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\text{_{10}} = 001110_{2}
-14\text{_{10}} = 110001_{2}
+1
110010_{2}

- To add signed numbers use regular addition but disregard carry out

Example \ 18\text{_{10}} - 14\text{_{10}} = 18\text{_{10}} + (-14\text{_{10}}) = 4\text{_{10}}
010010_{2}
+110010_{2}
000100_{2}

Review: Floating Point Representation

Numbers are represented by:

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

- \( S \) := 1-bit field; Sign bit
- \( E \) := 8-bit field; Exponent: Biased integer, \( 0 \leq E \leq 255 \).
- \( M \) := 23-bit field; Mantissa: Normalized fraction with hidden 1.

Single precision floating point number

\[
\begin{array}{cccc}
S & \text{exp} & \text{Mantissa} \\
\hline
0 & 22 & 31 \\
8-bit & 23-bit & 0 \\
\end{array}
\]
Floating Point Representation

Example:
What floating-point number is:
0xC1580000?

Floating Point Representation

• Double Precision Floating point:

64-bit representation: 1-bit sign, 11-bit (biased) exponent; 52-bit mantissa (with hidden 1).

\[ X = (-1)^s \times 2^{E-1023} \times 1 \cdot M \]

Double precision floating point number

<table>
<thead>
<tr>
<th>S</th>
<th>Exp</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-bit</td>
<td>20-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit</td>
</tr>
</tbody>
</table>
ASCII Character Representation

<table>
<thead>
<tr>
<th>Oct. Chr.</th>
<th>000</th>
<th>001</th>
<th>002</th>
<th>003</th>
<th>004</th>
<th>005</th>
<th>006</th>
<th>007</th>
<th>008</th>
</tr>
</thead>
<tbody>
<tr>
<td>nul</td>
<td>010</td>
<td>bs</td>
<td>011</td>
<td>ht</td>
<td>012</td>
<td>nl</td>
<td>013</td>
<td>vt</td>
<td>014</td>
</tr>
<tr>
<td>soh</td>
<td>020</td>
<td>dle</td>
<td>021</td>
<td>df</td>
<td>022</td>
<td>dc2</td>
<td>023</td>
<td>dc3</td>
<td>024</td>
</tr>
<tr>
<td>stx</td>
<td>025</td>
<td>dc4</td>
<td>026</td>
<td>enq</td>
<td>027</td>
<td>esc</td>
<td>028</td>
<td>fs</td>
<td>029</td>
</tr>
<tr>
<td>etx</td>
<td>030</td>
<td>sub</td>
<td>031</td>
<td>em</td>
<td>032</td>
<td>sp</td>
<td>033</td>
<td>can</td>
<td>034</td>
</tr>
<tr>
<td>etb</td>
<td>035</td>
<td>can</td>
<td>036</td>
<td>cr</td>
<td>037</td>
<td>sync</td>
<td>038</td>
<td>del</td>
<td>039</td>
</tr>
<tr>
<td>crl</td>
<td>040</td>
<td>sp</td>
<td>041</td>
<td>!</td>
<td>042</td>
<td>&quot;</td>
<td>043</td>
<td>#</td>
<td>044</td>
</tr>
<tr>
<td>nl</td>
<td>050</td>
<td>(</td>
<td>051</td>
<td>)</td>
<td>052</td>
<td>*</td>
<td>053</td>
<td>+</td>
<td>054</td>
</tr>
<tr>
<td>vt</td>
<td>060</td>
<td>0</td>
<td>061</td>
<td>1</td>
<td>062</td>
<td>2</td>
<td>063</td>
<td>3</td>
<td>064</td>
</tr>
<tr>
<td>np</td>
<td>070</td>
<td>6</td>
<td>071</td>
<td>9</td>
<td>072</td>
<td>:</td>
<td>073</td>
<td>;</td>
<td>074</td>
</tr>
<tr>
<td>enq</td>
<td>080</td>
<td>0</td>
<td>081</td>
<td>1</td>
<td>082</td>
<td>2</td>
<td>083</td>
<td>3</td>
<td>084</td>
</tr>
<tr>
<td>syn</td>
<td>090</td>
<td>0</td>
<td>091</td>
<td>1</td>
<td>092</td>
<td>2</td>
<td>093</td>
<td>3</td>
<td>094</td>
</tr>
<tr>
<td>nak</td>
<td>100</td>
<td>0</td>
<td>101</td>
<td>A</td>
<td>102</td>
<td>B</td>
<td>103</td>
<td>C</td>
<td>104</td>
</tr>
<tr>
<td>sp</td>
<td>110</td>
<td>0</td>
<td>111</td>
<td>I</td>
<td>112</td>
<td>J</td>
<td>113</td>
<td>K</td>
<td>114</td>
</tr>
<tr>
<td>soh</td>
<td>120</td>
<td>0</td>
<td>121</td>
<td>Q</td>
<td>122</td>
<td>R</td>
<td>123</td>
<td>S</td>
<td>124</td>
</tr>
<tr>
<td>!</td>
<td>130</td>
<td>0</td>
<td>131</td>
<td>Y</td>
<td>132</td>
<td>Z</td>
<td>133</td>
<td>[</td>
<td>134</td>
</tr>
<tr>
<td>&quot;</td>
<td>140</td>
<td>0</td>
<td>141</td>
<td>a</td>
<td>142</td>
<td>b</td>
<td>143</td>
<td>c</td>
<td>144</td>
</tr>
<tr>
<td>#</td>
<td>150</td>
<td>0</td>
<td>151</td>
<td>i</td>
<td>152</td>
<td>j</td>
<td>153</td>
<td>k</td>
<td>154</td>
</tr>
<tr>
<td>$</td>
<td>160</td>
<td>0</td>
<td>161</td>
<td>q</td>
<td>162</td>
<td>r</td>
<td>163</td>
<td>s</td>
<td>164</td>
</tr>
<tr>
<td>%</td>
<td>170</td>
<td>0</td>
<td>171</td>
<td>x</td>
<td>172</td>
<td>z</td>
<td>173</td>
<td>{</td>
<td>174</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
  - 4 bits is a nibble
  - 8 bits is a byte
  - 16 bits is a half-word
  - 32 bits is a word
  - 64 bits is a double-word

