

Platform Invocation Indepth
Licensed Attribution-NonCommercial 3.0
United States (CC BY-NC 3.0 US)
<https://creativecommons.org/licenses/by-nc/3.0/us/>

BlackCentipede

November 2017

Book Contributors

- Jamesbascle - For grammar correction.

Contents

1	Introduction	1
1.1	What you will need to get started	1
1.2	Minimum Knowledge	1
2	Introduction to P/Invoke	3
2.1	Getting Started	3
2.2	Compiling the Library	4
2.3	Configuring C# Project	5
2.4	Wrapping C Code in C#	6
2.5	Some background	8
3	Introduction to Pointer	9
3.1	Overview on Pointer	9
3.2	The Function Pointers	13
3.3	C# Counterpart	15
4	The Delegate Approach	17
4.1	Moving Forward	17
4.2	Loading the Library Dynamically	18
4.3	The Delegate Approach	23
5	Runtime Generated Functions for C	25

Chapter 1

Introduction

1.1 What you will need to get started

You will need Dotnet Core and Clang/LLVM compilers installed for this book. The book will assume you are working on Linux platform although knowledge gained here can be applied on any other platforms including Windows.

You can install Dotnet Core SDK from this URL: <https://www.microsoft.com/net/download/linux>
Clang/LLVM Compilers can be installed or compiled on your respective linux distribution.

TODO: Expound instructions for this part.

The book will be using Visual Studio Code IDE and CodeLite IDE.

1.2 Minimum Knowledge

You'll need to have some comprehension of C# and C languages before starting this book.

Chapter 2

Introduction to P/Invoke

2.1 Getting Started

First, create a Directory as 'ChapterTwo' for this project and create a new file, 'ChapTwo.c' under 'ChapterTwo' folder.

Let's assume we have a basic Addition function in a C Library that we want to call.

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

It's a simple Addition Operation at a first glance, but there are considerations that must be observed first before attempting to write platform invocation wrapper code for the function above:

1. 'int' datatype in C can be considered 2 bytes long or 4 bytes long or however long it may be depending on the architecture and compiler that the library is compiled on. In C Standard, int must be capable of containing **at least** the $[-32,767, +32,767]$ range; thus, it is at least 16 bits in size.
2. Due to 1, you can reasonably safeguard against data loss by substituting C# Int32(int) which contains 4 bytes or you may choose follow the standard strictly by supplying C# Int16(short) which contains 2 bytes, even though it can suffer data loss. The best approach is to avoid using "at least" integers in C and instead use fixed size integers provided by the compiler in "stdint.h" header if you have Foreign Function Interface kept in mind.
3. Sometimes you have to keep Endianness in mind although it is less of a concern in x86_64 architecture since little endianness is the default.

The best approach to writing the Addition function is to make it clear what sized integers you're attempting to add if possible.

```
#include <stdint.h>
int32_t Sum(int32_t a, int32_t b)
{
    return a+b;
}
```

2.2 Compiling the Library

This book assumes you have sufficient knowledge of C, we will still however, provide compilation instruction. The following command assumes that you have named your source code file as 'ChapTwo.c' as instructed at the beginning of this chapter.

```
clang -std=c99 -shared -fPIC -olibChapTwo.so ChapTwo.c
```

Here we examine and explain the compiler arguments:

1. '-std=c99' specify that we are compiling C source code under C99 Standard.
2. '-shared' specify that we want the program to be compiled as shared/dynamic library.
3. '-fPIC' specify that code must be position independent so that the resultant library can be loaded by other processes and have code be made available to be run anywhere in program address space regardless of code's address.
4. '-olibChapTwo.so' specify what the output library should be named. lib prefix in 'libChapTwo.so' is a matter of naming convention to be followed on Linux although compilers like clang and gcc do search libraries based on lib prefix when using '-l' option.

2.3 Configuring C# Project

Since we're already in "ChapterTwo" directory, we can go ahead and run 'dotnet new Console'. There are a few steps we need to take to add the C code to our C# project. First, we need to automate the compilation process of our C file and copy the compiled C library to the target directory for Debug, Release, or any other configurations.

Open up 'ChapterTwo.csproj' file with your favorite editor, and add the following under '</PropertyGroup>' inside '<Project>' tag.

```
<Target Name="CompileCProject" AfterTargets="AfterBuild">
  <exec Command="clang -std=c99 -shared -fPIC -olibChapTwo.so ChapTwo.c" />
  <Copy SourceFiles="libChapTwo.so" DestinationFolder="$(OutDir)" />
</Target>
```

The snippet above does few things after building our C# project:

1. Compile ChapTwo.c code as a shared library, libChapTwo.so
2. Copy libChapTwo.so into any target directory that C# is being built in.

This makes it significantly easier to modify our code without having to run any additional commands for it to take effect.

