Advanced C# for Low-Level Programming

Tyler Crandall

November 2018

Book Contributors

- Jamesbascle - For grammar corrections.

- Topping - For grammar corrections.

- SirJosh#3917 - For correcting Chapter 4 Snippet, renamed chapFourLib to lib variable.

- Monica#7056 - For suggestions and improvements.

Contents

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 which contains an instruction to install Dotnet Core on your respective distribution:

https://www.microsoft.com/net/download/linux

Clang/LLVM Compilers can be installed or compiled on your respective linux distribution, the table below will get you started.

Linux Distribution	Command Line or Link
Arch Linux	pacman -S llvm clang
Ubuntu	apt.llvm.org/
Debian	apt.llvm.org/
Red Hat Enterprise Linux	developers.redhat.com/blog/2018/07/07/yum-install-gcc7-clang/
Fedora	dnf install llvm clang
CentOS	Compile LLVM/Clang yourself ¯_(ツ)_/¯
OpenSUSE	zypper install llvm clang
Gentoo Linux	https://wiki.gentoo.org/wiki/Clang
Slackware Linux	Compile LLVM/Clang yourself ¯_(ツ)_/¯

1.2 Minimum Knowledge

You'll need to have some comprehension of C# and C languages before starting this book though this book will try to walk you through the basic of C/C++.

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 Endianess in mind although it is less of a concern in x86_64 architecture since little endianess 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>netcoreapp2.1</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 Sum(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}", Sum(1, 2)};
```
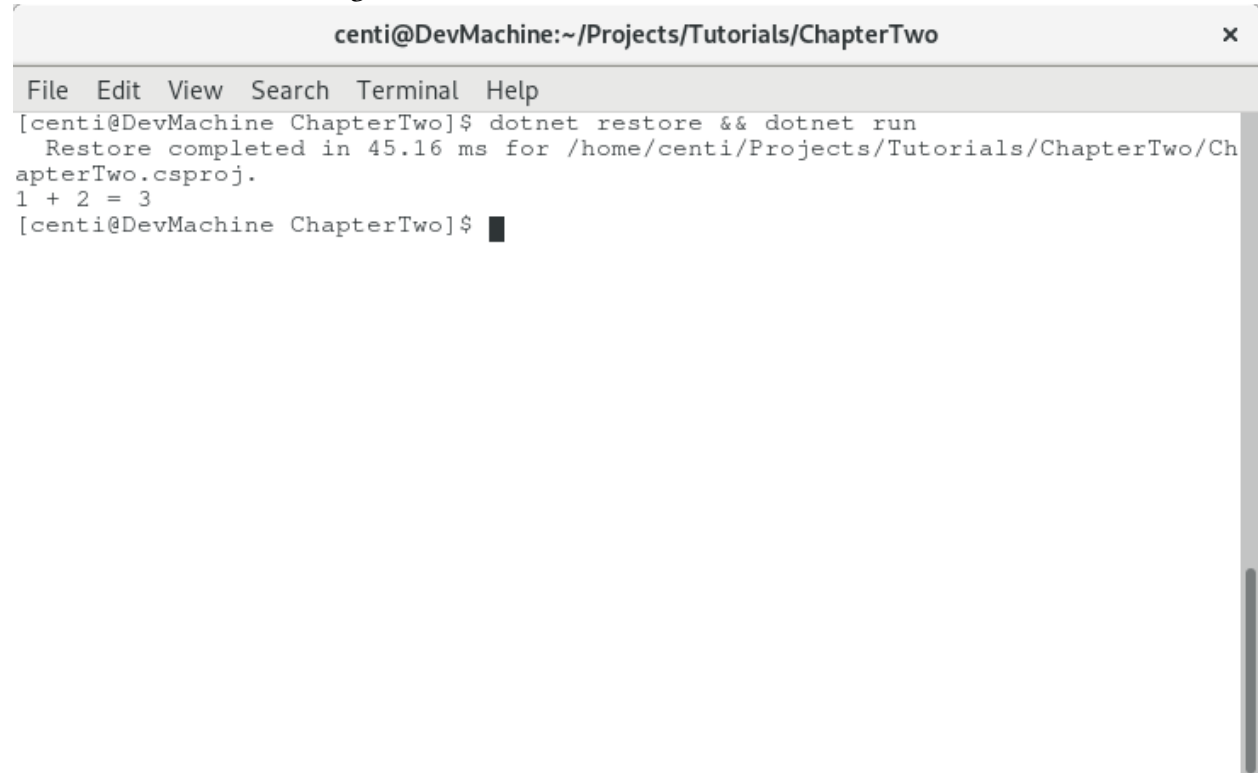
And your source code should look as follows:

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

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

