Under the Hood: Data Representations, Memory and Bit Operations

Computer Science 104
Lecture 3

Last time …..
Who can remind us what we covered last time?

• Representing positive and negative integer
  − Sign Magnitude
  − 1’s Complement
  − 2’s Complement
• 2’s Complement math
  − Addition, subtraction, negation
• Representing “real” numbers
  − Fixed Point
  − Rational
  − IEE Floating Point

Computer Memory
Where do we put these numbers?
• Registers [more on these later]

• Memory

Last time …..
Who can remind us what we covered last time?
• Register [more on these later]
  − In the processor
  − Compute directly on them
  − Few of them (~16 or 32)
• Memory
**Computer Memory**

Where do we put these numbers?
- Registers [more on these later]
  - In the processor
  - Compute directly on them
  - Few of them (~16 or 32)
- Memory [Our focus now]
  - External to processor
  - Load/store values to/from registers
  - Very large (multiple GB)

**Memory Organization**

Memory: billions of numbers...how to get the right one?
- We number our numbers!
- Each memory location has an address
- Processor asks to read or write specific address
  - Memory, please load address 0x123400
  - Memory, please write 0xFE into address 0x8765000
- Kind of like a giant array

**Memory Organization**

Memory: billions of numbers...how to get the right one?
- We number our numbers!
- Each memory location has an address
- Processor asks to read or write specific address
  - Memory, please load address 0x123400
  - Memory, please write 0xFE into address 0x8765000
- Kind of like a giant array

**Word of the Day: Endianess**

**Byte Order**

- **Big Endian:** byte 0 is 8 most significant bits IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
- **Little Endian:** byte 0 is 8 least significant bits Intel 80x86, DEC Vax, DEC Alpha

**Endianess**

Example: 12 as 32-bit int at address 0x1234:

<table>
<thead>
<tr>
<th>Address</th>
<th>Little Endian</th>
<th>Big Endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1237</td>
<td>0x00</td>
<td>0x0C</td>
</tr>
<tr>
<td>0x1236</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x1235</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>0x1234</td>
<td>0x0C</td>
<td>0x00</td>
</tr>
</tbody>
</table>

Programming impact: machines of different Endianess
- Network programs
  - htonl, htons, ntohl, ntohs
- Saved data files
Alignment

Alignment: require that objects fall on address that is multiple of their size.

32-bit integer
- Aligned if address % 4 = 0

64-bit integer?
- Aligned if ?

Results of un-aligned access? Depends...
- Nothing
- Accepted, but slow (very very slow?)
- Crash your program

A word about C

We will have a few examples in C
- Haven't learned details of C yet (Wednesday), but...

Syntax: mostly like Java
- But not OO: functions are not inside classes
- Has global variables (though we will avoid them generally)
- Lower-level: can work with specific memory addresses

Let's do a little Java...

Let's do a little Java...
Let's do a little Java...

```java
public class Example {
    public static void swap (int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
    public static void main (String[] args) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a = " + a + " b = " + b);
    }
}
```

What does this print? Why?

```
a = 42
b = 100
```

---

Let's do some different Java...

```java
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int a = 42;
        int b = 100;
        swap (a, b);
        System.out.println("a = " + a + " b = " + b);
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a = " + a.data + " b = " + b.data);
    }
}
```

What does this print? Why?

```
a = 42
b = 100
```
Let's do some different Java...

```java
class Ex2 {
    int data;
    void print() {
        System.out.println(data);
    }
}
```

What does this print? Why?

```
public static void swap(Ex2 x, Ex2 y) {
    int temp = x.data;
    x.data = y.data;
    y.data = temp;
}
```

What does this print? Why?

```
public class Ex2 {
    int data;
    public static void main(String[] args) {
        Ex2 a = new Ex2(42);
        Ex2 b = new Ex2(100);
        swap(a, b);
        System.out.println("a = " + a.data + " b = " + b.data);
        System.out.println("temp = " + temp);
    }
}
```

What does this print? Why?

```
public class Ex2 {
    int data;
    public static void main(String[] args) {
        Ex2 a = new Ex2(42);
        Ex2 b = new Example(100);
        swap(a, b);
        System.out.println("a = " + a.data + " b = " + b.data);
        System.out.println("temp = " + temp);
    }
}
```

What does this print? Why?
Let's do some different Java...

```java
public class Ex2 {
    int data;
    public Ex2 (int d) { data = d; }
    public static void swap (Ex2 x, Ex2 y) {
        int temp = x.data;
        x.data = y.data;
        y.data = temp;
    }
    public static void main (String[] args) {
        Example a = new Example (42);
        Example b = new Example (100);
        swap (a, b);
        System.out.println("a = " + a.data + 
                      " b = " + b.data);
    }
}
```

What does this print? Why?

