Reverse Engineering
The Fundamentals of CIL

Washi washiwashi1337@gmail.com
766F6964 766F6964@protonmail.com

An educational writeup submitted for

www.rtn-team.cc

September 25, 2018
2nd Edition

Contents

Chapter 1

Introduction

Welcome to *Fundamentals of CIL* - a comprehensive guide to the .NET Intermediate Language (IL). This paper will present the most important CIL code semantics essential to learning .NET reverse engineering. The examples illustrated in this paper utilize the C# programming language for these reasons:

- Popularity
- Widespread Usage
- Verbose English-esque Syntax

To get the most out of this paper, the reader is encouraged to have at least a basic understanding of the C# programming language.

1.1 What is CIL?

The next step is to define what CIL is, which Wikipedia summarizes as:[1, Wikipedia]

> Common Intermediate Language (CIL), formerly called Microsoft Intermediate Language (MSIL), is the lowest-level human-readable programming language defined by the Common Language Infrastructure (CLI) specification and is used by the .NET Framework and Mono. Languages which target a CLI-compatible runtime environment compile to CIL, which is assembled into an object code that has a bytecode-style format. CIL is an object-oriented assembly language, and is entirely stack-based. Its bytecode is translated into native code or – most commonly – executed by a virtual machine.
>
> CIL was originally known as Microsoft Intermediate Language (MSIL) during the beta releases of the .NET languages. Due to standardization of C# and the Common Language Infrastructure (CLI), the bytecode is now officially known as CIL.

The largest benefit CIL provides is having compiled code that is platform and processor independent - as long as the target platform supports the Common Language Infrastructure (CLI). In theory then, multiple environments, platforms, and CPU types could be supported by a single software project without needing code dedicated to a specific environment, platform, or CPU type. This works because the platform's Common Language Infrastructure (CLI) fills in any environment, platform, or processor specific code that may be required - similar to Java's Java Runtime Environment (JRE).

1.2 What Is Next?

The content in this chapter may seem abstract at this point, to help clarify things the next chapter introduces the basics of using CIL.

Chapter 2

Basic Concepts

In the previous chapter, the goals of this paper and a brief definition of CIL were covered. This chapter expands on the previous chapter by outlining the basics of using CIL.

2.1 The Stack

One of the key aspects of the Common Intermediate Language (CIL) is that it is a language that works on a stack. As with many other low-level languages, the stack is a temporary data storage container consisting of all kinds of values that are used for performing calculations or making calls to other functions and procedures. It is sometimes referred to as the *evaluation stack*. A stack is a Last-In-First-Out (LIFO) data structure, which means that every time one would add an item to it, it is appended to the end of the collection. The operation to add an item is commonly known as *pushing* and the last item in the collection is commonly known as the *top of the stack*. Removing items from the collection is known as *popping* and will only remove the top-most item from the collection. An example of the stack behaviour can be seen in Figure 2.1.

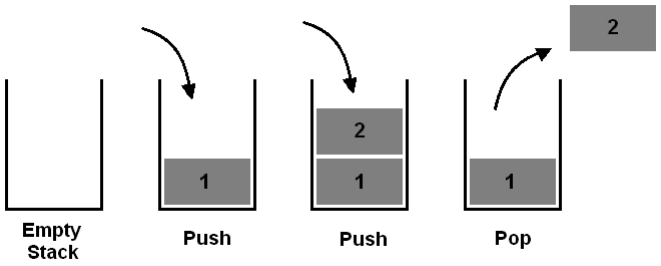

Figure 2.1: Illustration of Stack Behaviour

Some CIL instructions pop one or more values from the stack, which can be used to perform a certain operation. Sometimes a CIL instruction produces a result. This is then pushed back onto the stack for it to be used by another instruction.

2.2 Hello World!

A programming language tutorial would not be complete without a *Hello World!* example application.

```
1  .assembly FundamentalsOfCIL { }
2
3  .method public static void main(string[] args)
4  {
5      .entrypoint
6
7      ldstr "Hello, World!"
8      call void [mscorlib]System.Console::WriteLine(string)
9      ret
10 }
```

Listing 2.1: Hello World.

The emphasis of this code sample will be on the instructions found in the *.method* directive. The *.method* directive consists of three instructions which will be used to demonstrate the use of the stack mentioned in 2.1:

- **ldstr**: pushes a string onto the stack. The *Hello World* example pushes the string $Hello, World!$.
- **call**: invokes a method. If the method has defined parameters, then the arguments used are popped from the stack. If the invoked method returns a value, then the returned value is pushed onto the stack so the caller can retrieve the result. In the example, the $System.Console :: WriteLine(string)$ located in the library *mscorlib* is called. The method invoked requires a single *string* parameter, which uses the value pushed to the stack in the previous instruction. Since the method does not return a value, nothing is pushed to the stack after the invocation is made.
- **ret**: returns execution to the caller. Equivalent to a return statement in C# . If a method returns a value, it is important to note that the *ret* instruction would have been preceded by an instruction to push the returned value onto the stack.

2.3 Assembling IL Files

In addition to the standard libraries, the .NET Framework comes with an assembler named *ilasm*. *ilasm* takes an input *Intermediate Language* (IL) file and translates it to an executable file. The location of the *ilasm* assembler is typically `C:\Windows\Microsoft.NET\Framework(64)\vX.X.XXXXX` folder (where *X.X.XXXXX* represents a version of the .NET Framework).
Using the command line, the following command can be executed to assemble an IL source file:

```
ilasm MyFile.il
```

The output of the previous command is a *MyFile.exe* executable file that can be run like any other executable file.

2.4 Disassembling IL Files

The reverse process is also possible by using a disassembler. Microsoft provides a disassembler called *ildasm* with the .NET *Software Development Kit* (SDK). *ildasm* is typically located in the `C:\Program Files (x86)\Microsoft SDKs\Windows\vXX.X\bin\NETFX x.x.x Tools` directory (where *XX.X* represents a version of the .NET SDK).
ildasm provides a Graphical User Interface (GUI) which can be used by running the executable directly. Alternatively, the Command Line Interface (CLI) can be used by providing the file path of the target file to disassemble:

```
ildasm MyFile.exe
```

Other tools and disassembly methods are also available, including decompilers with the ability to translate IL code to a higher level language like C# . However, these tools and disassembly methods are beyond the scope of this paper.

2.5 Basic Arithmetic

In addition to supporting string data types (as shown in the *Hello World* example), the CIL supports integer and floating point numbers and the operations that can be performed on them.

2.5.1 Loading Numerical Values

Like any other value, the CIL stores numbers on the stack. To get the values onto the stack, the CIL provides different instructions depending on their data type and size requirements - similar to the **ldstr** instruction seen in the *Hello World* example. These instructions are shown in table 2.1.

Operation	**Description**
ldc.i4 *<int32 (value)>*	*Pushes* a 32-bit integer value onto the stack.
ldc.i4.s *<int8 (value)>*	*Pushes* an 8-bit integer as a 32-bit integer onto the stack.
ldc.i8 *<int64 (value)>*	*Pushes* a 64-bit integer value onto the stack.
ldc.r4 *<float32 (value)>*	*Pushes* a 32-bit floating point value onto the stack.
ldc.r8 *<float64 (value)>*	*Pushes* a 64-bit floating point value onto the stack.

Table 2.1: Operations for loading numerical values in CIL

Notice that the first two instructions of table 2.1 both *push* a 32-bit integer onto the stack. The difference between the two instructions is the size in bytes of the value *pushed* onto the stack. The **ldc.i4** instruction uses 1 byte for the mnemonic code and 4 bytes to represent the 32-bit integer value for a total of 5 bytes. The **ldc.i4.s** instruction still uses 1 byte for the mnemonic code and 1 byte for the value, a reduction in 3 bytes. In an attempt to generate more efficient code, the compiler may opt to use the second instruction over the first depending on the value. A value greater than 255 will have to use the first instruction since it is the only instruction providing enough byte space. Smaller values may be able to use the second instruction; however, resulting in more efficient and optimized code. The CIL provides a few more optimizations for a few specific values which require only 1 byte. These operations are listed in table 2.2:

Operation	**Description**
ldc.i4.0 to ldc.i4.8	*Pushes* a value between 0 and 8 onto the stack.
ldc.i4.m1	*Pushes* -1 onto the stack.

Table 2.2: Single byte operations for loading numerical values in CIL

2.5.2 Arithmetic Operations

Once a number has been pushed onto the stack, arithmetic operations can be performed on the *pushed* values. Table 2.3 lists the arithmetic operation instructions that can be performed:

Operation	Description
add add.ovf add.ovf.un	Adds two numerical values.
sub sub.ovf sub.ovf.un	Subtracts two numerical values.
mul mul.ovf mul.ovf.un	Multiplies two numerical values.
div div.un	Divides two numerical values.
rem rem.un	Computes the remainder of two numerical values.

Table 2.3: Basic arithmetic operations available in CIL

Some operations appear to have multiple instructions for that operation. For example, there are three instructions that can be used for addition arithmetic operations. One instruction performs the arithmetic operation normally. Instructions with **.ovf** have the runtime check for overflow and if overflow is found throw a $System.OverflowException$. In C# code this is a similar concept to the *checked* and *unchecked* statements. Instructions with **.un** perform the arithmetic operation as if the values in the arithmetic operation were unsigned.

2.5.3 Arithmetic Example

In the following code sample, the numbers 3 and 4 are added together and the result is written to the console:

```
1  .method public static void main(string[] args)
2  {
3      .entrypoint
4
5      ldc.i4 3
6      ldc.i4 4
7      add
8
9      call  void [mscorlib]System.Console::WriteLine(int32)
10     ret
11 }
```

Listing 2.2: Basic addition in CIL.