```
dotnet restore && dotnet run
```
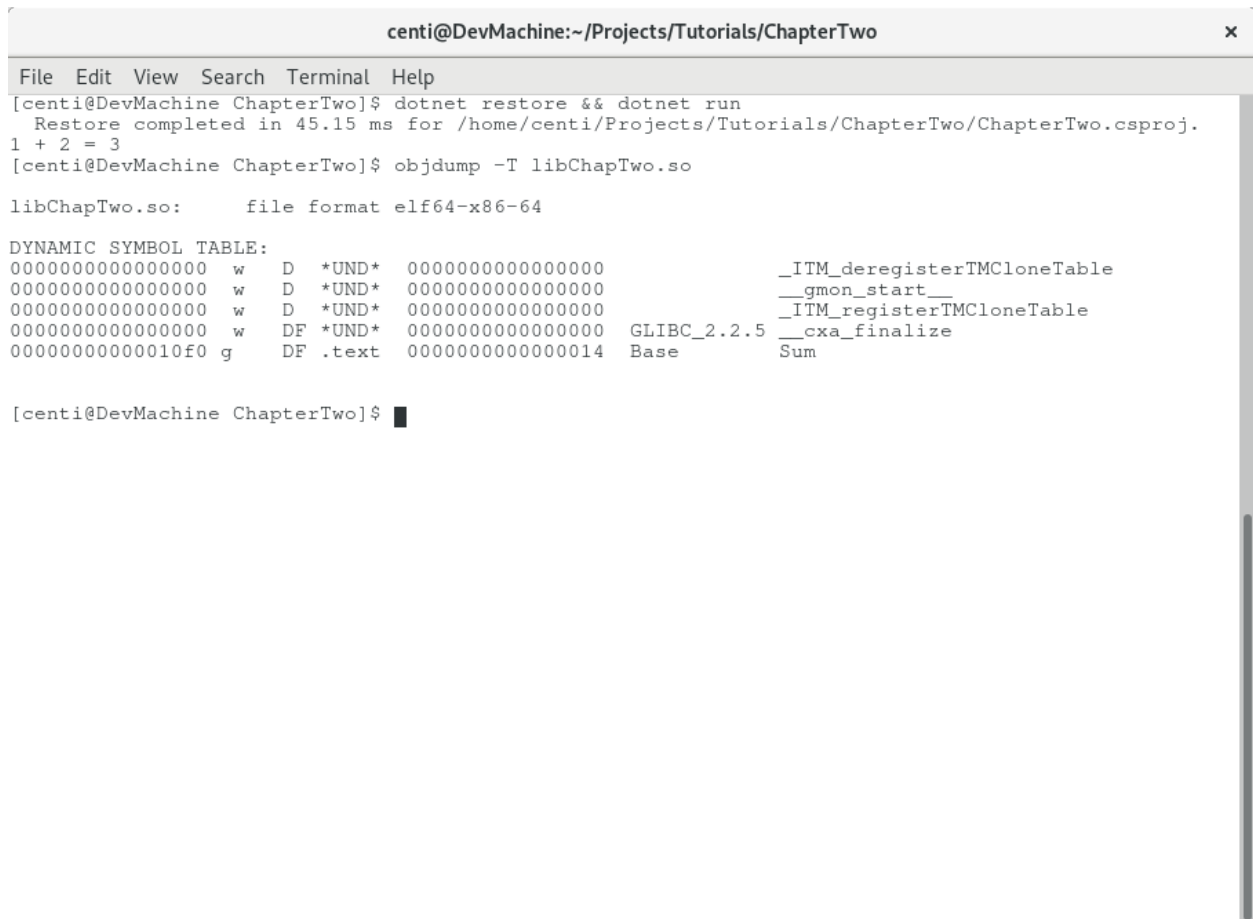
And we have the following:

It works as expected!

2.5   Some Backgrounds

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 "Sum" when P/Invoke defined method is called, 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 that can be resolved to an address to where code or variable is located. You can find a list of symbols by running "objdump -T libChapTwo.so" on your library and you'll have the following:

```
                        centi@DevMachine:~/Projects/Tutorials/ChapterTwo                    ✕

 File  Edit  View  Search  Terminal  Help
[centi@DevMachine ChapterTwo]$ dotnet restore && dotnet run
  Restore completed in 45.15 ms for /home/centi/Projects/Tutorials/ChapterTwo/ChapterTwo.csproj.
1 + 2 = 3
[centi@DevMachine ChapterTwo]$ objdump -T libChapTwo.so

libChapTwo.so:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000  w   D  *UND*  0000000000000000                 _ITM_deregisterTMCloneTable
0000000000000000  w   D  *UND*  0000000000000000                 __gmon_start__
0000000000000000  w   D  *UND*  0000000000000000                 _ITM_registerTMCloneTable
0000000000000000  w   DF *UND*  0000000000000000  GLIBC_2.2.5 __cxa_finalize
00000000000010f0 g   DF .text  0000000000000014  Base        Sum


[centi@DevMachine ChapterTwo]$ █
```

You will notice that the Sum 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.

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

// Let's allocate 12 bytes, or 3 int32_t.
int32_t* MyPointer = (int32_t*)malloc(sizeof(int32_t) * 3);

// 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;

// You can advance the pointer by 4 bytes or by Int32_t
// Compilter would advance the pointer to the size of type that is defined for our
    variable
MyPointer++;

// If you do this however...
MyPointer = (int32_t*)(((int16_t*)MyPointer)++);
// It would advance pointer by 2 bytes or size of int16_t instead of 4 bytes

// You however cannot do this:

MyPointer = (int32_t*)(((void*)MyPointer)++);

// Because no arithmetic operation can be done for void pointer.
// However if you have this instead...

MyPointer = (int32_t*)(((void**)MyPointer)++);

// That would become Pointer to Pointer to Void,
// it would increase by the size of pointer itself
```

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, "Chap-Three.c" and open with your favorite editor.

