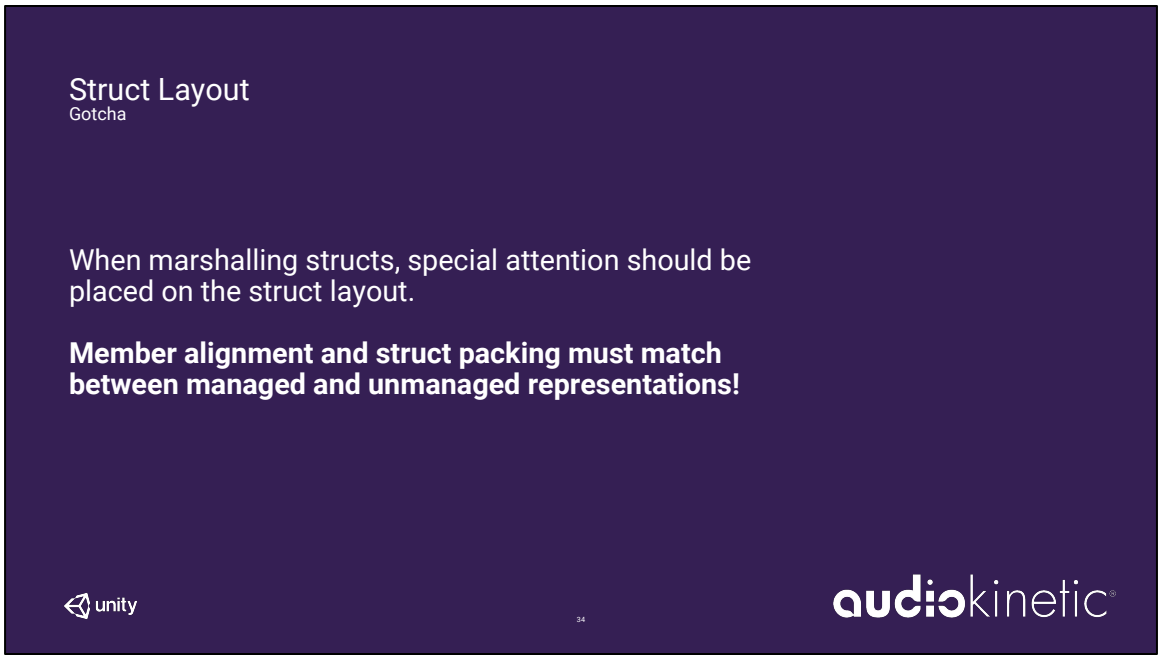
Marshalling Structs
Structure Layout

```
// Unity C#
namespace UnityEngine
{
  public struct Vector3 /* ... */
  {
    // ...
    public float x;
    public float y;
    public float z;
    // ...
  }
}
```

```
// C/C++
struct V3
{
  float X;
  float Y;
  float Z;
};
```

unity

For example, Unity's Vector classes can be easily represented in native code.
[Click]
Only the non-static, non-const fields of managed structs are taken into account for their bitwise representation.



When marshalling structs, special attention should be placed on the struct layout. Member alignment and struct packing must match between managed and native representations!

Advanced:
http://www.catb.org/esr/structure-packing/
Depending on compilation settings, enums can have different sizes.

C# Proxy For A C/C++ Object

unity

audiokinetic

Let's imagine that we wanted to expose a C++ class to C# with similar syntax so that coders in either language can feel comfortable with the interface.
This can all be done using the proxy design pattern combined with elements of the previous examples.

“Marshalling” C++ Classes

Exposing similar syntax

```
class CppClass
{
public:
  float Value = 0.0f;

  int Function()
  { return ++InternalCounter; }

private:
  int InternalCounter = 0;
};
```

unity

In this example, we have a basic class that exposes one public member variable and one public member function and hides some internal state.

“Marshalling” C++ Classes
Allocation/Deallocation

```
public partial class CppClassProxy
{
  private System.IntPtr CppPtr = System.IntPtr.Zero;
  public CppClassProxy()
  { CppPtr = NativePlugin.CreateCppClass(); }

  ~CppClassProxy()
  { NativePlugin.DestroyCppClass(CppPtr); CppPtr = System.IntPtr.Zero;  }
}
```

```
DLL_EXPORT CppClass* CALL_CONV CreateCppClass()
{ return new CppClass(); }

DLL_EXPORT void CALL_CONV DestroyCppClass(CppClass* c)
{ delete c; }
```

unity

I create native functions that allow me to allocate and deallocate instances of this type.
[Click]
Not seen here, I expose these functions to C# using P/Invoke.
I, then, create a proxy class that has a single data member responsible for maintaining a handle to the natively allocated data structure.

