September 5

- Fang-Yi's Office Hours Friday 10am ✍ 12pm (and another TBD)
- Programming Language Issues (JAVA vs. C, Pointers vs. Arrays)
- Representations
- Instructions

Just enough C

For our purposes C is almost identical to JAVA except:

C has "functions", JAVA has "methods". function == method without "class".
A global method.

C has "pointers" explicitly. JAVA has them but hides them under the covers.

C pointers

```
int i;      // simple integer variable
int a[10];  // array of integers
int *p;     // pointer to integer(s)
```

***(expression)** is content of address computed by expression.

a[k] == *(a+k)

a is a constant of type "**int ***"

a[k] = a[k+1] EQUIV *(a+k) = *(a+k+1)

Legal uses of C Pointers

```
p = &i;     // & means address of
p = a;      // no need for & on a
p = &a[5];  // address of 6th element of a
*p          // value of location pointed by p
*p = 1;     // change value of that location
*(p+1) = 1; // change value of next location
p[1] = 1;   // exactly the same as above
p++;        // step pointer to the next element
```

So what happens when

```
p = &i;
```

What is value of p[0]? What is value of p[1]?

C Pointers vs. object size

Does “p++” really add 1 to the pointer?
- NO! It adds 4.
- Why 4?

```
char *q;
...
q++; // really does add 1
```


Clear123

```
void clear1(int array[], int size) {
  for(int i=0; i<size; i++)
      array[i] = 0;
}

void clear2(int *array, int size) {
  for(int *p = &array[0]; p < &array[size]; p++)
      *p = 0;
}

void clear3(int *array, int size) {
  int *arrayend = array + size;
  while(array < arrayend) *array++ = 0;
}
```

Pointer summary

- In the “C” world and in the “machine” world:
  - a pointer is just the address of an object in memory
  - size of pointer is fixed regardless of size of object
  - to get to the next object increment by the objects size in bytes
  - to get the the i^{th} object add i*sizeof(object)
- More details:
  - int R[5] ✍ R is int* constant address of 20 bytes
  - R[i] ✍ *(R+i)
  - int *p = &R[3] ✍ p = (R+3) (p points 12 bytes after R)

Representations

You need to know your powers of 2!

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^10	1k=1024
2^20	1M
2^30	1G

Pointer Size vs. Addressable Space

- Pointers ARE addresses
- Number of unique addresses for N bits is 2^N

- With addresses that are 32 bits long you can address 4G bytes
- With addresses that are 13 bits long you can address 8k bytes
  - that's 2k words

C versus ASM

```
Swap(int v[], int k) {
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

```
Swap:
        muli $t0, $a1, 4
        add $t0, $a0, $t0
        lw $t1, 0($t0)
        lw $t2, 4($t0)
        sw $t2, 0($t0)
        sw $t1, 4($t0)
        jr $ra
```

Form of the Instructions

- Opcode
- Register (usually result destination)
- Operand 1
- Operand 2

  e.g.

  add $t0, $a0, $t0

Naming Registers

This is all just software "convention"

- $a0 - $a3 arguments to functions
- $v0 - $v1 results from functions
- $ra return address
- $s0 - $s7 "saved" registers
- $t0 - $t9 "temporary" registers
- $sp stack pointer

What are the operands?

- Registers e.g. $a0
- With load and store this is logical enough
- But small constants are VERY common
- So, some instructions allow “immediate” operands. E.g. muli $t0, $a1, 4
- ***Where do we get big constants?***

C versus ASM

```
Swap(int v[], int k) {
  int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

```
Swap:
        muli $t0, $a1, 4
        add $t0, $a0, $t0
        lw $t1, 0($t0)
        lw $t2, 4($t0)
        sw $t2, 0($t0)
        sw $t1, 4($t0)
        jr $ra
```

Next: Performance

- Measure, Report, and Summarize
- Make intelligent choices
- See through the marketing hype
- Key to understanding underlying organizational motivation

Why is some hardware better than others for different programs?

What factors of system performance are hardware related?
(e.g., Do we need a new machine, or a new operating system?)

How does the machine's instruction set affect performance?