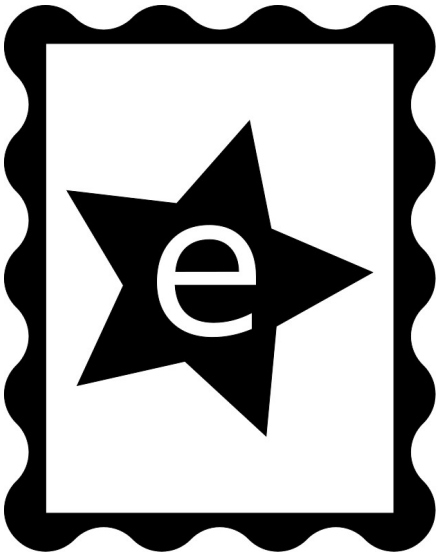


Make your own DLL from scratch (MFC)

(eToile 2016)

Introduction

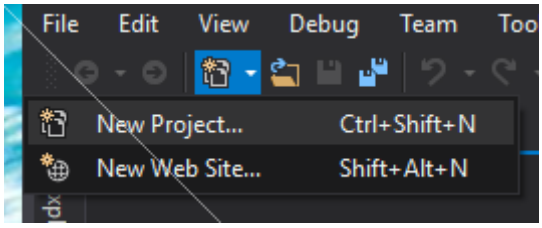
This package includes everything you need to know in order to deploy x86 and x64 DLLs for Unity integration. The included DLL project compiles under Visual Studio Community 2015, which can be downloaded for free from here:
https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx

You'll need to install the "C++ language" to be able to compile DLLs in Visual Studio.
In any case, eToile recommends to make a complete installation in order to avoid problems with the DLL example project and any further product development you may want to make.

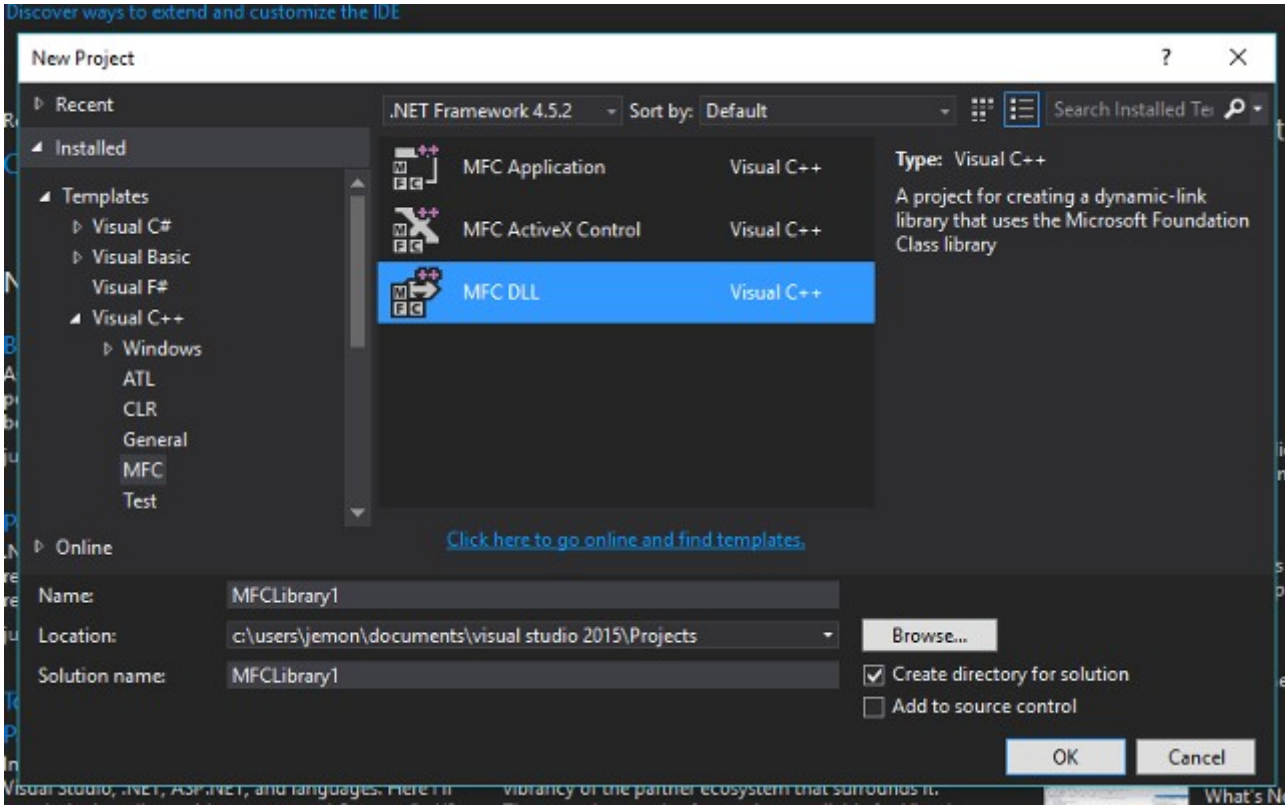
As you may expect, this DLL integration may also run on other development platforms because there is not any "Unity-Only" feature. This feature is not covered in this document.