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

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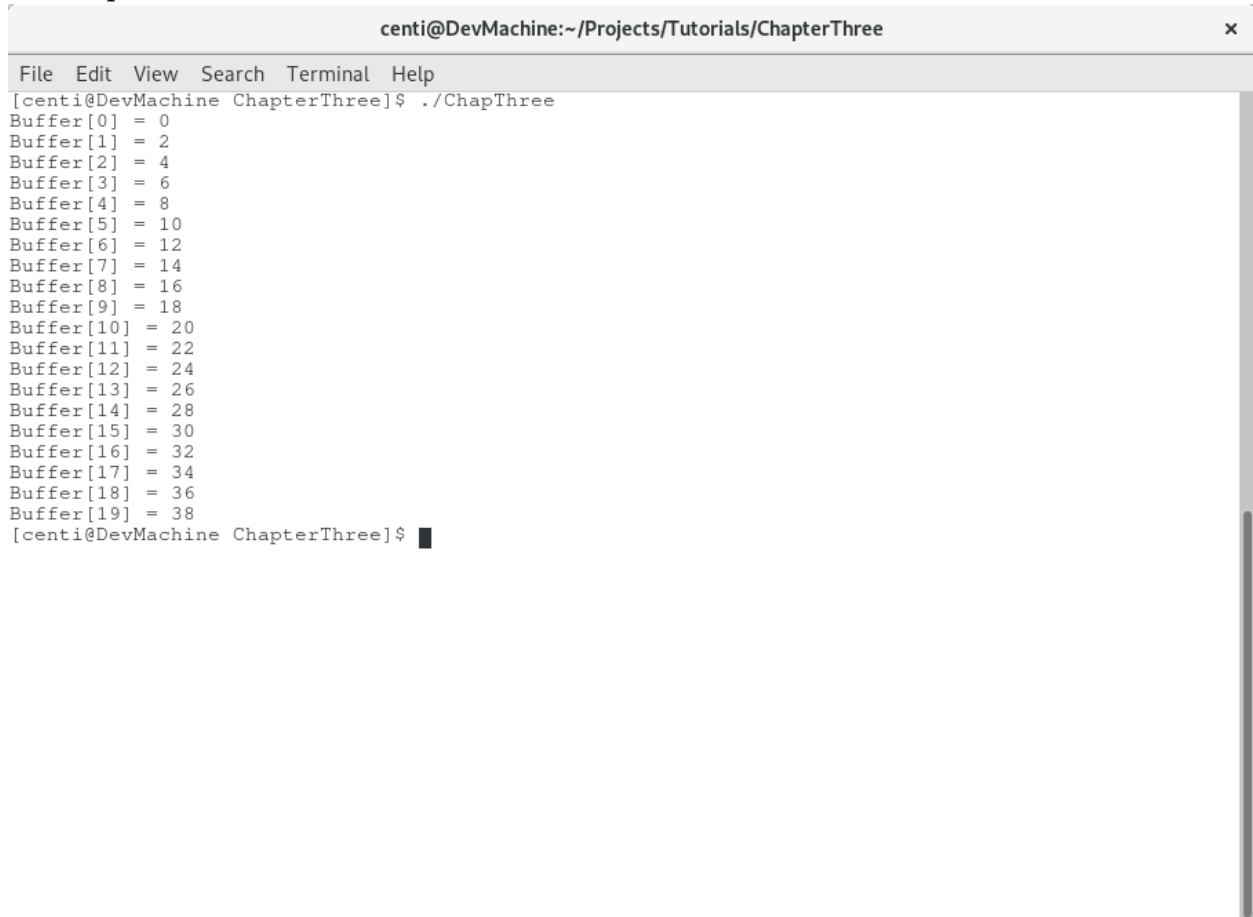
```c
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

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:

```
                       centi@DevMachine:~/Projects/Tutorials/ChapterThree            ×

 File   Edit   View   Search   Terminal   Help
[centi@DevMachine 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
[centi@DevMachine 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 resolves to 4 and then multiplied by 20, so we would have a buffer that have the capacity to hold 20 int32_t elements.

3. Malloc does not zero out the memory by default and in this specific case shown above, it's much more efficient since it's not necessary to zero out the memory since whatever memory/value stored in that allocated memory would be modified immediately after allocating. It however important that you need to zero out or assign a value to the memory otherwise when you attempt to read the said memory it would present an undefined behavior since you can't alway be sure the program will behave consistently when there are random value stored in the allocated memory and being processed by the program.

Because of the fact behind malloc/free that LibC does store information about the pointer that is allocated with those functions, it 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 (*Sum)(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:

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

int32_t (*Sum)(int32_t, int32_t);

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

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

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

    Sum = ActualSumFunction;

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

    Sum = FalseSumFunction;

