

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 Backgrounds	8
3	Introduction to Pointer	9
3.1	Overview on Pointer	9
3.2	The Function Pointers	13
3.3	C# Counterpart	14
4	The History On Delegate Approach	15
4.1	Note	15
4.2	Prior to Nov 2017	15
4.3	Precursor to Advanced DL Support	16
4.4	The Advanced DL Support Approach	17
5	Marshaling between C# and C Part 1	19
5.1	Struct Layout	19
5.1.1	Union	21
5.1.2	Packing and Size Options for Struct Layout	23
5.1.3	Technical Note on Packing	24
5.1.4	What is the Difference between sizeof and Marshal.SizeOf	25
5.1.5	CoreCLR vs Mono Struct Alignment	26
5.1.6	Fixed Size Array	26
5.1.7	By Value Array of Struct	27
6	Marshaling between C# and C Part 2	29
6.1	String Marshaling	29
6.2	Pointer Marshaling	33
6.3	Function Marshaling	34
6.4	Calling Conventions	35

6.4.1	Cdecl	35
6.4.2	StdCall	35
6.4.3	FastCall	35
6.4.4	ThisCall	35
6.5	Name Mangling	36
6.6	Internal Calls	39
7	Memory Management	41
7.1	Garbage Collection	41
7.2	Manual Memory Management	41
7.3	Memory Consideration when Binding Native Library	42
7.4	Note on Multi-Thread Platform Invocation	44
7.5	All about HandleRef, Saferef and CriticalRef	45
8	Introduction to Common Intermediate Language	47
8.1	CIL Fundamentals	47
8.2	Special Note about the Stack	48
8.3	Static vs Class Member Methods	49
8.4	Introducing the Dynamic Method	50
8.4.1	Dynamic Method Approach	50
8.4.2	Method Builder Approach	51
8.4.3	Assembly Loading via Reflection	52
8.5	Branches in CIL	53
8.6	OpCodes references	54
8.6.1	Note about Strict Emit Utility	54
8.7	Try Catch Finally Stubs	55
8.7.1	Fault Exception Handling	56
8.7.2	Filter Handling	57
9	Advanced CIL Programming	59
9.1	Constructing a Type at Runtime	59
9.1.1	Constructing a Struct	59
9.1.2	Properties	62
10	Embedding Mono and CoreCLR Runtime	63
10.1	Embedding CoreCLR in C Language	63
10.1.1	Introduction to Embedding CoreCLR	63
11	ADL In-depth	73
12	System.Linq.Expressions Programming	75
13	Introduction to Roslyn	77
14	Advanced Roslyn Development	79
15	Extending Roslyn	81

16 Introduction to LLVM	83
17 Advanced LLVM Programming	85
18 Advanced P/Invoke	87
19 Compiler Theory	89
20 Writing Your Own Compiler	91
21 Developing Your Own Runtime	93
22 Conclusion	95

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	<code>pacman -S llvm clang</code>
Ubuntu	<code>apt.llvm.org/</code>
Debian	<code>apt.llvm.org/</code>
Red Hat Enterprise Linux	<code>developers.redhat.com/blog/2018/07/07/yum-install-gcc7-clang/</code>
Fedora	<code>dnf install llvm clang</code>
CentOS	Compile LLVM/Clang yourself <code>^_(^)_/</code>
OpenSUSE	<code>zypper install llvm clang</code>
Gentoo Linux	<code>https://wiki.gentoo.org/wiki/Clang</code>
Slackware Linux	Compile LLVM/Clang yourself <code>^_(^)_/</code>

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));
```

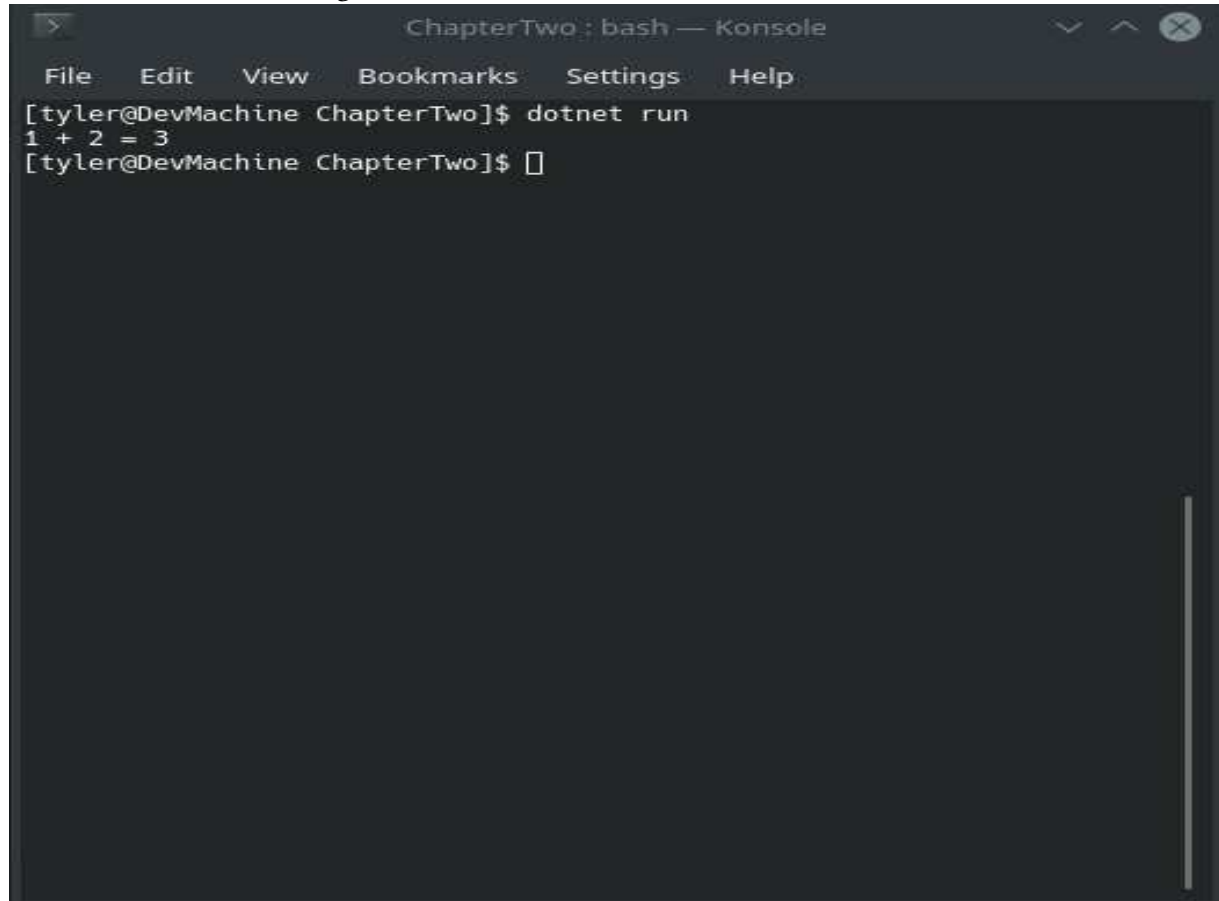
And your source code should look as follows:

```
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:

A screenshot of a terminal window titled "ChapterTwo : bash — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal shows the command "[tyler@DevMachine ChapterTwo]\$ dotnet run" being executed, followed by the output "1 + 2 = 3". The prompt "[tyler@DevMachine ChapterTwo]\$ " is visible on the next line, with a cursor. A vertical scrollbar is on the right side of the terminal window.

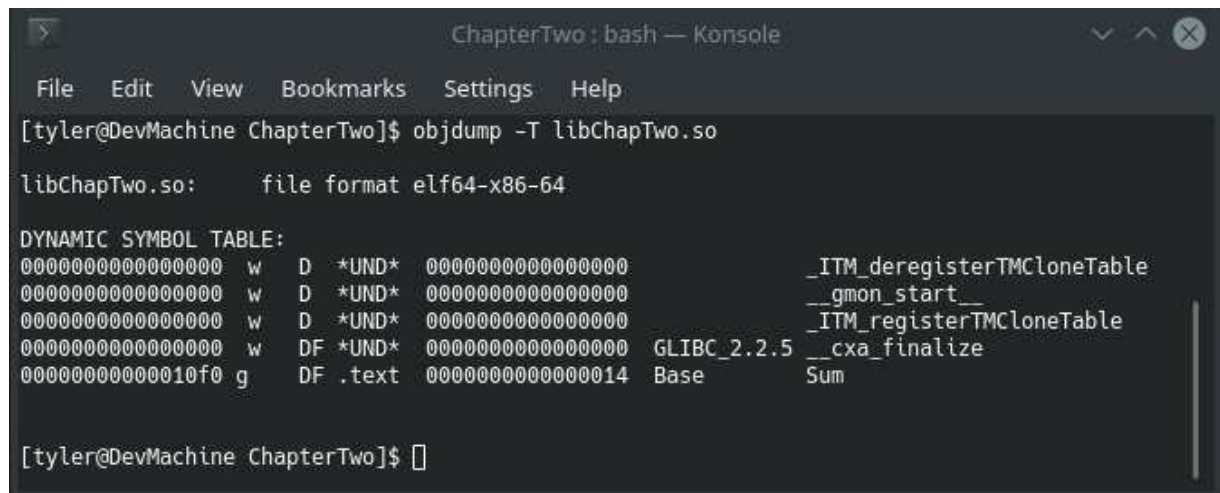
```
ChapterTwo : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterTwo]$ dotnet run
1 + 2 = 3
[tyler@DevMachine ChapterTwo]$ 
```

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:



```
ChapterTwo : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@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
000000000000010f0 g DF .text 0000000000000014      Base      Sum

[tyler@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.

```
#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
// Compiler 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, "ChapterThree.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.

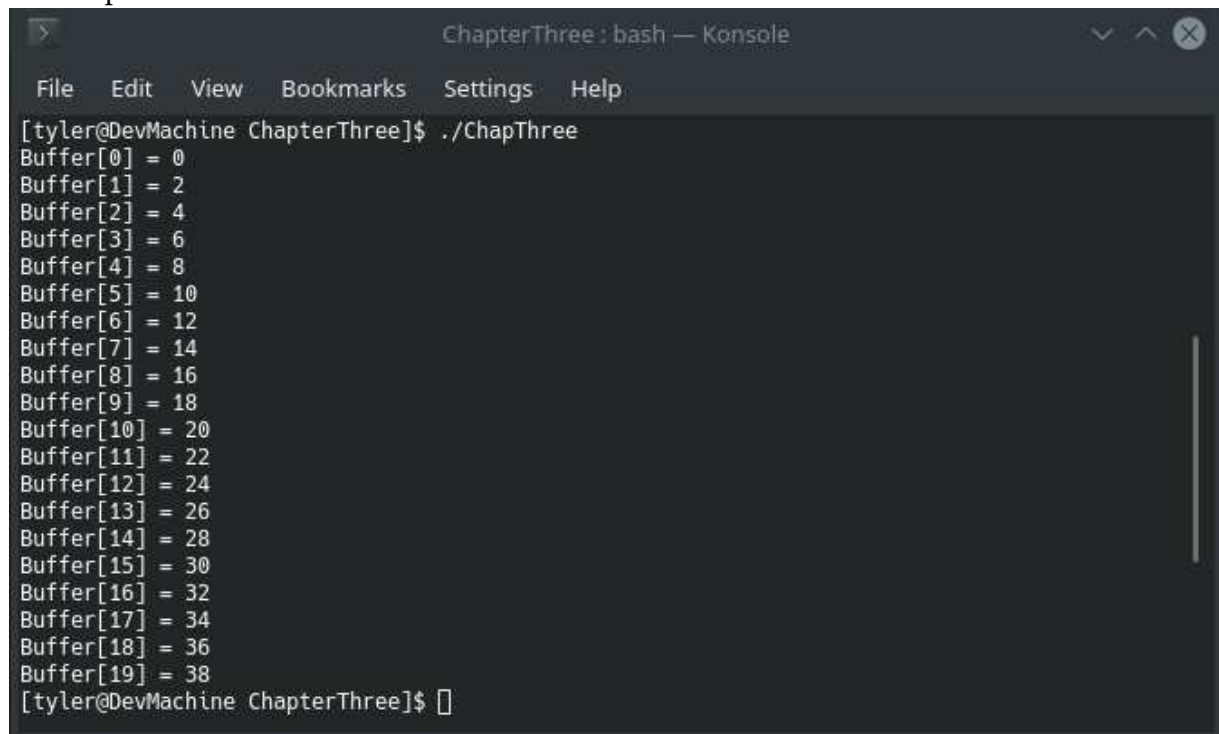
```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

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:

A screenshot of a terminal window titled "ChapterThree : bash — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal shows the command `./ChapThree` being executed, which outputs a list of memory addresses for an array named "Buffer" from index 0 to 19. The values are even numbers starting from 0 and increasing by 2 up to 38. The prompt `[tyler@DevMachine ChapterThree]$` is visible at the bottom.