Generating your MFC DLL from scratch

Assuming that you have already installed Unity (x86 or x64) and Visual Studio Community 2015, open Visual Studio and create a new project:

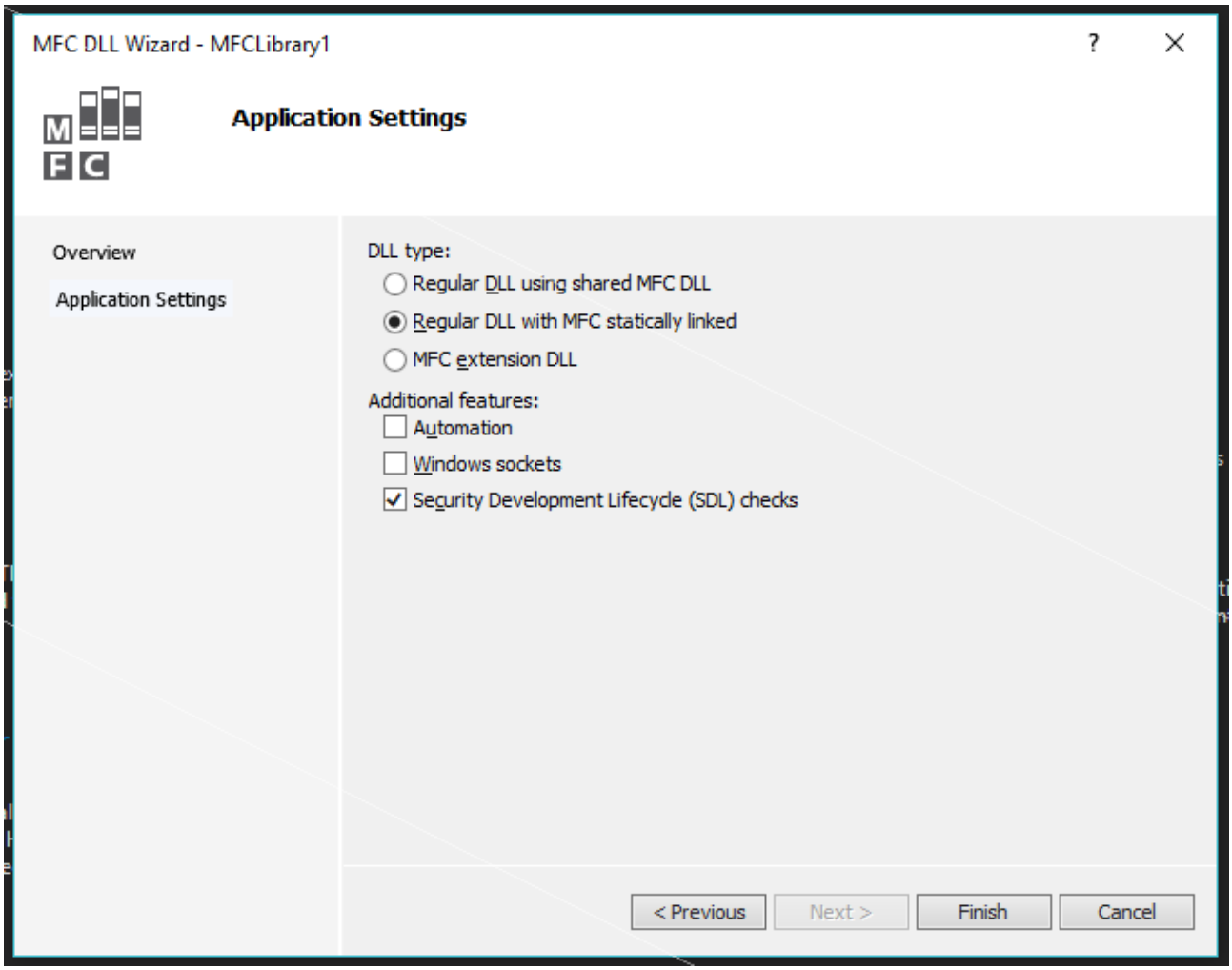


In the project creation window, you must select "Templates>Visual C++>MFC>MFC DLL":



Enter the name you wish for your project in this window and click the "OK" button (this example will use the default project name).

