Advanced C# for Low-Level Programming

Tyler Crandall

November 2018

Book Contributors

- Jamesbascle - For grammar corrections.
- Topping - For grammar corrections.
- SirJosh#3917 - For correcting Chapter 4 Snippet, renamed chapFourLib to lib variable.
- Monica#7056 - For suggestions and improvements.
- Tanner Gooding - For writing some snippets for Chapter 5 and suggestions on ideas for this book and investigated on Packing/Alignment of Struct.
- Rayan Desfossez (Absolute'Magic on Github) - For fixing snippets and providing console images and grammar corrections.
- John Kelly (@johnkellyoxford on Github) - For correction on CIL Try/Catch/Finally stubs chapters and for suggesting that chapters should be made for raw CIL rather than dynamically emitted CIL code.

Contents

Chapter 1

Introduction

1.1 What you will need to get started

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

You can install Dotnet Core SDK from this URL which contains an instruction to install Dotnet Core on your respective distribution:

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

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

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

1.2 Minimum Knowledge

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

Chapter 2

Introduction to P/Invoke

2.1 Getting Started

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

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

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

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

1. 'int' datatype in C can be considered 2 bytes long or 4 bytes long or however long it may be depending on the architecture and compiler that the library is compiled on. In C Standard, int must be capable of containing **at least** the [−32,767, +32,767] range; thus, it is at least 16 bits in size.

2. Due to 1, you can reasonably safeguard against data loss by substituting C# Int32(int) which contains 4 bytes or you may choose follow the standard strictly by supplying C# Int16(short) which contains 2 bytes, even though it can suffer data loss. The best approach is to avoid using "at least" integers in C and instead use fixed size integers provided by the compiler in "stdint.h" header if you have Foreign Function Interface kept in mind.

3. Sometimes you have to keep Endianess in mind although it is less of a concern in x86_64 architecture since little endianess is the default.

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

```
#include <stdint.h>

int32_t Sum(int32_t a, int32_t b)
{
 return a + b;
}
```

2.2 Compiling the Library

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

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

Here we examine and explain the compiler arguments:

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

2.3 Configuring C# Project

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

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

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

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

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

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

Your CSProj should look like the following:

