Week 1: History and Execution mechanics

Srikar Mylavarapu

Goals for Today

- understand why C# and the CLR exist/the problems that led to the solutions described
- How a C# application works (execution model, ch1)
- Play around with a few Visual Studio tools

Brief History of Computing --> Why the CLR and C# exist

Discovery --> Usage is often a slow process

- Something gets discovered (~1800-1820 Lithium discovered)
  - 1940s as a grease agent during WWII
  - takes until ~1950 to get used for mania
  - 1970 Bipolar med
- Number Theory (specifically primes) is almost as old as math
  - first major use in cryptography is in the last 100 years

Boolean Algebra

- 1850 ish
- venn diagrams
- a whole category of math based on 0s and 1s
- If we get a system that can compute 0's and 1s efficiently, all of Boolean Algebra becomes impossibly useful

Bit of a time skip

Computers are born in 1948

```
- 1956 first use of transistors at MIT
- 1975 Altair BASIC -- Microsoft's first product
```

At some level, a computer is simply a that can read instructions written in a specfic way, very efficiently

What does this mean to a computer?

- 01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100

- (first number, right to left)
- 0* 2 ^ 0 + 0 * 2 ^ 1 + 0 * 2 ^ 2 + 1 * 2 ^ 3 + 0 * 2 ^ 4 + 1 * 2 ^ 5 + 1 * 2 ^ 6 + 0 * 2 ^ 7

  = 0 + 0 + 8+ 0 + 32+ 64 = 104 --> ascii value of 'H'
- If you've never seen binary, some fun problems are 11 + 11; 11 * 11
- There's more to it, but at the end of the day (i.e. operators, call stacks etc.), this is what the computer/processor understands

Pretty common problem in life/translation in a foreign country

- You're trying to have one person who is fluent in one language communicate with another person who fluent in another
- What are your options?

Options

- Can have one try to learn the other language
- Create a setup where there's something in the middle

Why don't we write everything in Binary?/Speedy History of languages

Assembly

- Binary is functionally meaningless to people
- Assembly added some syntax/meaning to the code before it was read by the computer
  - It is called "assembly" because it is assembled into machine code (see code sample)
  - Still not really readable, but more workable than machine code

Languages from a higher view

- Human Language (let's pick English) <-7-6-5-4-3-2-1-> Computer Language (Binary)
    - arbitrary numbers
        - anything low is easy for the computer to read/faster for computer
        - anything high is easy for a human to write/understand, slower for a computer to read/execute

C

- Create a language based off of assembly that humans can write, have it compile into something that a program on the computer can execute
- Essentially give full control of the computer to the User
- have functions

C++

- superset of C with classes
- what's the potenial problem with 'perfectlyFine.cpp'
- "I spent a large part of my life fixing these kinds of problems" -Michael McCann

What if there's more than one type of OS/hardware etc.

- We write a program, how do we ensure it runs on Windows, Mac, Linux etc?
- solution is still the same translation problem, just scaled up a level

Java

- Created the Java Runtime Environment
- Basically if a program was written in Java and said program ran on the JRE, it could be run the same way wherever the JRE could run
- "write once, run anywhere"
- Solved the Memory Management problem w/ garbage collector
- .java compiles into bytecode and executes on the JRE

C#

- Microsoft's answer to Java
- before open source era
- .cs -C#complier-> intermediate language (IL) -CLR> result
- .cs compiles into IL and executes on the CLR (common language runtime)
- also has a garbage collector

Runtime execution, high level overview (based off of pg 12)

Using the "Hello World" application, lets assume we have the following

1. The file (helloworld.exe)
2. Console
3. Memory
4. JIT (Just in time) Compiler
5. Native Code

Steps/what happens when you hit run in VS

1. We have a main method (hidden in this file [when creating a sln, do not allow "top level statements"])
2. The main method gets called/instantiates a Console with an internal structure for all methods
3. The main method calls the first function, Console.WriteLine
   a. Method is not already in the Console's dynamic memory
   b. uses the JIT compiler to access the MsCorEE.dll to get Console.WriteLine and compile the native code for the function
   c. execute and store that function in dynamic memory (Richter has more indept steps about the JIT compiler but probably not necessary)
4. Now we get Console.WriteLine again
5. The function is already stored
   a. no need to go back to the .dll to recreate it
6. execute it from native code /endcontinueprogram

FCL Framework Class Library

- a bunch of frameworks that expose some functionality
- My current understanding is that when you create a new VS solution, each option is basically one of these.

Definitions

- Namespace: I just think of it as the sub object /definition you're trying to access within a given FCL
- Types: Types are basically the building blocks of C# --> second section is dedicated to understanding them.

Common Language Runtime grab bag

- Any language that can be converted into IL can run on it.
  - IronPython, Iron Ruby and a whole bunch of other stuff exist to be able to run "non C# stuff" on the CLR
    - Would like to add a easy way to show this later/ wasn't able to get it so far
- Managed vs Unmanaged Code
  - simply put, if an application runs on the CLR, it's managed (logic I'm using is "it is running on something w/ a garbage collector therefore it is memory managed")
  - C++ is the special

Modules and Assemblies

For our purposes, a module has a PE32 header, a CLR header, metadata, and the acutal IL code. (these will mean more as we go deeper)
For more detail, see chart on page 5
A module is not the smallest independent unit.
An Assembly is the smallest unit of reuse; contains one or more modules.

Module(s) + Resource(s) --> Compiler --> Assembly (Manifest saying whats in here) + [Module]

Activities

- run all of the "historical" files and see the compiled languages
- use the Ildecompiler in VS code to see IL code/what it looks like
- Run through the call stack of the IL code