Advanced:
https://ericlippert.com/2015/05/18/when-everything-you-know-is-wrong-part-one/

“Marshalling” C++ Classes
Accessors

```
public partial class CppClassProxy
{
  public float Value
  {
    get { return NativePlugin.CppClass_Value_get(CppPtr); }
    set { NativePlugin.CppClass_Value_set(CppPtr, value); }
  }
}
```

```
DLL_EXPORT float CALL_CONV CppClass_Value_get(CppClass* c)
{ return c ? c->Value : 0.0f; }

DLL_EXPORT void CALL_CONV CppClass_Value_set(CppClass* c, float v)
{ if (c) c->Value = v; }
```

unity

I expose the Value member variable by adding the accessors functions natively.
[Click]
And then creating a property in C# that calls the appropriate accessor.

“Marshalling” C++ Classes
Member Functions

```
public partial class CppClassProxy
{
  public int Function()
  {
    return NativePlugin.CppClass_Function(CppPtr);
  }
}
```

```
DLL_EXPORT int CALL_CONV CppClass_Function(CppClass* c)
{ return c ? c->Function() : 0; }
```

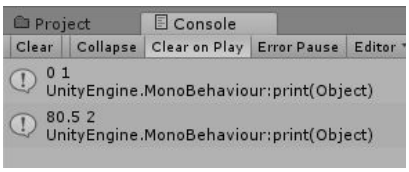
unity

The single member function is exposed by writing a new function that calls it.
On either the native side, managed side or both, null pointers are handled, arguments and return values are possibly converted and exceptions are possibly handled.

Accessing C++ Classes in Unity
Sample use and output

```
public class MarshallingStructuresTest02 :
UnityEngine.MonoBehaviour
{
  private void Start()
  {
    var c = new CppClassProxy();
    print(c.Value + " " + c.Function());
    c.Value = 80.5f;
    print(c.Value + " " + c.Function());
  }
}
```

```
struct CppClass
{
  float Value = 0.0f;
  int InternalCounter = 0;

  int Function()
  { return ++InternalCounter; }
};
```



unity

All of this results in a new C# proxy class that is syntactically and semantically similar in use to its C/C++ counterpart.

Null Pointers, Memory Leaks, and Referencing Deallocated Data
Gotchas

Null checks are important.

It is also important to ensure that allocated data is appropriately deallocated.

Memory management is now in your hands!
Get dirty, but remember to clean yourself up!

unity

audiokinetic

These, right here, are all of the GOTCHAS that you may have gotten used to not being too concerned with when dealing with C#. You are left on your own to manage memory. So be cautious, stay safe, stay clean and TEST!

Advanced Topics

0NLy 4 3L337 H4X0RZ

unity

audiokinetic

Now for a few advanced topics that are reserved for only the most 3L337 H4X0RZ.

Marshalling Strings

unity

audiokinetic®

Strings.

Marshalling Strings
Pseudo-code

```
// C# Pseudo-code
public class MagicalTextComponent :
UnityEngine.MonoBehaviour
{
  private void Start()
  {
    print(MakeTextMagical("TEXT"));
  }
}
```

unity

44

Let's imagine a scenario where we have a native function that receives the equivalent of a string and returns another string - possibly doing something wonderful and magical in between. [Click]

Advanced:

```
DLL_EXPORT const char* CALL_CONV MakeTextMagical(const char* text)
{
	static std::string textBuffer;
	if (text)
		textBuffer = std::string("<color=#FF0000>****</color>
<color=#0000FF>MAGICAL ") + text + std::string("</color>
<color=#FF0000>****</color>");
	else
		textBuffer.clear();

	return textBuffer.c_str();
}
```

Marshalling Single-Byte Character Strings

Interoperability

```
using System.Runtime.InteropServices;

public partial class NativePlugin
{
  [DllImport(PluginName, EntryPoint = "ModifyText")]
  private static extern System.IntPtr ModifyTextImpl([MarshalAs(UnmanagedType.LPStr)] string text);

  public static string ModifyText(string text)
  { return Marshal.PtrToStringAnsi(ModifyTextImpl(text)); }
}
```

```
DLL_EXPORT const char* CALL_CONV ModifyText(const char* text)
{
  return DoTextMagic(text);
}
```

unity

51

First, we'll take on single-byte character strings. With a function that receives a null-terminated char buffer and returns one, the C# representation looks like this.