The wizard is displayed; click on the “Next” button and you will see the following window:

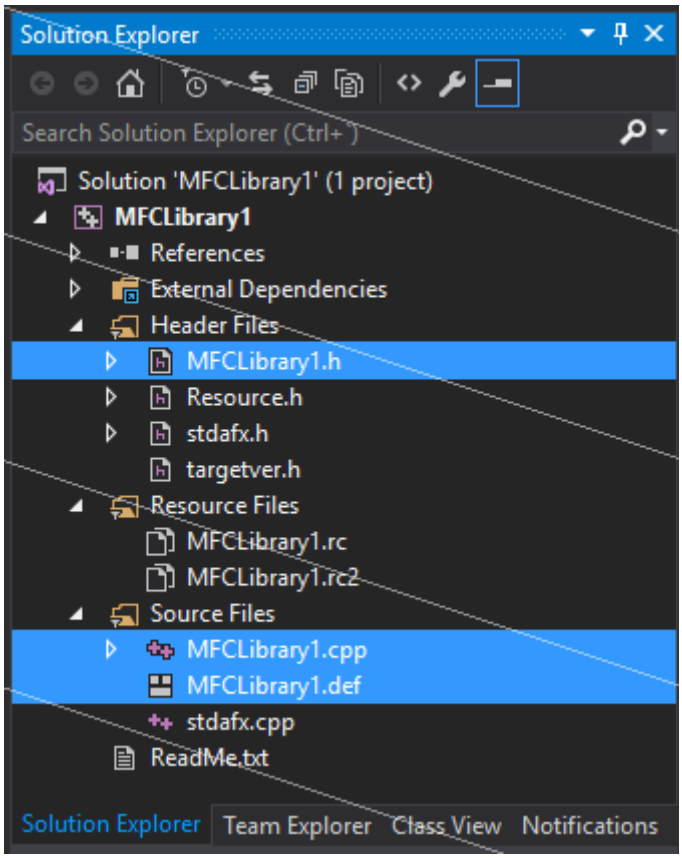


As you can see in the picture above, select the “Regular DLL with MFC statically linked” option to avoid most DLL dependencies and also to avoid redistributing secondary installers. More information here: https://msdn.microsoft.com/en-us/library/26h8x9sy.aspx.

Leave the “SDL” option untouched. This document will not describe SDL, but you can refer to this link for more information: https://msdn.microsoft.com/es-es/library/windows/desktop/84aed186-1d75-4366-8e61-8d258746bopq.aspx

Then click on “Finish” and the project will be automatically created in the location you have selected.

Look for the three main files we'll use to write the DLL in the "Solution explorer" window:



Now open the ".cpp" file. Here is where the main code will be written. We will implement a simple interface to reduce errors on both sides (Visual Studio and Unity).
So, the first half of the ".cpp" file should look like this:

```
// MFCLibrary1.cpp : Defines the initialization routines for the DLL.
//

#include "stdafx.h"
#include "MFCLibrary1.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#endif

// ...

// CMFCLibrary1App

BEGIN_MESSAGE_MAP(CMFCLibrary1App, CWinApp)
END_MESSAGE_MAP()
```

Pay attention to the `BEGIN_MESSAGE_MAP` line, your code must be added just before this macro.

The interfaces and the code must include the `extern "C"` header this way:

```
// CMFCLibrary1App
extern "C"

int WINAPI GetInt(int n)
{
    return 3 + n;
}

BEGIN_MESSAGE_MAP(CMFCLibrary1App, CWinApp)
END_MESSAGE_MAP()
```

Pay close attention to the `WINAPI` function definition. Our interface is already defined.
You can also declare variables out the curly braces and that stored data will be persistent, so it can be readed through an interface at any time.
This DLL implementation also admits the definition of parallel threads if constant execution is needed. You must stop the threads before closing your application.