Your CSProj should look like the following:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
  <Target Name="CompileCProject" AfterTargets="AfterBuild">
    <exec Command="clang -std=c99 -shared -fPIC -olibChapTwo.so ChapTwo.c" />
    <Copy SourceFiles="libChapTwo.so" DestinationFolder="$(OutDir)" />
  </Target>
</Project>
```

2.4 Wrapping C Code in C#

Open up Program.cs, add a new using directive at the top of your source code.

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

This line imports all of the platform invocation services which enables us to interact with our C library with ease.

Add the following lines under Program class:

```
[DllImport("ChapTwo")]  
static extern int Add(int a, int b);
```

The DllImport attribute declares that a static externally defined function is defined in a C library and to have CLR create a Platform Invocation stub to define the said function within external library.

It is required to declare the function with static and extern modifiers since it is a function that is both independent of state and externally defined.

Finally, modify the "Console.WriteLine" line to the following:

```
Console.WriteLine("1 + 2 = {0}", Add(1, 2));
```

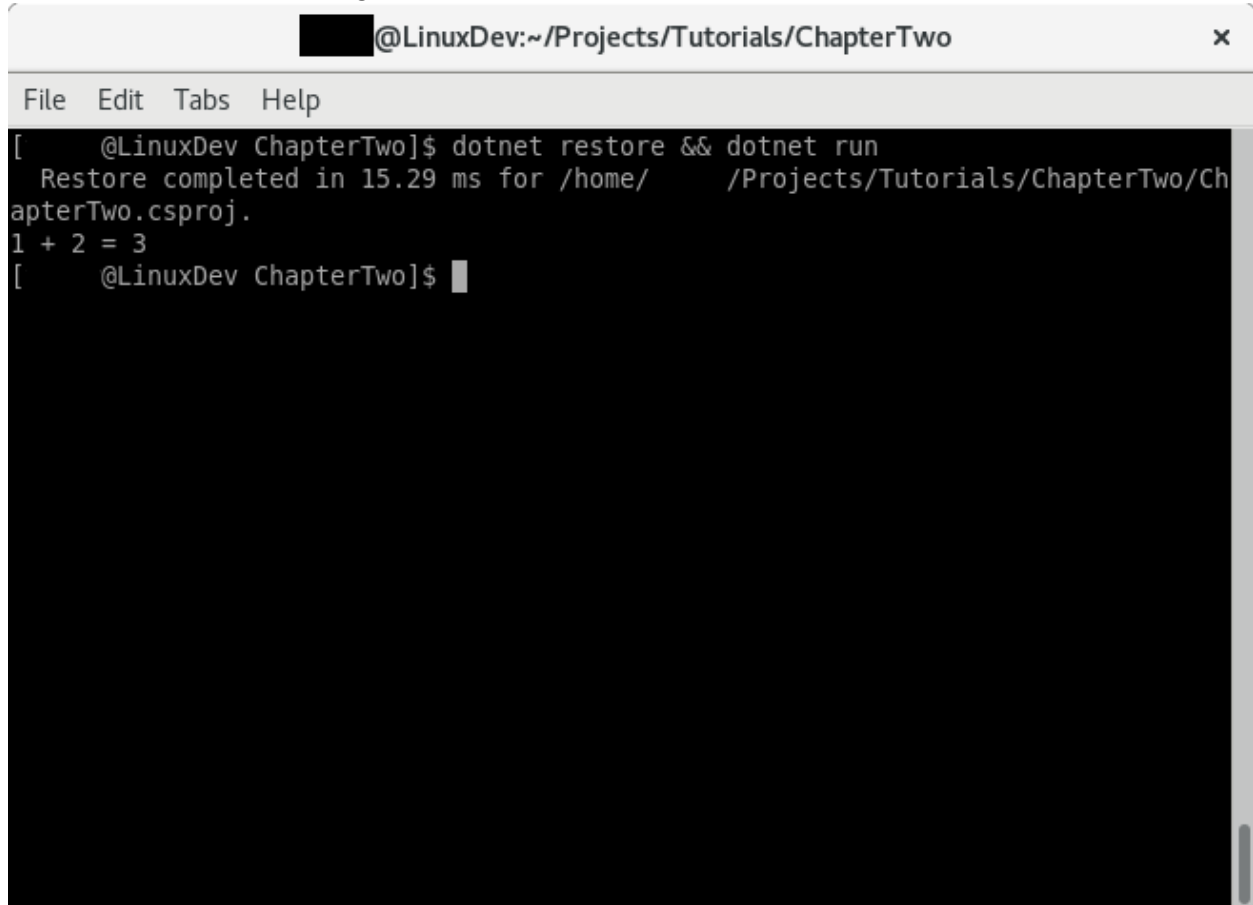
And your source code should look as follows:

```
using System;  
using System.Runtime.InteropServices;  
namespace ChapterTwo  
{  
    class Program  
    {  
        [DllImport("ChapTwo")]  
        static extern int Add(int a, int b);  
        static void Main(string[] args)  
        {  
            Console.WriteLine("1 + 2 = {0}", Add(1, 2));  
        }  
    }  
}
```

Finally, your program is ready to be executed. You can run:

```
dotnet restore && dotnet run
```

And we have the following:



The screenshot shows a terminal window titled "@LinuxDev:~/Projects/Tutorials/ChapterTwo". The terminal content is as follows:

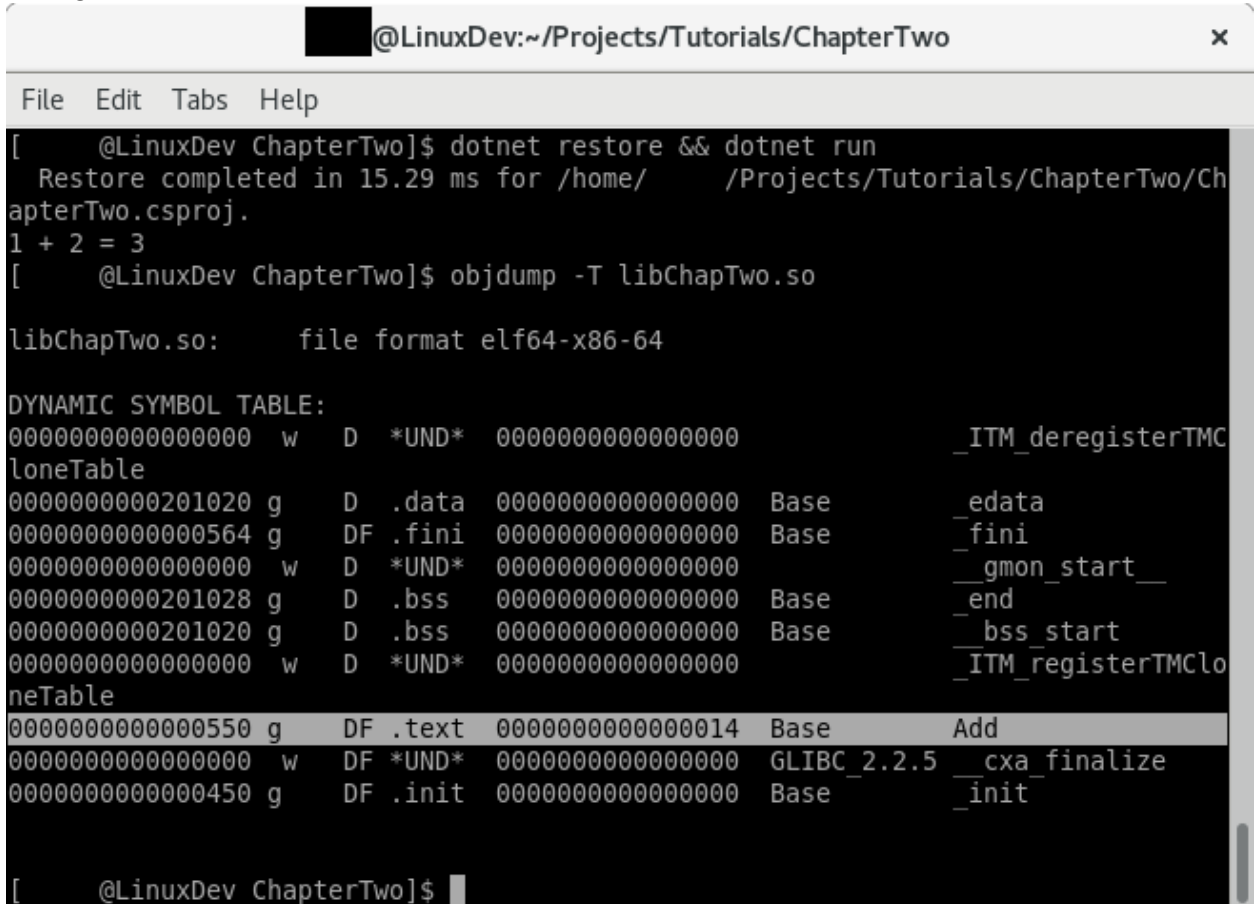
```
File Edit Tabs Help
[ @LinuxDev ChapterTwo]$ dotnet restore && dotnet run
Restore completed in 15.29 ms for /home/ /Projects/Tutorials/ChapterTwo/ChapterTwo.csproj.
1 + 2 = 3
[ @LinuxDev ChapterTwo]$
```

It works as expected!

2.5 Some background

There are few things happening when a function with DllImport is called, if this is the first time the function is being called, the Runtime will first load the external library immediately, then load the symbol "Add" by default, and finally generate a P/Invoke stub for that function to support the call to the external function.

The symbol is merely just that, a symbol that is exported by C Library. You can find a list of symbols by running "objdump -T libChapTwo.so" on your library and you'll have the following:



```

@LinuxDev:~/Projects/Tutorials/ChapterTwo
File Edit Tabs Help
[ @LinuxDev ChapterTwo]$ dotnet restore && dotnet run
Restore completed in 15.29 ms for /home/ /Projects/Tutorials/ChapterTwo/ChapterTwo.csproj.
1 + 2 = 3
[ @LinuxDev ChapterTwo]$ objdump -T libChapTwo.so

libChapTwo.so:      file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000 w  D  *UND*  0000000000000000      _ITM_deregisterTMC
loneTable
00000000000201020 g  D  .data  0000000000000000      Base      _edata
0000000000000564 g  DF .fini  0000000000000000      Base      _fini
0000000000000000 w  D  *UND*  0000000000000000      _gmon_start__
00000000000201028 g  D  .bss   0000000000000000      Base      _end
00000000000201020 g  D  .bss   0000000000000000      Base      _bss_start
0000000000000000 w  D  *UND*  0000000000000000      _ITM_registerTMCclo
neTable
0000000000000550 g  DF .text  0000000000000014      Base      Add
0000000000000000 w  DF *UND*  0000000000000000      GLIBC_2.2.5  _cxa_finalize
0000000000000450 g  DF .init  0000000000000000      Base      _init

[ @LinuxDev ChapterTwo]$

```

You will notice that the Add symbol is shown in the symbol table in your library, this is how the CLR looks up a function by entry name.

Chapter 3

Introduction to Pointer

3.1 Overview on Pointer

If you already have sufficient understanding about Pointer in both C# and C languages, you can skip this chapter. This chapter is for introducing beginners to the concept of pointer.

A pointer is essentially an address to an area of memory. You can represent what that pointer is supposed to be such as a pointer of integer, struct, classes, function or even another pointer. To read and write memory that pointer is pointing to, you have to first dereference that pointer and you can do so by using asterisk in front of a pointer variable to dereference it, not to be confused with multiplication operator.

