Under the Hood: Data Representations, Memory and Bit Operations

CPS 104
Lecture 3

Administrivia
• Homework #1 Due Jan 24
• Memory
• Bitwise operations
Outline
• Review/finish data representations
• Arrays
• Pointers
• Pointer Arithmetic
• Bitwise operations (AND, OR)
Reading
• Chapter 2 (next few lectures)

Review: 2’s Complement Negation and Addition
• To negate a number do:
  > Step 1: complement the digits
  > Step 2: add 1
Example
14_{10} = 001110_2
-14_{10} = 110001_2
+ 1
110010_2
• To add signed numbers use regular addition but disregard carry out
  > Example 18_{10} - 14_{10} = 18_{10} + (-14_{10}) = 4_{10}

Review: Floating Point Representation
Numbers are represented by:

\[ X = (-1)^s \times 2^{E-127} \times 1.M \]

\[ S := 1\text{-bit field; Sign bit} \]
\[ E := 8\text{-bit field; Exponent: Biased integer, } 0 \leq E \leq 255. \]
\[ M := 23\text{-bit field; Mantissa: Normalized fraction with hidden 1.} \]

Single precision floating point number

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Exp</td>
<td>Mantissa</td>
</tr>
</tbody>
</table>

Floating Point Representation

Example:
What floating-point number is:
0xC1580000?
ASCII Character Representation

<table>
<thead>
<tr>
<th>Oct. Chr</th>
<th>Chr</th>
<th>ASCII</th>
<th>Oct. Chr</th>
<th>Chr</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>nul</td>
<td>001</td>
<td>soh</td>
<td>013</td>
<td>vt</td>
</tr>
<tr>
<td>010</td>
<td>bs</td>
<td>014</td>
<td>op</td>
<td>015</td>
<td>cr</td>
</tr>
<tr>
<td>020</td>
<td>sp</td>
<td>021</td>
<td>tms</td>
<td>022</td>
<td>ch</td>
</tr>
<tr>
<td>030</td>
<td>eol</td>
<td>031</td>
<td>ext</td>
<td>032</td>
<td>esc</td>
</tr>
<tr>
<td>040</td>
<td>sp</td>
<td>041</td>
<td>nq</td>
<td>042</td>
<td>si</td>
</tr>
</tbody>
</table>

- Each character is represented by a 7-bit ASCII code.
- It is packed into 8-bits

Basic Data Types

**Bit**: 0, 1  
**Bit String**: sequence of bits of a particular length
  - 8 bits is a byte
  - 16 bits is a half-word
  - 32 bits is a word
  - 64 bits is a double-word

**Character**: 7-bit code

**Decimal (BCD code)**
- digits 0-9 encoded as 0000 thru 1001
- two decimal digits packed per 8 bit byte

**Floating Point**
- Single Precision (32-bit representation)
- Double Precision (64-bit representation)
- Extended Precision (128-bit representation)

Summary of Data Representations

- Computers operate on binary numbers (0s and 1s)
- Conversion to/from binary, oct, hex
- Signed binary numbers
  - 2’s complement
  - arithmetic, negation
- Floating point representation
  - hidden 1
  - biased exponent
  - single precision, double precision
- ASCII code for characters

A Program’s View of Memory

- **What is Memory?** a bunch of bits
- **Looks like** a large linear array
- **Find things by** indexing into array
  - unsigned integer
  - Most computers support byte (8-bit) addressing
    - Each byte has a unique address (location).
    - Byte of data at address 0x100 and 0x101
    - Word of data at address 0x100 and 0x104
  - 32-bit v.s. 64-bit addresses
    - we will assume 32-bit for rest of course, unless otherwise stated

Computer Memory

- **What is Computer Memory?**
- **What does it “look like” to the program?**
- **How do we find things in computer memory?**

Buzz Word Definition: Endianness

**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
**Buzz Word Definition: 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?