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>netcoreapp2.1</TargetFramework>
 </PropertyGroup>
 <Target Name="CompileCProject" AfterTargets="AfterBuild">
  <exec Command="clang -std=c99 -shared -fPIC -olibChapTwo.so ChapTwo.c" />
  <Copy SourceFiles="libChapTwo.so" DestinationFolder="$(OutDir)" />
 </Target>
</Project>
```

2.4 Wrapping C Code in C#

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

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

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

Add the following lines under Program class:

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

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

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

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

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

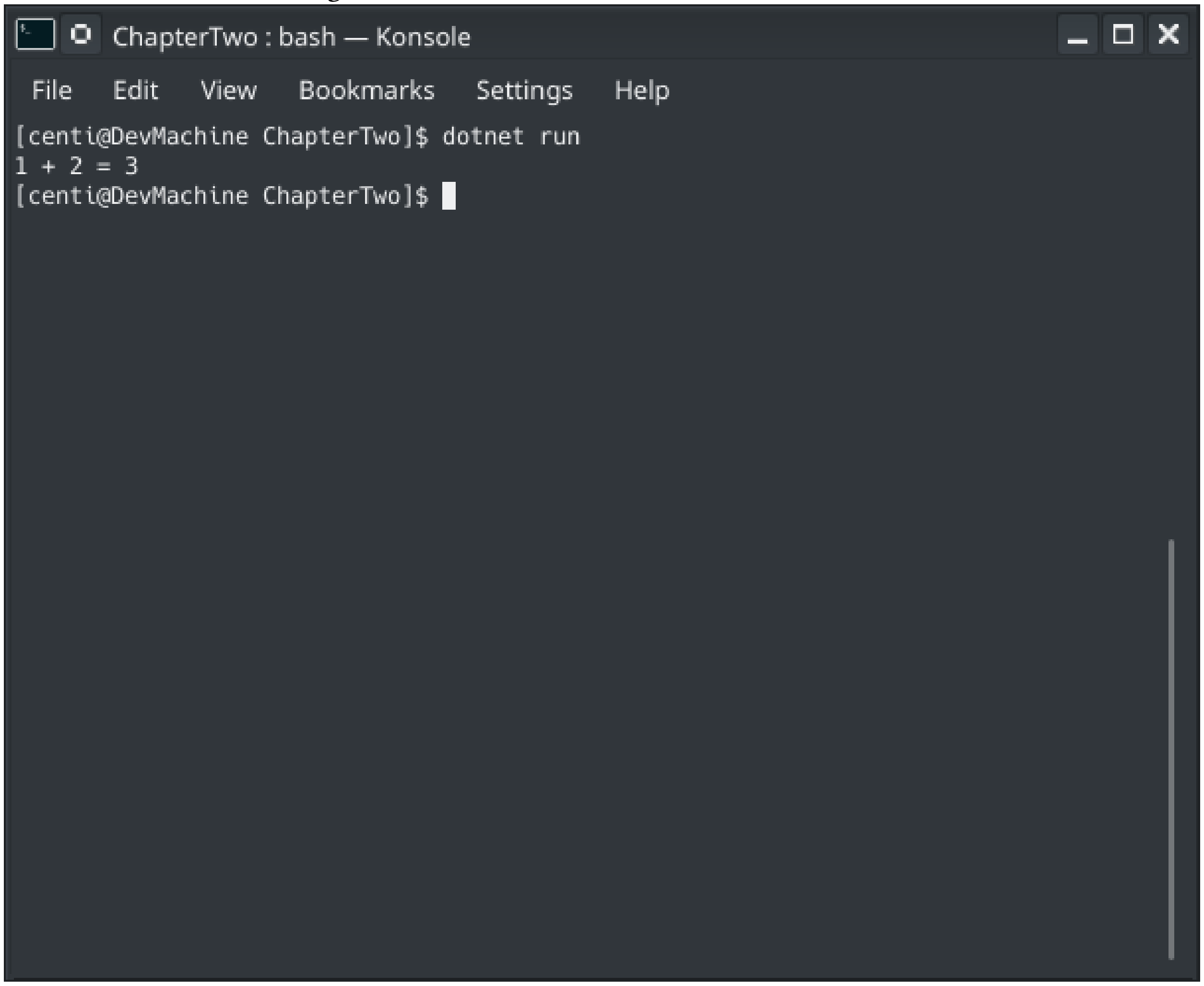
And your source code should look as follows:

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

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

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

And we have the following:



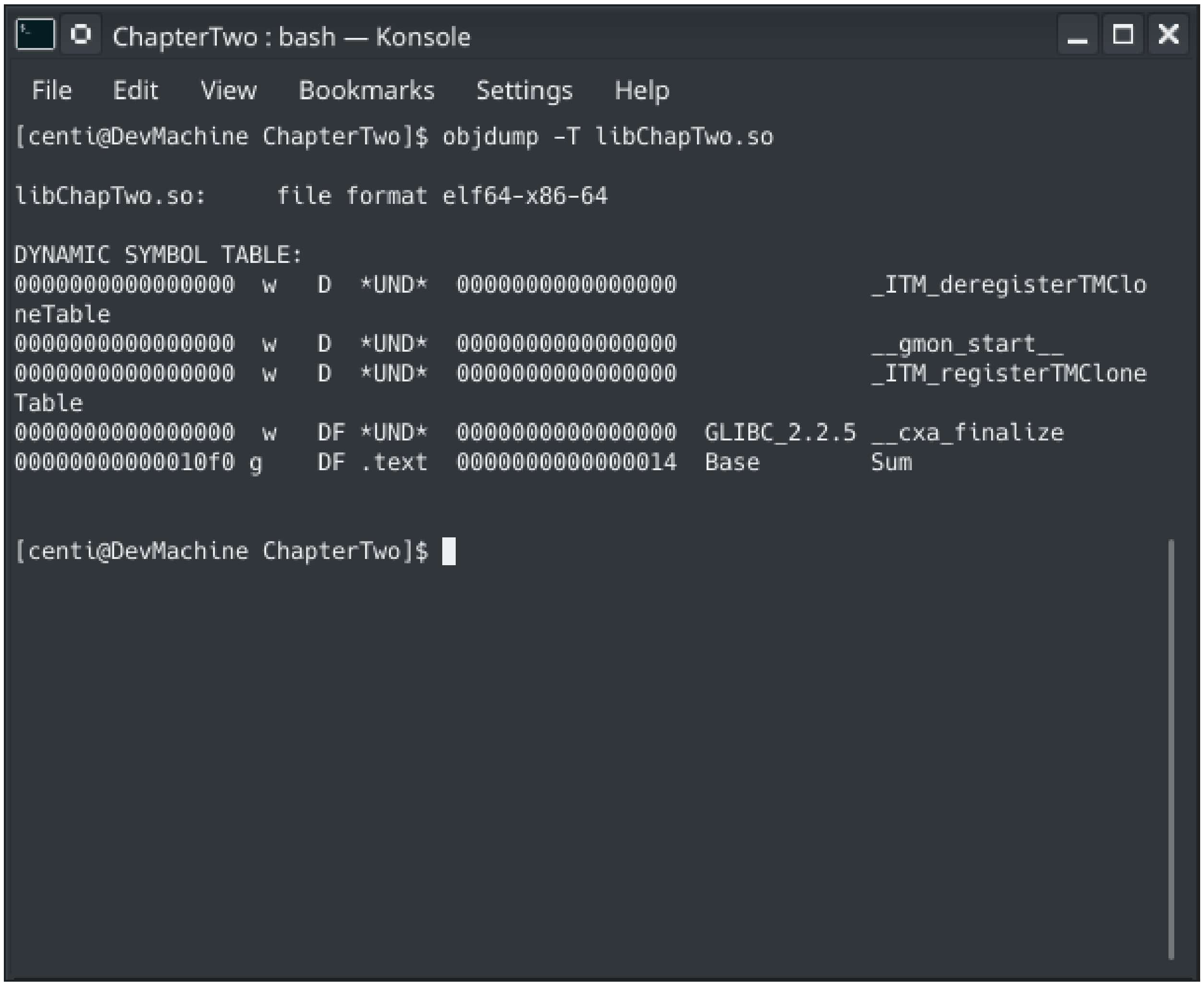
```
[centi@DevMachine ChapterTwo]$ dotnet run
1 + 2 = 3
[centi@DevMachine ChapterTwo]$
```

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