```
#include <stdlib.h>
#include <stdint.h>

int32_t* MyPointer = (int32_t*)malloc(sizeof(int32_t));

// You can access pointer like an array
MyPointer[0] = 12;

// You can also dereference a pointer to read or write the data in memory directly
*MyPointer = 12;
```

The C snippet above first allocate with malloc function a new buffer of memory up to a size of `int32_t` datatype and return a `void*` pointer which then get casted into `int32_t*` pointer type. You can access the pointer in two ways, writing it similar fashion as you would when accessing an array and to use `*` operator to dereference the pointer to read and write the first datatype the pointer is currently pointing to.

Let's get started by creating a new "ChapterThree" directory and create a new file, "ChapThree.c" and open with your favorite editor.

We will need three headers to provide the functionalities and types we need for this chapter.

```
#include <stdlib.h> // For malloc function
#include <stdio.h> // for printf function
#include <stdint.h> // for int32_t type
```

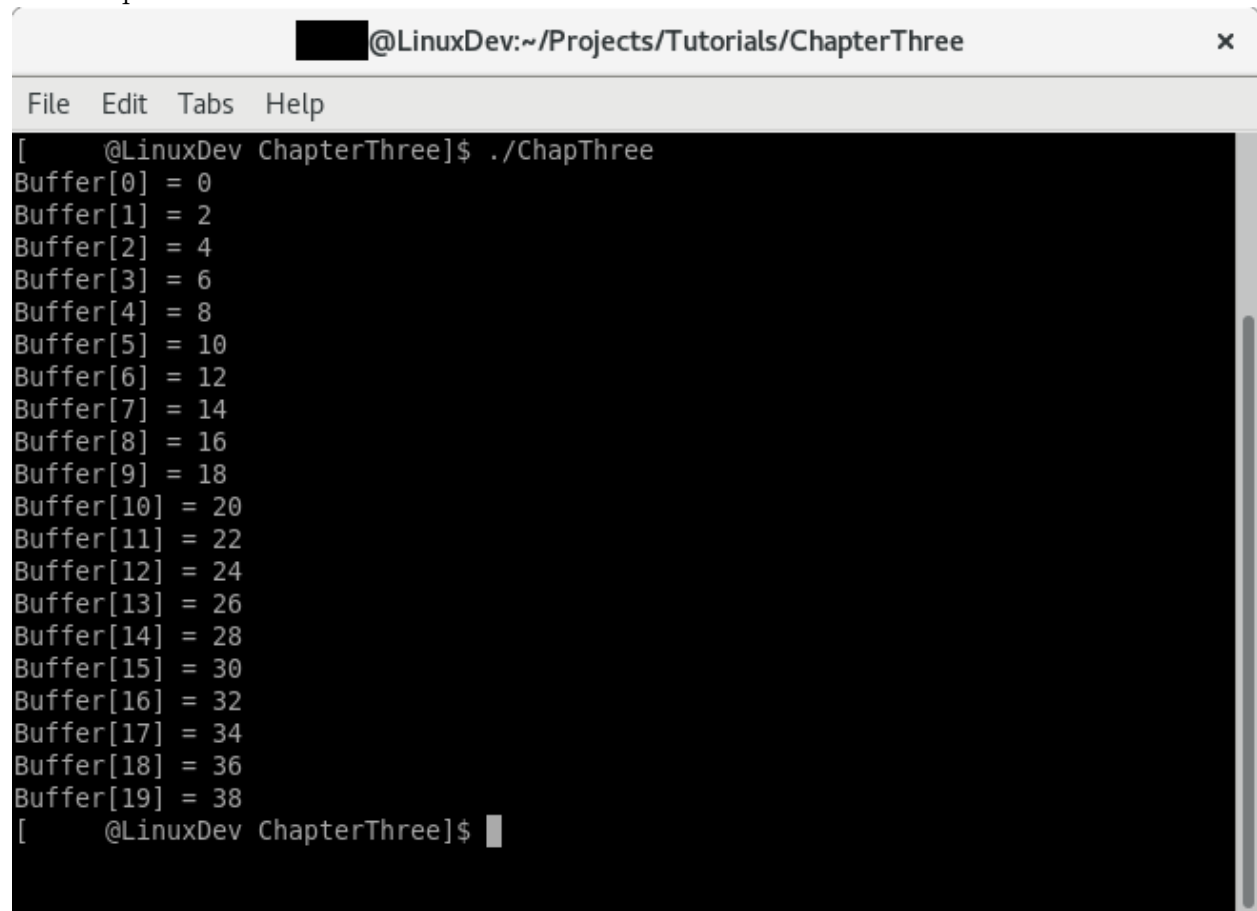
Let's declare a main function that allocate a buffer of 20 integers and return a pointer address to that buffer and we'll treat it as an array of integer.