```
[tyler@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
[tyler@DevMachine ChapterThree]$
```

It may be important to note a few characteristics that may not be fairly apparent about Malloc 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:

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

int32_t (*Sum)(int32_t, int32_t);

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

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

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

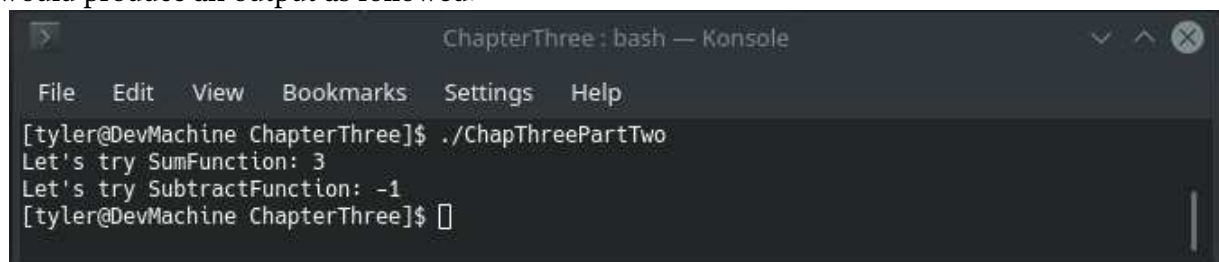
    Sum = SumFunction;

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

    Sum = SubtractFunction;

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

It would produce an output as followed:



```
ChapterThree: bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterThree]$ ./ChapThreePartTwo
Let's try SumFunction: 3
Let's try SubtractFunction: -1
[tyler@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:

```
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.

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:

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

It should look like the followings:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <OutputType>Exe</OutputType>
    <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
  </PropertyGroup>
</Project>
```

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 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:

```
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 name it "ChapterFour" 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 look like this:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.2</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="AdvancedDLSupport" Version="3.0.0" />
  </ItemGroup>
  <ItemGroup>
    <None Include="ChapterFour.c" />
  </ItemGroup>
  <Target Name="CompileCProject" AfterTargets="AfterBuild">
    <exec Command="clang -std=c99 -shared -fPIC -olibChapterFour.so ChapterFour.c" />
    <Copy SourceFiles="libChapterFour.so" DestinationFolder="$(OutDir)" />
  </Target>
</Project>
```

The C Library source could simply have a global variable and a function to increment the said variable as followed:

```
#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:

```
using System;
using AdvancedDLSupport;

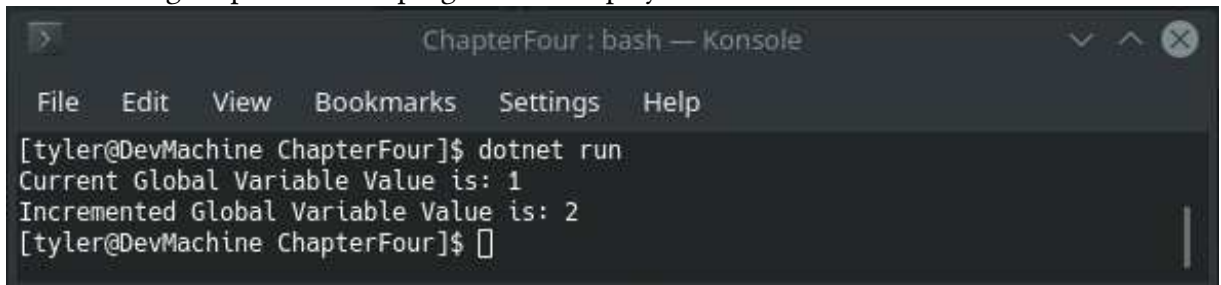
namespace ChapterFour
{
    public static class LibChapterFour
    {
        public static IChapterFour API { get; }

        static LibChapterFour()
        {
            API = NativeLibraryBuilder.Default.ActivateInterface<IChapterFour>("ChapterFour");
        }
    }

    public interface IChapterFour
    {
        int GlobalVariable { get; set; }
        void IncrementTheGlobalVariable();
    }

    static class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Current Global Variable Value is: {0}", LibChapterFour.API.
                GlobalVariable);
            LibChapterFour.API.IncrementTheGlobalVariable();
            Console.WriteLine("Incremented Global Variable Value is: {0}", LibChapterFour.API.
                GlobalVariable);
        }
    }
}
```

The following output from the program will display as followed:

A screenshot of a terminal window titled "ChapterFour: bash — Konsole". The window has a menu bar with "File", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal shows the command "[tyler@DevMachine ChapterFour]\$ dotnet run" and its output: "Current Global Variable Value is: 1" followed by "Incremented Global Variable Value is: 2". The prompt "[tyler@DevMachine ChapterFour]\$ " is visible at the bottom.

```
> ChapterFour: bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterFour]$ dotnet run
Current Global Variable Value is: 1
Incremented Global Variable Value is: 2
[tyler@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 repeatedly and you can also access the global variable via properties.

Chapter 5

Marshaling between C# and C Part 1

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[8]
}
```

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:

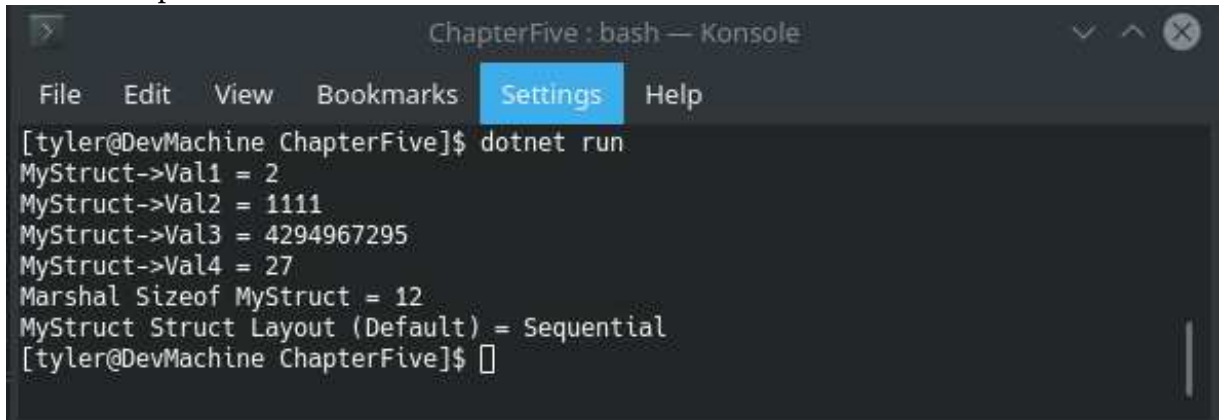
The fundamental rule of data alignment is that the safest (and most widely supported) approach relies on what Intel terms "the natural boundaries." Those are the ones that occur when you round up the size of a data item to the next largest size of two, four, eight or 16 bytes. For example, a 10-byte 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

You can verify the above with the following snippet:

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

namespace ChapterFive
{
    static unsafe class Program
    {
        public struct MyStruct
        {
            public byte Val1;
            public ushort Val2;
            public uint Val3;
            public byte Val4;
        }
        static void Main(string[] args)
        {
            var myStruct = (MyStruct*)Marshal.AllocHGlobal(10).ToPointer();
            myStruct->Val1 = 2;
            myStruct->Val2 = 1111;
            myStruct->Val3 = uint.MaxValue;
            myStruct->Val4 = 27;
            var bytePtr = (byte*) myStruct;
            var intPtr = (IntPtr) bytePtr;
            Console.WriteLine("MyStruct->Val1 = {0}", Marshal.ReadByte(intPtr));
            Console.WriteLine("MyStruct->Val2 = {0}", (ushort)Marshal.ReadInt16(intPtr, 2));
            Console.WriteLine("MyStruct->Val3 = {0}", (uint)Marshal.ReadInt32(intPtr, 4));
            Console.WriteLine("MyStruct->Val4 = {0}", Marshal.ReadByte(intPtr, 8));
            Console.WriteLine("Marshal Sizeof MyStruct = {0}", Marshal.SizeOf<MyStruct>());
            Console.WriteLine("MyStruct Struct Layout (Default) = {0}", typeof(MyStruct).
                StructLayoutAttribute.Value);
        }
    }
}
```

And the output will be:



```
ChapterFive : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterFive]$ dotnet run
MyStruct->Val1 = 2
MyStruct->Val2 = 1111
MyStruct->Val3 = 4294967295
MyStruct->Val4 = 27
Marshal Sizeof MyStruct = 12
MyStruct Struct Layout (Default) = Sequential
[tyler@DevMachine ChapterFive]$
```

The size of the struct shown above is 12 bytes rather than 8 bytes, because the sequential layout rule was being followed. As you can see, by default, we would leave some margin between fields in our struct and that in turn make it slightly wasteful. So if you wish to override the behavior on data alignment, you can use Explicit Layout as shown below:


```
[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 fit 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 fields are located in memory by using `FieldOffset`. If you have for an example have the following code:

```
[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:

```
[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 fit in the memory boundary. You can again, verify this by following the next page snippet and see the result.

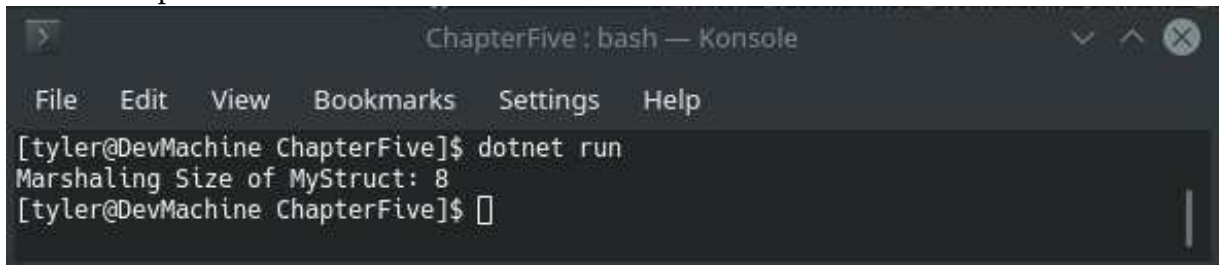
```
using System;
using System.Runtime.InteropServices;

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

public struct MyStruct
{
    public MyUnion Union;
    public uint A;
}

public class Program
{
    static unsafe void Main()
    {
        Console.WriteLine("Marshaling Size of MyStruct: {0}", Marshal.SizeOf<MyStruct>());
    }
}
```

And the output will be as followed:

A screenshot of a terminal window titled "ChapterFive : bash — Konsole". The terminal shows the command "dotnet run" being executed, which outputs "Marshaling Size of MyStruct: 8". The prompt is "[tyler@DevMachine ChapterFive]\$".