The string argument, that is passed to the function from the managed side, is decorated with the "MarshalAs" attribute so that it can be seen as the appropriate type on the native side, an LPStr - the .NET name for a native single-byte null-terminated character string.

Since the char* being returned can be easily marshalled as a System.IntPtr, we let that be... [Click] And simply let another PInvoke service do the work. "Marshal.PtrToStringAnsi" does exactly what it's name crudely implies. It takes an IntPtr argument

that was an "ANSI" character buffer and converts it to a managed string.

Marshalling Multi-Byte Character Strings
Interoperability

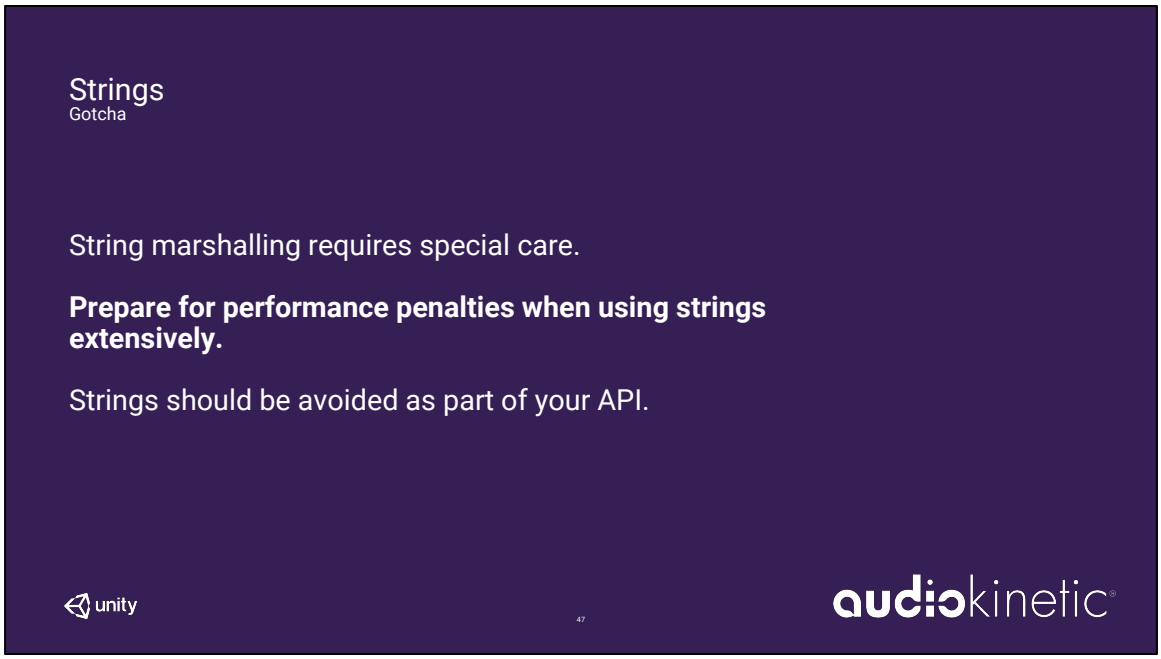
```
using System.Runtime.InteropServices;

public partial class NativePlugin
{
  [DllImport(PluginName, EntryPoint = "ModifyMultiByteText")]
  private static extern System.IntPtr ModifyText2Impl([MarshalAs(UnmanagedType.LPWStr)] string text);

  public static string ModifyMultiByteText(string text)
  { return Marshal.PtrToStringUni(ModifyText2Impl(text)); }
}
```

```
DLL_EXPORT const wchar_t* CALL_CONV ModifyMultiByteText(const wchar_t* text)
{
  return DoTextMagicW(text);
}
```

unity

For multi-byte strings, the technique is very similar.

The differences being that the IntPtr return value must be converted using "Marshal.PtrToStringUni" and that LPWStr is used instead of LPStr.