2.5.4 Bitwise Operations

In addition to the basic arithmetic operations CIL also supports bitwise operations. Bitwise operation instructions perform an action on the bits of a number. A list of all supported bitwise operation instructions can be found in table 2.4.

2.5.5 Bitwise Example

Just like arithmetic option instruction, bitwise operation instructions take either one, or two arguments and push the result on the stack. This can be seen in the code sample found in listing 2.3 which performs an xor operation on the numbers 3 and 4:

Operation	Description
and	Computes the *bitwise and* of two numerical values.
not	Computes the *bitwise complement* of a numerical value.
xor	Computes the *bitwise exclusive or* of two numerical values.
or	Computes the *bitwise complement* of two numerical values.
shl	*Shifts* an integer value to the left by a specified number of bits.
shr	*Shifts* an integer value to the right by a specified number of bits.
shr_un	*Shifts* an unsigned integer value to the left by a specified number of bits.

Table 2.4: Basic bitwise operations available in CIL

```
1  .method public static void main(string[] args)
2  {
3      .entrypoint
4
5      ldc.i4 3
6      ldc.i4 4
7      xor
8
9      call  void [mscorlib]System.Console::WriteLine(int32)
10     ret
11 }
```

Listing 2.3: Basic xor operations in CIL.

2.6 What Is Next?

The CIL provides several concepts found in other languages for interacting with strings and numbers. This chapter showed the basics of utilizing these concepts in the CIL. The concepts from this chapter will be utilized in the next chapter to introduce intermediate concepts utilizing the CIL.

Chapter 3

Intermediate Concepts

The basic concepts of using the CIL were covered in the previous chapter. With an understanding of the basic concepts, intermediate concepts such as variables, arrays, and control flow can be introduced.

3.1 Local Variables

Like other programming languages, the CIL allows variables to be declared and their values set to store and load data that may be referenced at a later time. The difference between variables in other programming languages and the CIL is how the variables are accessed once they have been declared. The CIL (and most decompilers/disassemblers) typically access local variables by their defined ordinal index - similar to C-based language arrays. The CIL can also access variables by a name token similar to how most modern programming languages access variables but this is uncommon.
The code snippet below may better illustrate how local variables are declared in the CIL as can be seen in listing 3.1.

```
1  .method public static void LocalTestMethod()
2  {
3      .locals init ([0] int32)
4      nop
5      ldc.i4.5
6      stloc.0
7      ldloc.0
8      call  void [mscorlib]System.Console::WriteLine(int32)
9      nop
10     ret
11 }
```

Listing 3.1: Defining and using local variables in CIL.

The **.locals** directive indicates that local variables will be declared in this scope. Usually, the **.locals** directive is followed by the **.init** directive to set the local variables memory segment to zero before a value is assigned to it. The contents between the parenthesis is where the local variables are defined. Each of these local variable definitions contains the ordinal index of the local variable and the local variables data type. Explanations for the remaining instructions will come shortly.
Pushing and *popping* data into defined local variables is done by using one of the following CIL instructions:
Table 3.1 contains information for the instructions in the previous code snippet that had not been introduced yet.
The first instruction from the previous code snippet *pushes* the value 5 onto the stack (refer to section 2.5.1 where this is introduced). The next instruction *pops* a value from the stack into the variable at index 0. This value is 5 because the value was placed there in the previous instruction. Next, the value in the variable at index 0 is *pushed* (back) onto the stack. This value is then provided as a parameter to the *WriteLine* function called in the next instruction.

Operation	Description
ldloc *<uint16 (index)>*	*Push* local variable at argument *index* onto the stack.
ldloc.0	*Push* local variable at index 0 onto the stack.
ldloc.1	*Push* local variable at index 1 onto the stack.
ldloc.2	*Push* local variable at index 2 onto the stack.
ldloc.3	*Push* local variable at index 3 onto the stack.
ldloc.s *<uint8 (index)>*	*Push* local variable at argument *index* onto the stack, short form.
ldloca *<uint16 (index)>*	*Push* address of local variable at argument *index* onto the stack.
ldloca.s *<uint8 (index)>*	*Push* address of local variable at argument *index* onto the stack, short form.
stloc *<uint16 (index)>*	*Pop* a value from the stack into local variable at argument *index*.
stloc.0	*Pop* a value from the stack into local variable at index 0.
stloc.1	*Pop* a value from the stack into local variable at index 1.
stloc.2	*Pop* a value from the stack into local variable at index 2.
stloc.3	*Pop* a value from the stack into local variable at index 3.
stloc.s *<uint8 (index)>*	*Pop* a value from the stack into local variable at argument *index*, short form.

Table 3.1: Instructions for *pushing* and *popping* local variable data in CIL

3.2 Arrays

Arrays are a useful data structure for holding values of the same type. While the idea of arrays is similar to the concepts of **Local Variables** introduced in section 3.1 their implementations are different.
In this paper, the array concepts are divided into four categories: *Creating Arrays*, *Initializing Arrays*, *Writing Data Into Arrays*, and *Reading Data From Arrays*.

3.2.1 Creating Arrays

In a language such as C# , an array would be created using code like can be seen in listing 3.2.

```
1 public void TestArray(){
2          int[] myarray = new int[3];
3 }
```

Listing 3.2: Array creation in C#.

To achieve the same result using the CIL, the **newarr** instruction is used. **Note:** The **newarr** can only be used for one-dimensional arrays. [2]

Operation	Description
newarr	*Pushes* an object reference to a new zero-based, one-dimensional array whose elements are of a specific type onto the stack.

The CIL complement to the C# code shown previously would look like the one found in listing 3.3.

```
1 .method public hidebysig instance default void TestArray () cil
     managed
2 {
3          .maxstack 1
4          .locals init (int32[] V_0)
5          ldc.i4.3
6          newarr [mscorlib]System.Int32
7          stloc.0
8          ret
9 }
```

Listing 3.3: Array creation in CIL.

Note: The **.maxstack** directive is *optional* since it is used specifically during code verification (while *required* is used in the [3]; the method may still execute without it) and is therefore beyond the scope of this paper. If **.maxstack** value was not specified explicitly, the compiler will calculate the correct **.maxstack** and use that instead.

> Every method specifies a maximum number of items that can be pushed onto the CIL evaluation stack. The value is stored in the IMAGE_COR_ILMETHOD structure that precedes the CIL body of each method. A method that specifies a maximum number of items less than the amount required by a static analysis of the method (using a traditional code flow graph without analysis of the data) is invalid (hence also unverifiable) and need not be supported by a conforming implementation of the CLI.
>
> Note: Maxstack is related to analysis of the program, not to the size of the stack at runtime. It does not specify the maximum size in bytes of a stack frame, but rather the number of items that must be tracked by an analysis tool.

The **newarr** instruction takes a single argument representing the size of the array to create. The code sample shown previously creates an array of size 3. **ldc.i4.3** (refer to section 2.5.1 where this is introduced) *pushes* the value 3 onto the stack before making the call to **newarr**.
Once the call to **newarr** has been made, the resulting array reference can be *popped* into a local variable for use later. In the code sample this can be seen on line 7 with **stloc.0**. Thereafter, any reference to the array *pushes* the variable onto the stack using **ldloc.0**. [2]

3.2.2 Initializing Arrays

There are two methods of initializing arrays in the CIL: explicit initialization and inline initialization. Explicit initialization is the same for C-based languages. Inline initialization allows the array size to be calculated from the items that are specified in the statement. The code sample found in listing 3.4 demonstrates explicit initialization and inline initialization.

```
1  public static void InitArray()
2  {
3      // Explicit initialization
4      int[] arr2 = new int[3];
5      arr2[0] = 10;
6      arr2[1] = 20;
7      arr2[2] = 30;
8
9      // Inline initialization
10     int[] arr = {10, 20, 30};
11 }
```

Listing 3.4: Array initialization in C#.

The explicit initialization method results in the CIL instructions as depicted in listing 3.5.

```
1  .method public hidebysig static void  InitArray() cil managed
2  {
3    .maxstack  8
4
5    ldc.i4.3
6    newarr     [mscorlib]System.Int32
7
8    dup
9    ldc.i4.0
10   ldc.i4.s   10
11   stelem.i4
```

```
12
13    dup
14    ldc.i4.1
15    ldc.i4.s    20
16    stelem.i4
17
18    ldc.i4.2
19    ldc.i4.s    30
20    stelem.i4
21
22    ret
23  }
```

Listing 3.5: Array initialization in CIL.

Lines 5 and 6 create an array reference of type *Int32* (see section 3.2.1).
The remaining code follows a similar pattern using the following instructions:

- **dup**: Duplicates the reference on the top of the stack, in this case of the array that was created.
- **ldc.i4.#**: *Pushes* the index in the array that is being referenced.
- **ldc.i4.s**: *Pushes* the value to be stored in the array at the specified index.
- **stelem.i4**: *Pops* the two parameters off of the stack and uses them to assign the value at a particular index in the array.

The inline initialization method results in the CIL instructions as found in listing 3.6.

```
1  .method public hidebysig static void  InitArray() cil managed
2  {
3    .maxstack  8
4    ldc.i4.3
5    newarr     [mscorlib]System.Int32
6    dup
7    ldtoken    field valuetype '<PrivateImplementationDetails>'/'
        __StaticArrayInitTypeSize=12' '<PrivateImplementationDetails>'
        ::'4B05543E313E9B7EB74D3ACF007D3CFAEDF99086'
8    call       void [mscorlib]System.Runtime.CompilerServices.
        RuntimeHelpers::InitializeArray(class [mscorlib]System.Array,
        valuetype [mscorlib]System.RuntimeFieldHandle)
9    pop
10   ret
11 }
```

Listing 3.6: Array initialization in CIL using the .NET Framework's compiler services namespace.