```
ChapterFive : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterFive]$ dotnet run
Marshaling Size of MyStruct: 8
[tyler@DevMachine ChapterFive]$
```

5.1.2 Packing and Size Options for Struct Layout

You can specify the packing option to respect byte boundary from a multiple of 1 byte to 8 bytes (.Net Core 3.0 and later will support 16 to 32 bytes boundary.)

Size will only be respected if the specified size is bigger than actual size of struct (including padding) however a different behavior results between Mono and CoreCLR when Size is specified to be smaller than actual size of struct. Read Subsection CoreCLR vs Mono that covers this part.

Here an example to demonstrate the differences of each struct with specified Packing and Size.

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

[StructLayout(LayoutKind.Explicit, Size=16, Pack=8)]
public struct A
{
    [FieldOffset(0)]
    public byte Var1;
}

[StructLayout(LayoutKind.Explicit, Size=1, Pack=8)]
public struct B
{
    [FieldOffset(0)]
    public byte Var1;
    [FieldOffset(1)]
    public ushort Var2;
}

[StructLayout(LayoutKind.Explicit, Pack=8)]
public struct C {
    [FieldOffset(0)]
    public ulong Val1;

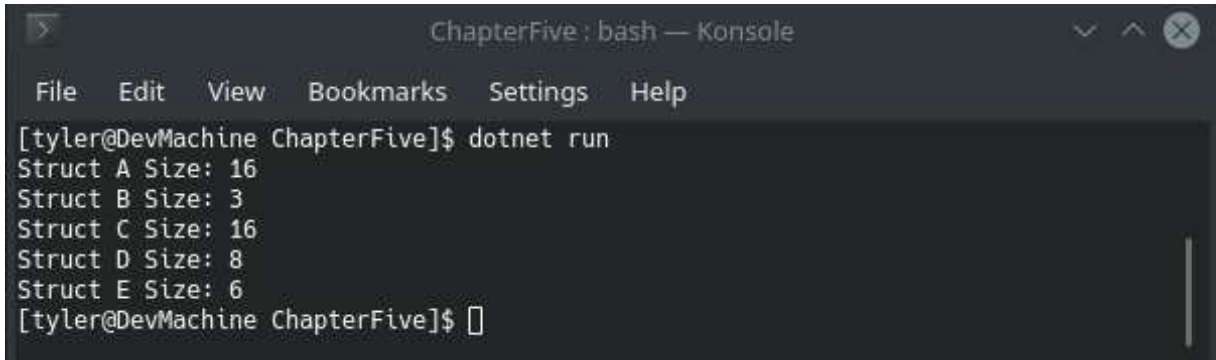
    [FieldOffset((8))]
    public byte Val2;
}

[StructLayout(LayoutKind.Explicit, Pack=8)]
public struct D
{
    [FieldOffset(0)]
    public byte Val1;
    [FieldOffset(1)]
    public int Val2;
}

[StructLayout(LayoutKind.Explicit, Pack=2)]
public struct E
{
    [FieldOffset(0)]
    public byte Val1;
    [FieldOffset(1)]
    public int Val2;
}

public class Program
{
    static void Main()
    {
        Console.WriteLine("Struct A Size: {0}", Marshal.SizeOf<A>());
        Console.WriteLine("Struct B Size: {0}", Marshal.SizeOf<B>());
        Console.WriteLine("Struct C Size: {0}", Marshal.SizeOf<C>());
        Console.WriteLine("Struct D Size: {0}", Marshal.SizeOf<D>());
        Console.WriteLine("Struct E Size: {0}", Marshal.SizeOf<E>());
    }
}
```

And the output for each struct is as followed:



```
ChapterFive : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterFive]$ dotnet run
Struct A Size: 16
Struct B Size: 3
Struct C Size: 16
Struct D Size: 8
Struct E Size: 6
[tyler@DevMachine ChapterFive]$
```

1. Struct A have Size specified and is larger than the actual size of struct with padding (which is 1 byte size for Struct A actual size.)
2. Struct B have size specified to 1 byte size struct with packing alignment of 8 bytes boundary. There are 2 separate behavior that results from this, in CoreCLR, the struct would result as 3 bytes size struct while Mono results in 4 bytes size.
3. Struct C size is 16 bytes even though it's actual size is 9 bytes, this is how padding affects the size of the struct. If padding is set to 1, we would see the Struct C size become 9 bytes long.
4. Struct D size is 8 bytes as described above, the actual size of struct is 5 bytes long and because of the specified padding, it is padded to 8 bytes boundary.
5. Struct E size is 6 bytes as opposed to Struct D, because packing is specified to pad struct to the multiple of 2 bytes.

5.1.3 Technical Note on Packing

While Natural Boundaries are something to keep in mind for 64 bit data alignment, it is done differently for 32 bit architecture, it's packed in the multiple of 4 bytes. And in CoreCLR prior to 3.0, 8 bytes packing is the most that can be used and .Net Core 3.0 and later will have packing supporting 16 and 32 bytes boundary to support `Vector128<T>` and `Vector256<T>`.

5.1.4 What is the Difference between sizeof and Marshal.SizeOf

Marshal.SizeOf tells you how much bytes are needed to allocate the structure in unmanaged environment and sizeof tells you how much memory required to allocate the structure in managed environment. The SizeOf keyword cannot accept managed object member in the struct such as reference type like string due to language limitation (but not technical limitation of CLR), but Unsafe.SizeOf can and it utilize the underlying CIL sizeof opcode.

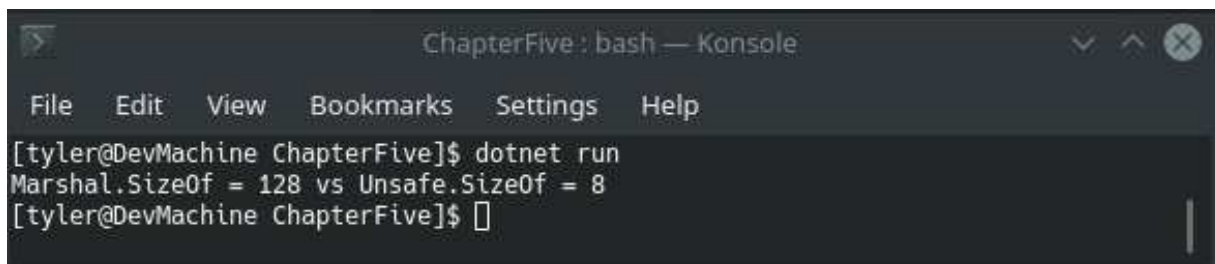
Let's assume we have the following code to demonstrate the difference:

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

[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Ansi)]
public struct A
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string Val1;
}

public class Program
{
    static unsafe void Main()
    {
        Console.WriteLine("Marshal.SizeOf = {0} vs Unsafe.SizeOf = {1}", Marshal.SizeOf<A>(),
            Unsafe.SizeOf<A>());
    }
}
```

And the output of that program will be:

A screenshot of a terminal window titled "ChapterFive: bash — Konsole". The terminal shows the command "dotnet run" being executed, which outputs "Marshal.SizeOf = 128 vs Unsafe.SizeOf = 8". The prompt is "[tyler@DevMachine ChapterFive]\$".

```
ChapterFive: bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterFive]$ dotnet run
Marshal.SizeOf = 128 vs Unsafe.SizeOf = 8
[tyler@DevMachine ChapterFive]$
```

In a managed environment, the string would be a reference type and on 64 bit architecture, the pointer would be 8 bytes long, 4 bytes long on a 32 bit architecture. However, because we explicitly define that our string in a struct is a By Value String, it is essentially a fixed length array of characters of specific encoding (in this case, Ansi Encoding.) The size of struct *should* be 128 bytes long for marshaling purpose.

5.1.5 CoreCLR vs Mono Struct Alignment

In CoreCLR, the struct size is emphasized on the packing, not on alignment, but on Mono, it's emphasized on alignment along with packing. So a struct defined as thus:

```
[StructLayout(LayoutKind.Explicit, Size=1, Pack=8)]
public struct A
{
    [FieldOffset(0)]
    public byte Var1;
    [FieldOffset(1)]
    public ushort Var2;
}
```

Have Marshal.SizeOf result of 3 on CoreCLR while it's 4 on Mono, Mono rounded it up to Natural Alignment Boundary.

5.1.6 Fixed Size Array

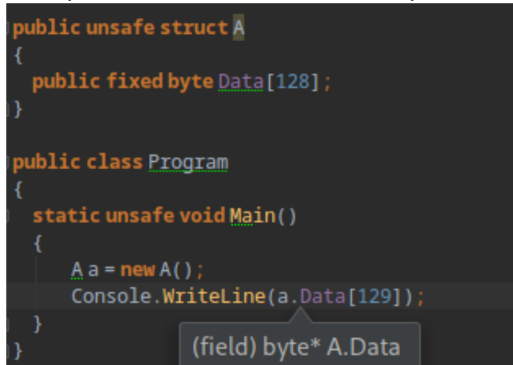
You can allocate a fixed sized array within a struct, but it is restricted to the following types: bool, byte, char, short, int, long, sbyte, ushort, uint, ulong, float, or double. Though there is a proposal for this change to allow user-defined structs to be used for fixed sized array:

<https://github.com/dotnet/csharplang/blob/master/proposals/fixed-sized-buffers.md>

The fixed statement require an unsafe modifier for struct and you can create a fixed size array by writing the following:

```
public unsafe struct A
{
    public fixed byte Data[128];
}
```

This is a struct that is allocated for 128 bytes with a fixed size array of bytes. There is one thing to note about this, fixed size buffer is **NOT** bound checked, it is a pointer to the first element of the array in struct hence this is why unsafe modifier is required.



```
public unsafe struct A
{
    public fixed byte Data[128];
}

public class Program
{
    static unsafe void Main()
    {
        A a = new A();
        Console.WriteLine(a.Data[129]);
    }
}
```

(field) byte* A.Data

5.1.7 By Value Array of Struct

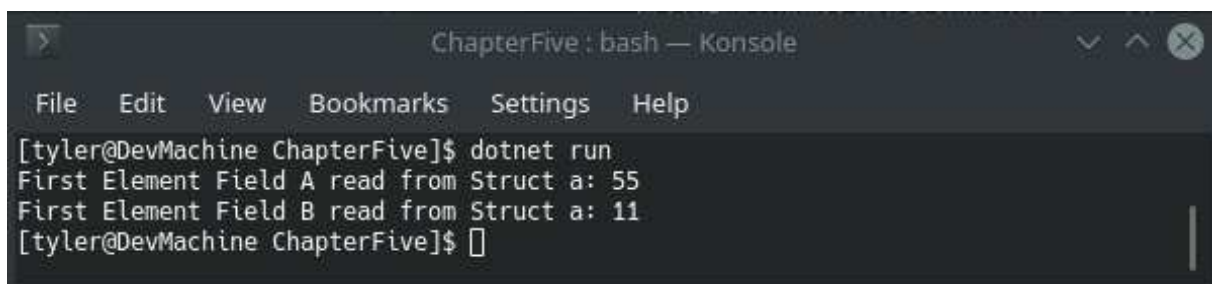
C# does not have a concept for By Value Array of Struct though there is an upcoming proposal to introduce support for this. At this time, fixed statement is restricted to certain primitive types. There are few possible approach that you can accomplish By Value Array of Structs, one of them is to allocate certain amount of bytes to accommodate the space required to store a number of struct elements with a by value byte array. The following code will demonstrate on how to accomplish this:

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

[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct Element
{
    public int A;
    public byte B;
}

public unsafe struct A
{
    public fixed byte Data[640]; // 128 elements * 5 bytes (Size of Element)
}

public class Program
{
    static unsafe void Main()
    {
        A a = new A();
        var ptrToByValueArrayOfStructs = (Element*)a.Data;
        ptrToByValueArrayOfStructs[0].A = 55;
        ptrToByValueArrayOfStructs[0].B = 11;
        Console.WriteLine("First Element Field A read from Struct a: {0}", Marshal.ReadInt32((
            IntPtr)a.Data, 0));
        Console.WriteLine("First Element Field B read from Struct a: {0}", a.Data[4]);
    }
}
```



```
ChapterFive : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterFive]$ dotnet run
First Element Field A read from Struct a: 55
First Element Field B read from Struct a: 11
[tyler@DevMachine ChapterFive]$
```

Another approach is a little more elegant approach that apply similar concept as above:

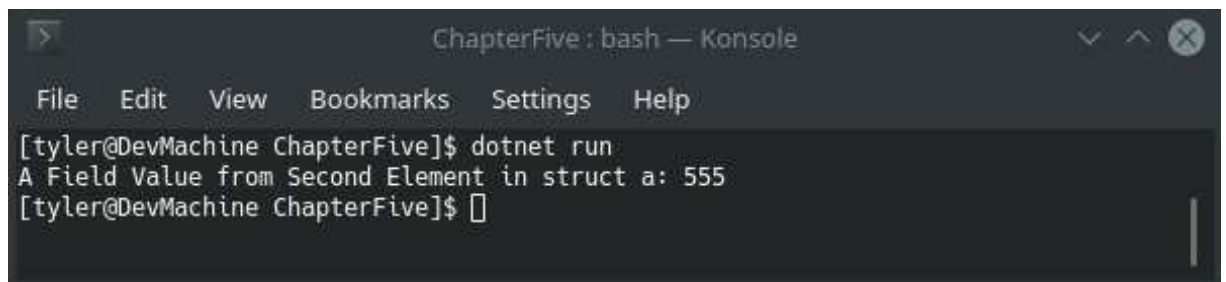
```
using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[StructLayout(LayoutKind.Sequential, Pack = 1)]
public struct Element
{
    public int A;
    public byte B;
}

[StructLayout(LayoutKind.Sequential, Pack = 0, Size = 5 * 128)]
public unsafe struct A
{
    public Element FirstElement;

    public ref Element this[int index]
    {
        get
        {
            if ((uint)(index) > 127)
                throw new IndexOutOfRangeException();
            fixed (Element* pElement = &FirstElement)
            {
                return ref Unsafe.AsRef<Element>(pElement + index);
            }
        }
    }
}

static class Program
{
    static void Main()
    {
        var a = new A();
        a[1].A = 555;
        Console.WriteLine("A Field Value from Second Element in struct a: {0}", a[1].A);
    }
}
```



```
ChapterFive: bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterFive]$ dotnet run
A Field Value from Second Element in struct a: 555
[tyler@DevMachine ChapterFive]$
```

This snippet was originally written by Tanner Gooding.