    printf("Let's try FalseSumFunction: %i\n", Sum(A, B));
}
```
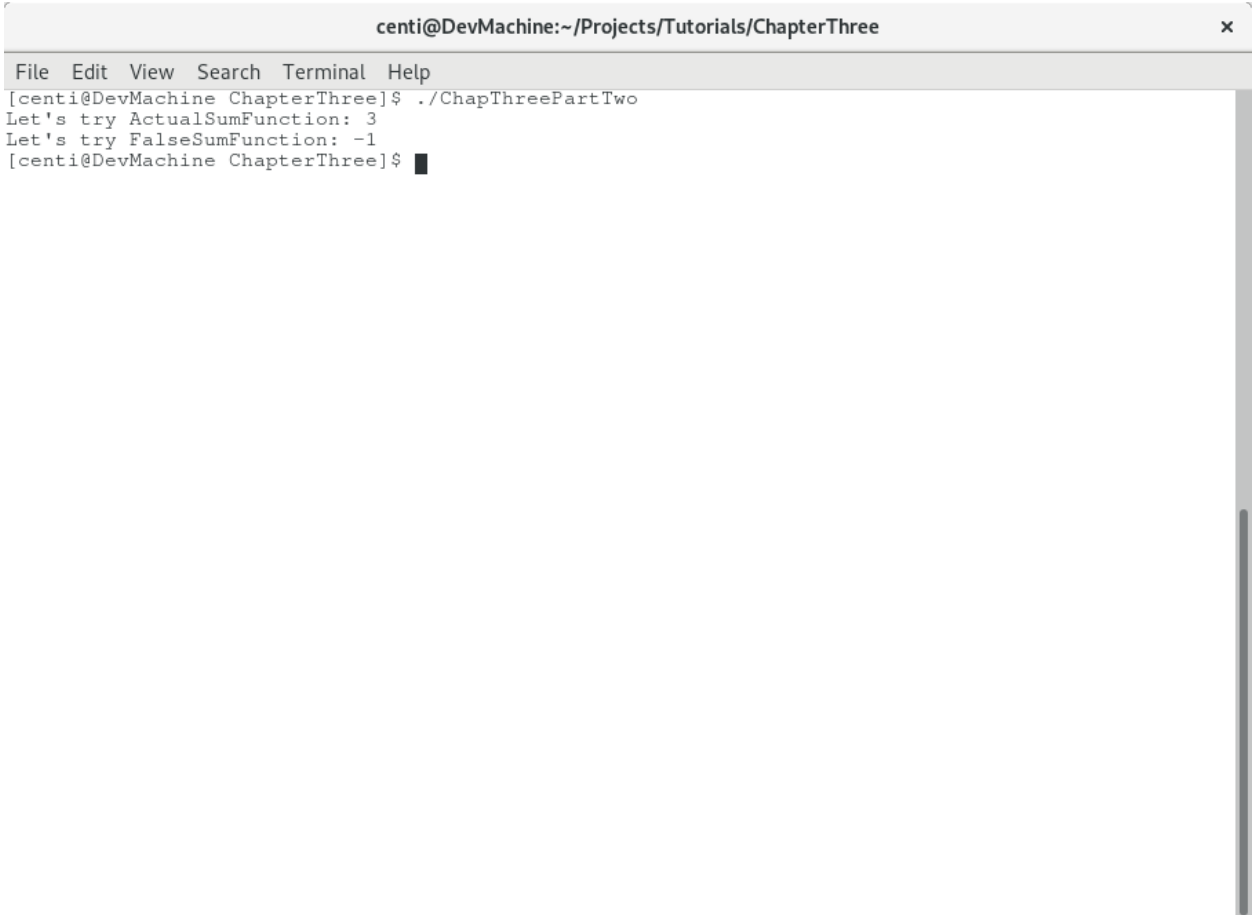
It would produce an output as followed:

```
                              centi@DevMachine:~/Projects/Tutorials/ChapterThree                          ×

  File   Edit   View   Search   Terminal   Help
[centi@DevMachine ChapterThree]$ ./ChapThreePartTwo
Let's try ActualSumFunction: 3
Let's try FalseSumFunction: -1
[centi@DevMachine 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:

```csharp
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;
}
```

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

    *MyIntegerPointer = 123;
  }
}
```

```csharp
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.