Lines 4 and 5 create an array reference of type *Int32* (see section 3.2.1).
Just like the previous example, the size is defined and the array is created. What really sets inline initialization apart is the use of **ldtoken**. The **ldtoken** instruction references a *valuetype field* from a different class specified by the alphanumeric string that follows. This value is automatically generated by the compiler. **ldtoken** uses the *runtime* reference of the field and *pushes* it onto the stack to be used by the next instruction.
The .NET Framework's *RuntimeHelpers* class contains the *InitializeArray* method which is used to complete the initialization of the array. Using the reference placed on the stack by **ldtoken**, *InitializeArray* deserializes the compiler generated alphanumeric string data into an array.
Finally a value is *popped* off the stack to clean up the stack frame and execution continues.

3.2.3 Writing Data Into Arrays

Arrays are used to hold data of similar data types. Depending on the data type, the CIL provides several instructions for storing data into an array.
Storing values in an array requires three parameters to be on the stack:

1. A reference to the array to store the value in.
2. The index in the array to store the value at.
3. The value to store in the index.

The explicit initialization example from the previous section showed an example of this when setting the initial values of the array.
A more concise example is shown in listing 3.7.

```
1 public void TestArray(){
2         int[] myarray = new int[3];
3         myarray[0] = 42;
4 }
```

Listing 3.7: Setting individual values in arrays using C#.

The CIL complement for the C# code above looks like the one found in listing 3.8.

```
1  .method public hidebysig instance default void WriteArray ()  cil
       managed
2  {
3          .maxstack 3
4          .locals init (int32[] V_0)
5          ldc.i4.3
6          newarr [mscorlib]System.Int32
7          stloc.0
8          ldloc.0
9          ldc.i4.0
10         ldc.i4.s 0x2a
11         stelem.i4
12         ret
13 }
```

Listing 3.8: Setting individual values in arrays using CIL.

Table 3.4 outlines more array instructions that can be used with arrays:
Lines 5 and 6 create an array reference of type *Int32* (see section 3.2.1).
The array reference created from the previous instructions is *popped* into the local variable at index 0. Next, the array reference is *pushed* back onto the stack. **ldc.i4.0** *pushes* 0 onto the stack, referencing the index in the array that is being set. The final argument, the value being stored in the array ($0x2A$ or 42 when converted to decimal from the example), is pushed onto the stack. Finally, the **stelem.i4** instruction uses the three parameters *pushed* onto the stack to set the value at the specified index in the array referenced.

3.2.4 Reading Data From Arrays

An array does not do any good if values are stored but cannot be read or used. To retrieve stored values from an array, only two arguments are needed.

1. A reference to the array the value is stored in.
2. The index in the array the value is stored at.

Operation	Description
stelem	Replaces the array element at a given index with the value on the stack, whose type is specified in the instruction.
stelem.i	Replaces the array element at a given index with the native int value on the stack.
stelem.i1	Replaces the array element at a given index with the int8 value on the stack.
stelemi.2	Replaces the array element at a given index with the int16 value on the stack.
stelem.i4	Replaces the array element at a given index with the int32 value on the stack.
stelem.i8	Replaces the array element at a given index with the int64 value on the stack.
stelem.r4	Replaces the array element at a given index with the float32 value on the stack.
stelem.r8	Replaces the array element at a given index with the float64 value on the stack.
stelem.ref	Replaces the array element at a given index with the object ref value (type O) on the stack.

Table 3.4: Instructions for *popping* individual elements from the stack and storing them into an array in CIL.

Table 3.6 references the instructions used on an array to retrieve a value stored in it.
A C# example program could look like the contents of listing 3.9.

```
1 public void TestArray(){
2         int[] myarray = new int[3];
3         myarray[0] = 42;
4         int value = myarray[0];
5 }
```

Listing 3.9: Reading individual elements from an array in C#.

Which would compile to the CIL code that is similar to listing 3.10:

```
1  .method public hidebysig instance default void TestArray () cil
       managed
2  {
3          .maxstack 3
4          .locals init (int32[] V_0, int32 V_1)
5          ldc.i4.3
6          newarr [mscorlib]System.Int32
7          stloc.0
8          ldloc.0
9          ldc.i4.0
10         ldc.i4.s 0x2a
11         stelem.i4
12         ldloc.0
13         ldc.i4.0
14         ldelem.i4
15         stloc.1
16         ret
17 }
```

Listing 3.10: Reading individual elements from an array in CIL.

The code is exactly the same as the code example from section 3.2.3 until line 12.
Line 12 *pushes* the reference to the array onto the stack. Then the the index in the array to be read (0) is *pushed* onto the stack. **ldelem.i4** uses the arguments *pushed* onto the stack to retrieve the value from

Operation	Description
ldelem.i	Loads the element at a specified array index onto the top of the stack as the type specified in the instruction.
ldelem.i	Loads the element with type native int at a specified array index onto the top of the stack as a native int.
ldelem.i1	Loads the element with type int8 at a specified array index onto the top of the stack as an int32.
ldelem.i2	Loads the element with type int16 at a specified array index onto the top of the stack as an int32.
ldelem.i4	Loads the element with type int32 at a specified array index onto the top of the stack as an int32.
ldelem.i8	Loads the element with type int64 at a specified array index onto the top of the stack as an int64.
ldelem.r4	Loads the element with type float32 at a specified array index onto the top of the stack as type F (float).
ldelem.r8	Loads the element with type float64 at a specified array index onto the top of the stack as type F (float).
ldelem.ref	Loads the element containing an object reference at a specified array index onto the top of the stack as type O (object reference).
ldelem.u1	Loads the element with type unsigned int8 at a specified array index onto the top of the stack as an int32.
ldelem.u2	Loads the element with type unsigned int16 at a specified array index onto the top of the stack as an int32.
ldelem.u4	Loads the element with type unsigned int32 at a specified array index onto the top of the stack as an int32.
ldelema	Loads the address of the array element at a specified array index onto the top of the stack as type (managed pointer).

Table 3.6: Instructions for reading and *pushing* individual elements from an array in CIL.

the array and place it on the stack. This value is then stored in the local variable at index 1 by **stloc.1** (see section 3.1).

3.3 Control Flow

Compiled applications are very rarely (if ever) run sequentially. When a condition is met, such as a local variable containing a unique value, the code can *jump* to a location where specific instructions can be executed. This *jump* is known as *Control Flow*.

3.3.1 Branches

The most common form of *Control Flow* is *Branches*, which jump to a location in memory when their instruction is executed.

Unconditional Branches

Unconditional Branches are the most basic types of *branches*. These instructions jump to a particular memory location, similar to a *goto* statement in some programming languages. The two *unconditional branch* instructions are shown in the following table:

Operation	Description
br *<label (target)>*	*Branch* to label.
br.s *<label (target)>*	*Branch* to *nearby* label.

Table 3.7: Unconditional Branch Instructions.

The key difference between the two is the *nearby*, also known as a short jump variant. The distinction comes into play when the destination address location is between -128 and 127 bytes away from the current instruction pointer. This was done intentionally to make the instruction mnemonic utilize only a single byte.
If the destination address location is further away the first variant must be used. Unlike the short instruction, the first variant uses four bytes to represent the instruction mnemonic.
The code sample in listing 3.11 provides C# code that utilizes an unconditional jump.

```
1 public static void main()
2 {
3     MyLabel:
4         Console.WriteLine("Hello World!");
5         goto MyLabel;
6 }
```

Listing 3.11: A simple goto-loop in C#.

The CIL representation of this code can be found in listing 3.12.

```
1 .method public static void main() {
2     .entrypoint
3
4 MyLabel:
5     ldstr "Hello, world!"
6     call void [mscorlib] System.Console::WriteLine(string)
7     br.s MyLabel
8 }
```

Listing 3.12: A simple loop in CIL.

The code emulates a do/while loop without a condition check. Once the code finishes executing the *Console.WriteLine* it jumps back to the line before it and executes again, infinitely.

Conditional Branches

Most branches need to execute code only when a condition is met, known as a *Conditional Branch*. **if** statements in all programming languages are an example of a *conditional branch*. The CIL provides the instructions in the following table as a means of utilizing *conditional branches* in the IL code:

Operation	**Description**
brtrue[.s] *<label (target)>*	*Branch* if non-zero (true).
brfalse[.s] *<label (target)>*	*Branch* if stack zero (false).
beq[.s] *<label (target)>*	*Branch* if equal.
bge[.un][.s] *<label (target)>*	*Branch* if greater or equal.
bgt[.un][.s] *<label (target)>*	*Branch* if greater.
ble[.un][.s] *<label (target)>*	*Branch* if less or equal.
blt[.un][.s] *<label (target)>*	*Branch* if less.
bne.un[.s] *<label (target)>*	*Branch* if not equal.

Table 3.8: Conditional Branch Instructions.

Listing 3.13 provides an implementation of a program that asks the user to guess the secret number:

```
1  public static void main()
2  {
3      Console.WriteLine("Guess the number: ");
4
5      if (int.Parse(Console.ReadLine()) == 1337)
6      {
7          Console.WriteLine("Correct");
8      }
9      else
10     {
11         Console.WriteLine("Incorrect");
12     }
13 }
```

Listing 3.13: Conditional statements in C#.