```
[centi@DevMachine ChapterTwo]$ objdump -T libChapTwo.so

libChapTwo.so:     file format elf64-x86-64

DYNAMIC SYMBOL TABLE:
0000000000000000  w   D  *UND*  0000000000000000              _ITM_deregisterTMClo
neTable
0000000000000000  w   D  *UND*  0000000000000000              __gmon_start__
0000000000000000  w   D  *UND*  0000000000000000              _ITM_registerTMClone
Table
0000000000000000  w   DF *UND*  0000000000000000  GLIBC_2.2.5 __cxa_finalize
00000000000010f0 g    DF .text  0000000000000014  Base        Sum


[centi@DevMachine ChapterTwo]$
```

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

Chapter 3

Introduction to Pointer

3.1 Overview on Pointer

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

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

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

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

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

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

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

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

// You however cannot do this:

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

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

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

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

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

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

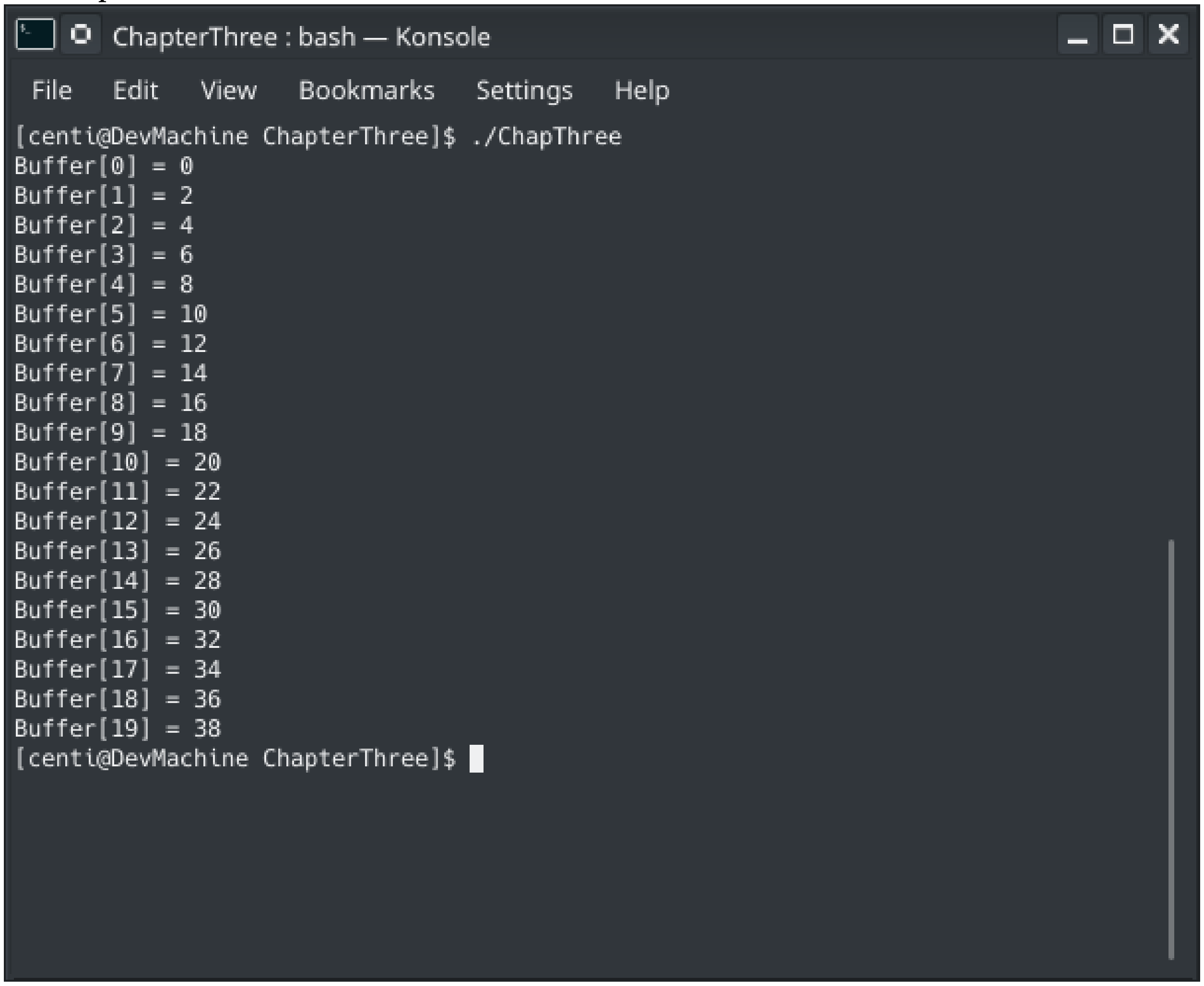
We will need three headers to provide the functionalties and types we need for this chapter.