References and Pointers

Java has references:
- Any variable of object type is a reference
- Point at objects (which are all in the heap)
  - Under the hood: is the memory address of the object
- Cannot explicitly manipulate them (e.g., add 4)

Some languages (C, assembly) have explicit pointers:
- Hold the memory address of something
- Can explicitly compute on them
- Can de-reference the pointer (*ptr) to get thing-pointed-to
- Can take the address-of (&x) to get something’s address
- Can do very unsafe things, shoot yourself in the foot

References and Pointers

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12340000C</td>
<td>42</td>
<td>x</td>
</tr>
<tr>
<td>0x12340000B</td>
<td>0x1234000C</td>
<td></td>
</tr>
<tr>
<td>0x12340004</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pointers are variables too (they also reside in memory)
- Their value is the address of another variable
- Address-of operator (unary-&)
  - Gives the address where a variable is (rather than its value)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x12340000C</td>
<td>42</td>
<td>x</td>
</tr>
<tr>
<td>0x12340000B</td>
<td>0x1234000C</td>
<td></td>
</tr>
<tr>
<td>0x12340004</td>
<td>42</td>
<td>y</td>
</tr>
</tbody>
</table>
Pointers

```
int x = 42;
int *p = &x;
int y = *p;
p = 99;
```

Dereference operator (*)

“Follow the pointer”

Gives what is pointed to, instead of pointer value.

Can be used to change values also

```
*p = 99;
p = &y;
p = 7;
```

Can change what p points to by assigning to p

Could try to assign a number to p, but asking for trouble

(Doing *p will access memory address 7, not a valid range of memory, program will crash)

Arrays

Java:

```
int [] x = new int [nElems];
```

C:

```
int data[42]; //if size is known constant
int * data2 = malloc [nElem * sizeof(*data2)];
```

- malloc takes number of bytes
- sizeof tells how many bytes something takes

C++:

- Has new int[nElems] like Java
- And malloc(size) like C

In C Pointers = Arrays

Pointers and arrays are “the same thing”

Note: no bounds checking either way

```
int * p = malloc(42 * sizeof(*p));
int data[100];
int x = 3;
int * q = & p;

*p = 3;
p[4] = 12;
q[0] = 11; //q[1] would be a bad idea
```
x[n] is the same as *(x + n)

- To make sense, x+n has to be done a little oddly
- Recall: some things take multiple bytes
  - One int spans 4 (or 8) addresses
  - x+1 has to go 4 (or 8) address if x is an int * to get to next int
  - Doesn’t make sense to go to next byte: pieces of two ints

Rule:
- If x is a T*, then x+n evaluates to x + (n * sizeof(T)).
- T is some type, e.g., int

Stupid C trivia: x[n] = *(x+n) = n[x];
- 3[myArray] works just fine…

Doesn’t make sense to go to next byte: pieces of two ints
More Pointers & Arrays

```
int * a = new int[100];
// 0 1 2 3 4 5 6 7 8 9
// 01 02 03 04 05 06 07 08 09 29
// a is a pointer
// *a is an int
// a[0] is an int (same as *a)
// a[1] is an int
// a+1 is a pointer
// a+32 is a pointer
// *(a+1) is an int (same as a[1])
// *(a+99) is an int
// *(a+100) is trouble
```

Memory Manager (Heap Manager)

malloc / new (C/C++)
Library routines that handle memory management for heap (allocation / deallocation)
Java has garbage collection (reclaim memory of unreferenced objects)
C/C++ must use free/delete, else memory leak
More on this Wed