The idea with this approach is that you leverage the `StructLayout` as mentioned in previous section to allocate large enough struct to accommodate the size of By Value Array and you introduce first element field so that field can be used as a reference to read/store other elements. The indexer property simply make the code easier to read, avoiding unneeded casting on external code and to ensure that it is bound checked. This approach is recommended for By Value Array of Struct Marshaling.

Chapter 6

Marshaling between C# and C Part 2

6.1 String Marshaling

For this part, we'll need to create a dotnet core console project for Chapter 6 and make sure to reference "AdvancedDLSupport" on nuget. You will need to add a build task for this project to compile C code after building C# code. Your CSProj file should have 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 -olibChapSix.so ChapSix.c" />
    <Copy SourceFiles="libChapSix.so" DestinationFolder="$(OutDir)" />
  </Target>
</Project>
```

Over the course of this lesson, we'll go over the differences between By Value String and C Ansi String (char*). It important to keep in mind that in C#, string is a reference type and is an immutable type as well meaning, the content of the string shouldn't be modified during runtime and in that case, we would use StringBuilder which can be used for this situation (we can specify the capacity for initial size of StringBuilder.) The marshaling will marshal the reference to content of string to C native code and back almost seamlessly.

And for this lesson, we will have a few functions to play with for string manipulation from C Library:

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

static char* DefaultMsg = "Hello, this is from native code";

void AnsiString(char* str)
{
    printf("%s\n", str);
}

typedef struct
{
    char StringData[128];
} ByValString;

ByValString* Initialize()
{
    return (ByValString*)malloc(sizeof(ByValString));
}

void SetDefaultMessage(ByValString* val)
{
    strcpy(val->StringData, DefaultMsg);
    val->StringData[strlen(DefaultMsg)] = 0;
}

void SetDefaultMessage2(char* val)
{
    strcpy(val, DefaultMsg);
    val[strlen(DefaultMsg)] = 0;
}
```

As you may have noticed, we are attempting to marshal both C Ansi String and By Value String.

```

using System;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Text;
using AdvancedDLSupport;

[StructLayout(LayoutKind.Explicit, CharSet = CharSet.Ansi, Size = 128, Pack = 1)]
public struct ByValCharArray
{
    [FieldOffset(0)] public byte FirstElement;

    public unsafe string StringVal => Encoding.ASCII.GetString((byte*)Unsafe.AsPointer(ref
        FirstElement), 128);
    public unsafe char this[int index]
    {
        get => StringVal[index];
        set => ((byte*)Unsafe.AsPointer(ref FirstElement))[index] = Encoding.ASCII.GetBytes(new
            [] {value})[0];
    }
}

public unsafe interface IChapterSixLib
{
    void AnsiString(char* str);
    void AnsiString([MarshalAs(UnmanagedType.LPStr)] string str);
    void AnsiString(ref ByValCharArray val);
    IntPtr Initialize();
    void SetDefaultMessage(ref ByValCharArray val);
    void SetDefaultMessage2([MarshalAs(UnmanagedType.LPStr)] StringBuilder val);
}

public static class ChapterSixLibrary
{
    public static IChapterSixLib API { get; }

    static ChapterSixLibrary()
    {
        API = NativeLibraryBuilder.Default.ActivateInterface<IChapterSixLib>("ChapterSix");
    }
}

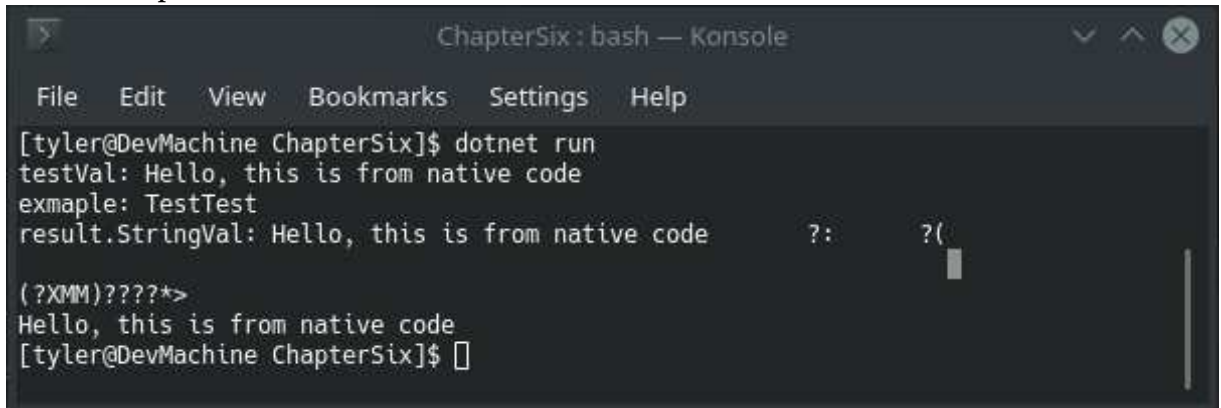
public class Program
{
    static unsafe void Main()
    {
        var testVal = new StringBuilder(128);
        ChapterSixLibrary.API.SetDefaultMessage2(testVal);
        var example = new StringBuilder("Test\0Test");
        var result = Unsafe.AsRef<ByValCharArray>(ChapterSixLibrary.API.Initialize().ToPointer())
            ;
        ChapterSixLibrary.API.SetDefaultMessage(ref result);

        Console.WriteLine("testVal: {0}", testVal);
        Console.WriteLine("exmaple: {0}", example);
        Console.WriteLine("result.StringVal: {0}", result.StringVal);

        ChapterSixLibrary.API.AnsiString(ref result);
    }
}

```

And the output will be as followed:



```
ChapterSix : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterSix]$ dotnet run
testVal: Hello, this is from native code
exmaple: TestTest
result.StringVal: Hello, this is from native code
(?XMM)????*>
Hello, this is from native code
[tyler@DevMachine ChapterSix]$
```

There are few things happening here, the 'testVal' line demonstrate the string builder with the capacity of 128 characters, but when running SetDefaultMessage2, the Marshaler will determine the size of string written to StringBuilder with either strlen or wcslen depending on the character set for SetDefaultMessage2 and have content of string copied over to StringBuilder. Due to the nature of strlen and wcslen, it will copy the string up to null terminated character and you can test this by adding \0 in DefaultMsg string in C source file and P/Invoke Marshaler will null terminated the string copying.

The 'example:' line demonstrate that StringBuilder will still print even with null terminated character in C#, it does not terminate the string at the first null terminated character, this illustrate that the process seen in first line is purely done by P/Invoke Marshaling.

The 'result.StringVal' line is a little more complex, but what is going on is that the ByValCharArray is quite literally a ValueType that store string data within it and it have a fixed size of 128 characters and when running SetDefaultMessage, it would print a string that includes null terminated characters and other data that aren't zeroed out when allocated, P/Invoke Marshaler have no process or handle on how this data is copied, this is useful if this the intended behavior you desire.

The last line demonstrate the print of null terminated string that is read from third line.

The ValueType char array can be useful for recycling same data container without allocating/reallocating/disposing data in an iteration and thus enabling a more efficient code and less memory pressure on the process.

6.2 Pointer Marshaling

Pointer marshaling is a little tricky in .Net due to the barrier between Managed and Unmanaged Programming, the memory ownership lesson will be covered in Chapter 7 that go in-depth on Garbage Collection and Manual Memory Management. There are few things to keep in mind:

A parameter with ref modifier indicate that a **reference** to the parameter should be supplied. Hypothetically if we have the following code:

```
void DoStuff(ref IntPtr ptr);
```

It would make a reference to the IntPtr parameter itself similarly to the following snippet in C Programming Language:

```
void* ptr;  
void** refParameter = &ptr;
```

The ref modifier can be applied to class, struct, primitive types, and even a delegate parameters.

C# can marshal pointer types as function parameters and return type if unsafe modifier is applied.

6.3 Function Marshaling

Function pointer is a part of Pointer Marshaling, but it's represented differently in C#.

You can declare a delegate type like so:

```
public delegate void MyFunc_dt(ref int val);  
public MyFunc_dt MyFunc;
```

When you declare a field, MyFunc it acts similarly to a function pointer, but it's a managed type which contains additional information. You can't directly declare an unmanaged function pointer a delegate as a field **in a struct** and make a pointer of that struct. There is a proposal being championed for this addition to support this behavior: <https://github.com/dotnet/csharplang/issues/80>

The current workaround for most circumstances is to use Marshal.FunctionPointerToDelegate or to define explicitly for the field to marshal as function pointer like so:

```
public delegate void Test_dt();  
public struct MyStruct  
{  
    [MarshalAs(UnmanagedType.FunctionPtr)]  
    public Test_dt Test;  
}
```

6.4 Calling Conventions

While this section will not go in-depth on the topic of calling convention, it's important to go over it. There are 4 calling conventions to know about: StdCall, Cdecl, FastCall, and ThisCall. It should be noted that WinApi although listed as one of CallingConvention member, it's not an actual calling convention however, it uses the default platform specific calling convention for Windows API which Stdcall is default on Windows and Cdecl for Windows CE.Net.

6.4.1 Cdecl

Cdecl calling convention have arguments passed on the stack in right to left order and return value would be returned on EAX register. The calling function cleans the stack, to clarify this, if for an example, that you are using a DllImport function, your program is the one that have to clean the stack, not the called function that you're calling to.

6.4.2 StdCall

StdCall is otherwise known as "Windows API" calling convention and is used almost exclusively by Microsoft platform. It behaves similar to Cdecl except for one important difference, the called function cleans the stack, not your program. It essentially does not allow variable length argument lists.

6.4.3 FastCall

The standard for FastCall is not consistent across compilers, so it should be used with caution. It's intended to be used when there are at most 3 32 bits arguments on registers and it's intended to be used for performance sensitive code. It's similar to CDecl where the calling function cleans the stack.

6.4.4 ThisCall

On GCC, It's nearly identical to Cdecl, except it push the this pointer onto the stack last as if it's the first parameter in the function prototype.

On Microsoft Visual C++ Compiler, the this pointer get passed in ECX register and the called function clean the stack similarly to StdCall.

6.5 Name Mangling

In low level programming language other than C tend to leverages name mangling so it can create a distinction between other class members with the same name by composing function name with arguments, class that it resides in, and namespace it's also resides in and etc. Let's assume we have the following C++ Snippet:

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

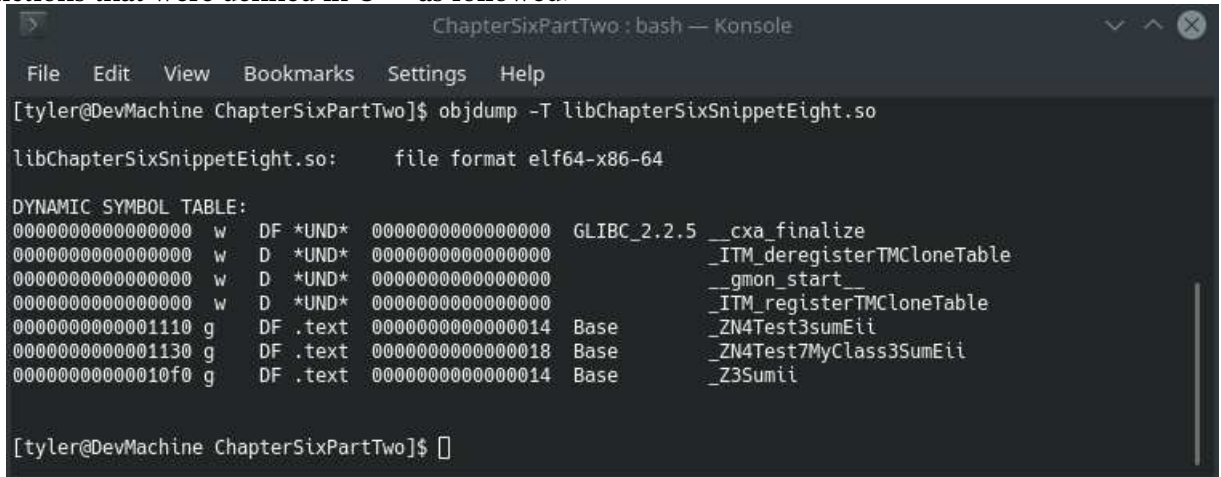
namespace Test
{
    int sum(int a, int b)
    {
        return a + b;
    }
    extern "C" class MyClass {
    public:
        int Sum(int a, int b);
    };