```
#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:



```
[centi@DevMachine ChapterThree]$ ./ChapThree
Buffer[0] = 0
Buffer[1] = 2
Buffer[2] = 4
Buffer[3] = 6
Buffer[4] = 8
Buffer[5] = 10
Buffer[6] = 12
Buffer[7] = 14
Buffer[8] = 16
Buffer[9] = 18
Buffer[10] = 20
Buffer[11] = 22
Buffer[12] = 24
Buffer[13] = 26
Buffer[14] = 28
Buffer[15] = 30
Buffer[16] = 32
Buffer[17] = 34
Buffer[18] = 36
Buffer[19] = 38
[centi@DevMachine ChapterThree]$
```

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

1. When you use malloc to allocate buffer, malloc keeps a record of how big the block of memory is so that it can be freed in later time, however you cannot and should not access that size information within malloc, keeping track of buffer size is your responsibility.

2. When using malloc, you allocate the amount of data you would need by using sizeof keyword to determine how many bytes capacity you need in a buffer to cover that information and you can multiply that element size by the number of elements you want allocated for your program. In the snippet above, size of int32_t type would resolves to 4 and then multiplied by 20, so we would have a buffer that have the capacity to hold 20 int32_t elements.

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

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

3.2 The Function Pointers

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

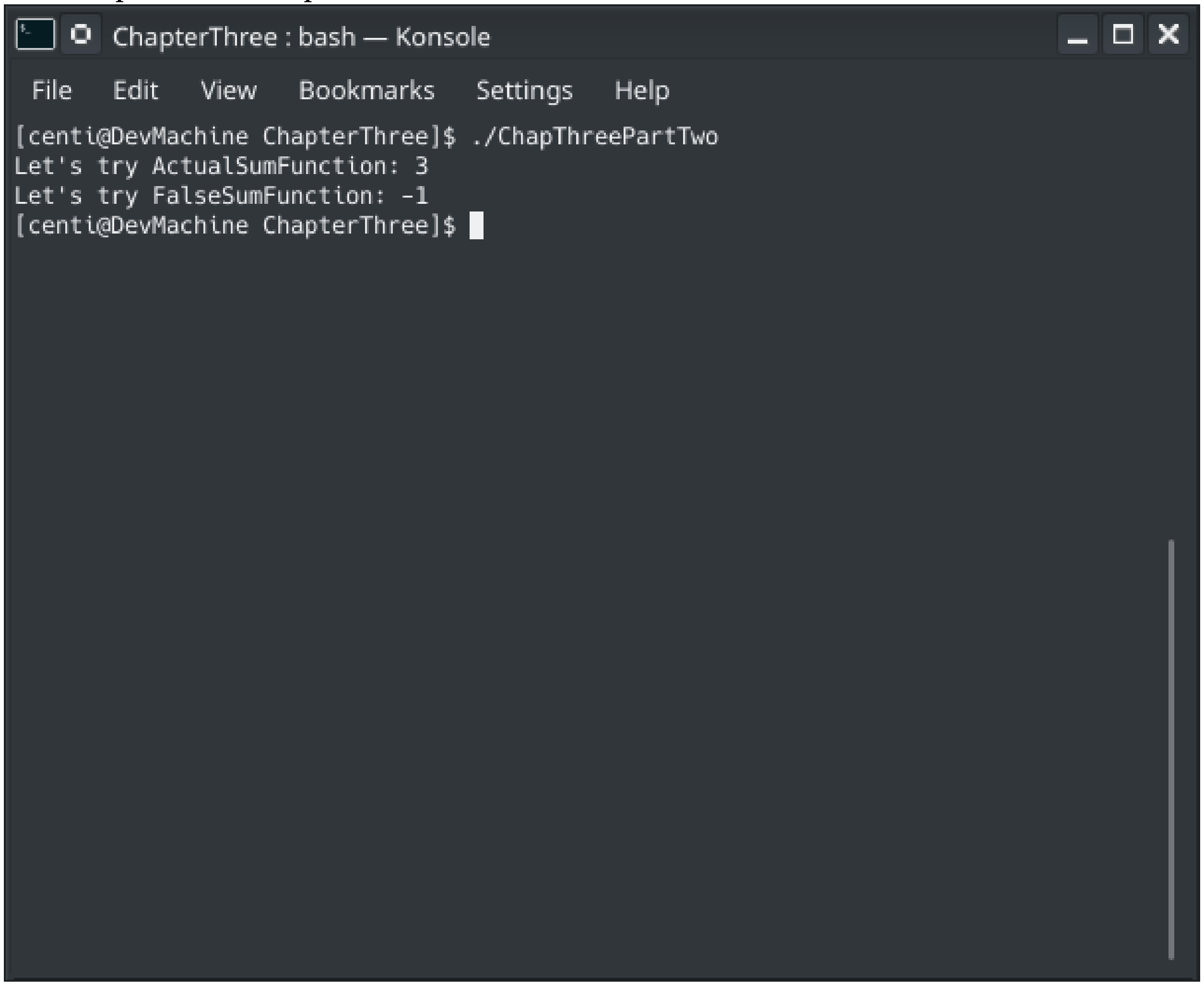
```
int32_t (*Sum)(int32_t, int32_t);
```