Open the ".h" file. This file should look like this:

```
// MFCLibrary1.h : main header file for the MFCLibrary1 DLL
//

#pragma once

#ifndef __AFXWIN_H__
    #error "include 'stdafx.h' before including this file for PCH"
#endif

#include "resource.h"       // main symbols


// CMFCLibrary1App
// See MFCLibrary1.cpp for the implementation of this class
//

class CMFCLibrary1App : public CWinApp
{
public:
    CMFCLibrary1App();

// Overrides
public:
    virtual BOOL InitInstance();

    DECLARE_MESSAGE_MAP()
};
```

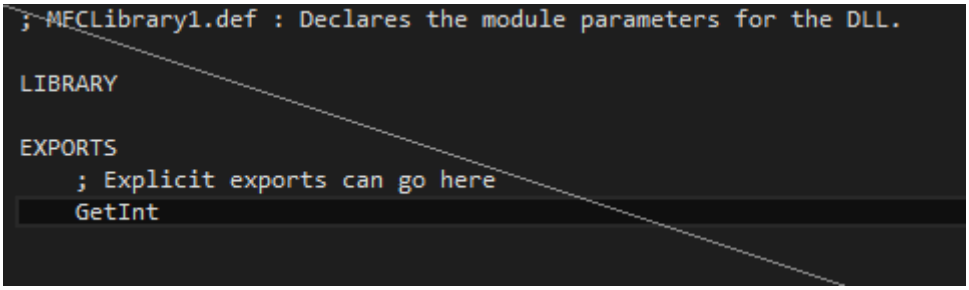
Your function declarations must be added at the very end of this file.

Add the declaration this way (you need this entire block):

```
#ifdef __cplusplus
extern "C"
{
#endif
	// Declare your interfaces here:
	int WINAPI GetInt(int);

#ifdef __cplusplus
}
#endif
```

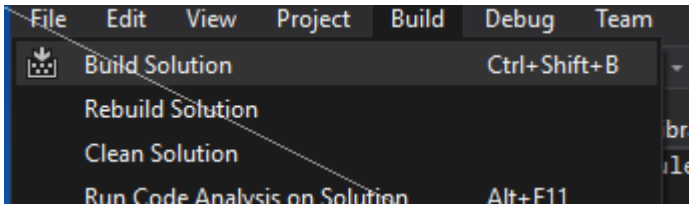
Note that arguments only include variable types, not variable names.

Finally, open the ".def" file, where you must add the interface names this way:

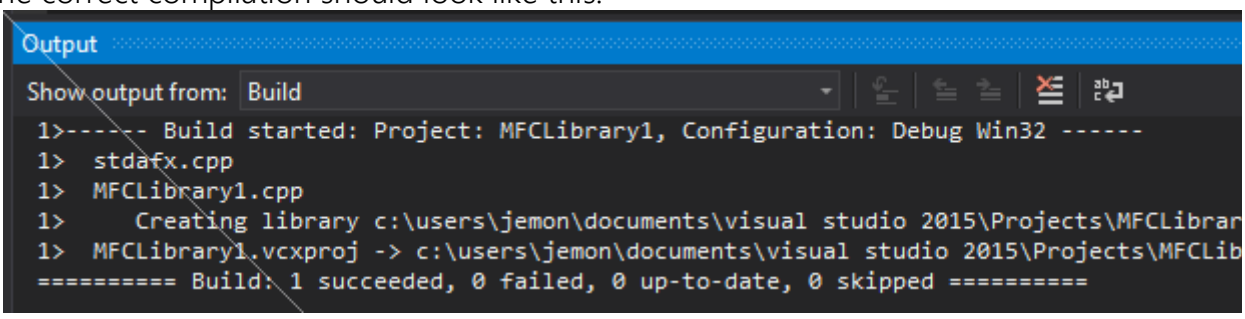


At this point, the `GetInt(int)` function is declared and defined, so it is ready to be accessed within Unity.

Build the project and see if there are any errors. If there are, you must fix them before continuing.



