P/Invoke and CIL

Mingsheng Hong
CS 215, Spring 2008

P/Invoke

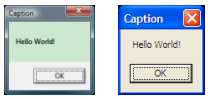
- Allows managed code to call unmanaged functions in COM objects, C/C++ DLLs, etc.
  - E.g. access to Win32 API
- To declare unmanaged functions
  - Use `DllImport` attribute and `static extern`

```
[DllImport("kernel32.dll")]
static extern int GetProcessHeap();
```

P/Invoke Example

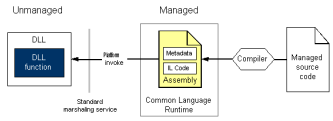
```
using System.Runtime.InteropServices;
namespace HelloWorld {
  class MyClass {
    [DllImport("user32.dll", CharSet=CharSet.Ansi)]
    static extern int MessageBox(int hwnd,
            string msg, string caption, int t);

    public static void Main() {
      MessageBox(0, "Hello World!", "Caption", 0);
    }
  }
}
```



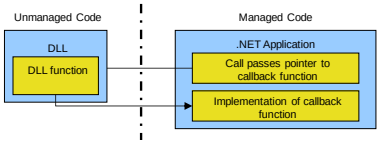
Steps in P/Invoke

- Locates implementing DLL
- Loads DLL into memory
- Finds function address
- Pushes arguments on stack, marshalling data
- Transfers control to unmanaged code



P/Invoke Callbacks

- Unmanaged code can call back to managed code
  - Unmanaged parameter is function pointer
  - In managed code, must supply parameter as delegate
  - P/Invoke creates callback thunk
    - Passes address of thunk as callback parameter



Callback Example

```
public class SampleClass {
  delegate bool CallBack(int hwnd, int lParam);

  [DllImport("user32.dll")]
  static extern int EnumWindows(CallBack x, int lParam);

  // report the window handle
  public bool Report(int hwnd, int lParam) {
    Console.Write("Window handle is " + hwnd);
    return true;
  }

  public static void Main() {
    CallBack myCallBack = new CallBack(Report);
    EnumWindows(myCallBack, 0);
  }
}
```

C++/CLI

- Write managed C++ code
  - compile with `/clr`
  - generates CIL from C++
  - new key words
    - `__gc, __box, __typeof, __interface, __property`
    - `#pragma managed/unmanaged`
- Very useful for native access to C++ libraries
  - build a "managed wrapper"

CIL

- Recall: two stage compilation
  - C# compiler: C# code → CIL code
  - Just-in-time (JIT) compiler: CIL → native code
- Common Intermediate Language
  - very close to C# (and other OO languages)
  - define classes, structs, inheritance, methods
  - assembly-like statements

CIL

- Stack language
  - instead of registers, everything is from stack
  - main operations take operands from stack
  - e.g.

```
int i = 137;
int j = 1;
int k = i + j;
```

Hello World Example

```
.assembly extern mscorlib {} //automatically added
.assembly hello {}
.class Program {
   .method static public void Main() cil managed {
        .entrypoint //designates this method as the entry pt
        .locals init (string name) //create a local var
        ldstr "World"  //load the string onto eval stack
        stloc.0 //store the string into the first local var
        ldstr "Hello, {0}!"
        ldloc name //load local var onto eval stack
        call void [mscorlib] System.Console::WriteLine(
  string, object) //call method with stack items as params
        ret
   }
}
```

Compile with `ilasm /exe /debug hello.il`

CIL Directives

- `.assembly`
- `.class`
  - define any type
  - `extends:` extend some other type
    - if extend `System.ValueType`, then value type
- `.method, .field, .property, .event`
- `.locals:` names and types for local vars
- `.entrypoint`
- `.maxstack`

Load/Store Operations

- `ldloc`/`stloc`
  - pushes contents of local var (or index) onto stack
  - pops and stores in local var (or index)
- `ldc`
  - `ldc.i4 50000`
  - `ldc.i4.1`
  - `ldc.i4.m1`
- `ldnull`
- `ldfld` / `stfld`
  - `ldsfld int32 A::fielda`

Load/Store Example

```
.locals init ([1] int32 a, [0] int32 b)
ldc.i4.5
stloc.0 ldc.i4 10
stloc.1
ldloc a
call void [mscorlib]
  System.Console::WriteLine(int32)
ldloc b
call void [mscorlib]
  System.Console::WriteLine(int32)
```

CIL Operations

- Integer operations
  - `add, mul, sub, div, rem, neg`
- Boxing
  - removes the value type from stack
  - creates an object on the managed heap that boxes the value type
  - places a reference to the newly created object back on the evaluation stack
  - e.g. `ldc.i4.3`
    `box int32`
- Data type conversions: `conv.*`

CIL Operations

- Construct objects
  - `newobj instance void A::.ctor()`
- Invoke functions
  - `call void [mscorlib]`
    `System.Console::WriteLine(string, object)`
  - `call instance int32 A::F()`
  - `callvirt instance int32 A::G()`

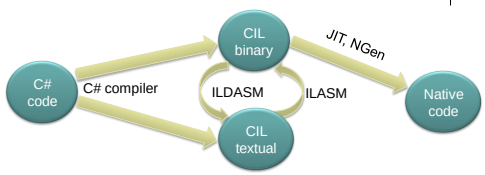
Control Flow Operations

- `ceq/cgt/clt`
  - pop top two elements of stack
  - check =, >, <
  - push true or false onto stack
- `br/beq/bne/bgt/blt/brfalse/brtrue`
  - do the comparison and jump
  - use to implement structured control flow

Control Flow Example

```
            ldc.i4.3
            ldc.i4.1
            cgt
            brtrue greater
            ldstr "{0} is less than or equal {1}"
            br end
greater:    ldstr "{0} is greater than {1}"
end:        ldc.i4.3
            box int32
            ldc.i4.1
            box int32
            call void [mscorlib]
  System.Console::WriteLine(string, object, object)
```

CIL Tools



- Roundtripping
  - `ildasm /out=new_program.il program.exe`
  - Edit the CIL code in `new_program.il`
  - `ilasm newadd.il`