In Unity, there are performance issues when working with strings. These and the complications related to marshalling strings simply make dealing with them a pain.

Don't do it, if it can be avoided.

Advanced:
https://docs.unity3d.com/Manual/BestPracticeUnderstandingPerformanceInUnity5.html

Marshalling Delegates

unity

audiokinetic®

Marshalling Delegates
Interoperability

```
using System.Runtime.InteropServices;
public partial class CallbackMarshallingTest : MonoBehaviour
{
   [UnmanagedFunctionPointer(CallingConvention.StdCall)]
   private delegate void LoggerInteropDelegate([MarshalAs(UnmanagedType.LPStr)] string message);

   [DllImport(NativePlugin.PluginName)]
   private static extern void DoExtensiveWork(LoggerInteropDelegate logger, int steps);
}
```

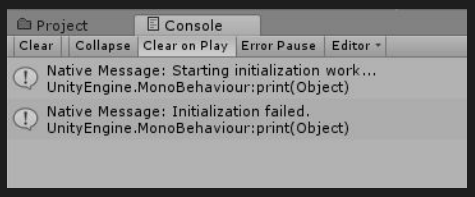
```
typedef void(CALL_CONV * Logger)(const char*);

DLL_EXPORT void CALL_CONV DoExtensiveWork(Logger logger, int steps)
{
   logger("Starting initialization work...");
   if (TestFailure(steps))
      logger("Initialization failed.");
}
```

unity

Let's say that we have a native function that we have exposed to managed code, and we want to pass a logger function as an argument to this function so that we can get a handle on what is happening inside it.
[Click]
We need to determine both the native and managed function signature of this logger, ensuring that all parameters are marshalled appropriately.
Please note that the managed delegate type is decorated with the UnmanagedFunctionPointer attribute which explicitly specifies that the delegate will be marshalled as a native function pointer.

Delegates as C/C++ Callbacks
Example

```
public partial class CallbackMarshallingTest : MonoBehaviour
{
  private LoggerInteropDelegate Logger = new LoggerInteropDelegate(LogFunction);

  [AOT.MonoPInvokeCallback(typeof(LoggerInteropDelegate))]
  private static void LogFunction(string message)
  {
    print("Native Message: " + message);
  }

  private void Start()
  {
    DoExtensiveWork(Logger, 0);
  }
}
```

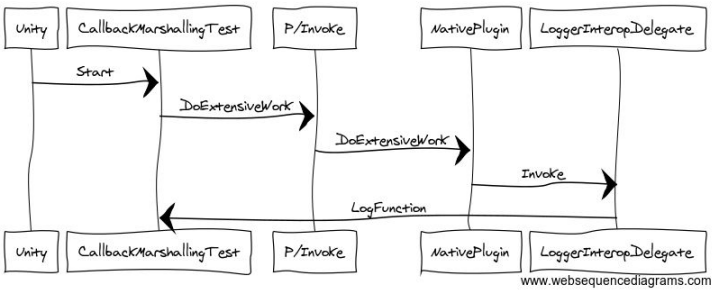


unity

50

In this example, the private Logger field is set to our managed LogFunction which is decorated with the AOT.MonoPInvokeCallback attribute.
This attribute is required to ensure that the function is compiled ahead-of-time on systems that do not support just-in-time compilation like iOS, XBoxOne and many more.
[Cick]
Finally, calling our C/C++ function in turn calls our delegate logger function resulting in the expected output.

Delegates as C/C++ Callbacks
Interaction Diagram



unity

Here’s the interaction diagram for the previous example.

Delegated Callback Problem Solver
Gotcha

- Unity's multithreading restrictions
- Garbage Collection

Devs should avoid passing managed delegates as native callbacks when they are uncertain of the ramifications!

unity

audiokinetic

Delegates have delegated you the callback problem solver. It is your responsibility to understand whether the delegate that you are sending to native code will be called from other threads or in time sensitive cases.
If called from other threads, as with regular Unity threading restrictions, these delegates must not make calls to gameObject APIs.
If called from a time sensitive context, one must be aware that GC can occur simply because managed code is being run!