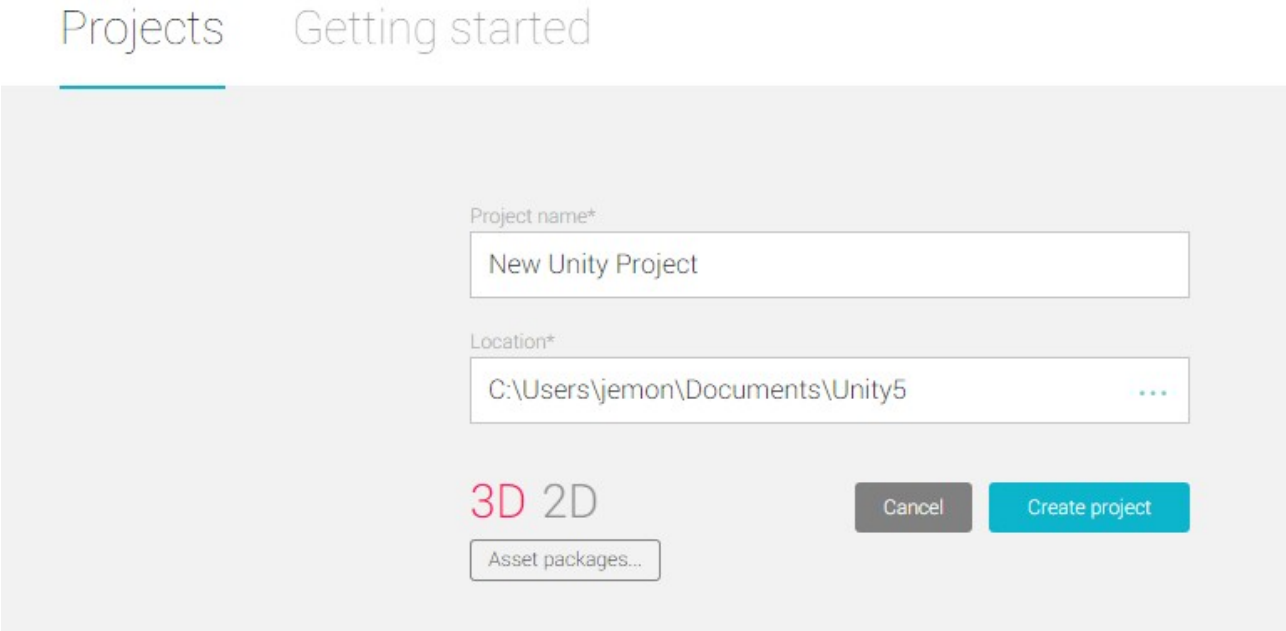
The correct compilation should look like this:



The DLL file was generated in the output folder inside your project folder in a secondary folder called “Release” or “Debug” depending on the compilation you have chosen. During development, it is recommended to use the “Debug” compilation, but once the code is running and needs to be shipped within the final product, the “Release” compilation must be used. The “Release” compilation has code optimizations and eliminates “Debug” dependencies.
There is another directive for building your project, that is the platform and can be “x86”, meaning 32bit, or “x64”, meaning 64bit.
Your DLL file will have the same name as your project, in this case: “MFCLibrary1.dll”.
The x86 version can be found at: “Documents / Visual Studio / 2015 / Projects / MFCLibrary1 / Release” folder.
The x64 version can be found at: “Documents / Visual Studio / 2015 / Projects / MFCLibrary1 / x64 / Release” folder.

Creating the Unity project for integration

The next step is to create a Unity project and then add this DLL to it. To do so, open Unity and create a new empty project:



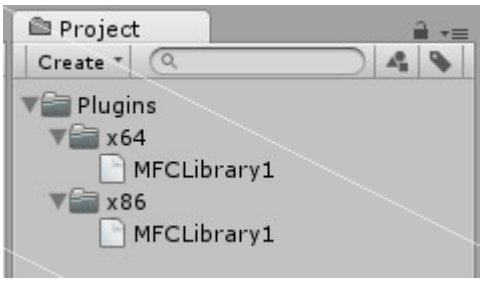
This example will use Unity 5, which imports the DLLs through an importer (older versions obligated a “Plugins” folder).
For compatibility, we will create the “Plugins” folder, but this is not compulsory.

Inside the "Plugins" folder, also create the "x86" and "x64" folders, so you can add both versions of your plugin and compile for both platforms.

Drag and drop each generated DLL file into its corresponding folder.

NOTE: You don't need to compile both versions of the DLL, but it is a good practice for compatibility and this way you will learn how to do it.

