Scribe Notes for September 10, 2013
Contributors: Daniel Windham, Greg Foster, **???**

Lecture 3: Data Representation

Key dichotomy: C vs. x86

- Let's consider two entities:
  - The C abstract machine: defines the meaning of a C program
  - The processor (Intel x86): defines the meaning of an executable (a binary program)
    - Historical side note: Intel's first processor was named 8086. Subsequent processors were backwards compatible and therefore named 80286, 80386, and 80486. It is therefore common to refer to them as x86 processors.

Compilation

- It is the job of the compiler to translate C into x86
- The compiler wants to optimize x86 code to run as fast as possible on the processor
- The compiler has to represent each kind of data type (e.g. int) in memory
- While the compiler could spread information across many locations, clumping information is better for the processor
- Key questions to explore:
  - WHERE does the compiler put information?
  - HOW do the compiler & processor represent information as bytes?

Warning

- If your C program executes undefined behavior, all rules are off for the entire program. There is no guarantee that any of this behavior will hold. If you access the 5^{th} element of a length-3 array, for instance, demons might flight out of your nose. No promises they won't.

Memory addresses

- On a 32-bit machine, there are 2^32 address spaces. These are labeled 0 to 2^32-1.
- Addresses are therefore 4 bytes long. They are written in hexadecimal notation, noted with the prefix 0x, e.g.:
  - first address is 0x00000000
  - last address is 0xffffffff
  - midpoint is 0x80000000

Storing objects in memory

- One job of the compiler is to assign memory to objects
  - C requires distinct live variables to refer to distinct objects. Assigning one variable does not affect any other.
  - In x86, distinct live variables occupy different regions of memory.

    - We can check this by printing the addresses of objects, e.g.:
      ```
      int local = 0;
      char local_arr[5];
      int* dynamic_arr = (int *)(malloc(sizeof(int)));
      printf("%p", &local);
      printf("%p", &local_arr);
      printf("%p", dynamic_arr);
      printf("%p", &dynamic_arr);
      ```
- Memory is partitioned into certain segments. From the compiler's point of view, in order from lowest to highest memory addresses:
    - **Text**: read only data and functions
    - **Data**: global, writeable addresses
    - **Heap**: dynamically allocated data
    - **Stack**: local variables of functions
    - **Kernel**: untouchable memory reserved for the operating system
- Some details:
    - Text is static and set at compile time.
    - The heap and stack both change as program runs. The heap grows up, the stack grows down.
    - Local variables are located in the stack
    - Dynamic allocations of memory contain extra meta-data to help catch bugs and track memory
    - Each time a program runs, stack and heap memory receive new random starting addresses. This is to defend against attacks. Within a given program, however, relative memory addresses are held constant from one execution to the next.
    - Memory addresses can be reused if all previous owners are dead. Liveness on the stack is controlled with malloc and free.
    - On the heap, the compiler allocates and frees memory