Which compiles to the CIL code that is similar to listing 3.14.

```
1  .method public static void main() {
2      .entrypoint
3
4      // Ask for a number:
5      ldstr "Guess the number: "
6      call void [mscorlib] System.Console::Write(string)
7
8      call string [mscorlib] System.Console::ReadLine()
9      call int32 [mscorlib] System.Int32::Parse(string)
10     ldc.i4 1337
11     bne.un FalseLabel // if number != 1337 go to FalseLabel
12
13 TrueLabel: // number equals 1337
14     ldstr "Correct!"
15     br EndIf
16
17 FalseLabel: // number did not equal 1337
18     ldstr "Incorrect!"
19
20 EndIf:
21     // Show result:
22     call void [mscorlib] System.Console::WriteLine(string)
23     ret
24 }
```

Listing 3.14: Conditional branches in CIL.

The conditional logic occurs on line 11 when the **bne.un** instruction is executed.

3.3.2 Switches

Another form of *Control Flow* supported by C-based languages is the **switch** statement. A **switch** statement is the equivalent of an **if**/**else if** statement condensed into a single scope where **case**s are defined. The **switch** statement in the CIL is limited to a single comparison value compared to those provided by other programming languages.
The **switch** instruction in the CIL implements a *jump* table. This means that the **case** is selected depending on the index that is *pushed* onto the stack instead of evaluating a conditional expression. When the **switch** instruction executes it will *pop* the index value from the stack and jump to the **case** label corresponding to that index. A **case** label can occur several times in the jump table, which allows for **case** blocks to have multiple possibilities for selection. If the index does not have a corresponding **case** block, then no jump is performed and execution continues at the instruction after the switch.
The code sample in listing 3.15 outlines a program utilizing a **switch** statement.

```
1  public static void main() {
2      Console.WriteLine("Enter a number: ");
3      switch(int.Parse(Console.ReadLine()))
4      {
5          case 0:
6              Console.WriteLine("Case 0");
7              break;
8          case 1:
9          case 3:
10             Console.WriteLine("Case 1 and 3");
11             break;
12         case 2:
13             Console.WriteLine("Case 2");
14             break;
15         default:
16             Console.WriteLine("Something else");
17             break;
18     }
19 }
```

Listing 3.15: A switch statement in C#.

When compiled, the CIL code will look similar to the code found in listing 3.16.

```
1  .method public static void main() {
2      .entrypoint
3
4      // Ask for a number:
5      ldstr  "Enter a number: "
6      call   void [mscorlib] System.Console::Write(string)
7
8      call   string [mscorlib] System.Console::ReadLine()
9      call   int32 [mscorlib] System.Int32::Parse(string)
10
11     // Perform switch
12     switch (Case_0, Case_1_And_3, Case_2, Case_1_And_3)
13 Default:
14     ldstr "Something else"
15     br    EndSwitch
16 Case_0:
17     ldstr "Case 0"
18     br    EndSwitch
19 Case_1_And_3:
20     ldstr "Case 1 and 3"
21     br    EndSwitch
22 Case_2:
23     ldstr "Case 2"
24
25 EndSwitch:
26     call   void [mscorlib] System.Console::WriteLine(string)
27     ret
28 }
```

Listing 3.16: A switch construction in CIL.

The **switch** statement on line 12 contains the jump table. When a value does not exist the code *pushes Something else* onto the stack before *jumping* to the end where it calls *Console.WriteLine*.

3.4 Summary

Programs that run sequentially are rare and the basic concepts learned in the previous chapter will only go so far. This chapter built on the basic concepts from the previous chapter making more complex programs possible. At the conclusion of this chapter the reader should be comfortable with Local Variables, Arrays, and Control Flow. The next chapter introduces more advanced concepts that can be utilized by CIL programs.

Chapter 4

Advanced Concepts

In this chapter, CIL advanced concepts will be covered to supplement the basic concepts from previous chapters by introducing Error Handling, Value Conversion, Type Casting, Boxing, and utilizing the WinAPI and other native code using PInvoke.

4.1 Error Handling

Nobody likes applications that crash. Especially those that crash unexpectedly. Language designers figured this out and provided the **try** and **catch** keywords to give the program a chance to handle and recover from any unexpected errors.
The CIL provides the same capability by utilizing multiple handler types to handle these unexpected errors (also referred to as exceptions). The handler types provided by the CIL are located in table 4.1.

Type	**Description**	**Category**
Catch	The catch block includes the exception type and the block of code to transfer control to. The catch clause with the matching exception type will be chosen as handler. Multiple catch blocks can be chained after the try block.	Resolving
Filter	The filter block does not match the exception type but provides a scope block where it evaluates whether it wants to handle the exception or not. If it pushes a 1 on the stack it indicates the runtime to handle the exception, otherwise 0 is pushed. After the evaluation block comes the handler block which contains the code to handle the exception.	Resolving
Finally	The finally block gets always executed regardless of whether an exception occurred and was handled or not.	Observing
Fault	The fault block is similar to the finally block, except that its only executed if the associated try block did throw an exception.	Observing

Table 4.1: CIL Exception Handler Types

Exception handlers can be categorized into two different groups:

- Resolving Handlers
- Observing Handlers

Resolving handlers *attempt* to *resolve* the exception, as their name suggests, so program execution can continue. The **catch** and **filter** handlers are the two kinds of resolving handlers provided by the CIL.
Observing handlers *observe* that an exception has occurred, as their name suggests, and may execute code as part of the code block but *do not resolve the exception.* The **finally** and **fault** handlers are the two kinds of observing handlers provided by the CIL [2].
Exception handling in the CIL is done by emitting each handler separately. By doing so, the CIL runtime can easily choose which code blocks to execute. For example, the **try**/**catch**/**finally** construct is emitted as two exception handlers: one for the **catch** block and one for the **finally** block. A code example showing this can be found in listing 4.1.

```
public static void ExceptionExample()
{
        try
        {
                Console.WriteLine(int.Parse("abc"));
        }
        catch (FormatException)
        {
                Console.WriteLine("Catch block");
        }
        finally
        {
                Console.WriteLine("Finally block");
        }
        Console.WriteLine("Afterwards");
}
```

Listing 4.1: A try-catch-finally construct in C#.

The CIL compiled instructions can be found in listing 4.2.

```
.method public hidebysig static void ExceptionExample () cil managed
{
    .maxstack  1
    .try
    {
        .try
        {
            ldstr      "abc"
            call       int32 [mscorlib]System.Int32::Parse(string)
            call       void [mscorlib]System.Console::WriteLine(
                int32)
            leave.s    EndTry
        }  // end .try
        catch [mscorlib]System.FormatException
        {
            pop
            ldstr      "Catch block"
            call       void [mscorlib]System.Console::WriteLine(
                string)
            leave.s    EndTry
        }  // end handler
    }  // end .try
    finally
    {
        ldstr      "Finally block"
        call       void [mscorlib]System.Console::WriteLine(string)
        endfinally
    }  // end handler
EndTry:
    ldstr      "Afterwards"
    call       void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

Listing 4.2: A try-catch-finally construct in CIL.

The example attempts to convert the string "abc" into an integer which results in a *FormatException* because the string is not a valid integer value. When the *FormatException* (or any exception for that matter) occurs, the CLR *pushes* the exception onto the stack for use with the handler blocks. Execution then leaves the **try** block via the **leave** instruction. More information on the **leave** instruction can be found in Table 4.3. The **catch** block then evaluates whether it can handle the exception that occurred. If the **catch** block can handle the exception, the exception is *popped* off the stack to clean up the stack and allow this exception block to reference the exception object - often to get the exception message. Next, the **finally** handler is executed and exited using the **endfinally** instruction. Information on the **endfinally** instruction is found in Table 4.3. Finally, execution continues with the code that comes after the **try/catch/finally** construct.
Within each emitted handler block, the CIL provides keyword instructions specific to exception handling can be used to tell the CIL runtime which handler blocks to execute. Table 4.3 lists the exception handling instructions provided.

Instruction	**Description**
throw	Throws an exception.
rethrow	Rethrows the current exception.
leave *<label (target)>*	Exits a protected region of code.
leave.s *<label (target)>*	Exits a protected region of code. Short form
endfault	Ends fault clause of an exception block.
endfilter	Ends an exception handling filter clause.
endfinally	Ends finally clause of an exception block.

Table 4.3: Exception handler instructions

4.2 Type Conversions

Sometimes, especially when complex objects are involved, data needs to be referenced differently. This section outlines how to cast data using *Number Conversions*, *Reference Types*, and explains *Boxing*.

4.2.1 Number Types

Number conversions are one of the most occurring conversions that take place in a program. In order to translate between number types, the CIL defines a bunch of standard, single-byte opcode instructions for these operations:

Operation	**Description**
conv.i	*Converts* to a native int (IntPtr).
conv.ovf.i	Same as conv.i but throws exception upon overflow.
conv.ovf.i.un	Same as conv.ovf.i but performs conversion as an unsigned integer.
conv.i1	*Converts* to an int8 (sbyte).
conv.ovf.i1	Same as conv.i1 but throws exception upon overflow.
conv.ovf.i1.un	Same as conv.ovf.i1 but performs conversion as an unsigned integer.
conv.i2	*Converts* to an int16 (short).
conv.ovf.i2	Same as conv.i2 but throws exception upon overflow.
conv.ovf.i2.un	Same as conv.ovf.i2 but performs conversion as an unsigned integer.
conv.i4	*Converts* to an int32 (int).
conv.ovf.i4	Same as conv.i4 but throws exception upon overflow.
conv.ovf.i4.un	Same as conv.ovf.i4 but performs conversion as an unsigned integer.
conv.i8	*Converts* to a int64 (long).
conv.ovf.i8	Same as conv.i8 but throws exception upon overflow.
conv.ovf.i8.un	Same as conv.ovf.i8 but performs conversion as an unsigned integer.
conv.u	*Converts* to a native unsigned int (UIntPtr).
conv.ovf.u	Same as conv.u but throws exception upon overflow.
conv.ovf.u.un	Same as conv.ovf.u but performs conversion as an unsigned integer.

conv.u1	*Converts* to an uint8 (byte).
conv.ovf.u1	Same as conv.u1 but throws exception upon overflow.
conv.ovf.u1.un	Same as conv.ovf.u1 but performs conversion as an unsigned integer.
conv.u2	*Converts* to an uint16 (ushort).
conv.ovf.u2	Same as conv.u2 but throws exception upon overflow.
conv.ovf.u2.un	Same as conv.ovf.u2 but performs conversion as an unsigned integer.
conv.u4	*Converts* to an uint32 (uint).
conv.ovf.u4	Same as conv.u4 but throws exception upon overflow.
conv.ovf.u4.un	Same as conv.ovf.u4 but performs conversion as an unsigned integer.
conv.u8	*Converts* to an uint64 (ulong).
conv.ovf.u8	Same as conv.u8 but throws exception upon overflow.
conv.ovf.u8.un	Same as conv.ovf.u8 but performs conversion as an unsigned integer.
conv.r4	*Converts* to a float32 (single).
conv.r8	*Converts* to a float64 (double).
conv.r.un	*Converts* an unsigned integer to a floating point number.

Table 4.4: Single-Byte Number Conversion Instructions

The code sample in Listing 4.3 and the resulting CIL code sample in listing **??** are used to explain this concept.