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

And build our library like so:

```
clang++ -shared -fPIC -olibChapterSixPartTwo.so ChapterSixPartTwo.cpp
```

Now when we attempts to dump our Symbol Tables, we would have a mangled names for our functions that were defined in C++ as followed:



```
ChapterSixPartTwo : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterSixPartTwo]$ objdump -T libChapterSixSnippetEight.so

libChapterSixSnippetEight.so:      file format elf64-x86-64

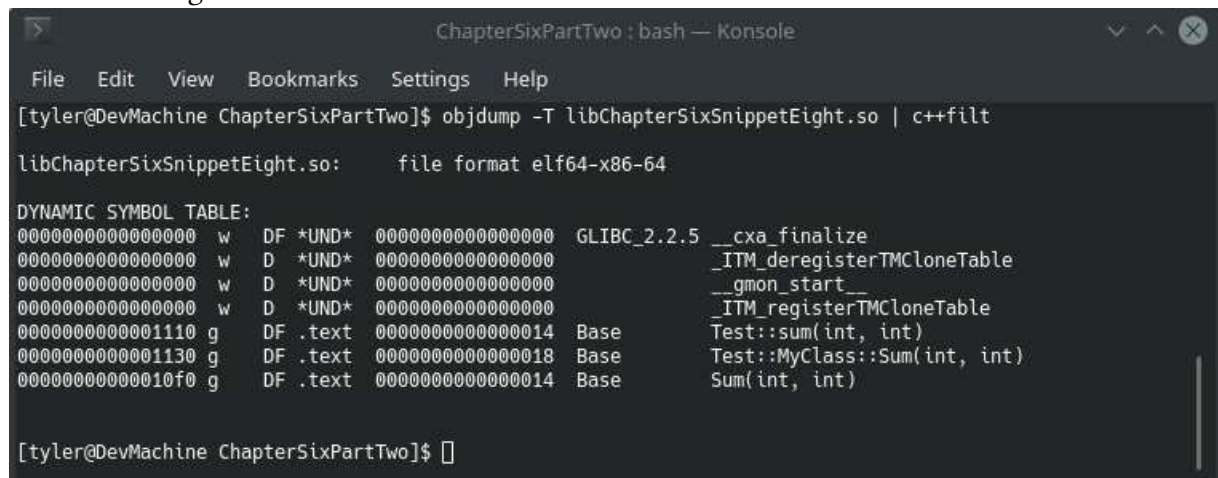
DYNAMIC SYMBOL TABLE:
0000000000000000 w DF *UND* 0000000000000000 GLIBC_2.2.5 __cxa_finalize
0000000000000000 w D *UND* 0000000000000000 _ITM_deregisterTMCloneTable
0000000000000000 w D *UND* 0000000000000000 _gmon_start__
0000000000000000 w D *UND* 0000000000000000 _ITM_registerTMCloneTable
0000000000001110 g DF .text 0000000000000014 Base _ZN4Test3sumEii
0000000000001130 g DF .text 0000000000000018 Base _ZN4Test7MyClass3SumEii
00000000000010f0 g DF .text 0000000000000014 Base _Z3Sumii

[tyler@DevMachine ChapterSixPartTwo]$
```

As you can see, our names are all mangled, we can unmangle the name to give us a better indication of which function it is for each one by running the following command:

```
objdump -T libChapterSixSnippetEight.so | c++filt
```


Which would give us this:



```
ChapterSixPartTwo : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterSixPartTwo]$ objdump -T libChapterSixSnippetEight.so | c++filt
libChapterSixSnippetEight.so:      file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000 w DF *UND* 0000000000000000 GLIBC_2.2.5 __cxa_finalize
0000000000000000 w D *UND* 0000000000000000 _ITM_deregisterTMCloneTable
0000000000000000 w D *UND* 0000000000000000 _gmon_start__
0000000000000000 w D *UND* 0000000000000000 _ITM_registerTMCloneTable
0000000000001110 g DF .text 0000000000000014 Base Test::sum(int, int)
0000000000001130 g DF .text 0000000000000018 Base Test::MyClass::Sum(int, int)
00000000000010f0 g DF .text 0000000000000014 Base Sum(int, int)

[tyler@DevMachine ChapterSixPartTwo]$
```

Now we can make a distinction of each function, you can call to all three C++ functions by providing the correct P/Invoke interface. Remember, C++ function utilizes this pointer for any non-static function which is a part of ThisCall calling convention.

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

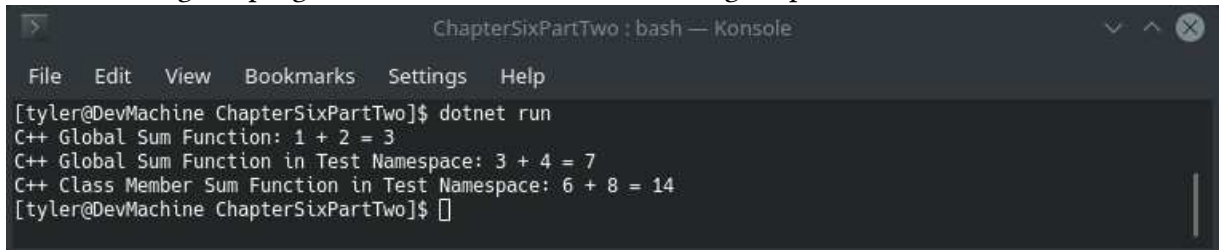
namespace ChapterSixPartTwo
{
    public static class LibraryChapterSixSnippetEight
    {
        public static IChapterSixSnippetEightLibrary API { get; }

        static LibraryChapterSixSnippetEight()
        {
            API = NativeLibraryBuilder.Default.ActivateInterface<IChapterSixSnippetEightLibrary>
                >("ChapterSixSnippetEight");
        }
    }

    public interface IChapterSixSnippetEightLibrary
    {
        [NativeSymbol("_Z3Sumii")]
        int Sum(int a, int b);
        [NativeSymbol("_ZN4Test3sumEii")]
        int Sum2(int a, int b);
        [NativeSymbol("_ZN4Test7MyClass3SumEii", CallingConvention = CallingConvention.ThisCall)]
        int Sum3(IntPtr thisPtr, int a, int b);
    }

    static class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("C++ Global Sum Function: 1 + 2 = {0}",
                LibraryChapterSixSnippetEight.API.Sum(1, 2));
            Console.WriteLine("C++ Global Sum Function in Test Namespace: 3 + 4 = {0}",
                LibraryChapterSixSnippetEight.API.Sum2(3, 4));
            Console.WriteLine("C++ Class Member Sum Function in Test Namespace: 6 + 8 = {0}",
                LibraryChapterSixSnippetEight.API.Sum3(IntPtr.Zero, 6, 8));
        }
    }
}
```

When running the program, we would have the following output:



```
ChapterSixPartTwo : bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine ChapterSixPartTwo]$ dotnet run
C++ Global Sum Function: 1 + 2 = 3
C++ Global Sum Function in Test Namespace: 3 + 4 = 7
C++ Class Member Sum Function in Test Namespace: 6 + 8 = 14
[tyler@DevMachine ChapterSixPartTwo]$
```

It should be noted that there are multiple variants of name mangling for C++ and other low level programming languages that you should ship your own low-level binary to ensure that the name mangling would be match with the P/Invoke C# interface provided in your code, otherwise you may have to implement something similar to `Objdump` and `C++filt` to dynamically rewrite your C# interface for binding C++ library across a variety of platforms which is no easy task to accomplish. It may change with `AdvancedDL Support Library` to support a number of name mangling in the future.

6.6 Internal Calls

The internal call is essentially an internally registered function to the run-time, .Net Framework make an extensive use of this particular function that is often implemented in run-time either for optimization or simplicity with underlying run-time code. It should be noted however that Internal Call often does not use any Marshaling by P/Invoke since that is handled at native level, it's very similar to Calli Opcode which allows you to make an invocation without any marshaling from P/Invoke which give you greater performance as opposed to the usual P/Invoke invocation.

You can create an internal method like so:

```
[MethodImpl(MethodImplOptions.InternalCall)]  
public static extern int Sum(int a, int b);
```

For demonstration and further lessons on this can be found in Chapter 10, Embedding Mono and CoreCLR Runtime.

Chapter 7

Memory Management

We'll summarize few things about Manual Memory Management and Garbage Collections and then how we'll handle the cases for both in C#. Not a lot will be covered for various memory management at this time, since this section is emphasized on work around.

7.1 Garbage Collection

A garbage collection is essentially a runtime memory management that manage the memory for you so you wouldn't have to manage the memory all the time. Garbage Collection may offer less control over memory management for developers who requires it, but there are some advantages to Garbage Collection, it compact memories improving data locality, it eliminate circular references and so forth.

7.2 Manual Memory Management

Manual Memory can be either the smart pointer or your usual Malloc/Calloc and Free memory allocation and deallocation operations. The advantages for managing memory manually is that you can dispose memory whenever you desired and determine whether or not if memory should be zeroed out. Some caution is needed when memory leak is much more prominent with manual memory leak such as Circular Reference. Circular Reference is a case when two reference type references each others, the program may not be able to determine which reference to dispose, Garbage Collection avoid this issue by having a GC Root, if both objects cannot be seen by GC root, it's considered garbage memory and will be disposed of.

7.3 Memory Consideration when Binding Native Library

As a general rule in memory ownership, you often do not want to re-implement the native memory allocation and deallocation on C# and instead have it managed by native library itself, this is because there may be some implementation details that may not be obvious at first glance. Consider the following header file snippet:

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

typedef struct {
    int32_t X;
    int32_t Y;
} *MyStructRef;

void PrintMyStructRef(MyStructRef mystruct);
MyStructRef CreateMyStructRef();
void DestroyMyStructRef(MyStructRef mystruct);
```

At a glance, you could re-implement the headers for your own implementation of CreateMyStructRef and DestroyMyStructRef, but there is a problem with this. Suppose we have the following snippet implemented in C# of what we've observed in the header:

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

public struct MyStruct
{
    public int X;
    public int Y;
}

public unsafe interface IMyStructLibrary
{
    void PrintMyStructRef(MyStruct* mystruct);
}

public static unsafe class MyStructLibrary
{
    public static IMyStructLibrary API { get; }
    static MyStructLibrary()
    {
        API = NativeLibraryBuilder.Default.ActivateInterface<IMyStructLibrary>("ChapterSeven");
    }

    public static MyStruct* CreateMyStruct()
    {
        return (MyStruct*)Marshal.AllocHGlobal(Marshal.SizeOf<MyStruct>());
    }
}

public class Program
{
    static unsafe void Main()
    {
        var myStruct = MyStructLibrary.CreateMyStruct();
        myStruct->X = 5;
        myStruct->Y = 10;
        MyStructLibrary.API.PrintMyStructRef(myStruct);
    }
}
```

If you were to make an invocation to external function with your own version of implemented struct and memory allocation, you may end up creating few problems:

1. Memory Leak
2. Undefined Behavior
3. Cause Segmentation Fault

Suppose we have internal library implementation for the header as followed:

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

typedef struct {
    int32_t X;
    int32_t Y;
} *MyStructRef;

typedef struct {
    int32_t X;
    int32_t Y;
    int32_t Z;
    intptr_t Context;
} MyInternalStruct, *MyInternalStructRef;

void PrintMyStructRef(MyStructRef mystruct)
{
    MyInternalStructRef data = (MyInternalStructRef)mystruct;
    if (data->Context == NULL)
    {
        printf("Corrupted MyStructRef!\n");
        return;
    }
    data->Z++;
    printf("MyStruct: %i %i %i", data->X, data->Y, data->Z);
}

MyStructRef CreateMyStructRef()
{
    MyInternalStructRef createdStruct = calloc(1, sizeof(MyInternalStruct));
    createdStruct->Context = malloc(sizeof(intptr_t));
    return (MyStructRef) createdStruct;
}

void DestroyMyStructRef(MyStructRef mystruct)
{
    MyInternalStructRef data = (MyInternalStructRef) mystruct;
    free(data->Context);
    free(data);
}
```

There are a few problems that you can observe here, if you were to make an assumption for such implementation of the library based on what you observed in the header, you'll end up creating several unneeded problems. When attempting to call `PrintMyStructRef`, you would notice that it increment the `Z` field of `MyStruct`, but your implementation version of allocating `MyStruct` is allocating too small of a memory to hold `Z` field and you may consequentially be writing memory in other memory and create undefined behavior in your program, and that assuming the call didn't fail when `Context` is equals to null. Also when you try to dispose your implementation version of `MyStruct` with Library implemented `DestroyMyStructRef`, you would have another unintended side effect, it would free up `Context` memory if it's not null and that can potentially cause Segmentation Fault in your program.

