

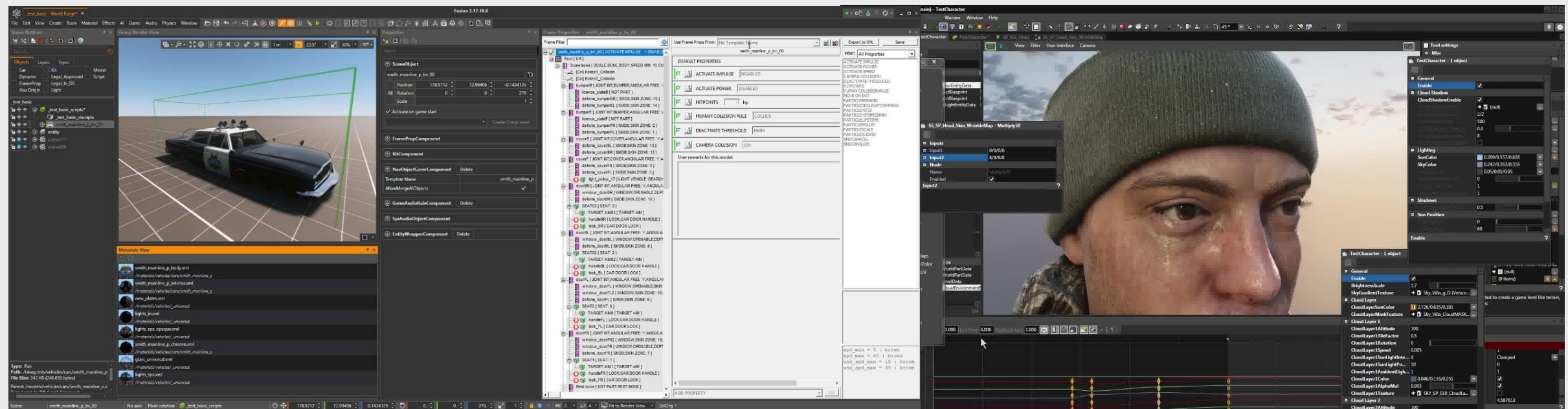
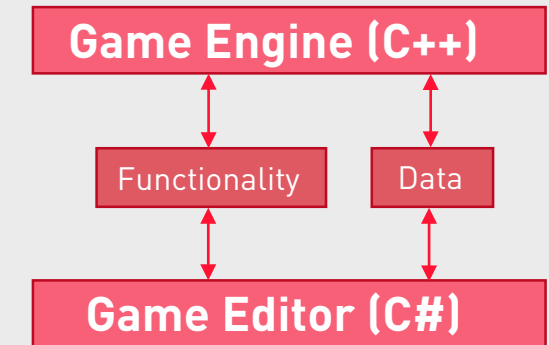


# C++/C# Interoperability

# C++/C# Interop

## WHAT? WHY?

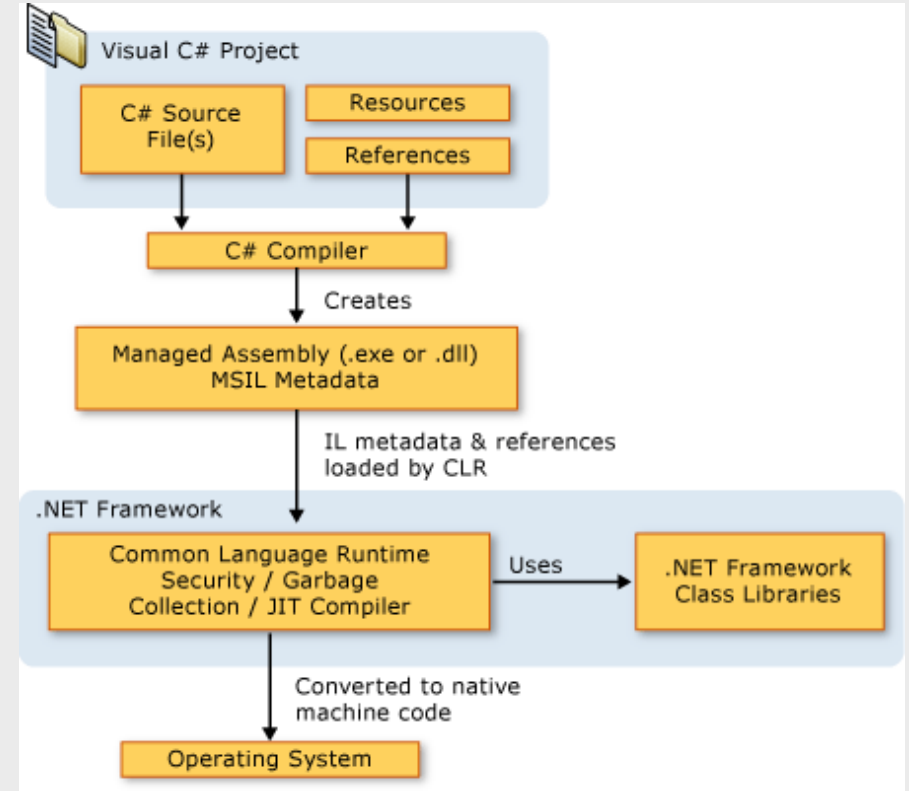
- Sometimes we want code that has been written and compiled in **different programming languages** to communicate with each other.
- Common example: game engine editors, hardware library integration in game engine (eg. Unity), etc.