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

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

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

int32_t (*Sum)(int32_t, int32_t);

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

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

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

      Sum = ActualSumFunction;

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

      Sum = FalseSumFunction;

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

It would produce an output as followed:


ChapterThree : bash — Konsole
File Edit View Bookmarks Settings Help

```
[centi@DevMachine ChapterThree]$ ./ChapThreePartTwo
Let's try ActualSumFunction: 3
Let's try FalseSumFunction: -1
[centi@DevMachine ChapterThree]$
```

3.3 C# Counterpart

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

```
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>
```

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

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

Chapter 4

The History On Delegate Approach

4.1 Note

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

4.2 Prior to Nov 2017

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

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

4.3 Precursor to Advanced DL Support

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

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

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

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

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

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

4.4 The Advanced DL Support Approach

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

```
dotnet new console
```

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

Your CsProj file should looks like this:

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

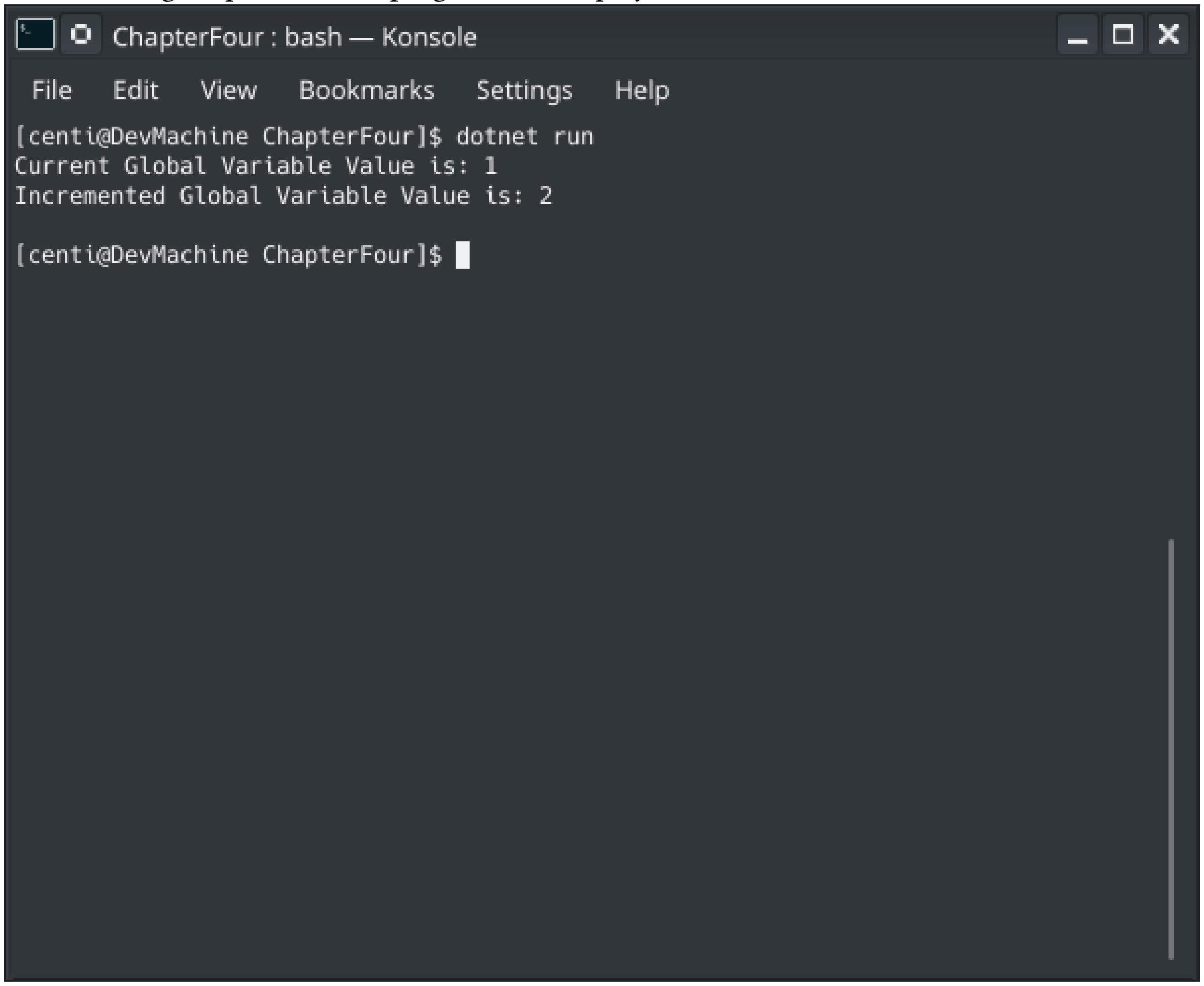
The C Library source could will simply have a global variable and a function to increment the said variable as followed:

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

The following output from the program will display as followed:


ChapterFour : bash — Konsole
File Edit View Bookmarks Settings Help

```
[centi@DevMachine ChapterFour]$ dotnet run
Current Global Variable Value is: 1
Incremented Global Variable Value is: 2

[centi@DevMachine ChapterFour]$ 
```

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

Chapter 5

Marshaling between C# and C 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[9]
}
```

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

To quote from that link:

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

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

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

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

5.1.1 Union

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

```
[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 fits in the memory boundary.

5.1.2 Packing and Size Options for Struct Layout

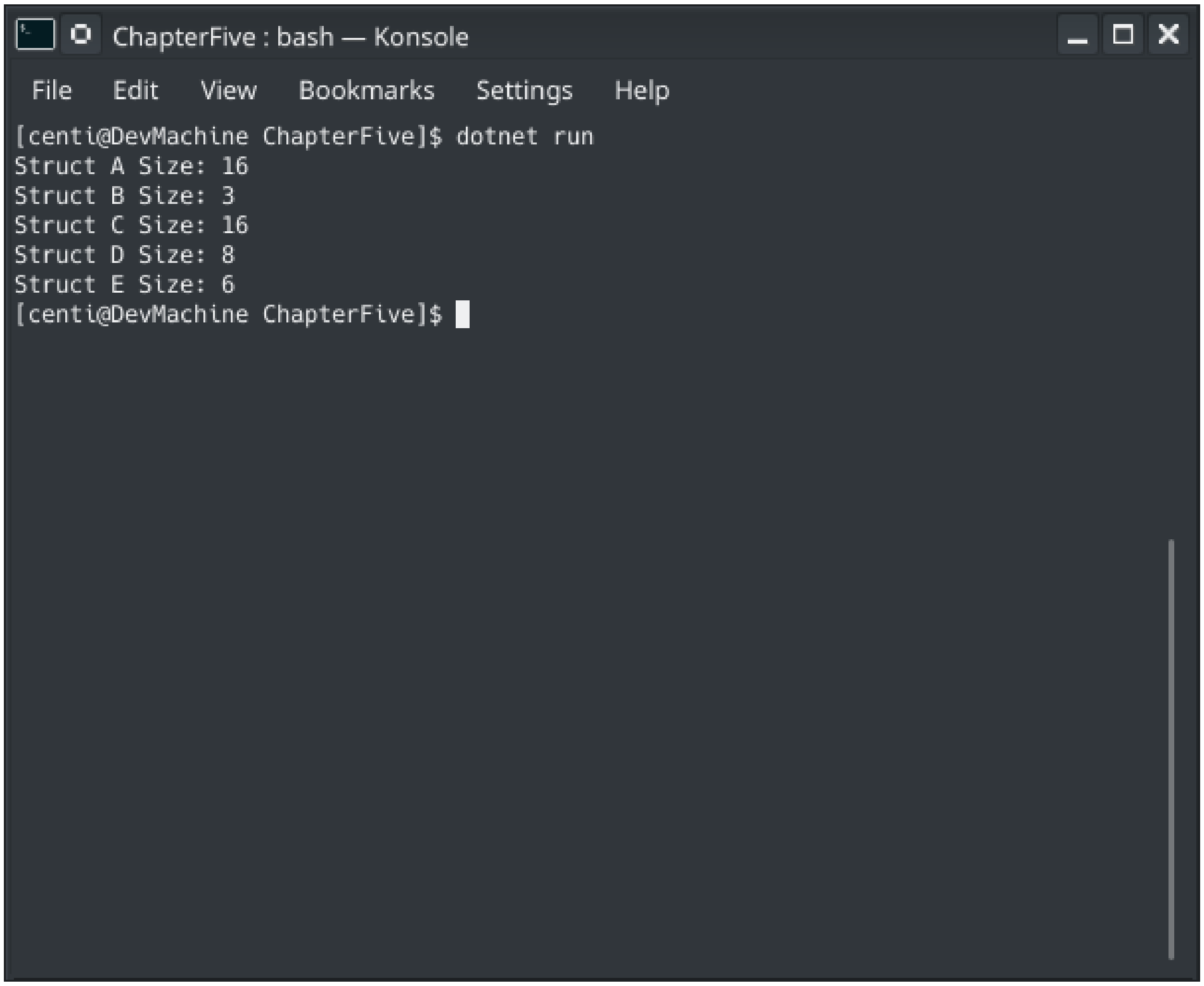
You can specify the packing option to respect byte boundary from a multiple of 1 byte to 8 bytes (.Net Core 3.0 and later will support 16 to 32 bytes boundary.)