When using **unsafe** modifier, 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:

```xml
<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 it's execution.

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

Chapter 4

The History On Delegate Approach

4.1 Note

It is recommended **not** to skip this chapter, because for the remainder of this book, this book will be making an extensive use of Advanced DL Support library. More information can be found here: https://github.com/Firwood-Software/AdvanceDLSupport

4.2 Prior to Nov 2017

There were at the time that variety of CLR implementations for C# does not conform to the same behavior expected for P/Invoke. C# at the time of writing does not have any way to reach the global variable through the normal DllImport attribute approach, and it has to be done by loading libdl and dlopen/dlsym/dlclose. Libdl is a library used to dynamically load external native libraries at runtime and you can retrieve the address to variables or functions by using dlsym which accepts the input for symbol.

In Mono, it would load the external native library with dlopen and you would be sharing the same instance for this external library when using dlopen/dlsym/dlclose. CoreCLR would load the library in other means than dlopen and that would create two instances of the same library which would reflect a different behavior.

4.3   Precursor to Advanced DL Support

ADL, Advanced DL Support, was created shortly after Nov 2017 to work around the problem with P/Invoke and inconsistent behavior with different implementations of CLR. It was initially accomplished by doing the followings in concept:

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

public static class DL {
  [DllImport("dl")]
  public static extern IntPtr dlopen(string library, int flags);
  [DllImport("dl")]
  public static extern IntPtr dlsym(IntPtr libraryHandle, string symbol);
  [DllImport("dl")]
  public static extern int dlclose(IntPtr libraryHandle);
}

public static class MySpecialLibrary {
  internal static IntPtr libraryHandle;
  static MySpecialLibrary()
  {
    libraryHandle = DL.dlopen("MySpecialLibrary.so", 1);
    Sum = Marshal.GetDelegateForFunctionPointer<Sum_dt>(DL.dlsym(libraryHandle, "Sum"));
  }
  public delegate int Sum_dt(int A, int B);
  public static Sum_dt Sum;

  public static void Main()
  {
    Console.WriteLine("1 + 2 = {0}", Sum(1, 2));
  }
}
```

But this is an extremely inefficient approach to wrap native libraries. Advanced DL Support utilizes CIL, Common Intermediate Language, the language that C# compiles to, to generate new types and return new instances of said types for you to utilize native libraries which can be disposed, and therefore do no longer have to be kept around for the duration of the program runtime. You can create new types and code while the program is running and that is thank to the Just-In-Time Compiler. You supplement an interface, abstract class or even a base class to ADL to generate a new type at runtime that binds all of the functions and variables and make it significantly easier to bind native library at an equivalent speed to DllImport attribute approach.

4.4 The Advanced DL Support Approach

Make sure you have a new directory created for Chapter 4 and run the following to initialize your Dotnet Console project:

```
dotnet new console
```

We will need to both reference "AdvancedDLSupport" from Nuget and to add compilation target for C Library that we will be wrapping with for this demonstration.

Your CsProj file should looks like this:

```xml
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="AdvancedDLSupport" Version="2.3.0" />
  </ItemGroup>
  <Target Name="CompileCProject" AfterTargets="AfterBuild">
    <exec Command="clang -std=c99 -shared -fPIC -olibChapFour.so ChapFour.c" />
    <Copy SourceFiles="libChapFour.so" DestinationFolder="$(OutDir)" />
  </Target>
</Project>
```