# C++/C# Interop

## WHAT? WHY?

- C++ (unmanaged language) versus C# (managed language)
- C#
  - Compiled to **Intermediate Language** (IL) → loaded into CLR (**Common Language Runtime**) → JIT (**Just-In-Time**) compilation to native machine code.
  - CLR provides functionality for the GC (Garbage Collector), Exception Handling and Resource Management.
  - Windows specific! (.NET framework).
    - But what about Unity and our C# scripts that runs on other platforms?
      - IL2CPP AOT (Ahead-Of-Time) compiler translates the IL to C++ code ☺
- C++
  - Compiles to native machine code straight away.
  - Supports inline assembly.



<https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework>

# C++/C# Interop

## WHAT? WHY?

- There are two ways to let C# and C++ communicate with each other:
  - Explicit Platform Invoke (**PInvoke**)
    - Works on multiple platforms.
    - No need to have the source code of the library.
  - Implicit Platform Invoke (**C++/CLI** (Common Language Interface))
    - C++ modified for CLI (specification developed by Microsoft).
    - Only supported on Windows.
    - Smaller overhead of marshalling, so better performance.
    - Better type safety.
    - More forgiving if the API is modified.
- So which one do we pick?
  - In most game companies they use **Explicit PInvoke**. Why?
    - Not platform specific.
    - No real need for an intermediate library (you can directly call the managed functions).
    - We don't care about some performance hits → It's only the editor.

```
struct Vector3
{
    float x, y, z;
    Vector3(float _x, float _y, float _z):x(_x),y(_y),z(_z)
    {}
};

__declspec(dllexport) Vector3 __stdcall MarshalStructureReturn()
{
    return Vector3(32.f, 64.f, 128.f);
}

__declspec(dllexport) void __stdcall MarshalStructureOutput(Vector3& v)
{
    v = Vector3(32.f, 64.f, 128.f);
}
```

```
#include "stdafx.h"
using namespace System;

int main(array<System::String ^> ^args)
{
    array<Byte>^ byteArray = gcnew array<Byte>(10);

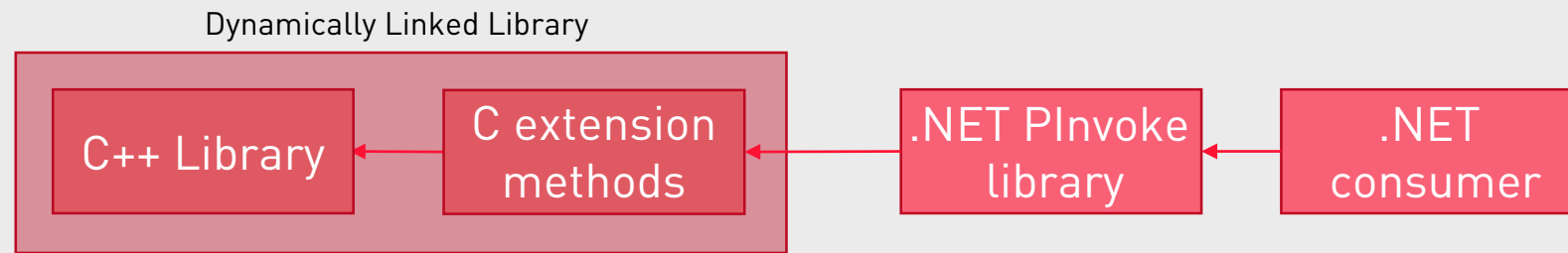
    for (short i = 0; i < byteArray->Length; ++i)
    {
        byteArray[i] = static_cast<Byte>(i);
    }

    return 0;
}
```



# C++/C# Interop

## PInvoke - Layout

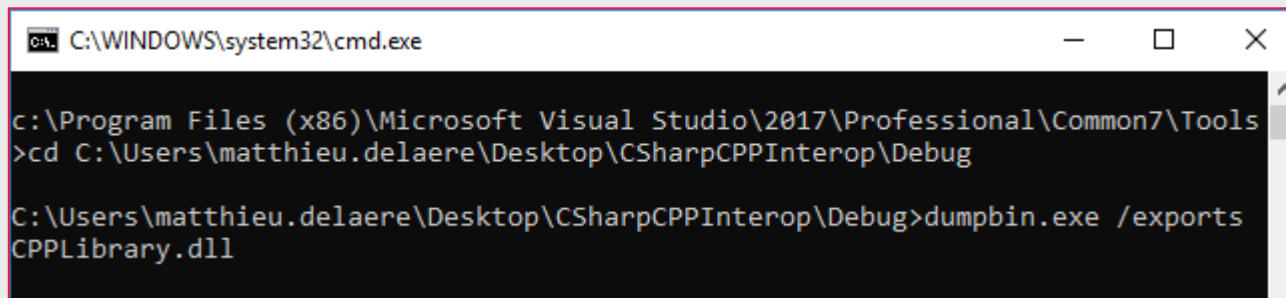


- PInvoke project consists out of at least three parts:
  - C++ Library (DLL).
  - C Wrapper (exposing the functions from the library).
  - .NET library or project that marshals calls from .NET to the C Wrapper and vice versa.
- It is possible to skip the C Wrapper, but this would result in platform- and compiler dependent solution! → **BAD!**
- Why?
  - There is **no common** specification for the **Application Binary Interface (ABI)** of C++ methods!
  - **ABI** defines among other things: **how parameters are passed** to functions (registers/stack), **who cleans** parameters from the stack (caller/callee), **where return value is placed**, exception propagation, etc.
  - Where an **API** exposes the public types/variables/function from a library or application.