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

```
[centi@DevMachine ChapterFive]$ dotnet run
Struct A Size: 16
Struct B Size: 3
Struct C Size: 16
Struct D Size: 8
Struct E Size: 6
[centi@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>.

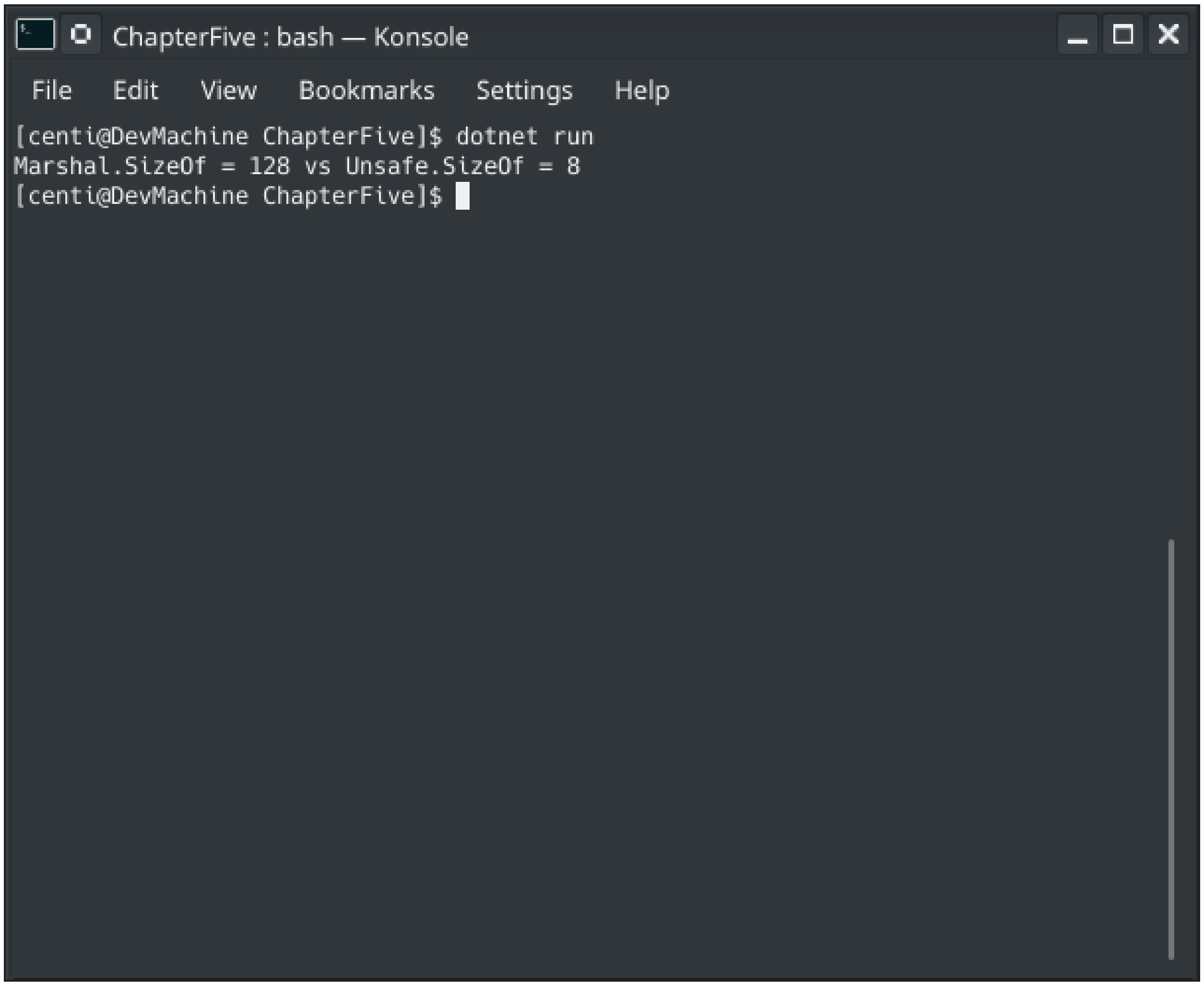
5.1.4 What is the Difference between sizeof and Marshal.SizeOf

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:


ChapterFive : bash — Konsole
File Edit View Bookmarks Settings Help

```
[centi@DevMachine ChapterFive]$ dotnet run
Marshal.SizeOf = 128 vs Unsafe.SizeOf = 8
[centi@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