**Memory Partitions**

- **Text for instructions**
  - add res, src1, src2
  - mem[res] = mem[src1] + mem[src2]

- **Data**
  - static (constants, globals)
  - dynamic (heap, new allocated)
  - grows up

- **Stack**
  - local variables
  - grows down

- Variables are names for memory locations
  - int x;

---

**A Simple Program’s Memory Layout**

```java
... int result;
main() {
    int x;
    ... result = x + result;
    ...
} mem[0x208] = mem[0x400] + mem[0x208]
```

**Reference (handle) vs. Pointer**

**Java**

- "The value of a reference type variable, in contrast to that of a primitive type, is a reference to (an address of) the value or set of values represented by the variable"
- Cannot manipulate the value

**C or C++**

- A pointer is a memory location that contains the address of another memory location
- Can manipulate the value (double edge sword)

---

**Pointers**

- “address of” operator 
  - don’t confuse with bitwise AND operator (later today)

**Given**

```java
int x; int *p;
```

**Then**

```java
p = &x; *p = 2; x = 2; produce the same result
```

- What happens for p = 2?;

**On 32-bit machine, p is 32-bits**

```
x 0x26c0f0 ...
```

---

**Vector Class vs. Arrays**

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

- **How are Vectors implemented?**
  - real understanding comes when more levels of abstraction are understood (the ridiculous...)

- **Programming close to HW**
  - (e.g., operating system, device drivers, etc.)

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

- **Can talk about memory easier in terms of arrays**
  - pointer to a vector?
Arrays

- In C++ allocate using array form of `new`
  ```cpp
  int *a = new int[100];
  double *b = new double[300];
  ```
- `new []` returns a pointer to a block of memory
  > how big? where?
- Size of block can be set at runtime
- `delete [] a;` // storage returned

In C:
```c
malloc(nbytes);
free(ptr);
```

Address Calculation

- `*a` is a pointer, what is `x+33`?
- A pointer, but where?
  > what does calculation depend on?
- Result of adding an int to a pointer depends on size of object pointed to
- Result of subtracting two pointers is an int
  ```cpp
  (d + 3) - d = _______
  ```

More Pointer Arithmetic

- address one past the end of an array is ok for pointer comparison only
- what's at `*(begin+44)`?
- what does `begin++` mean?
- how are pointers compared using `<` and using `==`?
- what is value of `end - begin`?

More Pointers & Arrays

```cpp
int * a = new int[100];
```
```cpp
01 9 9 32
33 98
```
- 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+1)` is an int (same as `a[1]`
- `*(a+99)` is an int
- `*(a+100)` is trouble

Array Example

```cpp
#include <iostream.h>
main()
{
  int *a = new int[100];
  int *p = a;
  int k;
  for (k = 0; k < 100; k++)
    {*
      p = k;
      p++;
    }
  cout << "entry 3 = " << a[3] << endl;
}
```

Array of Classes (Linked List)

```cpp
#include <iostream.h>
class node{
public:
  int me;
  node *next;
};
main()
{
  node *ar = new node[10];
  node *p = ar;
  int k;
  for (k = 0; k < 9; k++)
    {*
      p->me = k;
      p->next = &ar[k+1];
      p++;
    }
  p->me = 9;
  p->next = NULL;
  p = &ar[0];
  while (p != NULL)
    {*
      cout << p->me << " " << hex << p << " " << p->next << endl;
      p = p->next;
    }
```

- Given `ar = 0x10000`, what does memory layout look like?
### Memory Layout

#### Output

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Memory Contents</th>
<th>Symbol</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x26ca8</td>
<td>0</td>
<td>me</td>
<td>array[0]</td>
</tr>
<tr>
<td>0x26cb0</td>
<td>0</td>
<td>next</td>
<td>array[1]</td>
</tr>
<tr>
<td>0x26cb8</td>
<td>0x26cc0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cc8</td>
<td>0x26cd0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cd8</td>
<td>0x26ce0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26ce8</td>
<td>0x26cf0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26cf8</td>
<td>0x26d0b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d0b</td>
<td>0x26d14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d14</td>
<td>0x26d28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d28</td>
<td>0x26d3c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d3c</td>
<td>0x26d48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d48</td>
<td>0x26d5c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d5c</td>
<td>0x26d68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d68</td>
<td>0x26d7c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d7c</td>
<td>0x26d88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d88</td>
<td>0x26d9c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26d9c</td>
<td>0x26da8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x26da8</td>
<td>0x26db4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `me` is int (4 bytes)
- `next` is node* (4 bytes)