```
1 public static void Main()
2 {
3     int x = 3;
4     byte y = (byte) x;
5     Console.WriteLine(y);
6 }
```

Listing 4.3: Casting numerical values in C#.

The program code initializes an integer with a value. The example was purposefully set up this way, instead of using *(byte) ¡integer value¿* because the underlying type of byte is an unsigned integer. The next instruction converts this value to a byte representation which is then displayed via *Console.WriteLine*.

```
1  .method public static hidebysig void Main() cil managed
2  {
3      .entrypoint
4      .maxstack 1
5      .locals init ( [0] int32, [1] uint8 )
6
7      nop
8      ldc.i4.3
9      stloc.0
10
11     ldloc.0
12     conv.u1
13     stloc.1
14
15     ldloc.1
16     call void [mscorlib]System.Console::WriteLine(int32)
17
18     nop
19     ret
20 }
```

Listing 4.4: Casting numerical values in CIL.

As seen from the section 3.1, the CIL code utilizes two local variables. **conv.u1** on line 12 of the CIL example converts the integer value to an unsigned byte and stored in the second (index 1) variable.

4.2.2 Reference Types

Often, a variable or a value on the stack with a specific type needs to be converted to a different type. For example, if a value on the stack is a *System.Object* type but needs to be *System.String* type; then an attempt can be made to cast the *System.Object* type on the stack to a *System.String* type. The CIL provides two operations to perform this conversion which are listed in Table 4.5.

Operation	Description
castclass *<type (class)>*	*Casts* an object to the specified type and throws an exception when it fails.
isinst *<type (class)>*	*Casts* an object to the specified type and returns null when it fails.

Table 4.5: Type Conversion Instructions

The difference between the two is that the **castclass** instruction *throws* an exception when the conversion fails; while the **isinst** instruction returns a **null** value instead. The **null** value return type allows a value to be tested to determine if it is a specific type or not. The C# keywords **is** and **as** operate in this way. An example of this can be found in Listing 4.5 and the resulting CIL code in Listing 4.6.

```
1 public static void Main()
2 {
3     var o = SomeMethod();
4     if (o is MyClass)
5     {
6         Console.WriteLine("o is of type MyClass");
7     }
8 }
```

Listing 4.5: Reference type testing and casting in C#.

A message is displayed if the resulting value from *SomeMethod* is an instance of *MyClass*.

```
1  .method public static hidebysig void Main() cil managed
2  {
3     .entrypoint
4     .maxstack 2
5     .locals init ( [0] object )
6
7     call object MyApp.Program::SomeMethod()
8     stloc.0
9     ldloc.0
10    isinst class MyApp.Program.MyClass
11    ldnull
12    bne.un EndIf
13
14    ldstr "o is of type MyClass"
15    call void [mscorlib]System.Console::WriteLine(string)
16
17 EndIf:
18    ret
19 }
```

Listing 4.6: Reference type testing and casting in CIL.

Line 10 shows the **isinst** instruction providing the *MyApp.Program.MyClass* comparison. The next line *pushes* **null** onto the stack before doing a comparison (as demonstrated in section **??**).

4.2.3 Boxing/Unboxing

Type casting reference types is different than when type casting value types. In order to convert from one type to another, a process called *un/boxing* is used. *Boxing* is the act of converting a value type to a *System.Object* type or to an interface, turning it into a reference type.
Conversely, unboxing is the reverse process.
To support the *un/boxing* mechanisms, CIL defines the three instructions in Table 4.6.

Operation	**Description**
box *<type (class)>*	*Boxes* a value type into an object.
unbox *<type (class)>*	*Unboxes* an object to a managed pointer of the specified value type.
unbox.any *<type (class)>*	*Unboxes* an object to the specified type.

Table 4.6: Un/Boxing Instructions

Listing 4.7 shows a code example that will demonstrate these instructions in the compiled CIL code found in Listing 4.8.

```
1  public static void Main()
2  {
3      int o = (int) SomeMethod();
4  }
5
6  private static object SomeMethod()
7  {
8      return 0;
9  }
```

Listing 4.7: Boxing of values in C#.

A method is called with a signature return type of *System.Object* which returns 0. The callee casts this result to an integer value.

```
1  .method public static hidebysig void Main() cil managed
2  {
3      .entrypoint
4      .maxstack 1
5      .locals init ( [0] int32 )
6
7      call object MyApp.Program::SomeMethod()
8      unbox.any valuetype [mscorlib]System.Int32
9      stloc.0
10
11     ret
12 }
13
14 .method private static hidebysig object SomeMethod() cil managed
15 {
16     ldc.i4.0
17     box valuetype [mscorlib]System.Int32
18     ret
19 }
```

Listing 4.8: Boxing of stack values in CIL.

Notice that the **unbox** instruction pushes a managed pointer to the value type instead of the value type itself. To be able to actually use the value, it is required to follow such an instruction with an instruction to *load* it. For example, an **ldobj** instruction, which pushes the actual value referenced by the pointer onto the stack. The opcode **unbox.any** does this in one instruction, and is often used instead. It is important to note that **unbox.any** can also have a reference type as operand. This would have the same effect as using the **castclass** opcode with the same operand.

4.3 PInvoke

PInvoke (Platform Invoke) describes a technology to access functions and structs in unmanaged libraries from managed code. Wikipedia defines it as[4]:

> a feature of Common Language Infrastructure implementations, like Microsoft's Common Language Runtime, that enables managed code to call native code.

In order to reference native assemblies, a wrapper definition must be generated using the *DllImport* annotation and the *extern* keyword on the wrapper declaration. At a minimum, the *DllImport* annotation requires the name of the native DLL to reference. Additional parameters can be provided to set the character set, marshalling, whether the native function utilizes get last error, and more. The *CharSet* field, for example, tells the annotation that all string parameters should be marshalled into the specified character set. A complete list of all DLLImport fields can be found on msdn. [5]
The signature of the managed method may need to massaged into a format that the native library can understand, especially when different system architectures can be supported. For example, *IntPtr* is often used instead of integers so that the same declaration can be used for both 32-bit and 64-bit systems.
Once the wrapper is defined, it can be referenced like a normal method call.
The source code in Listing 4.9 provides an example of how this is done using C# to use the *MessageBox* function from *user32.dll*.

```
1 [DllImport("user32.dll", CharSet = CharSet.Unicode)]
2 public static extern int MessageBox(IntPtr hWnd, string text, string
      caption, uint type);
```

Listing 4.9: A P/Invoke signature of the MessageBox API found in user32.dll.

Since *MessageBox* is declared in *user32.dll*, the *Relative Virtual Address (RVA)* will always be 0. The compiled CIL code for this example is found in Listing 4.10.

```
1 .method public static hidebysig pinvokeimpl ("user32.dll" as "
    MessageBox" unicode winapi )
2           default int32 MessageBox (native int hWnd, string text,
                string caption, unsigned int32 'type')  cil managed
                preservesig
3     {
4         // Method begins at RVA 0x0
5     }
```

Listing 4.10: A P/Invoke signature of the MessageBox API found in user32.dll.

There are a few specialized keywords used in the generated code, but little else can be done by developers other than defining the wrapper.

4.4 Summary

This chapter demonstrated some of the advanced features the CIL provides to make it a more robust language. The next chapter expands the understanding of the CIL from a stack-based programming language into an object-oriented programming language.

Chapter 5

Object-Oriented Programming

The previous chapters have exclusively looked at the CIL programming language stack-based features. This chapter introduces how the CIL provides object-oriented capabilities in addition to the stack-based features.

5.1 Defining Complex Types

Every type in CIL is a class and can be defined by using the **.class** keyword. A simple example can be found in Listing 5.1.