The C Library source could will simply have a global variable and a function to increment the said variable as followed:
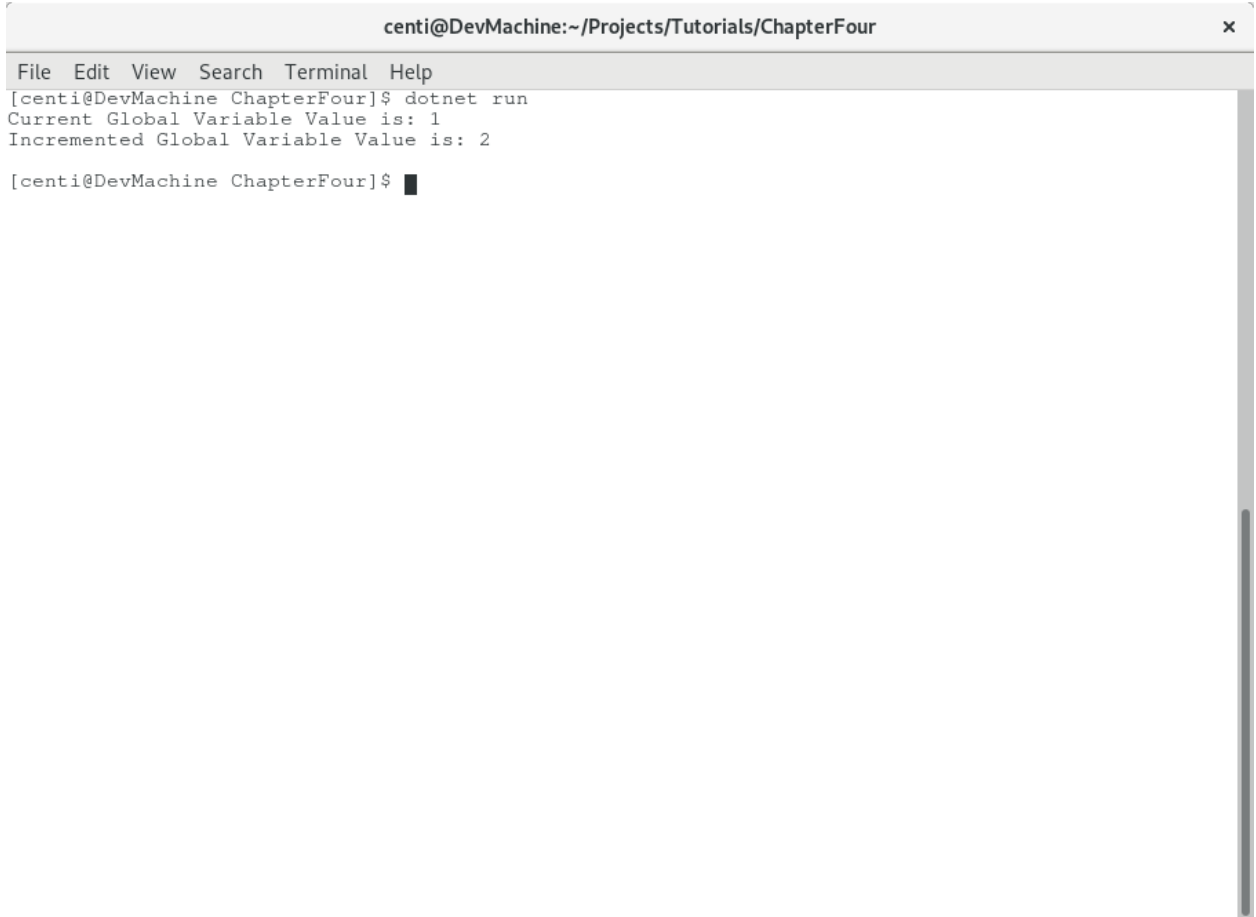
```c
#include <stdlib.h>
#include <stdint.h>

int32_t GlobalVariable = 1;
void IncrementTheGlobalVariable()
{
    ++GlobalVariable;
}
```

For demonstration of ADL, the native library can be binded in C# by simply creating an interface for library and it supports properties:

```csharp
using System;
using AdvancedDLSupport;
public interface IChapFourLib
{
    int GlobalVariable { get; set; }
    void IncrementTheGlobalVariable();
}
static class Program
{
    static void Main()
    {
        var lib = NativeLibraryBuilder.Default.ActivateInterface<IChapFourLib>("ChapFour");
        Console.WriteLine("Current Global Variable Value is: {0}", lib.GlobalVariable);
        lib.IncrementTheGlobalVariable();
        Console.WriteLine("Incremented Global Variable Value is: {0}", lib.GlobalVariable);
        Console.ReadLine();
    }
}
```

The following output from the program will display as followed:

```
centi@DevMachine:~/Projects/Tutorials/ChapterFour                              ×

File  Edit  View  Search  Terminal  Help
[centi@DevMachine ChapterFour]$ dotnet run
Current Global Variable Value is: 1
Incremented Global Variable Value is: 2

[centi@DevMachine ChapterFour]$ █
```

As you can tell, the amount of time saved in writing the binding for native library compared to both DllImport and the Delegate Approach are very significant. You simply only have to write an interface to the native library and that eliminates the need for writing DllImport attribute repeatingly and you can also access the global variable via properties.