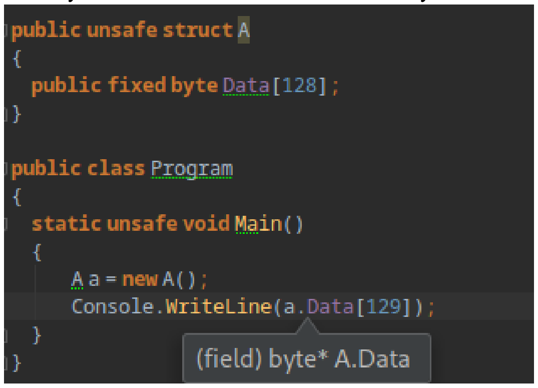
You can allocate a fixed sized array within a struct, but it is restricted to the following types: bool, byte, char, short, int, long, sbyte, ushort, uint, ulong, float, or double. Though there is a proposal for this change to allow user-defined structs to be used for fixed sized array:

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]);
  }
}
```

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;
        Console.WriteLine(ptrToByValueArrayOfStructs[0].A);
    }
}
```

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();
        Console.WriteLine(a[1].A);
    }
}
```

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.

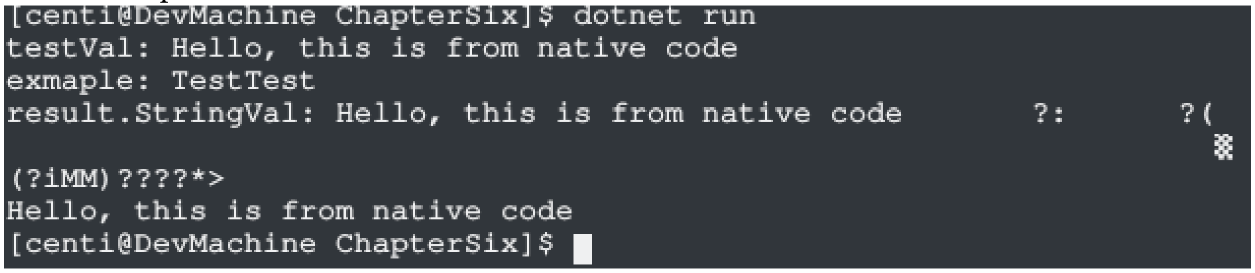
And for this lesson, we will have a few functions to play with for string manipulation from C Library:

```c
#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 IChapSixLib
{
    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 class Program
{
    static unsafe void Main()
    {
        var lib = NativeLibraryBuilder.Default.ActivateInterface<IChapSixLib>("ChapSix");
        var testVal = new StringBuilder(128);
        lib.SetDefaultMessage2(testVal);
        var example = new StringBuilder("Test\0Test");
        var result = Unsafe.AsRef<ByValCharArray>(lib.Initialize().ToPointer());
        lib.SetDefaultMessage(ref result);

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

        lib.AnsiString(ref result);
    }
}
```

And the output will be as followed:



```
[centi@DevMachine ChapterSix]$ dotnet run
testVal: Hello, this is from native code
exmaple: TestTest
result.StringVal: Hello, this is from native code       ?:      ?(
                                                                 ▒
(?iMM)????*>
Hello, this is from native code
[centi@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. 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

6.5 Name Mangling

6.6 Internal Calls

Chapter 7

Introduction to Common Intermediate Language

7.1 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:

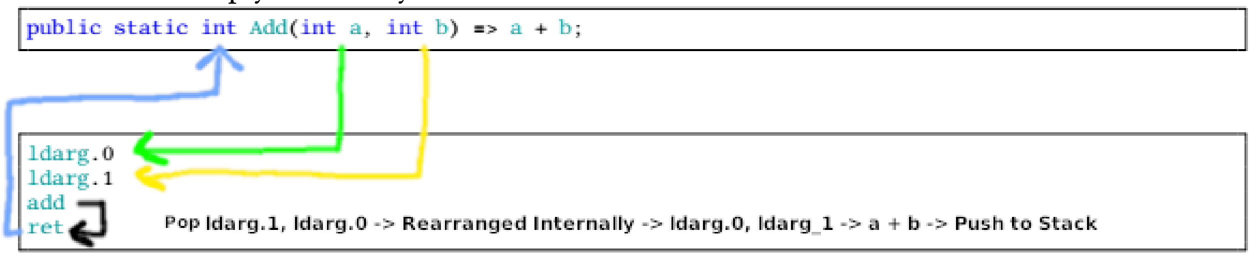
```
ldarg.0
ldarg.1
add
ret
```

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:



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

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

Pop ldarg.1, ldarg.0 -> Rearranged Internally -> ldarg.0, ldarg_1 -> a + b -> Push to Stack

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.

7.2 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);
```

7.3 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
  }
}
```

7.4 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.

7.4.1 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.

7.4.2 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 resides 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 instanced 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.

7.4.3 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.

7.5 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.Emits, 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

7.6 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.comen-usdotnetapisystem.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 explains that "value1 is added to value2" and that clearly define which value is left hand and vice versa.

7.6.1 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.

7.7 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.Emits, 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 attempts 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.

7.7.1 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.

7.7.2 Filter Handling

Chapter 8

Advanced CIL Programming

8.1 Constructing a Type at Runtime

One of the more advanced use of Runtime code emitting involves the use