### Array of Classes with Inheritance

```cpp
#include <iostream.h>

class course {
public:
    int number;
    int score;
    float average;
};

class node : public course {
public:
    int me;
    node *next;
};

main() {
    node *ar = new node[10];
    node *p = ar;
    int *num_ptr, *me_ptr;
    int k;
    for (k = 0; k < 9; k++) {
        p->me = k;
        p->number = 104;
        p->score = k*20;
        p->average = 0.96;
        p->next = &ar[k+1];
        p++;
    }
    p->me = 9;
    p->number = 104;
    p->score = k*20;
    p->average = 0.96;
    p->next = NULL;
    num_ptr = &p->number;
    me_ptr = &p->me;
    cout << p->me << " " << *me_ptr
         << " " << p->number << " " << *num_ptr << endl;
}```

### Strings as Arrays

- 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()

- `strlen()` returns the # of characters in a string
  - same as # elements in char array?

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

### Outline

- **Memory**
  - Important concept: bits in memory can represent anything
  - Data (char, int, float, int*, char*, etc.)
  - Instructions (the commands of the machine)
- **Bit Manipulations**
Bit Manipulations

Problem
• 32-bit word contains many values
  ➢ e.g., input device, sensors, etc.
  ➢ current x,y position of mouse and which button (left, mid, right)
• Assume x, y position is 0-255
• How many bits for position?
• How many for button?

Goal
• Extract position and button from 32-bit word
• Need operations on individual bits of word

Bitwise AND / OR

• & operator performs bitwise AND
• | operator performs bitwise OR

  For bit
  0 & 0 = 0 0 | 0 = 0
  0 & 1 = 0 0 | 1 = 1
  1 & 0 = 0 1 | 0 = 1
  1 & 1 = 1 1 | 1 = 1

  For multiple bits, apply operation to individual bits in same position

AND          OR
011010       011010
101110       101110
001010       111110

Mouse Example

• 32-bit word with x, y and button fields
  ➢ bits 0-7 contain x position
  ➢ bits 8-15 contain y position
  ➢ bits 16-17 contain button (0 = left, 1 = middle, 2 = right)
• To extract value need to clear all other bits
• How do I use bitwise operations to do this?

  button   y   x
  0x1a34c = 01 1010 0011 0100 1100

Mouse Solution

• AND with a bit mask
  ➢ specific values that clear some bits, but pass others through
• To extract x position use mask 0x000ff
  xpos = 0x1a34c & 0x000ff

  button   y   x
  0x1a34c = 01 1010 0011 0100 1100
  0x000ff = 00 1111 1111 0000 0000
  0x0a300 = 00 1010 0011 0000 0000

• To extract y position use mask 0x0ff00
  ypos = 0x1a34c & 0x0ff00

• Similarly, button is extracted with mask 0x30000
  button = 0x1a34c & 0x30000
• Not quite done...why?

  button   y   x
  0x1a34c = 01 1010 0011 0100 1100
  0x000ff = 00 1111 1111 0000 0000
  0x0a300 = 00 1010 0011 0000 0000

More of the Mouse Solution

• To extract y position use mask 0x0ff00
  ypos = 0x1a34c & 0x0ff00
• Similarly, button is extracted with mask 0x30000
  button = 0x1a34c & 0x30000
• Not quite done...why?

  button   y   x
  0x1a34c = 01 1010 0011 0100 1100
  0x000ff = 00 1111 1111 0000 0000
  0x0a300 = 00 1010 0011 0000 0000

The SHIFT operator

• >> is shift right, << is shift left, operands are int and number of positions to shift
  ➢ (1 << 3) is ...000001 -> ...0001000 (it’s 2³)
  ➢ 0xff00 is 0xff << 8, and 0xff is 0xff00 >> 8
  ➢ So, true ypos value is
  ypos = (0x1a34c & 0x0ff00) >> 8
  button = (0x1a34c & 0x30000) >> 16
Extracting Parts of Floating Point Number

- See web page for full code

```c
#define EXP_BITS 8
#define MANTISSA_BITS 23
#define SIGN_MASK 0x80000000
#define EXP_MASK  0x7f800000
#define MANTISSA_MASK 0x007fffff

class myfloat {
    public:
        int sign;
        unsigned int exp;
        unsigned int mantissa;

    num->sign = (x & SIGN_MASK) >> (EXP_BITS + MANTISSA_BITS);
    num->exp  = (x & EXP_MASK) >> MANTISSA_BITS;
    num->mantissa = x & MANTISSA_MASK;
```

Summary

- Homework #1 Jan 24
- Computer memory is linear array of bytes
- Pointer is memory location that contains address of another memory location
- Bitwise operations
- Code examples are linked to course web page
- We'll visit these topics again throughout semester

Next Time

- Instruction set architecture (ISA)

Reading

- Chapter 2