# C++/C# Interop

## PInvoke - Name Mangling

- C++ compilers use **name mangling** to encode the types of parameters and return values of functions into exported symbol names.
- A PInvoke call cannot accept multiple symbol names, so we would only be able to support one type of C++ compiler.
- **Solution:** C Wrapper → will assign an unambiguous symbol name to each function
  - Be warned, comes at the cost of **type safety** in the ABI since the parameters and return types are no longer encoded in the symbol name!
- How can we see the exported symbol names of our functions?
  - Tools/Visual Studio Command Prompt → **dumpbin.exe**.
    - Put path to location of the CPPLibrary.dll using the **cd** command.
    - Execute **dumpbin.exe** with the **/exports** option for the **CPPLibrary.dll**.



```
C:\WINDOWS\system32\cmd.exe

c:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\Common7\Tools
>cd C:\Users\matthieu.delaere\Desktop\CSharpCPPInterop\Debug

C:\Users\matthieu.delaere\Desktop\CSharpCPPInterop\Debug>dumpbin.exe /exports
CPPLibrary.dll
```

# C++/C# Interop

## PInvoke - Name Mangling

- Before we can start inspecting the exported names of our function, we need to tell the compiler we want to export a function. We can do this by using the following attribute: **`__declspec( extended-decl-modifier-seq )`**
- To specify we want to export our functions (with parameters, data, objects, etc.) we need to use the following specifier: **`dllexport`**
- There is also the option to import functions: **`dllimport`**
- Hence we get the following global function for example:

```
__declspec(dllexport) int AddCPPMangling(int a, int b)
{ return a + b; }
```

- Let's inspect the exported symbol name:

```
ordinal hint RVA      name
1      0 00011262 ?AddCPPMangling@CppLibrary@@YAHHH@Z = @ILT+605(?AddCPPMangling@CppLibrary@@YAHHH@Z)
```

# C++/C# Interop

## Pinvoke - Name Mangling

- Let's inspect the exported symbol name:

```
ordinal hint RVA      name
1      0 00011262 ?AddCPPMangling@CppLibrary@@YAHHH@Z = @ILT+605(?AddCPPMangling@CppLibrary@@YAHHH@Z)
```

- What is this... 😞
  - This is the result of our C++ Compiler. In this symbol name there are a few things encoded...
  - ?AddCPPMangling@CppLibrary: **function** called AddCPPMangling which is part of the **namespace** CppLibrary.
  - @@: end of class name
  - Y: **Type code** → 'Y' == Unqualified FunctionTypeCode
  - A: **Calling convention** → 'A' == \_\_cdecl (default C/C++ calling convention)
  - HHH: **return type, arguments list types** → 'H' == int → int(int, int)
  - @Z: end of arguments list

[https://github.com/gchatelet/gcc\\_cpp\\_mangling\\_documentation](https://github.com/gchatelet/gcc_cpp_mangling_documentation) & <http://www.kegel.com/mangle.html>



# C++/C# Interop

## PInvoke - Calling Conventions

- Ok, we can pretty much understand all these encoded symbols, but what is this thing called **Calling Conventions**?
- Calling conventions describe **the interface of called code**. It specifies the method that a compiler sets up to **access a subroutine**. So what does it describe?
  - **Where parameters, return values and return addresses are placed** (registers vs stack vs mix).
  - In case we use **registers**, which registers must be **reserved for the caller** (known as: callee-saved registers).
  - The **order** parameters are passed.
  - How the task of **setting up for and cleaning after** a function call is **divided** between the caller and the callee.
  - **Where** the **previous frame pointer is stored** (Game Tech 1 ☺).
  - ...
- <https://docs.microsoft.com/nl-nl/cpp/cpp/calling-conventions?view=vs-2017>

# C++/C# Interop

## PInvoke - Calling Conventions

- The following calling conventions are supported by the Visual C/C++ compiler:

Keyword	Stack cleanup	Parameter passing
<b>__cdecl</b>	caller	Pushes parameters on the stack, in reverse order (right to left)
__clrcall	n/a	Load parameters onto CLR expression stack in order (left to right)
<b>__stdcall</b>	callee	Pushes parameters on the stack, in reverse order (right to left)
__fastcall	callee	Stored in registers, then pushed on stack
__thiscall	callee	Pushed on stack, <b>this</b> pointer stored in ECX
__vectorcall	callee	Stored in registers, then pushed on stack in reverse order (right to left)

- By default the Visual C/C++ compiler uses `__cdecl` (x86). Let's change this to `__stdcall` and inspect the symbol name:

```
__declspec(dllexport) int __stdcall AddCPPMangling(int a, int b)
{ return a + b; }
```

ordinal	hint	RVA	name
1	0	000110FA	?AddCPPMangling@CppLibrary@@YGHH@Z

# C++/C# Interop

## PInvoke - Name Mangling

- So we have our exported C++ function. How can we use this in C#?
- Well in C# we need to **import** this DLL function and “bind” it to a function inside C#.

```
using System.Runtime.InteropServices;
```

```
[DllImport("CPPLibrary.dll",  
    EntryPoint = "?AddCPPMangling@CppLibrary@@YGHH@Z",  
    CallingConvention = CallingConvention.StdCall)]  
1 reference  
static public extern int AddCPPMangling(int a, int b);
```

- To be able to call our function we need to define the **EntryPoint**, which is our symbol name. We also define our **CallingConvention** (by default in C# this is StdCall).
  - **WARNING:** if you use the default calling conventions you have a mismatch!  
C++{\_\_cdecl} vs C# {\_\_stdcall}.
- Using this encoded symbol name from our C++ compiler is kind of annoying right? Every time we change the name, parameters, return value, etc we need to figure out the encoded name and change the entry point as well as the function definition!  
Can we avoid this?
- **YES WE CAN!**

# C++/C# Interop

## PInvoke - Name Mangling

- We can force the compiler to export the function as it was a **C function** instead of C++. Basically preventing C++ mangling!