Automation

unity

audiokinetic®

Automation

- SWIG
  (Simplified Wrapper and Interface Generator)
- CppSharp

unity

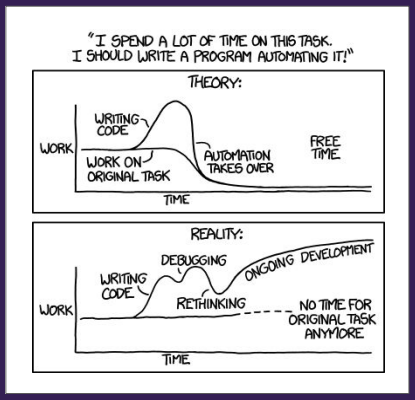
So after all of these examples and code snippets, you are now closer to being able to do this by yourself. But you wonder if you could possibly get all of this done for free.

There are a few products that exist to fill this need: [Click]
SWIG, the Simplified Wrapper and Interface Generator [Click]
and
CppSharp

These tools offer free interface code generation at the expense of readability. Often this doesn't matter, cause when it works, it works...

Automation is No Panacea
Gotcha

Hours can be wasted fighting against your automation tool to consistently produce the required output when it might be expedient to simply handcraft a solution for specific edge cases.



https://xkcd.com/1319/

unity

55

audiokinetic

Until it doesn't work.

Automation is no silver bullet. Sometimes the best thing to do is roll your own solution. That's why it's good to have these tips that I've laid out for you at your disposal.

Advanced:
https://xkcd.com/1319/
From http://www.swig.org/exec.html:
Extensibility. SWIG provides a variety of customization options that allow you to blow your whole leg off if that's what you want to do. SWIG is not here to enforce programming morality.

Conclusion

https://github.com/lazerfalcon/Unite2018_Native

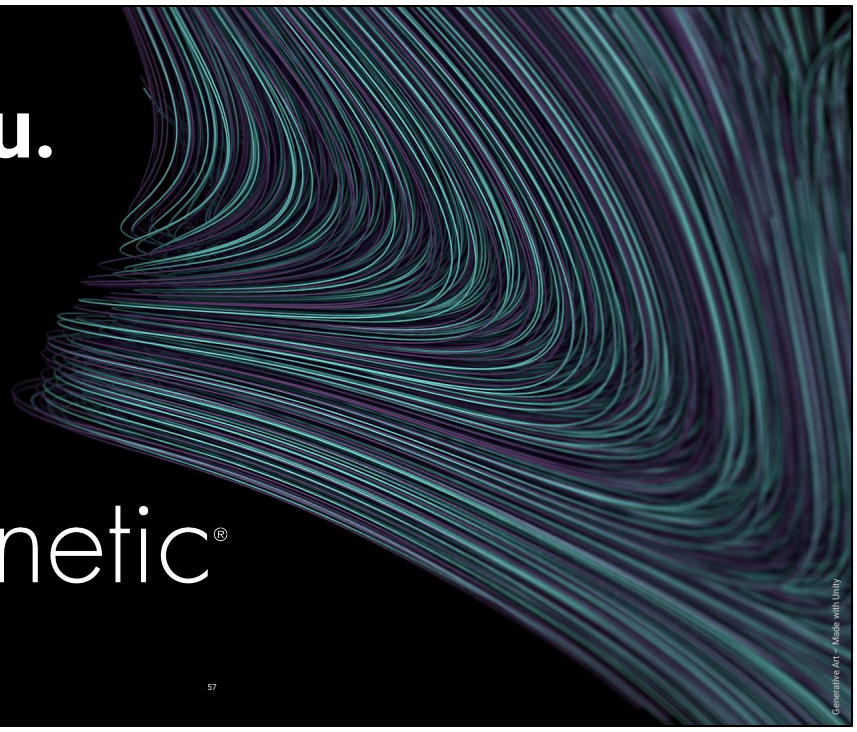
- Marshalling
  - Primitive Types
  - One-Dimensional Arrays
  - Enums
  - Structures
  - Strings
  - Delegates as Callbacks
- Gotchas
  - Native Plug-ins are not unloaded
  - Exceptions
  - Static Initialization
  - Use Strings sparingly
  - Memory Management
  - Access Violations
  - ...

unity

audiokinetic

In conclusion, I've spoken to you about how to marshal data, and I've discussed a number of peculiarities and problems that you will likely run into if you are working with native code.
Working with native code has the potential to offer significant benefits; however, the onus is on you to ensure that data is handled with care. Please play safe and worry less while you expose C/C++ to Unity.

Thank you.



audiokinetic®

unity