Chapter 5

Marshaling between C# and C

For this chapter, you'll need to create a ChapterFive directory and initialize a Dotnet Console project and to reference "AdvancedDLSupport" from nuget.

5.1 Struct Layout

The Layout in Struct is by default set to Sequential and you cannot use Auto layout for marshaling between Managed and Unmanaged code. Explicit Layout allows the you to explicitly define the field offsets in struct layout. This is also what enables you to create a union in struct.

```
// Assuming Ptr = (byte*)MyStruct*;
public struct MyStruct {
  public byte Val1; // Starts at Ptr[0]
  public ushort Val2; // Starts at Ptr[2]
  public uint Val3; // Starts at Ptr[4]
  public byte Val4; // Starts at Ptr[9]
}
```

You may have noticed that the Val1 occupied 2 byte slots in the struct rather than Val2 being placed immediately after Val1. This is due to data alignment. More information on that can be found here: https://software.intel.com/en-us/articles/data-alignment-when-migrating-to-64-bit-intel-architecture

To quote from that link:

> The fundamental rule of data alignment is that the safest (and most widely supported) approach relies on what Intel terms "the natural boundaries." Those are the ones that occur when you round up the size of a data item to the next largest size of two, four, eight or 16 bytes. For example, a 10-byte float should be aligned on a 16-byte address, whereas 64-bit integers should be aligned to an eight-byte address. Because this is a 64-bit architecture, pointer sizes are all eight bytes wide, and so they too should align on eight-byte boundaries. - Intel 2018

The size of the struct shown above is 12 bytes rather than 8 bytes, because the sequential layout rule was being followed. However if you wish to override the behavior on data alignment, you can use Explicit Layout as shown below:

```csharp
[StructLayout(LayoutKind.Explicit)]
public struct Example
{
  [FieldOffset(0)]
  public byte Val1;
  [FieldOffset(2)]
  public ushort Val2;
  [FieldOffset(4)]
  public int Val3;
  [FieldOffset(1)]
  public byte Val4;
}
```

In this struct, the size would become 8 bytes, because there is no padding required for any dangling member to fits in alignment, so everything fits neatly inside the 8 bytes boundary. Also, in explicit layout, the arrangement of members in struct does not affect how memory is laid out, you define the offsets yourself, so you essentially can arrange the members however you like, though it's recommended to keep member arrangement sequential for readability.

5.1.1   Union

Due to the nature of explicit layout, we can overlap where the field are located in memory by using FieldOffset. If you have for an example have the following code:

```csharp
[StructLayout(LayoutKind.Explicit)]
public struct MyStruct
{
  [FieldOffset(0)]
  public sbyte SignedByteVal1;
  [FieldOffset(0)]
  public byte UnsignedByteVal1;
}
```

The size of struct would remain at 1 byte, so no padding is added and both fields are storing into the same memory in the struct.

It is however recommended that you create separate structs for each union definition that you may come across in C native library binding as demonstrated:

```csharp
[StructLayout(LayoutKind.Explicit)]
public struct MyUnion
{
  [FieldOffset(0)]
  public sbyte SignedByteVal1;
  [FieldOffset(0)]
  public byte UnsignedByteVal1;
}

[StructLayout(LayoutKind.Sequential)]
public struct MyStruct
{
  public MyUnion Union;
  public uint A;
}
```

The resulting size of this struct would be 8 bytes, because it still follows the padding rule for sequential in MyStruct by padding the union struct to fits in the memory boundary.

5.1.2   Fixed Size Array

5.1.3   Array of Struct

5.2   Pointer Marshaling

5.3   Function Marshaling