The rule of thumb is to always leave it to the library to allocate/deallocate memory for you when possible otherwise you may expose yourself to the risk of native library changing the implementation down the road by adding or removing struct and changing the allocation/deallocation implementation for those struct. It's extremely common in C to have what is referred to a black box struct that is only a pointer, but it doesn't provide you any additional information in it's fields and other contexts.

Black Box struct can be as simple as this in a header file:

```
typedef struct *MyContextRef;
```

In other cases as shown above, it can be an incomplete type as well.

7.4 Note on Multi-Thread Platform Invocation

This book will touch very lightly on Multi-Thread for native programming since it's a vast problem that is worth talking in it's own book. When working with native library binding, there are several approaches on how a native library want to you to implement when utilizing their API and shared resources in a multi-threaded fashion. There are a significant number of implementation of threading solutions for native library shared resources which can include, but not limited to:

1. PThread Mutex
2. Lock Variable Synchronization Mechanism
3. Semaphore

Some of those libraries may be using specific tools to enable thread safety such as

1. Boost Library
2. LibUV

All in all, you must refer to the documentation for those native library to make the best determination of which tool to use to enable multi-threaded solution for your library.

7.5 All about HandleRef, Saferef and CriticalRef

Chapter 8

Introduction to Common Intermediate Language

8.1 CIL Fundamentals

The way CIL code get written are arranged by stack. So assume we have the following code in C#:

```
public static int Add(int a, int b) => a + b;
```

That code would be compiled as follow:

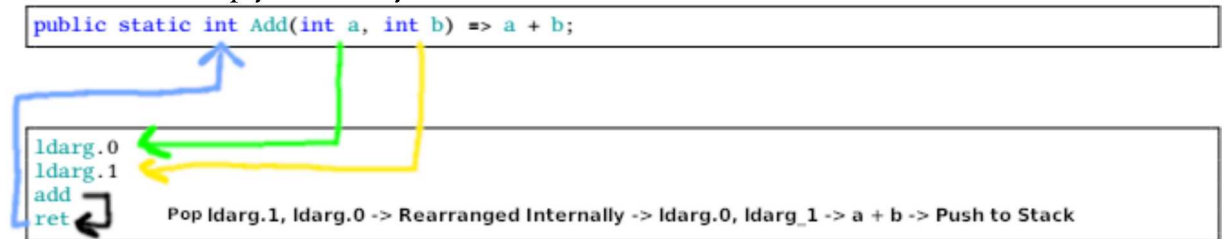
```
ldarg.0  
ldarg.1  
add  
ret
```

There are few things happening here: The shorthand CIL opcode, "ldarg.0", loads the first parameter, int a, to the stack. "ldarg.0" is a short hand opcode that you do not have to specify an argument for which parameter to load after opcode, so it save space and serves as a shortcut for runtime to infers what operation to build for that opcode.

Then "ldarg.1" do a similar operation to the above, but load the second parameter onto the stack. The "add" operation requires no arguments, but it pop off 2 items off the stack, so in "add" opcode point of view, when it pop the stack, it would see "ldarg.1" first and then "ldarg.0" second, this is something to keep in mind if you were to plan on writing a new runtime. The "add" opcode will arrange the items that were popped off the stack and do an addition operation from there (it uses the left-hand value type for addition) and then push the result of the addition to the stack.

Now that we have one remaining item in our stack, the "ret" opcode will pop the item off the stack and return that value for function return value.

In an effort to help you actually be able to visualize and understand how this works:



There is another thing to note about CIL, the local variables have to be declared and defined before body of CIL code can be written.

8.2 Special Note about the Stack

Note: This is a simple warning for those who use Stack in .Net Framework.

Stack is a Last In First Out, LIFO, so basically the last item you **push** to the stack is going to be the first item you get when you **pop** the stack. In .Net Framework, when you attempt to use the `IEnumerator` of `Stack<T>`, it will use the pop order in the way you read, so if you for an example have the following code:

```
Stack<string> originalStack;
Stack<string> tempStack = new Stack<string>(originalStack);
originalStack.Clear();

while (tempStack.Count != 0){
    originalStack.Push(tempStack.Pop());
}
```

There is a few things happening here, the stack would be reversed when the following code get called:

```
new Stack<string>(originalStack);
```

The stack already get reversed, because remember, the `IEnumerator` in `Stack<T>` is read by pop order, so when you pop and push to alternative stack, it rearrange the items like so:

ABC -> CBA -> ABC

To avoid this, you simply just have the following code instead of having to do any additional operations:

```
Stack<string> originalStack;
originalStack = new Stack<string>(originalStack);
```

8.3 Static vs Class Member Methods

In CIL, there is a special rule to follow for Static Method and Class Member Methods, static load the first parameter using "ldarg.0", but in class member method, it would load first parameter using "ldarg.1", not "ldarg.0". Because in class member method, "this" is a hidden parameter that get loaded when "ldarg.0" get called and this is a part of a calling convention in C#.

```
public class MyClass
{
    public int DoStuff(int a, int b)
    {
        return a + b;
        // This is a class method, the this parameter is supplied,
        // we need to use ldarg.1 instead of ldarg.0 for loading first parameter
        // ldarg.1
        // ldarg.2
        // add
        // ret
    }

    public static int DoStuff(int a, int b)
    {
        return a + b;

        // This is a static method, the this parameter is not supplied:
        // ldarg.0
        // ldarg.1
        // add
        // ret
    }
}
```

8.4 Introducing the Dynamic Method

For creating and building CIL code at runtime, there are three common approaches to this:

1. `DynamicMethod`
2. `MethodBuilder`
3. Assembly Loading via Reflection

Although there are technically infinite amount of approaches you can do it which can involve breaking the Runtime, using unsafe code, or creating your own compiler that works similarly to Roslyn compiler infrastructure.

8.4.1 Dynamic Method Approach

A dynamic method can be defined by using the constructor and specifying the name of method, the return type and parameter types.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class Program
{
    public delegate int Add_dt(int a, int b);

    static void Main(string[] args)
    {
        var method = new DynamicMethod("Add", typeof(int), new[] {typeof(int), typeof(int)});
        var il = method.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0);
        il.Emit(OpCodes.Ldarg_1);
        il.Emit(OpCodes.Add_Ovf);
        il.Emit(OpCodes.Ret);
        var add = (Add_dt) method.CreateDelegate(typeof(Add_dt));
        Console.WriteLine("2 + 2 = {0}", add(2, 2));
    }
}
```

The `DynamicMethod` is a shortcut that uses pre-defined dynamic assembly and nest the delegate as a Global Method which are methods that are defined globally in assembly without needing to reside in a type.

8.4.2 Method Builder Approach

A dynamic method can also be defined by first defining a dynamic assembly, module, and a type under it. It offers an additional amount of control how you emit your code by managing where code should reside in. You can also optionally choose to define method as a global method similarly to Dynamic Method above.

```
using System;
using System.Reflection;
using System.Reflection.Emit;

class Program
{
    public delegate int Add_dt(int a, int b);

    static void Main(string[] args)
    {
        var assBuilder = AssemblyBuilder.DefineDynamicAssembly(new AssemblyName("DynamicAss"),
            AssemblyBuilderAccess.RunAndCollect);
        var modBuilder = assBuilder.DefineDynamicModule("DynamicMod");
        var typeBuilder = modBuilder.DefineType("Math",
            TypeAttributes.Public | TypeAttributes.Class | TypeAttributes.Sealed | TypeAttributes
                .Abstract |
            TypeAttributes.AnsiClass);
        var method = typeBuilder.DefineMethod("Add", MethodAttributes.Public | MethodAttributes.
            Static, CallingConventions.Standard,
            typeof(int), new[] {typeof(int), typeof(int)});
        var il = method.GetILGenerator();
        il.Emit(OpCodes.Ldarg_0);
        il.Emit(OpCodes.Ldarg_1);
        il.Emit(OpCodes.Add_Ovf);
        il.Emit(OpCodes.Ret);
        var definedType = typeBuilder.CreateType();
        var add = (Add_dt)definedType.GetMethod("Add").CreateDelegate(typeof(Add_dt));
        Console.WriteLine("2 + 2 = {0}", add(2, 2));
    }
}
```

There are a number of objects that have to be defined and it chain all the way to the top starting with an Assembly, the module, the type to contains our method and finally the method itself. There are few things to explain on MethodAttributes and TypeAttributes here, similarly to C#, we have to define our methods and types with visibility modifiers and constraints. For a static class, it's usually defined by a combination of TypeAttributes.Class, TypeAttributes.Sealed, and TypeAttributes.Abstract which essentially informs the CLR that it's a type that can't be instantiated since it's sealed which cannot be inherited from and that it's an abstract which means it doesn't have a constructor to begin with. It's in all essence, a static class.

8.4.3 Assembly Loading via Reflection

In an exceptional cases, you may have a custom compiler that generates a custom assembly library, this is one form of an unorthodox dynamic code emitting at runtime. One such form can be done through Roslyn compiler infrastructure which allows you to compile C# snippet into a .Net assembly which can then be loaded dynamically.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

class Program
{
    public delegate int Add_dt(int a, int b);

    private static readonly IReadOnlyCollection<MetadataReference> _references = new[] {
        MetadataReference.CreateFromFile(typeof(object).GetTypeInfo().Assembly.Location)
    };
    static void Main(string[] args)
    {
        var syntaxTree = CSharpSyntaxTree.ParseText(@"
            public static class Math {
                public static int Add(int a, int b) => a + b;
            }
        ");
        var compilation = CSharpCompilation.Create("Example", new[] {syntaxTree},
            options: new CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary)).
            AddReferences(_references);
        var assemblyPath = Path.Combine(Path.GetDirectoryName(typeof(Program).Assembly.Location),
            "Example.dll");
        compilation.Emit(assemblyPath);

        var assembly = Assembly.LoadFile(assemblyPath);
        var mathType = assembly.GetTypes().First(I => I.Name == "Math");
        var method = mathType.GetMethod("Add");
        var add = (Add_dt)method.CreateDelegate(typeof(Add_dt));
        Console.WriteLine("2 + 2 = {0}", add(2, 2));
    }
}
```

The above constructs a .Net Assembly by leveraging Roslyn, we starts by defining our System library for .Net Assembly to reference on (so Runtime objects can be defined.) Then we have a simple snippet for C# code to compile by using CSharpSyntaxTree that simply deconstruct our parsed text into SyntaxTree which we can then pass to CSharpCompilation object to emit the compiled code as a new .Net Assembly. Through reflection, we can then load the newly created .Net Assembly and select our Math type and it's method to call Add function.

8.5 Branches in CIL

The CIL arrangement for branching works by specifying which "line" of CIL code to jump to or if talking about marked labels in `System.Reflection.Emit`, then it would jump to specified marked label. We can begin a demonstration of this by dynamically emitting a simple for loop code:

In a normal C# snippet for a For-Loop, it can be written like this:

```
for (int index = 0; index < 10; ++index)
{
    Console.WriteLine(index);
}
```

Now the same representation for above in CIL can be written as this:

```
using System;
using System.Reflection.Emit;

class Program
{
    public delegate void Example_dt();

    static void Main(string[] args)
    {
        var method = new DynamicMethod("Test", typeof(void), new Type[0]);
        method.InitLocals = true;
        var il = method.GetILGenerator();
        var index = il.DeclareLocal(typeof(int));
        var bodyLabel = il.DefineLabel();
        var conditionLabel = il.DefineLabel();
        il.Emit(OpCodes.Br, conditionLabel);
        il.MarkLabel(bodyLabel);
        il.EmitWriteLine(index);
        il.Emit(OpCodes.Ldloc_0);
        il.Emit(OpCodes.Ldc_I4_1);
        il.Emit(OpCodes.Add);
        il.Emit(OpCodes.Stloc_0);
        il.MarkLabel(conditionLabel);
        il.Emit(OpCodes.Ldloc_0);
        il.Emit(OpCodes.Ldc_I4, 10);
        il.Emit(OpCodes.Clt);
        il.Emit(OpCodes.Brtrue, bodyLabel);
        il.Emit(OpCodes.Ret);
        var example = (Example_dt) method.CreateDelegate(typeof(Example_dt));
        example();
    }
}
```

And to help visualize what's happening here, the snippet below is a C# equivalence to the CIL representation above:

```
var index = 0;
goto condition;
body:
Console.WriteLine(index);
++index;
condition:
if (index < 10) goto body;
```

There are a number of things happening in the snippet above and it can be broken down into these steps:

1. Define our local variable, an index
2. Define our marks, a body and a condition
3. Emit br to condition label since for-loop requires condition to be evaluated first
4. Mark the label body at this point
5. Emit invocation for Console.WriteLine with our index variable for argument
6. Increment the index variable by 1 at the end of body stub
7. Mark label condition at this point
8. Emit Ldloc_0 (our local variable is defined first) and Ldc_I4 with an argument of 10
9. Emit the CLT (Compare Less Than) comparison opcode to compare index to 10
10. Emit brtrue with body label as an argument, return to body if above condition return true

8.6 OpCodes references

You will often find a huge variety of OpCodes, but you might be asking how would you find out what each OpCode do.