To do this add the following specifier: **extern "C"**

```
extern "C" __declspec(dllexport) int __stdcall AddCMangling(int a, int b)
{ return a + b; }
```

- This results in the following symbol name:

ordinal	hint	RVA	name
1	0	0001128F	_AddCMangling@8 = @ILT+650(_AddCMangling@8)

- This is better, but still some elements are encoded in the symbol name:
  - A leading underscore followed by the function name.
  - Total amount of bytes of the parameters (in this case 2 integers == 8 bytes, hence @8).

# C++/C# Interop

## PInvoke - Name Mangling

- We can plug this into C# using the same technique (using the symbol name).

```
[DllImport("CPPLibrary.dll",  
    EntryPoint = "_AddCMangling@8",  
    CallingConvention = CallingConvention.StdCall)]  
1 reference  
static public extern int AddCMangling(int a, int b);
```

- What if I tell you this also works:

```
[DllImport("CPPLibrary.dll",  
    CallingConvention = CallingConvention.StdCall)]  
1 reference  
static public extern int AddCMangling(int a, int b);
```

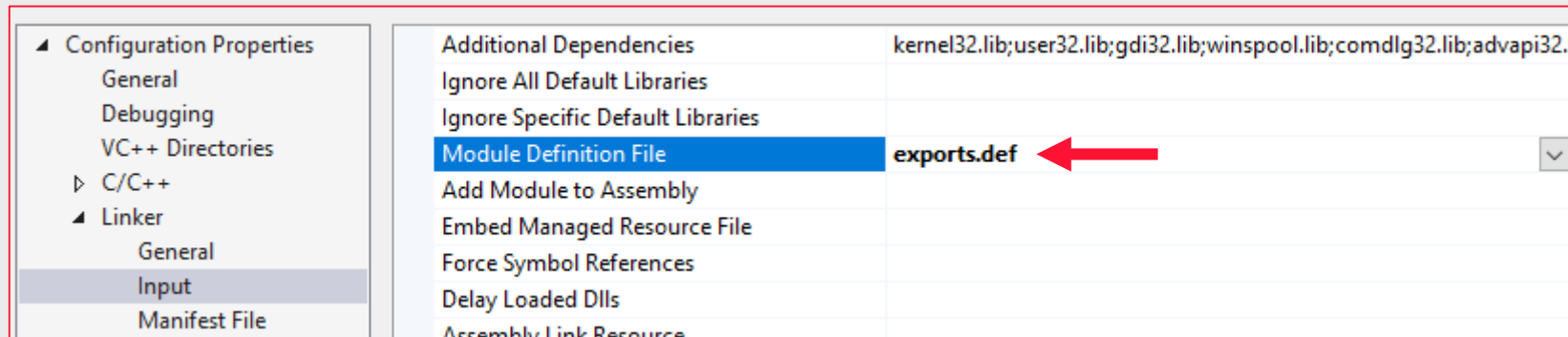
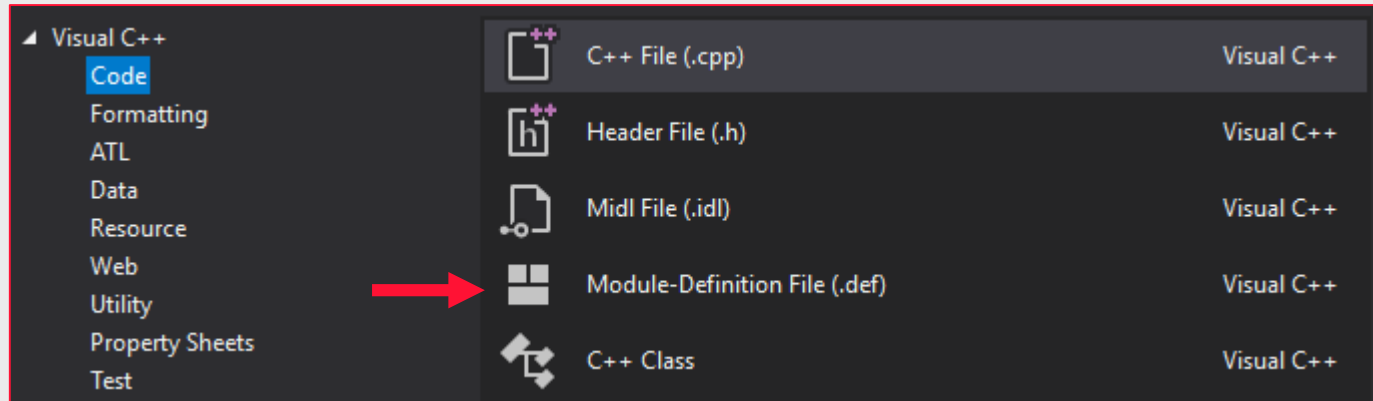
- If you export a function as a C function, **if both names match**, you don't have to explicitly identify the function with the EntryPoint.
- Why would you not match the names?
  - Different naming standards.
  - Avoid case-sensitive API function names.
  - Accommodate functions that take different data types.



# C++/C# Interop

## Pinvoke - Name Mangling

- If I don't match the names, is there a way to even further optimize this encoded symbol name?
- Use **.def** files!