```
int main()
{
    int32_t* buffer = (int32_t*)malloc(sizeof(int32_t)*20);
    // Let's do some math on our buffer!
    for (int32_t I = 0; I < 20; ++I)
    {
        buffer[I] = I * 2;
    }

    // Now let's print what our buffer is going to look like:
    for (int32_t I = 0; I < 20; ++I)
    {
        printf("Buffer[%i] = %i\n", I, buffer[I]);
    }

    free(buffer);
}
```

The output would be shown as this:



```
@LinuxDev:~/Projects/Tutorials/ChapterThree
File Edit Tabs Help
[ @LinuxDev ChapterThree]$ ./ChapThree
Buffer[0] = 0
Buffer[1] = 2
Buffer[2] = 4
Buffer[3] = 6
Buffer[4] = 8
Buffer[5] = 10
Buffer[6] = 12
Buffer[7] = 14
Buffer[8] = 16
Buffer[9] = 18
Buffer[10] = 20
Buffer[11] = 22
Buffer[12] = 24
Buffer[13] = 26
Buffer[14] = 28
Buffer[15] = 30
Buffer[16] = 32
Buffer[17] = 34
Buffer[18] = 36
Buffer[19] = 38
[ @LinuxDev ChapterThree]$
```

There are few things that may be confusing in the snippet above and why it is an acceptable practice in C:

1. When you use malloc to allocate buffer, malloc keeps a record of how big the block of memory is so that it can be freed in later time, however you cannot and should not access that size information within malloc, keeping track of buffer size is your responsibility.
2. When using malloc, you allocate the amount of data you would need by using sizeof keyword to determine how many bytes capacity you need in a buffer to cover that information and you can multiply that element size by the number of elements you want allocated for your program. In the snippet above, size of int32_t type would resolve to 4 and then multiplied by 20, so we would have a buffer that have the capacity to hold 20 int32_t elements.

Because of the fact behind malloc/free that LibC does store information about the pointer that is allocated with those functions, it is discouraged to use different library or framework to free that memory, although most library and framework may likely use the same functions.

3.2 The Function Pointers

Function pointers are a bit of a tongue twister, because the way it is defined in C can be confusing.

```
int32_t (*Add)(int32_t, int32_t);
```

The snippet above is a declaration of function pointer for a function that returns a `int32_t` after accepting two `int32_t` parameters. It currently pointing at nothing and would cause segmentation fault when you attempt to call it, so you have to assign a function for it to be used.

One example of this use case is that we can dynamically modify the behavior of our program during runtime and essentially allow our program to switch logic at different points during program execution like so:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

int32_t (*Add)(int32_t, int32_t);

int32_t ActualAddingFunction(int32_t a, int32_t b)
{
    return a + b;
}

int32_t FalseAddingFunction(int32_t a, int32_t b)
{
    return a - b;
}

int main()
{
    int A = 1;
    int B = 2;

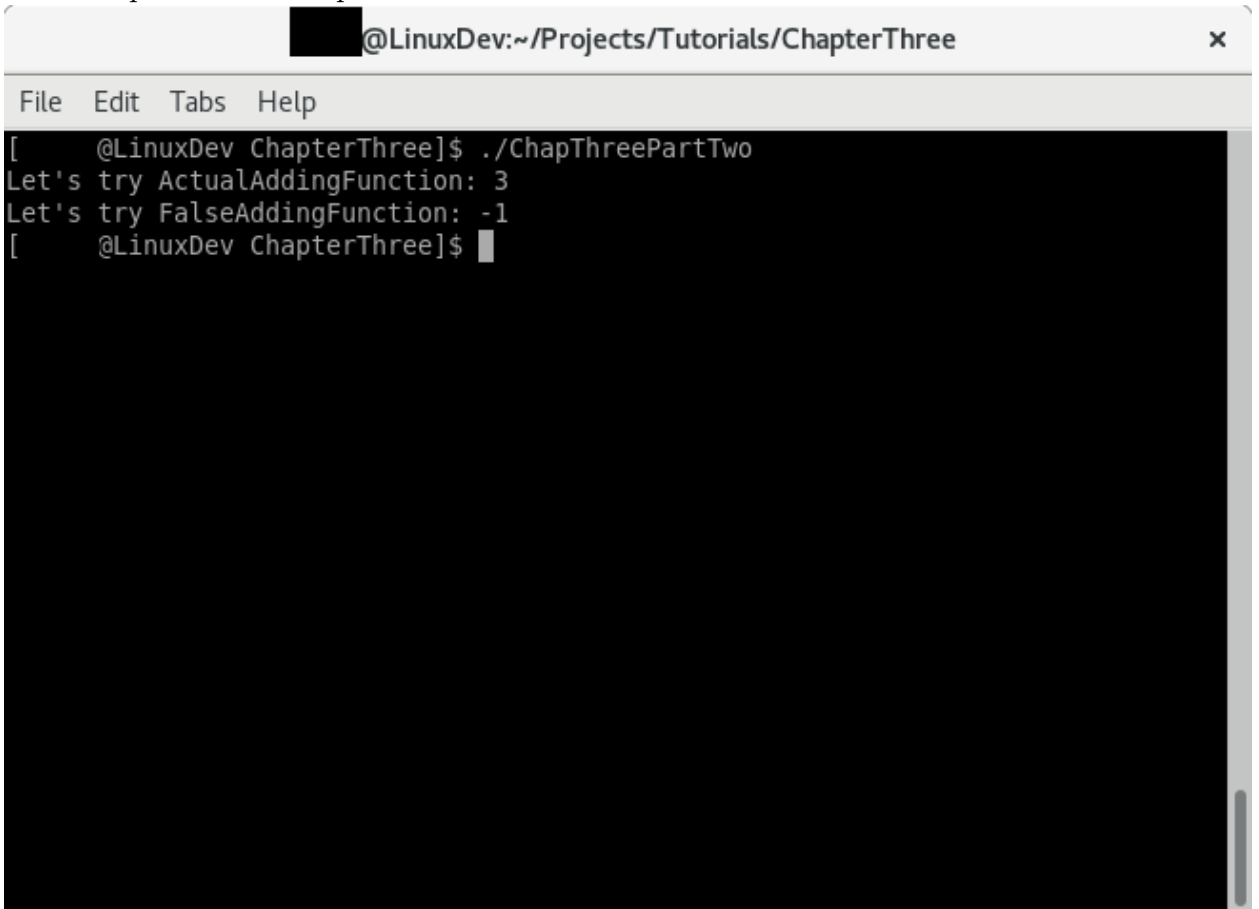
    Add = ActualAddingFunction;

    printf("Let's try ActualAddingFunction: %i\n", Add(A, B));

    Add = FalseAddingFunction;

    printf("Let's try FalseAddingFunction: %i\n", Add(A, B));
}
```