```
1  .assembly Test {}
2  .class public MyClass
3  {
4      .method public static void Main()
5      {
6          .entrypoint
7
8          ldstr "Hello, world!"
9          call void [mscorlib]System.Console::WriteLine(string)
10         ret
11     }
12 }
```

Listing 5.1: A simple class definition in CIL, containing one static member Main.

The **.class** keyword is applied to *MyClass*.

5.1.1 Access Modifiers

In the example shown in Listing 5.1, the class definition contains **public**. This keyword is an access modifier that signifies that the type is public and can be accessed by anyone, even another application. Table 5.1 lists all access modifiers that are available in the CIL language:

Attribute	**Description**
	No access modifier means that the type is internal to the assembly (**internal** in C#).
public	The type is public to everyone (**public** in C#).
nested public	The type is a nested class that is accessible to everyone (**public** in C#).
nested private	The type is a nested class that is only accessible by its enclosing class (**private** in C#).
nested family	The type is a nested class that is only accessible by classes that derive from the enclosing class (**protected** in C#).
nested assembly	The type is a nested class that is only accessible by classes within the same assembly (**internal** in C#).
nested famorassem	The type is a nested class that is only accessible by classes that derive from the enclosing class, and by all classes within the same assembly (**protected internal** in C#).

nested famandassem	The type is a nested class that is only accessible by classes that derive from the enclosing class within the same assembly.

Table 5.1: CIL Class Access Modifiers

Notice that specific opcodes exist for nested classes. Every nested class should have at least one of these access modifiers specified. Otherwise the assembler will produce an invalid assembly.

5.1.2 Layout Modifiers

The CIL defines layout modifiers to describe how a class structure is represented in memory. Table 5.2 lists the layout modifiers provided by the CIL.

Attribute	**Description**
auto	The memory layout is specified by the runtime (this is default).
sequential	The memory layout of the class is sequential, meaning that every field appears after the other in the order as they are defined (In C# this has the same effect as giving your class the *StructLayout* with the Value property set to Sequential).
explicit	The offset of each field in the class is explicitly specified. This is in particular very useful if only a few of the fields of the memory chunk are important, which can therefore be the only fields to be defined in this class (In C# this is realized with the *FieldOffset* attribute).

Table 5.2: CIL Class Structure Layout Modifiers

5.1.3 Encoding Modifiers

Characters and strings can be defined in several ways. The CIL provides *Encoding Modifiers* that allow the programmer to specify how the runtime should interpret characters for a class. The encoding modifier operands provided by the CIL are listed in Table 5.3.

Attribute	**Description**
	No encoding modifier means the character encoding is specified by the runtime (this is default).
ansi	The ANSI character encoding is used.
unicode	The Unicode character encoding is used.

Table 5.3: CIL Class Encoding Modifiers

5.2 Basic Polymorphism

5.2.1 Extending from Object

The class in section 5.1 is a static class, this means that no instance can be created from it. To be able to create instances of a class, it is required to extend from another class that eventually derives from **System.Object**. This is done by the **extends** keyword as can be seen in listing 5.2

```
1 .class public MyClass
2     extends [mscorlib] System.Object
3 {
4 }
```

Listing 5.2: A class definition that extends System.Object from mscorlib.

Next to extending **System.Object**, we also need a constructor. Unlike C# , it is always required to define a constructor, even the parameterless one. Constructors are normal methods but with a special

name (**.ctor** or **.cctor**) and a few extra method modifiers specified. A simple class in C# that is defined like in listing 5.3.

```
1 public class MyClass
2 {
3     public MyClass()
4         : base()
5     {
6     }
7 }
```

Listing 5.3: A class in C# with a parameterless constructor calling the constructor of the base class.

can be implemented using the following piece of code:

```
1 .class public ansi beforefieldinit MyClass
2     extends [mscorlib] System.Object
3 {
4     .method public hidebysig specialname rtspecialname void .ctor()
5     {
6         ldarg.0
7         call instance void [mscorlib] System.Object::.ctor()
8         ret
9     }
10 }
```

Listing 5.4: A class in CIL with a parameterless constructor calling the constructor of the bass class.

Notice that in the body of the constructor we see that the constructor method of **System.Object** is called as well. This is the result of the **base()** clause in the constructor of the original C# snippet. However, while in C# this clause is often redundant, it is always present in a CIL implementation. It is required by every constructor to eventually call the constructor of the base class in order to initialize the object properly. More about calling members from classes and subclasses in section 5.3.2.

5.2.2 Abstract classes

Some classes only serve a purpose to provide a base for its derived classes and should not be initialized on their own. These are also known as *abstract classes*. To support this, CIL defines the **abstract** type modifier. The simple demonstration of an abstract class in C# as found in listing 5.5

```
1 public abstract class MyAbstractClass
2 {
3    public virtual void MyVirtualMethod()
4    {
5       Console.WriteLine("This method can be overridden.");
6    }
7    public abstract void MyAbstractMethod();
8 }
9 public sealed class MyDerivedClass : MyAbstractClass
10 {
11    public override void MyAbstractMethod()
12    {
13       Console.WriteLine("This method had to be overwritten.");
14    }
15 }
```

Listing 5.5: An abstract class and a sealed class that derives from the abstract class in C#.

can be implemented using the CIL code in listing 5.6.

```
.class public abstract MyAbstractClass
    extends [mscorlib] System.Object
{
    .method public hidebysig specialname rtspecialname void .ctor()
    {
        ldarg.0
        call instance void [mscorlib] System.Object::.ctor()
        ret
    }

    .method public virtual void MyVirtualMethod()
    {
        ldstr "This method can be overridden."
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }

    .method public virtual abstract void MyAbstractMethod() { }
}

.class public sealed MyDerivedClass
    extends MyAbstractClass
{
    .method public hidebysig specialname rtspecialname void .ctor()
    {
        ldarg.0
        call instance void MyAbstractClass::.ctor()
        ret
    }

    .method public virtual void MyAbstractMethod()
    {
        ldstr "This method had to be overwritten."
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
}
```

Listing 5.6: An abstract class and a sealed class that derives from the abstract class in CIL.

Notice that the methods that are defined in the abstract class are flagged with the **virtual** attribute. Just like in C# , this indicates that the method can be overridden by any derived class. Furthermore, from the example it can be seen that the **abstract** modifier is also applicable to methods, which indicate that the programmer must override the method in the derived class. Finally, this example also introduces the **sealed** type modifier, which prohibits any other class deriving from the type *MyDerivedClass* and maps one-to-one with the C# version.

5.2.3 Interfaces

Interfaces are very similar to abstract classes, except for the fact that each member is abstract, meaning that each member defined in the interface does not have a body and has to be implemented by the derived class. It is also possible to implement multiple interfaces at once, whilst only one abstract class can be extended at a time. Interfaces are simple **.class** declarations with the **interface** and **abstract** type modifiers specified. Interfaces are implemented by a class using the **implements** keyword.
An example of how interfaces appear in C# can be found in listing 5.7.

```
1  public interface MyFirstInterface
2  {
3      void MyFirstMethod();
4  }
5
6  public interface MySecondInterface
7  {
8      void MySecondMethod();
9  }
10
11 public class MyClass : MyFirstInterface, MySecondInterface
12 {
13     public void MyFirstMethod()
14     {
15         Console.WriteLine("First method");
16     }
17
18     public void MySecondMethod()
19     {
20         Console.WriteLine("Second method");
21     }
22 }
```

Listing 5.7: Two interface definitions and a class implementing them in C#.

This roughly equates to the following CIL code:

```
.class interface public abstract MyFirstInterface
{
    .method public virtual abstract void MyFirstMethod() { }
}

.class interface public abstract MySecondInterface
{
    .method public virtual abstract void MySecondMethod() { }
}

.class public MyClass
    extends [mscorlib] System.Object
    implements MyFirstInterface, MySecondInterface
{
    .method public hidebysig specialname rtspecialname void .ctor()
    {
        ldarg.0
        call instance void [mscorlib]System.Object::.ctor()
        ret
    }

    .method public final virtual void MyFirstMethod()
    {
        ldstr "First method"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }

    .method public final virtual void MySecondMethod()
    {
        ldstr "Second method"
        call void [mscorlib]System.Console::WriteLine(string)
        ret
    }
}
```

Listing 5.8: Two interface definitions and a class implementing them in CIL.

In the CIL code, the implemented interface methods are flagged as both **virtual** and **final**. The methods are virtual because they implement the interface. The **final** attribute means that the method cannot be overridden by any further derivations of the class. This is comparable to the **sealed** method modifier in C# for overridden abstract methods.

5.3 Interacting with objects

Now that we know how to define custom types, it is time to actually create instances and interact with them.

5.3.1 Creating objects

Creating an instance of a specific type is a relatively easy process. The only thing that is required is invoking one of the constructors that is declared in the class. This is done using the **newobj** opcode as found in listing 5.10.

```
1 public static void MyMethod()
2 {
3         MyClass myObject = new MyClass();
4 }
```

Listing 5.9: Creating objects in C#.

```
1 .method public static void MyMethod()
2 {
3     .locals init ( [0] class MyClass )
4
5     newobj instance void MyClass::.ctor()
6     stloc.0
7     ret
8 }
```

Listing 5.10: Creating objects in CIL.

The workings of the **newobj** are very similar to a normal **call**. If the constructor that is invoked requires any parameters to be specified, then these should be pushed in the same order as they appear in the definition, before the **newobj** is evaluated.

5.3.2 Calling instance members

So far we have been calling methods using the **call** opcode. The syntax of CIL requires you to fully specify which method is to be called. This includes both the method name and the declaring type. But what happens when a virtual or even an abstract method is to be called on an object whose specific type is unknown? This problem can be illustrated by the CIL snippet found in listing 5.11:

```
1  .class public MyBaseClass
2      extends [mscorlib] System.Object
3  {
4      // (...)
5
6      .method public virtual void MyVirtualMethod()
7      {
8          ldstr "This is the base implementation."
9          call void [mscorlib]System.Console::WriteLine(string)
10         ret
11     }
12 }
13
14 .class public MyDerivedClass
15     extends MyBaseClass
16 {
```

```
17      // (...)
18
19      .method public virtual void MyVirtualMethod()
20      {
21          ldstr "This is the new implementation."
22          call void [mscorlib]System.Console::WriteLine(string)
23          ret
24      }
25  }
```

Listing 5.11: A CIL definition of a base class and a deriving class. The base class defines a virtual method and the deriving class overrides the method.

In the code above we can see two classes, one deriving from the other. The virtual method *MyVirtualMethod* is defined and it has two implementations. Since object oriented programming allows types to be derived from another, when we try to call the virtual method on an instance of *MyBaseClass*, there is the possibility that this variable actually contains an instance of *MyDerivedClass*. Therefore, calling the virtual method like in the *Main* method below might actually result in the wrong implementation to be called.