The common method is to use documentation which is well maintained: <https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes>

For each OpCode page, it would list the description, the stack transitional behavior and the relevant overload for emit. To understand the stack transition behavior, hypothetically, you're reading on Add opcode, it would list the following Stack Transitional Behavior:

1. value1 is pushed onto the stack.
2. value2 is pushed onto the stack.
3. value2 and value1 are popped from the stack; value1 is added to value2.
4. The result is pushed onto the stack.

It should be fairly clear what's happening, but if you attempt to read the same for subtraction opcode, then it would be important to know which value is a left hand value and which is the right hand value. The third item will explain that "value1 is added to value2" and that clearly define which value is left hand and vice versa.

8.6.1 Note about Strict Emit Utility

Strict Emit is also a free library provided by Firwood Organization that offers a large set of extension methods for ILGenerator to further simplify and document your code.

8.7 Try Catch Finally Stubs

The try/catch/finally in CIL are particularly similar to C# counterpart, but there are some key differences:

1. There is no "Try" clause in CIL emitting IE in System.Reflection.Emit, it's simply Begin/End Exception Block that enclose entire Try/Catch/Finally clauses. Although in actual CIL Output, the .try clauses will be emitted.
2. Nested Exception Handling will have Catch/Fault blocks filter exception based on the current nested exception block.

In the following example, the code will attempt to create a scenario that the Overflow Exception by attempting to increment an integer that have been signed to maximum value possible.

```
using System;
using System.Reflection.Emit;

class Program
{
    public delegate void Test_dt();
    static unsafe void Main(string[] args)
    {
        var example = new DynamicMethod("Test", typeof(void), new Type[0]);
        var il = example.GetILGenerator();
        var val = il.DeclareLocal(typeof(int));
        il.DeclareLocal(typeof(OverflowException));

        il.Emit(OpCodes.Ldc_I4, int.MaxValue);
        il.Emit(OpCodes.Stloc_0);

        il.BeginExceptionBlock();
        il.Emit(OpCodes.Ldloc_0);
        il.Emit(OpCodes.Ldc_I4_1);
        il.Emit(OpCodes.Add_Ovf);
        il.Emit(OpCodes.Stloc_0);
        il.BeginCatchBlock(typeof(OverflowException));
        il.Emit(OpCodes.Stloc_1);
        il.Emit(OpCodes.Ldloc_1);
        il.Emit(OpCodes.Call, typeof(OverflowException).GetMethod("ToString"));
        il.Emit(OpCodes.Call, typeof(Console).GetMethod("WriteLine", new []{typeof(string)}));
        il.BeginFinallyBlock();
        il.EmitWriteLine(val);
        il.EndExceptionBlock();
        il.Emit(OpCodes.Ret);

        var test = (Test_dt)example.CreateDelegate(typeof(Test_dt));
        test();
    }
}
```

The way BeginCatchBlock works is that when the specified exception were caught, it would have exception available on the stack that you can pop or store into local variable, in this example, it get stored into the second local variable and then it get loaded to print to standard output.

BeginFinallyBlock except a stub of code that will happen in either cases when exception is thrown and when code run successfully without exception.

8.7.1 Fault Exception Handling

The fault clause is specific to CIL and isn't a valid keyword for C#, and it is similar to finally clause, except it's whenever exception get thrown, it doesn't get executed when program exit try clause normally. This means, similarly to finally, it rethrows the exception at the end of it. It is the same in function to this C#:

```
try
{
    // CODE
}
catch
{
    // equivalent of fault block code
    throw;
}
```

To demonstrate the fault handler:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

public class Program
{
    public delegate int DoAdd_dt(int a);
    public static void Main()
    {
        var myTypeBuilder = AssemblyBuilder.DefineDynamicAssembly(new AssemblyName("DynamicAss"),
            AssemblyBuilderAccess.Run).DefineDynamicModule("AdderExceptionMod").DefineType("Adder");
        var myMethodBuilder = myTypeBuilder.DefineMethod("DoAdd", MethodAttributes.Public |
            MethodAttributes.Static, typeof(int)
            ), new Type[] { typeof(int) });

        var il = myMethodBuilder.GetILGenerator();
        var end = il.DefineLabel();
        il.DeclareLocal(typeof(int));
        il.BeginExceptionBlock();
        il.Emit(OpCodes.Ldarg_0);
        il.Emit(OpCodes.Ldc_I4, int.MaxValue);
        il.Emit(OpCodes.Add_Ovf);
        il.Emit(OpCodes.Stloc_0);
        il.Emit(OpCodes.Leave_S, end);
        il.BeginFaultBlock();
        il.EmitWriteLine("Fault block called.");
        il.EndExceptionBlock();
        il.MarkLabel(end);
        il.Emit(OpCodes.Ldloc_0);
        il.Emit(OpCodes.Ret);

        var add = (DoAdd_dt)myTypeBuilder.CreateType().GetMethod("DoAdd").CreateDelegate(typeof(
            DoAdd_dt));
        Console.WriteLine(add(1));
    }
}
```

Note: This code may not works on CoreCLR, but it does reflect the expected behavior on Mono Runtime

The CIL code above can be expressed in a similar C# Snippet, but instead of catch, it would be using fault clause:

```
int DoAdd(int a)
{
    int variable = 0;
    try
    {
        variable = a + int.MaxValue;
        goto End;
    }
    fault
    {
        Console.WriteLine("Fault block called.");
        throw; // rethrow the exception
    }
End:
    return variable;
}
```

To summarize, the Fault Handler is essentially a catch all clause that only run whenever an exception get thrown.

8.7.2 Filter Handling

Chapter 9

Advanced CIL Programming

9.1 Constructing a Type at Runtime

One of the more advanced use of Runtime code emitting involves the use of TypeBuilder and dynamically building new code while your program is actively running. There are multiple ways of how this can save you time writing codes and improve your program performance. So let's starts with the basics, creating simple types and adding properties and methods to those types.

9.1.1 Constructing a Struct

We'll need to construct a Dynamic Assembly and Dynamic Module for this code and this can be accomplished by importing System.Reflection and System.Reflection.Emit namespaces and writing the following snippet:

```
var assemblyBuilder = AssemblyBuilder.DefineDynamicAssembly(new AssemblyName("DynamicAssembly"),
    AssemblyBuilderAccess.RunAndCollect);

var modBuilder = assemblyBuilder.DefineDynamicModule("DynamicModule");
```

Within ModuleBuilder you can now define your types, enumeration, methods, and so forth. You can even define a global method as well.

Now we'll define a struct, in this case, a public struct would have the following type attributes: AutoLayout, AnsiClass, Class, SequentialLayout, Sealed, BeforeFieldInit

Let's go over what each Type Attribute is:

1. AutoLayout - Have Class Fields Laid out by Common Language Runtime or CLR
2. AnsiClass - LPTSTR is interpreted as ANSI Encoding (LPTSTR is a TChar String which can be either wide char or char depending on whether Unicode Encoding is being used)
3. Class - To construct the type as a Class Semantic (To accepts This Pointer for non-static methods and et cetera)
4. SequentialLayout - Have fields arranged in a sequential order in memory
5. Sealed - Disallow Inheritance in Struct
6. BeforeFieldInit - Avoid initializing the type when calling it's static methods.

Now we can construct our new Struct Type as followed:

```
var structBuilder = modBuilder.DefineType("MyStruct",
    TypeAttributes.AutoLayout | TypeAttributes.AnsiClass | TypeAttributes.Class |
    TypeAttributes.SequentialLayout | TypeAttributes.Sealed |
    TypeAttributes.BeforeFieldInit);
```

Now we'll attempt to implement the same as this struct in the following snippet in C#:

```
public struct MyStruct
{
    public int A;

    public void DoStuff()
    {
        A = 5;
    }
}
```

So far we've created a public struct MyStruct type, now to implement the rest including the "A" field and the "DoStuff" non-static method, we'll need to first define our field, "A" as an integer.

```
var myStructFieldA = structBuilder.DefineField("A", typeof(int), FieldAttributes.Public);
```

Note the declaration of myStructFieldA, it will come in handy for CIL emitting. Now we're ready to create a non-static method for our struct. Our method would require 2 signatures: Public and hidebysign. Hidebysignature allows you to have method overloading since it hide by signature, not by method name.

```
var methodBuilder =
    structBuilder.DefineMethod("DoStuff", MethodAttributes.Public | MethodAttributes.HideBySig);
```

At this point, you can define method parameters or even generic parameters, we will go over generic parameter after this method construction. Now we'll need to implement our method with methodBuilder variable. We'll create a new variable, "il" for holding a reference to our GetILGenerator() which we can then use to generate our IL code in our method.

```
var il = methodBuilder.GetILGenerator();
```

Note: This is a non-static method, it will have a This pointer supplied.

If you observe on earlier code, you'll notice that we can make use of prior variable of defined variables and structs that we can then use for emitting CIL with as arguments. The implemented CIL for DoStuff method would be as followed:

```
il.Emit(OpCodes.Ldarg_0);
il.Emit(OpCodes.Ldc_I4_5);
il.Emit(OpCodes.Stfld, myStructFieldA);
il.Emit(OpCodes.Ret);
```

Now we've successfully constructed our new Struct type! We'll need to finalize the Type by invoking CreateType() for structBuilder.

```
var newStructType = structBuilder.CreateType();
```

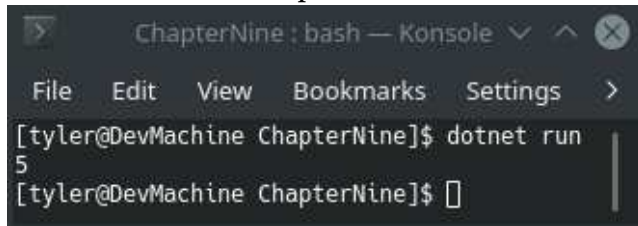
Note that when we retain the variable holding the newly created type of Struct, we can "activate" the struct through the use of Activator object provided in System namespace which allows us to supply a type as a parameter to construct a type.

```
var newStruct = Activator.CreateInstance(newStructType);
```


Now we have our newly constructed type object that we can then call method or retrieve value from, we must first query for method field from our type, so we can then invoke or get value from those methods and fields.