# C++/C# Interop

## PInvoke - Name Mangling

- Now you can control the symbol name, without any “weird” symbols.

```
1 LIBRARY CPPLibrary
2 EXPORTS
3     AddCMangling @1
4
```

ordinal	hint	RVA	name
1	0	0001128F	AddCMangling = @ILT+650(_AddCMangling@8)

```
[DllImport("CPPLibrary.dll",
    EntryPoint = "AddCMangling",
    CallingConvention = CallingConvention.StdCall)]
1 reference
static public extern int SomeOtherRandomName(int a, int b);
```

- The C# code above will also work without the .def file, so what is the benefit of using this file?
  - It's an alternative to the `extern "C" __declspec(dllexport)`.
  - You can set other options and still have the code being readable.

```
int __stdcall AddCMangling(int a, int b)
{ return a + b; }
```

[https://docs.microsoft.com/en-us/previous-versions/28d6s79h\(v=vs.140\)](https://docs.microsoft.com/en-us/previous-versions/28d6s79h(v=vs.140))

# C++/C# Interop

## PInvoke - Classes

- Let's create a class with one member function in C++. Notice the **dllexport** specifier!

```
class __declspec(dllexport) SomeUnmanagedClass
{
public:
    int SomeFunction() { return 42; };
private:
    int m_SomeIntegerDatamember = 32;
    float m_SomeFloatDatamember = 64.0f;
};
```

ordinal	hint	RVA	name
1	0	0001112C	??0SomeUnmanagedClass@Cpplibrary@@QAE@XZ = @ILT+295(??0SomeUnmanagedClass@Cpplibrary@@QAE@XZ)
2	1	00011230	??4SomeUnmanagedClass@Cpplibrary@@QAEAAV01@@Z = @ILT+555(??4SomeUnmanagedClass@Cpplibrary@@QAEAAV01@@Z)
3	2	00011136	??4SomeUnmanagedClass@Cpplibrary@@QAEAAV01@ABV01@@Z = @ILT+305(??4SomeUnmanagedClass@Cpplibrary@@QAEAAV01@ABV01@@Z)
4	3	000110E6	?SomeFunction@SomeUnmanagedClass@Cpplibrary@@QAEHXZ = @ILT+225(?SomeFunction@SomeUnmanagedClass@Cpplibrary@@QAEHXZ)

- When inspecting the symbol names, we see **4 exported functions!** What are the other functions?
  - ??0** == constructor
  - ??4** == assignment operator
    - First: T& T::operator=(const T&)
    - Second: T& T::operator=(T)

# C++/C# Interop

## Pinvoke - Classes

- We don't want these C++ exported symbol names, we want to export them as C functions. What could be the problem?
  - There is **no such thing as classes in C**, so we can't export constructors and destructors as C functions!
- Solution:
  - Create some **extern C functions** that **creates and destroys an instance** of our class.  
→ This is our **C Wrapper**, often referred to as our **Bridge**!
- Disadvantages:
  - Explicitly have to call these functions. So **we depend on the user!**
  - **Type safety and exception handling.**
- We need to capture those functions in C# as we did before, but we also need to be able to store the **returned pointer** in a C# variable. Which type of variable, because C# doesn't have pointers?
  - → **IntPtr** == a struct that represents a pointer or a handle!  
<https://docs.microsoft.com/en-us/dotnet/api/system.intptr?view=netframework-4.7.2>

# C++/C# Interop

## PInvoke - Classes

```
class SomeUnmanagedClass
{
public:
    int SomeFunction() { return 42; };
private:
    int m_SomeIntegerDatamember = 32;
    float m_SomeFloatDatamember = 64.0f;
};

extern "C" __declspec(dllexport) SomeUnmanagedClass* __stdcall CreateSomeUnmanagedClass()
{
    return new SomeUnmanagedClass();
}

extern "C" __declspec(dllexport) void __stdcall DestroySomeUnmanagedClass(SomeUnmanagedClass* pObject)
{
    if(pObject)
    {
        delete pObject;
        pObject = nullptr;
    }
}
```

```
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
1 reference
static public extern IntPtr CreateSomeUnmanagedClass();

[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
1 reference
static public extern void DestroySomeUnmanagedClass(IntPtr pObject);
```

```
IntPtr pUnmanagedClassObject = CreateSomeUnmanagedClass();
DestroySomeUnmanagedClass(pUnmanagedClassObject);
pUnmanagedClassObject = IntPtr.Zero;
```

- This works like a charm, but how do we call our **member function** called "SomeFunction"?
  - Solution: create a C function where you pass a pointer to the object (held in C# → IntPtr) and call the member function.



# C++/C# Interop

## PInvoke - Classes

```
class SomeUnmanagedClass
{
public:
    int SomeFunction() { return 42; };
private:
    int m_SomeIntegerDatamember = 32;
    float m_SomeFloatDatamember = 64.0f;
};

extern "C" __declspec(dllexport) SomeUnmanagedClass* __stdcall CreateSomeUnmanagedClass()
{
    return new SomeUnmanagedClass();
}

extern "C" __declspec(dllexport) void __stdcall DestroySomeUnmanagedClass(SomeUnmanagedClass* pObject)
{
    if(pObject)
    {
        delete pObject;
        pObject = nullptr;
    }
}

extern "C" __declspec(dllexport) int CallSomeFunction(SomeUnmanagedClass* pObject)
{
    if (pObject)
        return pObject->SomeFunction();
    return 0;
}
```



```
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
2 references
static public extern IntPtr CreateSomeUnmanagedClass();

[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
1 reference
static public extern void DestroySomeUnmanagedClass(IntPtr pObject);

[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.ThisCall)]
2 references
static public extern int CallSomeFunction(IntPtr pObject);
```

```
IntPtr pUnmanagedClassObject = CreateSomeUnmanagedClass();
result = CallSomeFunction(pUnmanagedClassObject);
DestroySomeUnmanagedClass(pUnmanagedClassObject);
pUnmanagedClassObject = IntPtr.Zero;
```

- Why the **ThisCall** convention?