```
1   .method public static void Main()
2   {
3       .entrypoint
4       .locals init ( [0] class MyBaseClass )
5
6       // Create a new MyDerivedClass object and store it in variable
            0.
7       newobj instance void MyDerivedClass::.ctor()
8       stloc.0
9
10      // Call the virtual method.
11      ldloc.0
12      call instance void MyBaseClass::MyVirtualMethod()
13      ret
14  }
```

Listing 5.12: Calling a virtual method. The code does not take polymorphism into account and will therefore invoke the original implementation of the method.

The code will now always invoke the one defined in the base class, regardless of what the actual type of the object is. In order to take polymorphism into account, CIL defines the **callvirt** opcode. This will solve the issue. Listing 5.13 shows a correct implementation of the main method.

```
1   .method public static void Main()
2   {
3       .entrypoint
4       .locals init ( [0] class MyBaseClass )
5
6       // Create a new MyDerivedClass object and store it in variable
            0.
7       newobj instance void MyDerivedClass::.ctor()
8       stloc.0
9
10      // Call the virtual method.
11      ldloc.0
```

```
12      callvirt instance void MyBaseClass::MyVirtualMethod()
13      ret
14  }
```

Listing 5.13: Calling a virtual method of the derived class. The code takes polymorphism into account and will therefore eventually call the implementation of the derived class.

5.4 Generics

Generic types are types that are more of a blueprint of a type, rather than a type on themselves. They provide re-usability of your code, whilst still preserving type safety. They are used a lot in collection types and methods that operate on them. A classic example of a generic type is the *List<T>* class in the *System.Collections.Generic* namespace of *mscorlib*. In this example, the List class is the generic type, and T is a generic type parameter.

5.4.1 Defining generic types

In IL, a generic type is, like any other type, a **.class** definition, but its name is followed up by an array of generic type parameters.

```
1  .class public MyGenericClass`2<T1, T2>
2      extends [mscorlib] System.Object
3  {
4      // ...
5  }
```

Listing 5.14: A generic class with two generic parameters T1 and T2.

In the example given by listing 5.14, T1 and T2 are the type parameters that are required to instantiate or access an object of the generic type. By convention, the name of each generic type is suffixed with the number of generic type parameters. However, this is not required, and types that do not follow this standard will still function properly.
After defining the generic type parameters, they can be used within definitions and code of members inside the class. Listing 5.15 shows an example.

```
1  .class public MyGenericClass`2<T1, T2>
2      extends [mscorlib] System.Object
3  {
4      // ...
5
6      .method public hidebysig !0 MyMethod(!1 parameter)
7      {
8          .locals init ( [0] !0 )
9
10         ldloca.s 0
11         initobj !0
12
13         ldloc.0
14         ret
15     }
16 }
```

Listing 5.15: A method that uses the generic arguments of the class.

Notice that the return type of MyMethod is **!0**. This refers to the first generic type parameter T1. It also has a parameter of type **!1**, which refers to the second type parameter T2. In general, a generic

type parameter is referenced by a single exclamation mark followed by an index. This is different from C# , since in C# the type parameters are referred to by their name instead.

5.4.2 Defining generic methods

Generic parameters can also be defined by methods. The syntax is very similar, except that now when referring to a type parameter that is defined by a method, we use two exclamation marks instead of one. Therefore, in the example given by listing 5.16, the return type **!!0** refers to the type parameter T3 defined in the method declaration, but the parameter type **!0** refers to the type parameter T1 defined in the class declaration:

```
1  .class public MyGenericClass`2<T1, T2>
2      extends [mscorlib] System.Object
3  {
4      // ...
5
6      .method public hidebysig !!0 MyMethod<T3>(!0 parameter)
7      {
8          .locals init ( [0] !!0 )
9
10         ldloca.s 0
11         initobj !!0
12
13         ldloc.0
14         ret
15     }
16 }
```

Listing 5.16: A method defining a type parameter T3.

5.4.3 Using generic types and methods

Listing 5.17 provides an example of how the MyGenericClass type could be used.

```
1  .method public static void main()
2  {
3      .entrypoint
4
5      // Declare generic variable.
6      .locals init ( [0] class MyGenericClass`2<string, int32> )
7
8      // Create generic object.
9      newobj instance void class MyGenericClass`2<string, int32>::.
           ctor()
10     stloc.0
11
12     // Call generic method.
13     ldloc.0
14     ldc.i4.1
15     callvirt instance !!0 class MyGenericClass`2<string, int32>::
           MyMethod<bool>(!0)
16
17     call void [mscorlib] System.Console::WriteLine(bool)
18     ret
19 }
```

Listing 5.17: Using generic types and methods in CIL.

Keep in mind that while calling the generic method, we still use **!!0** to denote return type of MyMethod instead of the substitute **bool** and **!0** to refer to the first type parameter of the declaring type MyGenericClass.

Chapter 6

Uncommon Instructions

There are some instructions in CIL which are only used in very special cases. Even though they do not appear that often in assemblies produced by the C# compiler, it is still important to acknowledge their existence and look at a few examples.

6.1 Argument iterators

A lot of native libraries define functions with a variable amount of parameters. While C# provides developers with the *params* keyword, it is incompatible with that the native libraries expose. A classic example would be the definition of the *sprintf* function, which formats a string based on a given set of arguments. The sprintf can be defined in C# using the P/Invoke signature as seen in listing 6.1, using the fairly uncommon **__arglist** keyword.

```
1 [DllImport("msvcrt.dll",
2     CharSet = CharSet.Ansi,
3     CallingConvention = CallingConvention.Cdecl)]
4 private static extern int sprintf(
5     StringBuilder buffer,
6     string format,
7     __arglist);
```

Listing 6.1: The __arglist keyword in C#.

If it is desired to actually implement such a procedure, one could use the **__arglist** keyword again to obtain the handle to the arguments. Combining this with an instance of the **ArgIterator** structure we can obtain the values of the arguments using managed code only [6].

```
1 private static int sprintf(StringBuilder buffer, string format,
      __arglist)
2 {
3     var iterator = new ArgIterator(__arglist);
4     // ...
5 }
```

Listing 6.2: Obtaining the __arglist in C#.

To support this, a dedicated opcode **arglist** is defined to enable exactly this behaviour, as shown in listings 6.3.

```
.method public hidebysig static vararg void sprintf (
    class [mscorlib] System.Text.StringBuilder buffer,
    string format) cil managed
{
    .locals init (
        [0] valuetype [mscorlib]System.ArgIterator iterator
    )

    ldloca.s iterator
    arglist
    call instance void [mscorlib]System.ArgIterator::.ctor(valuetype
        [mscorlib]System.RuntimeArgumentHandle)

    // ...
}
```

Listing 6.3: An example of a usage of the arglist opcode in CIL.

Notice that the method itself does not encode the argument list as a parameter like the C# implementation does. Rather, a special modifier **vararg** is added to the method definition instead to indicate that the method has a variable amount of arguments specified.

6.2 Typed references opcodes

Typed references are objects containing a managed pointer to some place in memory, and the type of the value that is stored at said location. They are used in for example argument iterators described in section 6.1 to determine the type of a specific argument in the argument list [7].
C# defines three fairly undocumented keywords to support the workings of TypedReferences, as can be seen in listing 6.4.

```
public static void TypedReferencesTest(int x)
{
    TypedReference typedReference = __makeref(x);
    Type type = __reftype(typedReference);
    int value = __refvalue(typedReference, int);
}
```

Listing 6.4: An example of typed references using C#.

The three corresponding IL opcodes are described in table 6.1.

Instruction	Description
mkrefany	The mkrefany opcode creates a typed reference from a given object. It roughly is the equivalent of the __makeref keyword in C# , except that it requires an address to the object instead of the object itself.
refanytype	The refanytype opcode retrieves the type token embedded in a typed reference. It is essentially the equivalent of the __reftype keyword in C# , except that it pushes the handle of the type, and not the type itself.
refanyval	The refanyval opcode retrieves the address (type &) embedded in a typed reference. It is roughly the equivalent to the __refvalue keyword in C# , except that it pushes the address to the value instead of the actual value

Table 6.1: TypedReference opcodes

This means that the C# example can be implemented using the code found in listing 6.5.