```
var method = newStruct.GetType().GetMethod("DoStuff");
method.Invoke(newStruct, null);
var fieldA = newStruct.GetType().GetField("A");
Console.WriteLine(fieldA.GetValue(newStruct));
```

And we'll have our output as followed:



```
ChapterNine: bash — Konsole
File Edit View Bookmarks Settings >
[tyler@DevMachine ChapterNine]$ dotnet run
5
[tyler@DevMachine ChapterNine]$
```

The rest of our snippet is as followed:

```
using System;
using System.Reflection;
using System.Reflection.Emit;

namespace ChapterNine
{
    class Program
    {
        static void Main(string[] args)
        {
            var assemblyBuilder = AssemblyBuilder.DefineDynamicAssembly(new AssemblyName("DynamicAssembly"), AssemblyBuilderAccess.RunAndCollect);

            var modBuilder = assemblyBuilder.DefineDynamicModule("DynamicModule");

            var structBuilder = modBuilder.DefineType("MyStruct",
                TypeAttributes.AutoLayout | TypeAttributes.AnsiClass | TypeAttributes.Class |
                TypeAttributes.SequentialLayout | TypeAttributes.Sealed |
                TypeAttributes.BeforeFieldInit);

            var myStructFieldA = structBuilder.DefineField("A", typeof(int), FieldAttributes.Public);

            var methodBuilder =
                structBuilder.DefineMethod("DoStuff", MethodAttributes.Public | MethodAttributes.HideBySig);

            var il = methodBuilder.GetILGenerator();

            il.Emit(OpCodes.Ldarg_0);
            il.Emit(OpCodes.Ldc_I4_5);
            il.Emit(OpCodes.Stfld, myStructFieldA);
            il.Emit(OpCodes.Ret);

            var newStructType = structBuilder.CreateType();
            var newStruct = Activator.CreateInstance(newStructType);
            var method = newStruct.GetType().GetMethod("DoStuff");
            method.Invoke(newStruct, null);
            var fieldA = newStruct.GetType().GetField("A");
            Console.WriteLine(fieldA.GetValue(newStruct));
        }
    }
}
```

9.1.2 Properties

Chapter 10

Embedding Mono and CoreCLR Runtime

10.1 Embedding CoreCLR in C Language

10.1.1 Introduction to Embedding CoreCLR

In CoreCLR, it have a feature that enables you to embed CoreCLR into an existing applications, by using this wisely you can harness the power and features of multiple programming languages and ecosystems. Before we can jump into embedding CoreCLR into a native projects, we need to observe a few steps:

1. CoreCLR requires a paths to C# Application/Library and Trusted Platform Libraries paths that are composed of paths to CoreCLR Base Class Libraries and your Application in a list with ':' delimiter.
2. Once you've obtained both information, you would then need to create 2 collections that resemble Key-Value Pairs, PropertyKeys and PropertyValues containing the information for TPAList and AppPath.
3. Once CoreCLR is initialized with provided information, you would have Domain ID and CoreCLR JIT Context Pointer to run C# executable or to create delegate from C# function.

There are going to be a few assumptions to be made for this demonstration, it's assuming that the Dotnet Core SDK is installed under **/opt/dotnet** directory. The native libraries and all of the trusted platform assembly libraries are stored in **dotnet/shared/Microsoft.NETCore.App/{Version}/** directory. For this demonstration, we'll be observing the TPAList implementation that were re-implemented from C++ implementation in CoreCLR example:

<https://github.com/dotnet/coreclr/blob/master/src/coreclr/hosts/unixcoreruncommon/coreruncommon.cpp>

For TPAList implementation, it'll do the followings to obtain a list of paths:

1. Determine the size of buffer allocation required to compose a TPAList by iterating through directory for matched libraries and add it to TPAList.
2. Once Buffer size is determined, the TPAList get allocated of specified size.
3. Reiterate through directory and append into TPAList with ':' delimiter
4. Append C# DLL Library Path into TPAList
5. Return TPAList

We would have the following snippet:

```
char* AddFilesFromDirectoryToTpaList(char* path, char* app)
{
    DIR* dir = opendir(path);
    int32_t pathLen = strlen(path);
    if (dir == NULL)
    {
        return NULL;
    }

    uint64_t requiredSize = strlen(app) + 2; // 1 for : and 1 for null terminator
    while (1)
    {
        struct dirent* entry = readdir(dir);
        if (entry == NULL)
        {
            break;
        }
        char subBuff[4];
        int32_t len = strlen(entry->d_name);
        if (len < 5)
            continue;
        memcpy(subBuff, entry->d_name + (len - 4), 4);
        if (strcmp(subBuff, ".dll") != 0 ||
            strcmp(subBuff, ".exe") != 0)
        {
            requiredSize += pathLen + strlen(entry->d_name) + 1;
            continue;
        }
    }
    char* tpaList = (char*)calloc(sizeof(char), requiredSize);
    char* insert = tpaList;
    seekdir(dir, 0);

    while (1)
    {
        struct dirent* entry = readdir(dir);
        if (entry == NULL)
        {
            break;
        }
        char subBuff[4];
        int32_t len = strlen(entry->d_name);
        if (len < 5)
            continue;
        memcpy(subBuff, entry->d_name + (len - 4), 4);
        if (strcmp(subBuff, ".dll") == 0 ||
            strcmp(subBuff, ".exe") == 0)
        {
            strcpy(insert, path);
            insert += pathLen;
            strcpy(insert, entry->d_name);
            insert += len + 1;
            *(insert - 1) = ':';
            continue;
        }
    }
    strcpy(insert, app);
    *(insert + strlen(app)) = ':';
    closedir(dir);
    return tpaList;
}
```

With header file that exposes the interface for our CoreCLR library with preprocessor removed from <https://github.com/dotnet/coreclr/blob/master/src/coreclr/hosts/inc/coreclrhost.h>:

```
#ifndef EMBEDDEDCORECLR_CORECLRHOST_H
#define EMBEDDEDCORECLR_CORECLRHOST_H

extern int coreclr_initialize(const char* exePath,
                           const char* appDomainFriendlyName,
                           int propertyCount, const char** propertyKeys,
                           const char** propertyValues, void* hostHandle,
                           unsigned int* domainId);

extern int coreclr_shutdown(void* hostHandle, unsigned int domainId);

extern int coreclr_shutdown_2(void* hostHandle, unsigned int domainId,
                             int* latchedExitCode);

extern int coreclr_create_delegate(void* hostHandle, unsigned int domainId,
                                  const char* entryPointAssemblyName,
                                  const char* entryPointTypeName,
                                  const char* entryPointMethodName, void** delegate);

extern int coreclr_execute_assembly(void* hostHandle, unsigned int domainId,
                                    int argc, const char** argv,
                                    const char* managedAssemblyPath,
                                    unsigned int* exitCode);

#endif //EMBEDDEDCORECLR_CORECLRHOST_H
```

With almost everything in place, before we can finally initialize our CoreCLR in our native application, we need to first create our C# Library for this demonstration, we can use the following snippet for demonstration purpose:

```
namespace Chapter10
{
    public static class Program
    {
        public static int Test() => 5;
    }
}
```

We need to then build and then copy the compiled library into our native application and ensure it is copied upon build so C Executable Program can find the C# Library in the same directory that it resides in.

Finally we can initialize our CoreCLR in a native application! For completeness sake, we will be covering a few extra steps involved to try and keep build somewhat clean. I want to emphasize on a note that you **CAN** copy the Trusted Platform Assembly libraries in an adjacent directory of your C Executable Program and have C Executable linked to CoreCLR native library in that library so you can package it and share it on other machine.

There are few things happening in this snippets which can be broken down in these steps:

1. Obtain the Directory Path to current C Executable Path
2. Compose a path to C# Library Path from Directory Path above
3. Compose a TPAList by running AddFilesFromDirectoryToTpaList function
4. Compose a Property Key/Value Pairs containing the parameters required for CoreCLR initialization
5. Initialize CoreCLR
6. Construct a Delegate to C# Test method and obtain function pointer that can be called from C
7. Execute said C# method in C
8. Shutdown everything and free memories

```

int main() {
    // Let's get directory name of the path to current C Executable
    char* buff = (char*)calloc(sizeof(char), 66535);
    readlink("/proc/self/exe", buff, 66535);
    char* dir = dirname(buff);
    free(buff);

    // Let's compose a full path to our C# Dll Library Path
    char* appName = "Chapter10.dll";
    // Directory + '/' + appName + \0 (null terminator)
    char* appPath = (char*)malloc(strlen(appName) + strlen(dir) + 2);
    char* insert = appPath;
    strcpy(insert, dir);
    insert += strlen(dir);
    *insert = '/';
    ++insert;
    strcpy(insert, appName);

    // Compose a full Trusted Platform Assemblies List
    char* tpaList = AddFilesFromDirectoryToTpaList("/opt/dotnet/shared/Microsoft.NETCore.
App/2.2.6/", appPath);

    const char *propertyKeys[] = {
        "TRUSTED_PLATFORM_ASSEMBLIES",
        "APP_PATHS"
    };
    const char *propertyValues[] = {
        // TRUSTED_PLATFORM_ASSEMBLIES
        tpaList,
        // APP_PATHS
        appPath
    };
    void* clr;
    uint32_t domainID;
    int result = coreclr_initialize(appPath, "Chapter10", 2, propertyKeys, propertyValues
        , &clr, &domainID);
    if (result)
    {
        printf("Failed to initialize!\n");
        return -1;
    }

    int32_t (*Test)();
    coreclr_create_delegate(clr, domainID, "Chapter10", "Chapter10.Program", "Test", (
        void**)&Test);

    printf("Running C# Snippet: %i\n", Test());

    coreclr_shutdown(clr, domainID);
    free(tpaList);
    free(dir);
    return 0;
}

```

And we'll have the following output for our program:

```

Chapter10Build: bash — Konsole
File Edit View Bookmarks Settings Help
[tyler@DevMachine Chapter10Build]$ ./Chapter10
Running C# Snippet: 5
[tyler@DevMachine Chapter10Build]$ 

```


You would need to ensure you link in libcoreclr.so library when compiling the above snippets, the following would work assuming your path and dotnet version matches:

```
clang -oChapter10 main.c
      -rpath /opt/dotnet/shared/Microsoft.NETCore.App/2.2.6/
      -L/opt/dotnet/shared/Microsoft.NETCore.App/2.2.6/
      -lcoreclr
```

Note: The '-rpath' informs the compiler to find the library at runtime, not at link time.

The following is the full snippet for the demonstration of this project:

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <libgen.h>
#include "coreclrhost.h"

char* AddFilesFromDirectoryToTpaList(char* path, char* app)
{
    DIR* dir = opendir(path);
    int32_t pathLen = strlen(path);
    if (dir == NULL)
    {
        return NULL;
    }

    uint64_t requiredSize = strlen(app) + 2; // 1 for : and 1 for null terminator
    while (1)
    {
        struct dirent* entry = readdir(dir);
        if (entry == NULL)
        {
            break;
        }
        char subBuff[4];
        int32_t len = strlen(entry->d_name);
        if (len < 5)
            continue;
        memcpy(subBuff, entry->d_name + (len - 4), 4);
        if (strcmp(subBuff, ".dll") != 0 ||
            strcmp(subBuff, ".exe") != 0)
        {
            requiredSize += pathLen + strlen(entry->d_name) + 1;
            continue;
        }
    }
    char* tpaList = (char*)calloc(sizeof(char), requiredSize);
    char* insert = tpaList;
    seekdir(dir, 0);

    while (1)
    {
        struct dirent* entry = readdir(dir);
        if (entry == NULL)
        {
            break;
        }
        char subBuff[4];
        int32_t len = strlen(entry->d_name);
        if (len < 5)
            continue;
        memcpy(subBuff, entry->d_name + (len - 4), 4);
        if (strcmp(subBuff, ".dll") == 0 ||
            strcmp(subBuff, ".exe") == 0)
        {
            strcpy(insert, path);
            insert += pathLen;
            strcpy(insert, entry->d_name);
            insert += len + 1;
            *(insert - 1) = ':';
            continue;
        }
    }
    strcpy(insert, app);
    *(insert + strlen(app)) = ':';
}
```

```

    closedir(dir);
    return tpaList;
}

int main() {
    // Let's get directory name of the path to current C Executable
    char* buff = (char*)calloc(sizeof(char), 66535);
    readlink("/proc/self/exe", buff, 66535);
    char* dir = dirname(buff);
    free(buff);

    // Let's compose a full path to our C# Dll Library Path
    char* appName = "Chapter10.dll";
    // Directory + '/' + appName + \0 (null terminator)
    char* appPath = (char*)malloc(strlen(appName) + strlen(dir) + 2);
    char* insert = appPath;
    strcpy(insert, dir);
    insert += strlen(dir);
    *insert = '/';
    ++insert;
    strcpy(insert, appName);

    // Compose a full Trusted Platform Assemblies List
    char* tpaList = AddFilesFromDirectoryToTpaList("/opt/dotnet/shared/Microsoft.NETCore.
        App/2.2.6/", appPath);

    const char *propertyKeys[] = {
        "TRUSTED_PLATFORM_ASSEMBLIES",
        "APP_PATHS"
    };
    const char *propertyValues[] = {
        // TRUSTED_PLATFORM_ASSEMBLIES
        tpaList,
        // APP_PATHS
        appPath
    };
    void* clr;
    uint32_t domainID;
    int result = coreclr_initialize(appPath, "Chapter10", 2, propertyKeys, propertyValues
        , &clr, &domainID);
    if (result)
    {
        printf("Failed to initialize!\n");
        return -1;
    }

    int32_t (*Test)();
    coreclr_create_delegate(clr, domainID, "Chapter10", "Chapter10.Program", "Test", (
        void*)&Test);

    printf("Running C# Snippet: %i\n", Test());

    coreclr_shutdown(clr, domainID);
    free(tpaList);
    free(dir);
    return 0;
}

```


Chapter 11

ADL In-depth

Chapter 12

System.Linq.Expressions Programming

Chapter 13

Introduction to Roslyn

Chapter 14

Advanced Roslyn Development

Chapter 15

Extending Roslyn

Chapter 16

Introduction to LLVM

Chapter 17

Advanced LLVM Programming

Chapter 18

Advanced P/Invoke

Chapter 19

Compiler Theory

Chapter 20

Writing Your Own Compiler

Chapter 21

Developing Your Own Runtime

Chapter 22

Conclusion