- **Character**: ASCII 7 bit code
- **Decimal**: (BCD code)
  - digits 0-9 encoded as 0000 thru 1001
  - two decimal digits packed per 8 bit byte

- **Integers**: 2's Complement (32-bit representation).
  - How many +/- #'s?
  - Where is decimal pt?
  - How are +/- exponents represented?

- **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

Computer Memory

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

Buzz Word Definition: 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
**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 ?

![Alignment Diagram]

---

**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

```c
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

```c
int x; int *p;
p = &x;
```

Then

```c
*p = 2; and x = 2; produce the same result
```

• What happens for `p = 2;`;

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

<table>
<thead>
<tr>
<th>x</th>
<th>0x26cf0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>0x26d00</td>
</tr>
</tbody>
</table>

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`
  
  ```
  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
  
  ```
  malloc(nbytes);
  free(ptr);
  ```

---

Address Calculation

- `x` 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

  ```
  (d + 3) - d = _________
  ```

```plaintext

int * a = new int[100]

```

```

```plaintext
da[33] is the same as *(a+33)
if a is 0x00a0, then a+1 is 0x00a4, a+2 is 0x00a8
(decimal 160, 164, 168)
```

```

```plaintext
double * d = new double[200];

```

```

```plaintext
*(d+33) is the same as d[33]
if d is 0x00b0, then d+1 is 0x00b8, d+2 is 0x00c0
(decimal 176, 184, 192)
```
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?

```c
char * a = new char[44];
char * begin = a;
char * end = a + 44;

while (begin < end)
{
    *begin = 'z';
    begin++;
}
```

More Pointers & Arrays

```c
int * a = new int[100];
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
```
Array Example

```c++
#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)

```c++
#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>Me</th>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x26ca8</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x26cb0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0x26cb8</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0x26cc0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0x26cc8</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0x26cd0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0x26cd8</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0x26ce0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0x26ce8</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0x26cf0</td>
<td>0</td>
</tr>
</tbody>
</table>

Source Symbols:
- 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 << " " << num_ptr << endl;
}
```
Memory Layout

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?

```c
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
• Per bit
  0 & 0 = 0
  0 & 1 = 0
  1 & 0 = 0
  1 & 1 = 1
• For multiple bits, apply operation to individual bits in same position

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>011010</td>
<td>011010</td>
</tr>
<tr>
<td>101110</td>
<td>101110</td>
</tr>
<tr>
<td>001010</td>
<td>111110</td>
</tr>
</tbody>
</table>
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?

\[
\begin{array}{c|c|c|c}
\text{button} & \text{y} & \text{x} \\
0x1a34c & 01 & 1010 0011 0100 1100
\end{array}
\]

Mouse Solution

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

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\text{button} & \text{y} & \text{x} \\
0x1a34c & 01 & 1010 0011 0100 1100 \\
0x000ff & 00 & 0000 0000 1111 1111 \\
0x0004c & 00 & 0000 0000 0100 1100
\end{array}
\]
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?

\[
\begin{array}{c|cccc}
\text{button} & y & x \\
0x1a34c & 01 & 1010 & 0011 & 0100 & 1100 \\
0x000ff & 00 & 1111 & 1111 & 0000 & 0000 \\
0x0a300 & 00 & 1010 & 0011 & 0000 & 0000 \\
\end{array}
\]

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