Strings as Arrays

```
0 1 15 16 42 43
// 01 15 16 42 43
```

A string is an array of characters with '0' at the end
Each element is one byte, ASCII code
'0' is null (ASCII code 0)

Strlen()

```
int strlen(char * s)
// pre: '^0' terminated
// post: returns # chars
{
    int count=0;
    while (*s++)
        count++;
    return count;
}
```

Vector Class vs. Arrays

Vector Class
- insulates programmers
- array bounds checking
- automatically growing/shrinking when more items are added/deleted

How are Vectors implemented?
- Arrays, re-allocated as needed

Arrays can be more efficient
- but be leery of claims that C-style arrays required for efficiency

Outline

Memory
- Memory holds numbers
- Everything (floats, colors, sound, letters) is a number

Bit Manipulations
Bit Manipulation: Motivating Example

Suppose I want to track

- 20 Million people
- 30 properties per person (e.g., IS_MALE, IS_STUDENT,…)
  - Each may or may not apply to any given person
- How could I represent this?

Bad Idea 1:

```java
class Person {
    static final int IS_MALE = 0;
    static final int IS_STUDENT = 1;
    ...
    Vector<Integer> props;
    ...
    boolean hasProperty(int p) {
        for (int i = 0; i < props.size(); i++)
            if (props.get(i).intValue() == p) return true;
        return false;
    }
```

P.S. Undergrad friend of mine did this, code was very slow

Okish Idea

```java
class Person {
    boolean is_male;
    boolean is_student;
    ...
}
```

Why is this not so great?

- Byte is smallest addressable size: typically `bool = 1 byte`
- Wastes space
- As we will see later, wasting space can hurt performance (caches)

Great idea

```java
class Person {
    static final int IS_MALE    = 0x1; //0000…0001
    static final int IS_STUDENT = 0x2; //0000…0010
    static final int KNOWS_JAVA = 0x4; //0000…0100
    static final int KNOWS_C   = 0x8; //0000…1000
    ... int props;
    bool hasProperty(int p) {
        return (props & p) != 0;
    }
```

Use one `int`, and manipulate/test individual bits
Fast, and only need 1 32-bit int (4 bytes) for 32 properties!

Bit-wise operations: AND

Previous slide used bit-wise & (AND):

- Do not confuse AND (a & b) with address-of (& a)
- Do not confuse logical AND (a && b)
- Each result bit will be 1 iff corresponding bit in a AND b is 1:

```
1101010 & 0100100
0100000
```

- Using bits as sets, corresponds to intersection
**Bit-wise operations: OR**

- Each result bit will be 1 iff corresponding bit in `a` OR `b` is 1:

```
1101010
| 0100100
```

- Using bits as sets, corresponds to union
  - Add `p` to `props`: `props = props | p`;

**Bit-wise operations: XOR**

- OR is inclusive (either or both)
- XOR is exclusive-or (either, but not both):

```
1101010
^ 0100100
```

**Bit-wise operations: NOT**

- Bit-wise negation ~ (NOT)
  - Flip each bit
    ```
    ~1101010
    0010101
    ```
  - Can combine NOT and AND for removal:
    - Remove `p` from `props`: `props = props & ~p`;
    - Suppose `props` is 11011101 and `p` is 00010000:
      ```
      ~p = 11101111
      Props & ~p:
      11011101
      11101111
      11011111
      ```

**Bit-wise operations: SHIFT**

- Left shift (`<<`)
  - Moves left, bringing in 0s at right, excess bits "fall off"
  - `x << k` corresponds to `x * 2^k`

- Logical (or unsigned) right shift (`>>`)
  - Moves bits right, bringing in 0s at left, excess bits "fall off"
  - `x >> k` corresponds to `x / 2^k` for unsigned `x`

- Arithmetic (or signed) right shift (`>>`)
  - Moves bits right, bringing in (sign bit) at left
  - `x >> k` corresponds to `x / 2^k` for signed `x`

**Summary**

**Homework #1 Sept 14**
Computer memory is linear array of bytes
Pointers contain address of other data
Bitwise operations

**Next Time**
Dive into C
**Reading**
Finish Ch 1, Start on Ch 2