```
1  .method public hidebysig static void TypedReferences (int32 x) cil
       managed
2  {
3      .locals init (
4          [0] valuetype [mscorlib]System.TypedReference,
5          [1] class [mscorlib]System.Type,
6          [2] int32
7      )
8
9      // Create a typed reference to the value x (__makeref).
10     ldarga.s x
11     mkrefany [mscorlib]System.Int32
12     stloc.0
13
14     // Obtain the type of the typed reference (__reftype).
15     ldloc.0
16     refanytype
17     call class [mscorlib]System.Type [mscorlib]System.Type::
           GetTypeFromHandle(valuetype [mscorlib]System.
           RuntimeTypeHandle)
18     stloc.1
19
20     // Obtain the value of the typed reference (_refvalue).
21     ldloc.0
22     refanyval [mscorlib]System.Int32
23     ldind.i4
24     stloc.2
25
26     ret
27 }
```

Listing 6.5: Using typed references in CIL.

6.3 Memory related opcodes

Sometimes it is desired to initialize or copy a large chunk of memory from one place to another. While there exists native methods in the standard windows libraries that do these kinds of operations really efficiently (think of the functions memset and memcpy), PInvoke might not be the best solution here. The procedure to call might be stored in another library, go with a slightly different name, or simply not exist at all on other platforms. Since PInvoke requires the programmer to give the name of the function and the path to the library that the function is declared in, it would make the application platform dependent. Fortunately, CIL defines dedicated opcodes for these kinds of operations that take away this problem, which are listed in table 6.2.

Instruction	**Description**
cpobj	The cpobj opcode copies the value type at the address of an object to the address of the destination object [8].
cpblk	The cpblk opcode copies a specified number of bytes from a source address to a destination address. It is similar to the *memcpy* function in C [9].
initblk	The initblk opcode initializes a block of memory at a specific address to a given size and initial value. It is similar to the *memset* function in C [10].

Table 6.2: Bulk memory opcodes

An example of how **initblk** could be used can be found in listing 6.6:

```
1  .method public static void main() cil managed
2  {
3      .entrypoint
4      .locals init (
5          [0] uint8*
6      )
7
8      // Allocate 10 bytes on the stack.
9      ldc.i4.s 10
10     localloc
11     stloc.0
12
13     // Set all 10 bytes to the value 1.
14     ldloc.0
15     ldc.i4.1
16     ldc.i4.s 10
17     initblk
18
19     ret
20 }
```

Listing 6.6: Allocating blocks of memory using the initblk instruction.

The opcodes **cpobj** and **cpblk** work very similar. Given two pointer variables, one could transfer the contents of the first variable to the second one like in listing 6.7.

```
1
2  .method public static void main() cil managed
3  {
4      .entrypoint
5      .locals init (
6          [0] uint8* ,
7          [1] uint8*
8      )
9      // ...
10     ldloc.1      // Destination
11     ldloc.0      // Source
12     ldc.i4.s 10 // Size
13     cpblk
14     // ...
15     ret
16 }
```

Listing 6.7: Copying blocks of memory using the cpblk instruction.

6.4 Prefix opcodes

Some instructions in CIL can be prefixed by special opcodes

Instruction	Description
unaligned.	The unaligned. prefix opcode specifies that the address on the stack might not be aligned to the natural size of the immediately following ldind, stind, ldfld, stfld, ldobj, stobj, initblk, or cpblk instruction. Code generators that do not restrict their output to a 32-bit word size shall use unaligned.
volatile.	The volatile. opcode specifies that the address on top of the stack might be volatile, and the results of reading that location cannot be cached or that multiple stores to that location cannot be suppressed.
tail.	Performs a postfixed method call instruction such that the current methods stack frame is removed before the actual call instruction is executed. C# does not support tail recursion, and therefore this opcode is never emitted. It can appear in applications compiled by for example the F# compiler, which does support this feature.

To show off how tail call can be used, let us consider the example given by listing 6.8.

```
1  public static void Main()
2  {
3      MyRecursiveMethod(0);
4  }
5
6  public static void MyRecursiveMethod(int x)
7  {
8      Console.WriteLine(x);
9      MyRecursiveMethod(x + 1);
10 }
```

Listing 6.8: A situation that could benefit from tail calls in C#.

If this example snippet would be compiled using the standard C# compiler and be executed, a stack overflow exception would eventually be thrown, since the application enters an infinite recursion. However, if we try to implement the C# code in CIL like the snippet in listing 6.9 using tail call, then upon each iteration, the current stack frame is removed before it calls the current method again, stopping the stack from growing, preventing a stack overflow to happen.

```
1  .method public static void main() cil managed
2  {
3      .entrypoint
4
5      ldc.i4.0
6      call void MyRecursiveMethod(int32)
7      ret
8  }
9
10 .method public static void MyRecursiveMethod(int32 x) cil managed
11 {
12     ldarg.0
13     call void [mscorlib] System.Console::WriteLine(int32)
14
15     ldarg.0
16     ldc.i4.1
```

```
17      add
18
19      tail.
20      call void MyMethod(int32)
21      ret
22  }
```

Listing 6.9: An example of tail calling a method in CIL.

Notice that **tail. call** is always used at the end of the method. It is therefore always required to suffix the call with a **ret** instruction, or else the runtime will flag the method body as invalid [11].

6.5 Invocation opcodes

The CIL defines two operations that affect the control flow of the program which are virtually never outputted by the C# compiler itself. These are shown in table 6.4.

Instruction	Description
calli	Calls a method whose function pointer is pushed onto the stack.
jmp	Exits current method and jumps to the method specified in the operand.

Table 6.4: Unsafe invocation opcodes

The **calli** is often accompanied with either the **ldftn** or the **ldvirtftn** instruction, which push the address of the function to be called onto the stack. Listing 6.10 example shows how **calli** could be used.

```
1  .method public static void MyMethod(int32)
2  {
3      ldarg.0
4      call void [mscorlib]System.Console::WriteLine(int32)
5
6      ret
7  }
8
9  .method public static void Main(string[] args)
10 {
11     .entrypoint
12
13     ldc.i4.1
14     ldftn void MyMethod(int32)
15     calli void (int32)
16
17     ldstr "Back to Main"
18     call void [mscorlib]System.Console::WriteLine(string)
19
20     ret
21 }
```

Listing 6.10: Using the calli instruction in CIL to indirectly call methods.

Notice the arguments of the function are pushed *before* the address of the function (line 13 and 14). This may seem counter-intuitive at first since it deviates from calling instance methods as described in section 5.3.2, where the arguments are pushed *after* the instance object instead. Also it should be verified that the signature of the method specified in the instruction's operand matches the one from the function

that was referenced on the stack (line 15). Doing this incorrectly will result in the CLR throwing an *System.InvalidProgramException*, which eventually leads to a program crash.
The **jmp** opcode is similar to the **tail.call** operation, except that it requires the caller to have the same signature as the callee. An example is given in listing 6.11.

```
1  .method public static void MyMethod(int32 arg)
2  {
3      ldstr "MyMethod"
4      call void [mscorlib]System.Console::WriteLine(string)
5
6      ldarg.0
7      call void [mscorlib]System.Console::WriteLine(int32)
8
9      jmp void MyOtherMethod(int32)
10 }
11
12 .method public static void MyOtherMethod(int32 arg)
13 {
14     ldstr "MyOtherMethod"
15     call void [mscorlib]System.Console::WriteLine(string)
16
17     ldarg.0
18     call void [mscorlib]System.Console::WriteLine(int32)
19
20     ret
21 }
```

Listing 6.11: Using the jmp instruction to transfer control to another method.

Notice that on line 9 the jump to the other method is performed, and that the stack has been emptied before this instruction is being called. This is an additional requirement for the **jmp** opcode, as it reuses the same stack frame the caller was using. A result of this is that the arguments passed to the caller will be reused by the callee as well. Calling **MyMethod** with a specific argument value will therefore print out that same argument value twice.

Chapter 7

Closing words

7.1 Summary

To summarize the paper we would like to address a couple of things. We covered everything from the basics to more advanced topics like object-oriented programming and uncommon opcodes. The goal of this writeup was to cover various aspects of CIL and to provide a compact and easy to understand reference for beginners and advanced programmers. The reader should now posses a solid understanding of CIL to start his journey into reverse engineering .NET software.

7.2 Acknowledgments

Special thanks to CodeBlue for his comments, proof-reading and review of the paper. This writeup would not have been possible without his help. We would also like to show our gratitude to all interested readers and the RTN community.

Chapter 8

References

[1] *Wikipedia: Common Intermediate Language*. URL: https://en.wikipedia.org/wiki/Common_Intermediate_Language (visited on 09/11/2017).

[2] Vijay Mukhi Akash Saraf and Sonal Mukhi. *C# TO IL*. BPB Publications, 2001.

[3] *CIL Instruction Set*. URL: https://www.cs.helsinki.fi/u/vihavain/k14/code_generation/project/CIL_Instruction_Set_MS_Partition_III.pdf (visited on 04/16/2018).

[4] *Wikipedia: PInvoke*. URL: https://en.wikipedia.org/wiki/Platform_Invocation_Services (visited on 09/11/2017).

[5] *DllImportAttribute Class*. URL: https://msdn.microsoft.com/en-us/library/system.runtime.interopservices.dllimportattribute(v=vs.110).aspx (visited on 01/06/2018).

[6] *ArgIterator Structure*. URL: https://msdn.microsoft.com/en-us/library/system.argiterator(v=vs.110).aspx (visited on 01/06/2018).

[7] *TypedReference Structure*. URL: https://msdn.microsoft.com/en-us/library/system.typedreference(v=vs.110).aspx (visited on 01/06/2018).

[8] *Cpobj Field*. URL: https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.cpobj(v=vs.110).aspx (visited on 01/06/2018).

[9] *Cpblk Field*. URL: https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.cpblk(v=vs.110).aspx (visited on 01/06/2018).

[10] *Initblk Field*. URL: https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.initblk(v=vs.110).aspx (visited on 01/06/2018).

[11] *Standard ECMA-335*. URL: https://www.ecma-international.org/publications/standards/Ecma-335.htm (visited on 09/11/2017).