- Equal to StdCall but storing the **this** pointer in register ECX. So we don't have to change the calling convention in C++ ☺ But you can use any option you like as long as **the stack stays in balance!**

# C++/C# Interop

## PInvoke - Classes

- Being dependent on the user to create and destroy the unmanaged object is not a good idea. We can fix this by wrapping our functionality in a C# class!

```
public class SomeManagedClass
{
    private IntPtr _nativePtr;

    References
    public SomeManagedClass() //Constructor
    { _nativePtr = CreateSomeUnmanagedClass(); }

    References
    public int SomeFunction()
    { return CallSomeFunction(_nativePtr); }
}
```

- In our C# class we call our Create function in the **constructor**. But where do we call the Destroy function?
  - C# has a function called the **finalizer**, which is similar to a destructor in C++. This function is being called whenever the GC wants to destroy the object. A few remarks:
    - Cannot be define in a struct and a class can only have **one finalizer**.
    - Finalizers cannot be **inherited** or **overloaded**.
    - Finalizers cannot be **called!** They are invoked automatically.
    - Finalizer does not take modifiers or have parameters.

# C++/C# Interop

## PInvoke - Classes

- C# also has an **interface** that is being used to release unmanaged resources called **IDisposable**. This interface also allows us to manually trigger the destruction of the object instead of relying on the GC.
- We can also make our C wrapper functions private and part of our C# wrapper class.

```
//Public functions
0 references
public int SomeFunction()
{ return CallSomeFunction(_nativePtr); }

//PRIVATE IMPORT FUNCTIONS
1 reference
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
static private extern IntPtr CreateSomeUnmanagedClass();

1 reference
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
static private extern void DestroySomeUnmanagedClass(IntPtr pObj);

1 reference
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.ThisCall)]
static private extern int CallSomeFunction(IntPtr pObj);
```

```
2 references
public class SomeManagedClass : IDisposable
{
    private IntPtr _nativePtr;
    private bool _disposed = false;

    0 references
    public SomeManagedClass() //Constructor
    {
        _nativePtr = CreateSomeUnmanagedClass();
    }

    //Implement IDisposable interface
    0 references
    public void Dispose()
    {
        Dispose(true); //Deterministic destruction
        //No need to call the finalizer since we've
        //now cleaned the unmanaged memory
        GC.SuppressFinalize(this);
    }

    2 references
    protected virtual void Dispose(bool bDisposing)
    {
        if (!this._disposed)
        {
            if (bDisposing)
            {
                //Free any dependent IDisposable objects
            }

            if (_nativePtr != IntPtr.Zero)
            {
                DestroySomeUnmanagedClass(_nativePtr);
                _nativePtr = IntPtr.Zero;
            }

            _disposed = true; //Track has been disposed
        }
    }

    //Finalizer, called when the GC occurs, but only if the
    //IDisposable.Dispose method wasn't already called!
    0 references
    ~SomeManagedClass()
    {
        Dispose(false);
    }
}
```

# C++/C# Interop

## PInvoke - Classes

- If you want to use your class in a container that uses a hashcode to index (eg Hashtable, Dictionary, etc.), you also need to implement the **GetHashCode** function.
- So what do we use to generate the hash, something that is already unique?
  - Our native pointer!
- Because we'll be using our native pointer a good practise is to make it **readonly** so it can only be set during declaration or within the constructor.

```
//readonly: Allows the field only to be modified in its declaration or within objects constructor
//Important because we use the IntPtr to do the hashing instead of the members (default implementation)
private readonly IntPtr _nativePtr;
```

```
//Implement GetHashCode function, necessary if you want to use your class in collections such as dictionaries
//Also for the Equal function (see below)
0 references
public override int GetHashCode()
{
    return _nativePtr.GetHashCode();
}

2 references
public override bool Equals(object obj)
{
    var tempObj = obj as SomeManagedClass;
    if (tempObj == null)
        return false;
    return this == tempObj;

    //Incorrect! Subtle bug: happens with unmanaged code in dictionaries that can be automatically disposed and reused.
    //return _nativePtr == tempObj._nativePtr;
}
```

# C++/C# Interop

## PInvoke - Marshaling

- Let's take a look at how we can transfer data between managed and unmanaged code. The process of transforming the memory representation of an object to a data format suitable for storage or transmission between programs is called **marshaling**.
- Most data types have a common representation in both managed and unmanaged memory and do not require special handling by the interop marshaler. These types are called **blittable types**.
- Typical **non-blittable types** are: Boolean, Array, Class, Char and String.

C++	C#	.NET
unsigned long	ulong	Ulong
unsigned int	uint	UInt32
unsigned short	ushort	UInt16
unsigned char	byte	Char
long	long	Int64
int	int	Int32
short	short	Int16
float	float	Single
double	double	Double
(someType) *	IntPtr	IntPtr

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



# C++/C# Interop

## PInvoke - Marshaling

- A **bool** is NOT a blittable type. In C# a bool/Boolean has a size of **4 bytes**, while in C++ it has a size of 1 byte.

```
//--- Bool ---  
__declspec(dllexport) bool __stdcall MarshalBool(bool input)  
{ return !input; }
```

```
//--- Bool ---  
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]  
[return: MarshalAs(UnmanagedType.I1)]  
2 references  
public static extern bool MarshalBool([MarshalAs(UnmanagedType.I1)] bool a);
```

- You can specify how data should be marshalled using the **MarshalAs** attribute. This can be done for both the parameters and the return value.