Now your project should look like this:

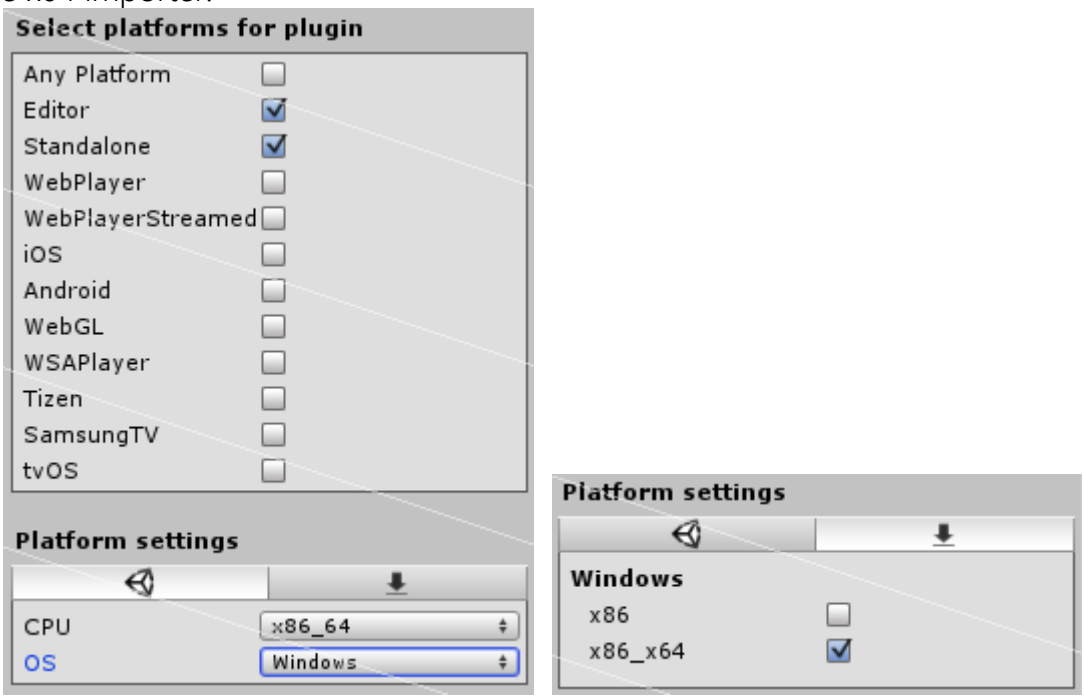


Both DLLs has different platform, so the importer must be different given that the code is not managed but native. You must configure the importers as follows.

Here the x86 importer:

If you are working on Windows, you can run the DLL in the Unity editor.

Here the x64 importer:



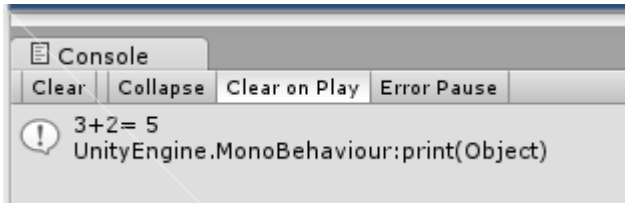
Don't forget to do the correct importing of DLL files, otherwise Unity will detect duplicated DLLs and will not compile correctly, given that both DLLs have the same name.
In the "Build settings" dialog, you can select the platform you wish to export, and Unity will chose the correct DLL to be included within the final product.

Create an empty C# source file, and name it "DllAccess.cs".
Your source code should look like this:

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

public class DllAccess : MonoBehaviour {

    [DllImport("MFCLibrary1")] public static extern int GetInt(int n);

    // Use this for initialization
    void Start () {
        print("3+2= " + GetInt(2).ToString());
       }

       // Update is called once per frame
       void Update () {

       }
}
```

Attach this source code to the "Main Camera", dragging the ".cs" file from the "Project" view to the "Hierarchy" view, and dropping it over the "Main Camera" GameObject.

Run your project and the console should display this:



The `InteropServices` library is needed to access the `DllImport` directive that makes loading and accessing the DLL dynamically possible.

Unity loads the DLL files into memory at the first run, so you must restart Unity in order to update a DLL file. It is recommended to open the container folder ("Assets/Plugins/x86" or "Assets/Plugins/x64") and to overwrite the DLL file while Unity is closed.
Once replaced, open Unity and your project again.

Feel free to experiment with the interfaces included in the Visual Studio example project to access some more interesting data types.
Normally, strings and byte arrays are the most important and complex types to be accessed.
For example: with a byte array, you can dynamically read an image or a video frame from a DLL and paint it into a Unity Texture2D in the fastest way.

Ending

If you find errors in the package or document, please contact me at jmonsuarez@gmail.com and product will be updated. Any comments, critics or improvements are also very welcome.

Hope you enjoyed the tutorial.

eToile, 2016