It would produce an output as followed:

A terminal window titled "@LinuxDev:~/Projects/Tutorials/ChapterThree" with a menu bar (File, Edit, Tabs, Help). The terminal shows the execution of a program. The prompt is "[@LinuxDev ChapterThree]\$". The first command is "./ChapThreePartTwo". The output is "Let's try ActualAddingFunction: 3" followed by "Let's try FalseAddingFunction: -1". The prompt returns to "[@LinuxDev ChapterThree]\$".

```
@LinuxDev:~/Projects/Tutorials/ChapterThree
File Edit Tabs Help
[ @LinuxDev ChapterThree]$ ./ChapThreePartTwo
Let's try ActualAddingFunction: 3
Let's try FalseAddingFunction: -1
[ @LinuxDev ChapterThree]$
```

3.3 C# Counterpart

C# does support pointer in a similar fashion as C so long that the code blocks, methods, or types are defined with unsafe modifier. Those can be accomplished by using snippets below as a demonstration:

```
public unsafe void DoBufferAllocation()
{
    int* MyIntegerPointer = (int*)Marshal
        .AllocHGlobal(Marshal.SizeOf<int>()).ToPointer();

    // You can use it in a similar fashion as you would in C.
    *MyIntegerPointer = 123;
}
```

```
public unsafe class Demo {
    public void DoBufferAllocation()
    {
        int* MyIntegerPointer = (int*)Marshal
            .AllocHGlobal(Marshal.SizeOf<int>()).ToPointer();

        *MyIntegerPointer = 123;
    }
}
```

```
public void DoBufferAllocation()
{
    unsafe {
        int* MyIntegerPointer = (int*)Marshal
            .AllocHGlobal(Marshal.SizeOf<int>()).ToPointer();

        *MyIntegerPointer = 123;
    }
}
```

Though this is not required, you can use IntPtr and Marshal static class to accomplish everything of above.

However, compiler will throw an error unless you explicitly specify that you want to compile unsafe code. For CoreCLR, you can do so by adding the following into your csproj file inside the <PropertyGroup> tags:

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

Marshal static class from System.Runtime.InteropServices offer a variety of functions for marshaling pointers to usable data types and vice versa. You can also generate a function in runtime in CLR and pass it over to C program so that C program can use your newly created function during its execution.

Passing a runtime generated function to C will be covered in Chapter 5.

Chapter 4

The Delegate Approach

4.1 Moving Forward

Let's create a new directory for this chapter, name it "ChapterFour". As the chapter implied, we now want to access variables that are defined in C Library, but unfortunately, C# does not offer an easy approach to accomplish this, so we must look into Marshal static class that is provided in

"System.Runtime.InteropServices" namespace.

Let's create a new C source code file, "ChapFour.c" and open it with your favorite editor. Add the following code to your source code:

```
#include <stdint.h>

int32_t MyVariable = 123;

void ResetMyVariable()
{
    MyVariable = 123;
}
```

You can compile the following code by running this command:

```
clang -std=c99 -shared -fPIC -olibChapFour.so ChapFour.c
```

You can export a list of symbols by running

```
objdump -T libChapFour.so
```

You will notice that we have a symbol for "MyVariable".

```
@LinuxDev ChapterFour]$ objdump -T libChapFour.so

libChapFour.so:      file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000 w  D  *UND*  0000000000000000      _ITM_deregisterTMCLoneTable
00000000000201024 g  D  .data  0000000000000000      Base      _edata
000000000000005a4 g  DF .fini  0000000000000000      Base      _fini
0000000000000000 w  D  *UND*  0000000000000000      __gmon_start__
00000000000000590 g  DF .text  0000000000000013      Base      ResetMyVariable
00000000000201028 g  D  .bss   0000000000000000      Base      _end
00000000000201024 g  D  .bss   0000000000000000      Base      _bss_start
00000000000201020 g  DO .data  0000000000000004      Base      MyVariable
0000000000000000 w  D  *UND*  0000000000000000      _ITM_registerTMCLoneTable
0000000000000000 w  DF *UND*  0000000000000000      GLIBC_2.2.5  __cxa_finalize
0000000000000490 g  DF .init  0000000000000000      Base      _init

[ @LinuxDev ChapterFour]$
```