# C++/C# Interop

## PInvoke - Marshaling

- **Arrays** can be marshaled in different ways. Some things that you can do to take input account:
  - **Input vs Output**
  - **Type** stored in array

```
//--- Arrays ---
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
1 reference
public static extern void MarshalIntegerArray([MarshalAs(UnmanagedType.LPArray,
SizeConst = 5, ArraySubType = UnmanagedType.I4)]
int[] ar);

[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
2 references
public static extern int MarshalIntegerArrayV2(IntPtr pArray, UInt32 length);
```

```
//--- Arrays ---
__declspec(dllexport) void __stdcall MarshalIntegerArray(int ar[5])
{
    ar[0] = 16; ar[1] = 32; ar[2] = 64; ar[3] = 128; ar[4] = 256;
}

__declspec(dllexport) int __stdcall MarshalIntegerArrayV2(int* pArray, unsigned int length)
{
    if (!pArray)
        return 0;

    int average = 0;
    for(unsigned int i = 0; i < length; ++i, ++pArray)
    {
        average += *pArray;
    }
    return average / length;
}
```

```
//--- Array ---
int[] someArray = new int[] {0, 0, 0, 0, 0};
MarshalIntegerArray(someArray);

someArray = new int[] {10, 20, 30, 40, 50, 60};
IntPtr unmanagedPtr = Marshal.AllocHGlobal(Marshal.SizeOf(someArray[0]) * someArray.Length);
Marshal.Copy(someArray, 0, unmanagedPtr, someArray.Length);
result = MarshalIntegerArrayV2(unmanagedPtr, (UInt32) someArray.Length);
```

# C++/C# Interop

## PInvoke - Marshaling

- Some important types and methods in our example:
  - **IntPtr**: pointing to the first element of our array.
  - **Marshal.AllocHGlobal**: allocates a block of unmanaged memory and returns an IntPtr object that points to the beginning of this block.
  - **Marshal.Copy**: copies data from a managed array to an unmanaged memory pointer, or the other way around.
- Some other useful functionality:
  - **Marshal.ReadByte** & **Marshal.WriteByte**: read or write a single byte at a given offset of index.
  - **Marshal.PtrToStructure** & **Marshal.StructureToPtr**: marshals data from an unmanaged block of memory to a managed object, or the other way around.
  - **Marshal.SizeOf**: returns the unmanaged size, in bytes, of a class.

# C++/C# Interop

## PInvoke - Marshaling

```
public class IntArrayWrapper
{
    private int[] _array;
    private IntPtr _nativeArray;

    1 reference
    public int Length
    {
        get => _array.Length;
    }

    1 reference
    public IntPtr ArrayPtr => _nativeArray;

    1 reference
    public IntArrayWrapper(int[] array)
    {
        _nativeArray = Marshal.AllocHGlobal(sizeof(int) * array.Length);
        _array = new int[array.Length];
        Array = array; //Force marshal copy at initialize time
    }

    1 reference
    public int[] Array
    {
        get
        {
            Marshal.Copy(_nativeArray, _array, 0, _array.Length);
            return _array;
        }
        set => Marshal.Copy(value, 0, _nativeArray, value.Length);
    }

    0 references
    public int this[int index]
    {
        get => Marshal.ReadInt32(_nativeArray, index * sizeof(int));
        set => Marshal.WriteInt32(_nativeArray, index * sizeof(int), value);
    }
}
```

# C++/C# Interop

## PInvoke - Marshaling

- Same applies for **strings**. Here we even have to take into account the **Character Set**.

```
#ifdef CPPLIBRARY_EXPORTS
    #define CPPLIBRARY_API __declspec(dllexport)
#else
    #define CPPLIBRARY_API __declspec(dllimport)
#endif
```

```
//--- Strings ---
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall, CharSet = CharSet.Ansi)]
1 reference
public static extern void MarshalStringStringInput([MarshalAs(UnmanagedType.LPStr)] string s);

[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
1 reference
public static extern IntPtr MarshalStringConstCharPtrOutput();

[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]
1 reference
public static extern void MarshalStringBuffer([MarshalAs(UnmanagedType.LPStr)] StringBuilder buffer,
ref UInt32 size);
```

```
//--- String ---
MarshalStringStringInput("SomeStringPassedToFunction");

IntPtr constCharPtr = MarshalStringConstCharPtrOutput();
string returnText = Marshal.PtrToStringAnsi(constCharPtr);

UInt32 sbSize = 65;
var sb = new StringBuilder((int) sbSize);
MarshalStringBuffer(sb, ref sbSize);
string rawSb = sb.ToString();
```

```
//--- Strings ---
CPPLIBRARY_API void __stdcall MarshalStringStringInput(char* p)
{
    std::string receivedString = std::string(p);
}

CPPLIBRARY_API const char* __stdcall MarshalStringConstCharPtrOutput()
{
    return "SomeOtherString";
}

CPPLIBRARY_API void __stdcall MarshalStringBuffer(char* pBuffer, unsigned long* pSize)
{
    if (!pSize)
        return;

    char txt[] = "BlaBlaBlaBlaBlaBlaBlaBlaBlaBlaBlaBlaBlaBlaBlaBla";
    unsigned long size = strlen(txt) + 1; //null-terminated string
    if((pBuffer != nullptr) && (*pSize >= size))
    {
        strcpy_s(pBuffer, size, txt);
    }
    *pSize = size;
}
```

<https://docs.microsoft.com/en-us/dotnet/framework/interop/default-marshaling-for-strings>



# C++/C# Interop

## PInvoke - Marshaling

- **Character Set:**
  - Ansi (default)
  - Unicode
  - Auto
- Keywords:
  - **ref**: argument is passed by reference.
    - Must be initialized!
  - **in**: arguments is passed by reference, but cannot be modified by the called method.
    - Must be initialized!
  - **out**: argument is passed by reference.
    - Doesn't have to be initialized.
- Both the method definition and the calling method must explicitly use the keywords!
- Keywords cause different run-time behaviour, they are **not** considered part of the **method signature at compile time!**

