Lecture 1: Course Overview

Computer Systems Organization (Spring 2022)
CSCI-UA 201, Section 1

Instructor: Prof. Douglas Moody

Slides adapted from
Randal E. Bryant and David R. O'Hallaron (CMU)
Mohamed Zahran (NYU) Joanna Klukowska
(NYU)

Well, not that kind of organization

Abstraction is good, but ...

- Most CS and CE courses emphasize abstraction
    - Abstract data types
    - Asymptotic analysis (like the Big-O notation)
- These abstractions have limits
    - Especially in the presence of bugs
    - Need to understand details of underlying implementations
- Useful outcomes from taking CS201
    - Become more effective programmers
        - Able to find and eliminate bugs efficiently
        - Able to understand and tune for program performance
- Prepare for later “systems” classes in CS
    - Compilers,
    - Operating Systems,
    - Networks,
    - Computer Architecture,
    - Embedded Systems,
    - etc.

This class adds to your CV:

- C programming
- Unix / Linux familiarity
- X86-64 assembly
- Low level debugging
- Reverse engineering
- Understanding of computer systems
- ...

NYU CLASSES:

Data Structures - Organization of Data with software

Coding - Python , Java – Use high level coding languages

CS Organization - Architecture for executing instructions

Programmers' Reality #1:

ints are not integers, floats/doubles are not real numbers

Is x^2 >= 0 ?

- in a math class: YES (when x is an integer or a real number)
- on a computer: IT DEPENDS on x
  - for example: when x is an int

    30,000 * 30,000 = 900,000,000

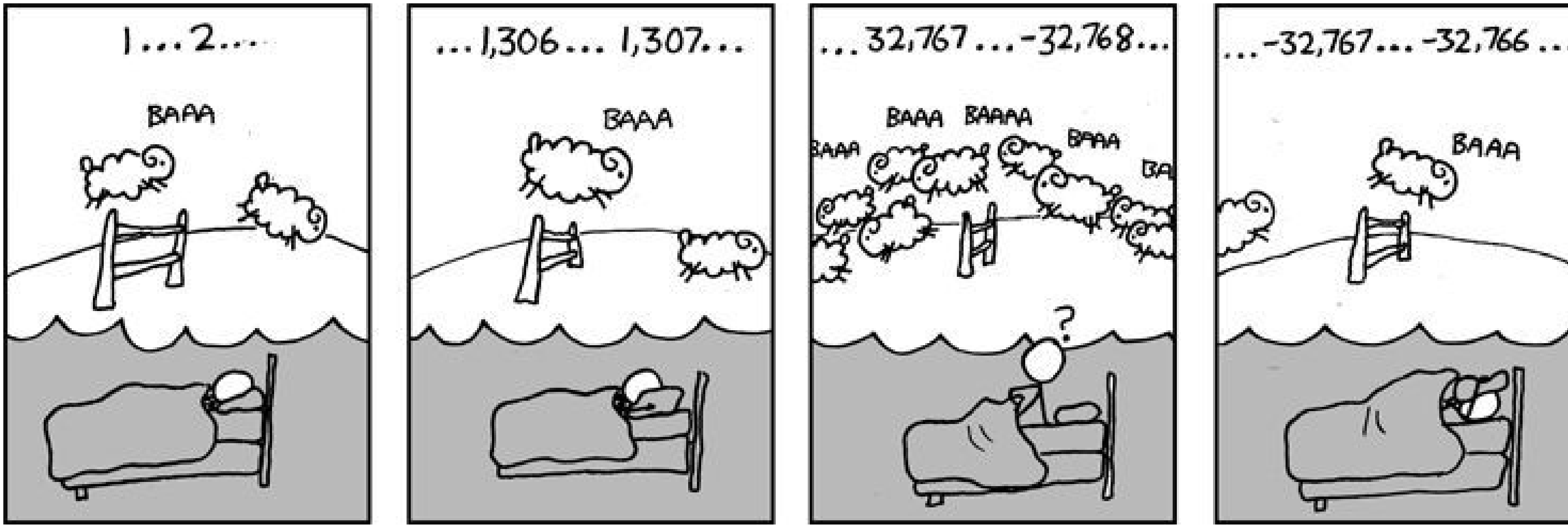
    50,000 * 50,000 = ???

Is (x+y) + z = x + (y+z)?

- in math class: YES (when x is an integer or a real number)
- on a computer: IT DEPENDS on x, y, z
  - for example: when x, y, z are of type

    float (1e20 + -1e20) + 3.14 = 3.14

    1e20 + (-1e20 + 3.14) = ???



32,767 + 1 = -32,766

Programmers' Reality #2:

you need to know assembly

- Chances are, you'll never write programs in assembly
  - Compilers are much better & more patient than you are

- But: understanding assembly is key to machine-level execution model
  - Debugging
  - Performance tuning
  - Writing system software (e.g. compilers , OS)
  - Reverse engineering software
  - Creating / fighting malware

Programmers' Reality #3:

memory matters

- Memory is not unbounded
    - It must be allocated and managed
    - All running applications and data have to be in memory, all applications can address lots of memory. Where does it all go?

- Memory referencing bugs especially wicked
    - Effects are distant in both time and space (i.e., may not happen until much later or in a different part of the program or data structure)

- Memory performance is not uniform
    - Cache and virtual memory effects can greatly affect program's performance
    - Adapting program to characteristics of memory system can lead to major speed improvements

Example: Array access

```
#include <stdio.h>

int main ( ) {
     int d = 3;
     printf("d = %d\n", d);
     int a[1];
     int i;
     for (i = 0; i < 5; i ++ ) {
          a[i] = 214748364;
     }
     printf("d = %d\n", d);
}
```

OUTPUT (one possibility):

d = 3
d = 214748364

Memory referencing errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- How can I deal with this?
  - Program in Java, Ruby, Python, ML, …
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

Programmers' Reality #4:
there is more to performance than asymptotic analysis

- (But do not tell your teachers in Data Structures and Algorithms courses that I said that!)

- Constant factors matter too!
- Optimization has to happen at multiple levels: algorithm, data representation, details of implementation.
- Optimizing implementation requires understanding of the underlying system.
    - How programs are compiled and executed
    - How to measure program's performance and identify bottlenecks
    - How to improve performance without destroying code modularity and generality

Example:
What is Big-O notation of these two programs?

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

About 7 times faster on
Intel® Core™ i7-3930K CPU @ 3.20GHz × 12 .

WHY?

Programmers' Reality #5:
computers do more than execute programs

- They need to get data in and out
  - I/O system critical to program reliability and performance

- They communicate with each other over networks
  - Many system-level issues arise in presence of network

MOBILE DEVICE HIERARCHY