4.2 Loading the Library Dynamically

As mentioned above, we cannot normally access library that is loaded by the CLR already directly. In Mono Runtime, it will use DL library to load the library only once in this method, but **CoreCLR as of this time of writing (Nov 24, 2017) will load entirely new instance of library and return new handle instead of returning a shared handle of the library already loaded by the CLR.**

Both DL library (for Linux Platform) and Kernel32.dll(Windows Platform) provide functions to load library and to return a handle for the same library if it is already previously loaded and haven't been freed first.

For the course of this chapter, we will load 3 new functions from DL library in order to enable us to access library that may already be loaded, dlopen, dlsym, and dlclose functions.

Let's get started by running the following command:

```
dotnet new Console
```

Open up "ChapterFour.csproj" file and add the following snippet under </PropertyGroup> tag inside <Project> tags.

```
<Target Name="CompileCProject" AfterTargets="AfterBuild">
  <exec Command="clang -std=c99 -shared -fPIC -olibChapFour.so ChapFour.c" />
  <Copy SourceFiles="libChapFour.so" DestinationFolder="$(OutDir)" />
</Target>
```

Add the following line in `<PropertyGroup>` tags:

```
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

Your `ChapterFour.csproj` should look like the following:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  </PropertyGroup>
  <Target Name="CompileCProject" AfterTargets="AfterBuild">
    <exec Command="clang -std=c99 -shared -fPIC -olibChapFour.so ChapFour.c" />
    <Copy SourceFiles="libChapFour.so" DestinationFolder="$(OutDir)" />
  </Target>
</Project>
```

Now open "Program.cs" source code file and append the using directive at the top of the source code:

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

Now we need to load the externally defined functions for dynamically loading a library and to load the reset function from "libChapThree.so" library. Append the following codes inside Program class:

```
[DllImport("dl")]
static extern IntPtr dlopen(string file, int flag = 1);

[DllImport("dl")]
static extern IntPtr dlsym(IntPtr handle, string symbol);

[DllImport("dl")]
static extern int dlclose(IntPtr handle);

[DllImport("ChapFour")]
static extern void ResetMyVariable();

delegate void ResetMyVariable_dt();
```

Finally replace Main method with the following snippet:

```
static unsafe void Main(string[] args)
{
    var libPath = "./libChapFour.so";
    IntPtr libraryHandle = dlopen(libPath);
    int* ptr = (int*)dlsym(libraryHandle, "MyVariable").ToPointer();
    Console.WriteLine("MyVariable Value: {0}", *ptr);

    *ptr = *ptr + 1;
    Console.WriteLine("Incremented Variable: {0}", *ptr);

    ResetMyVariable();
    Console.WriteLine("MyVariable after CLR Loaded ResetMyVariable Execution: {0}", *ptr);

    var resetVariable = Marshal.GetDelegateForFunctionPointer<ResetMyVariable_dt>
        (dlsym(libraryHandle, "ResetMyVariable"));
    resetVariable();
    Console.WriteLine("MyVariable after DL Loaded ResetMyVariable Execution: {0}", *ptr);

    dlclose(libraryHandle);
}
```


There are few things happening here:

1. ResetMyVariable_dt is a delegate type declaration to convert a simple function pointer to a usable function in C#. The _dt suffix is an acronym for delegate type.
2. The optional parameter, flag, in dlopen function is default to RTLD_LAZY flag which is 0x00001.
3. Method must be marked as Unsafe whenever pointer datatype such as "int*" is used.
4. Library Path is simply a path to the library that is in the current directory.
5. The dlsym returns a pointer to a symbol which then can be interacted with in C# program and C library.
6. When running this code in Mono and Dotnet Core, the result will be drastically different from one another. Mono will load the library only once and share the same handle while Dotnet core would create 2 different instances of the same library and therefore data that are defined in each library will not match.

Your source code for Program.cs should look like below:

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

namespace ChapterFour
{
    class Program
    {
        [DllImport("dl")]
        static extern IntPtr dlopen(string file, int flag = 1);

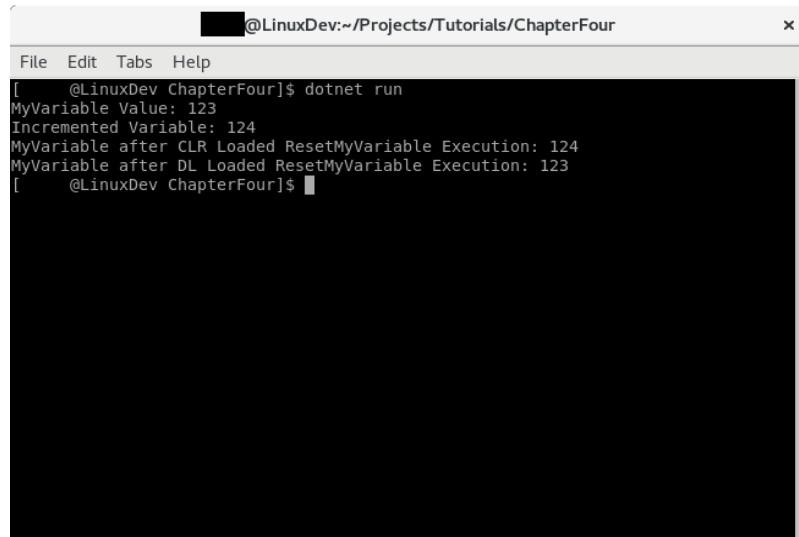
        [DllImport("dl")]
        static extern IntPtr dlsym(IntPtr handle, string symbol);