# C++/C# Interop

## PInvoke - Marshaling

- There are other options as well:
  - **SAFEARRAY**: data type that is a **safe array descriptor**, holding various pieces of **information about the instance of a array** (number of dimensions, dimension bounds, lock and pointer to actual array).
    - <https://docs.microsoft.com/nl-nl/windows/desktop/api/oidl/ns-oidl-tagsafearray>
    - <https://docs.microsoft.com/en-us/dotnet/framework/interop/default-marshaling-for-arrays>
  - **BSTR**: data type (pointer) that **consists of a length prefix**, a **data string** and a **terminator**. It points to the first character of the data string, not to the length prefix.
    - <https://docs.microsoft.com/nl-nl/previous-versions/windows/desktop/automat/bstr>
    - <https://docs.microsoft.com/en-us/cpp/dotnet/how-to-marshall-com-strings-using-cpp-interop?view=vs-2017>
    - **Marshal.StringToBSTR**, **Marshal.PtrToStringBSTR** & **Marshal.FreeBSTR**

# C++/C# Interop

## PInvoke - Marshaling

- In C# there is a big difference between a **class** and a **struct**!
- **Class:**
  - **Reference** (pointer) types, thus passed by reference.
  - Lives on the heap.
  - The reference can be null.
  - Support inheritance & interfaces.
- **Struct:**
  - **Value** types, thus passed by value.
  - Lives wherever the variable or field is defined (inline, often on the stack).
  - Cannot have a null reference (unless Nullable).
  - Cannot support inheritance, but does support interfaces.
- In C++ the **physical layout** of the class or struct depends on the declaration. This is fixed and doesn't change. In C# the CLR controls the physical layout in managed memory. This means the CLR can **reorder your member**. Using the **StructLayoutAttribute** you can force the members to be laid out in a specific order!
- <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.structlayoutattribute?view=netframework-4.7.2>

# C++/C# Interop

## PInvoke - Marshaling

```
//--- Structs ---  
[StructLayout(LayoutKind.Sequential)]  
5 references  
public struct Vector3  
{  
    public float x, y, z;  
}  
  
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]  
1 reference  
public static extern Vector3 MarshalStructureReturn();  
  
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]  
1 reference  
public static extern void MarshalStructureOutput(ref Vector3 v);
```

```
[StructLayout(LayoutKind.Explicit)]  
3 references  
public struct Package  
{  
    [FieldOffset(0)] public float data;  
    [FieldOffset(4)] public uint packedDataUInt;  
    [FieldOffset(4)] public int packedDataInt;  
}  
  
[DllImport("CPPLibrary.dll", CallingConvention = CallingConvention.StdCall)]  
1 reference  
public static extern void MarshalStructureWithUnionOutput(ref Package p);
```

```
//--- Structs ---  
struct Vector3  
{  
    float x, y, z;  
    Vector3(float _x, float _y, float _z):x(_x),y(_y),z(_z)  
    {}  
};  
CPPLIBRARY_API Vector3 __stdcall MarshalStructureReturn()  
{  
    return Vector3(32.f, 64.f, 128.f);  
}  
CPPLIBRARY_API void __stdcall MarshalStructureOutput(Vector3& v)  
{  
    v = Vector3(32.f, 64.f, 128.f);  
}
```

```
struct Package  
{  
    float data;  
    union PackedData  
    {  
        unsigned int packedDataUInt;  
        int packedDataInt;  
    } packedData;  
  
    Package(float _data, PackedData _packedData):data(_data), packedData(_packedData)  
    {}  
};  
CPPLIBRARY_API void __stdcall MarshalStructureWithUnionOutput(Package& p)  
{  
    Package::PackedData packedData;  
    packedData.packedDataInt = -256;  
    p = Package(32.f, packedData);  
}
```

# C++/C# Interop

## PInvoke - Marshaling

- Some extra's:
  - **Templates:**
    - Templates are instantiated by the C++ compiler. So even if you would use Generic Methods in C#, there is no way to link these together. So because of this there is **no way to use templates**, except by **defining specialized functions as C bridge functions** and call the template function with the proper type inside this bridge function.
  - **Function Pointers:**
    - Function pointers are just pointers pointing to code (function) with a specific signature (parameters and return type). **C# has something similar called delegates.**
    - If we want to marshal this type from managed to unmanaged, we **need a pointer** and not a **delegate object**. We can do this using **Marshal.GetFunctionPointerForDelegate**, or the other way around using **Marshal.GetDelegateForFunctionPointer**. Just make sure both signatures match!
    - <https://docs.microsoft.com/en-us/cpp/dotnet/how-to-marshal-function-pointers-using-pinvoke?view=vs-2017>
  - For classes, arrays and boxed (<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing>) value types, the **ICustomMarshaler** interface can be used to **provide custom marshalling**. Feel free to play around with this 😊
  - <https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.icustommarshaler?view=netframework-4.7.2>

# C++/C# Interop

## Exercise - Space Invaders

- Make the **bridge** between C++ and C# for the game Space Invaders. The game itself runs in C++. We just use Unity3D to **visualize** the state of the game and to accept **input** from the user.
- Open the project in the following order (first time use):
  1. Open the Unity3D project (UnityProject/Assets/Scenes/SampleScene)
    - This loads the reference to the CSharpLibrary in Unity and VS project.
  2. Close Unity3D
    - Whenever you want to rebuild the CPPLibrary you have to close Unity3D. This is due to the fact that once an unmanaged library is loaded Unity3D doesn't unload it.
  3. Open CSharpCPPInterop.sln and build everything
    - This will also copy the .dll's to the correct folder.
      - CPPLibrary.dll in root folder, CSharpLibrary.dll in Plugins folder
- Only **complete** the following classes: CPPLibrary/GameBridge.cpp and CSharpLibrary/SpaceInvadersLibrary. → See **//\*\* ADD CODE \*\*//**
- **DO NOT MODIFY EXISTING CODE (IN BOTH C++ and C#)**