        [DllImport("dl")]
        static extern int dlclose(IntPtr handle);

        [DllImport("ChapFour")]
        static extern void ResetMyVariable();

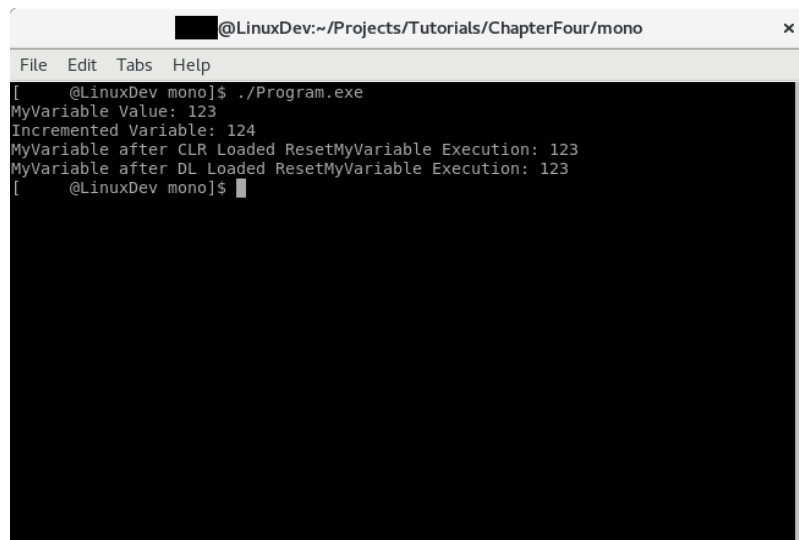
        delegate void ResetMyVariable_dt();
        static unsafe void Main(string[] args)
        {
            var libPath = "./libChapFour.so";
            IntPtr libraryHandle = dlopen(libPath);
            int* ptr = (int*)dlsym(libraryHandle, "MyVariable").ToPointer();
            Console.WriteLine("MyVariable Value: {0}", *ptr);
            *ptr = *ptr + 1;
            Console.WriteLine("Incremented Variable: {0}", *ptr);
            ResetMyVariable();
            Console.WriteLine("MyVariable after CLR Loaded ResetMyVariable Execution: {0}", *ptr);
            var resetVariable = Marshal
                .GetDelegateForFunctionPointer
                (<ResetMyVariable_dt>
                 (dlsym(libraryHandle, "ResetMyVariable")));
            resetVariable();
            Console.WriteLine("MyVariable after DL Loaded ResetMyVariable Execution: {0}", *ptr);
            dlclose(libraryHandle);
        }
    }
}
```

In dotnet core, we have the following:

A terminal window titled "@LinuxDev:~/Projects/Tutorials/ChapterFour" with a menu bar (File, Edit, Tabs, Help). The output shows the execution of a .NET application. The variable MyVariable starts at 123, is incremented to 124 by the CLR, and then reset to 123 by the DL library.

```
@LinuxDev ChapterFour]$ dotnet run
MyVariable Value: 123
Incremented Variable: 124
MyVariable after CLR Loaded ResetMyVariable Execution: 124
MyVariable after DL Loaded ResetMyVariable Execution: 123
[ @LinuxDev ChapterFour]$
```

In Mono, we have the following:

A terminal window titled "@LinuxDev:~/Projects/Tutorials/ChapterFour/mono" with a menu bar (File, Edit, Tabs, Help). The output shows the execution of a Mono application. The variable MyVariable starts at 123, is incremented to 124 by the CLR, and then reset to 123 by the DL library. The output is identical to the dotnet run screenshot.

```
@LinuxDev mono]$ ./Program.exe
MyVariable Value: 123
Incremented Variable: 124
MyVariable after CLR Loaded ResetMyVariable Execution: 123
MyVariable after DL Loaded ResetMyVariable Execution: 123
[ @LinuxDev mono]$
```

4.3 The Delegate Approach

The difference in each output is highlighted and as you can see, when executing CLR loaded function, MyVariable stored in library loaded by DL will not be affected as it affects only CoreCLR's copy of MyVariable. Mono runtime uses the same handle as DL library handle, so therefore MyVariable will be affected. There is a different approach to tackle this particular issue. We can limit our use of P/Invoke services provided by CLR to just loading DL library itself. We can use DL library to load all externally defined functions and symbols within C# and keep result and behavior consistent as demonstrated in both consoles above on the DL Loaded ResetMyVariable Execution line. This approach is referred as the Delegate Approach.

TODO: Add Delegate Stub and Project to demonstrate how to work around the issue described above.

Chapter 5

Runtime Generated Functions for C

