Announcements

- **Lab 0 is due at 5pm. How is it going?**
  - Arrays yesterday in section and today in lecture.
- **Stuck with something?**
  - Post on the discussion board; office hours after lecture.
- **Lab 1 posted, due Monday, July 8.**
- **Vote for next week’s section time today.**
  - See website or email.
  - Current favorite (13/17): Friday immediately after lecture.
- **Reading for each lecture posted on the website.**
  - I won’t be announcing these in general.
  - Memory, Data, and Addressing: CS:APP sections 2.0 – 2.1
  - Integer Representation: CS:APP sections 2.2 - 2.3
- **You can call me Ben.**

Assignment in C (review)

- **Left-hand-side = right-hand-side;**
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

  int x, y;
  x = 0x0000015F;
  y = 0x3CD02700;
  int* z;
  z = &y;
  // What does this do?

  & = ‘address of’
  * = ‘value at address’
  or ‘dereference’

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>x</td>
<td>0x00</td>
</tr>
<tr>
<td>0x04</td>
<td>0x08</td>
<td>0x0C</td>
<td>0x10</td>
<td>0x14</td>
<td>0x18</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x20</td>
<td>0x24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  (little endian)

Assignment in C (review)

- **Left-hand-side = right-hand-side;**
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

  int x, y;
  x = 0x0000015F;
  y = 0x3CD02700;
  int* z;
  z = &y;
  // Get address of y, put it in z.
  *z = x;
  // What does this do?

  & = ‘address of’
  * = ‘value at address’
  or ‘dereference’

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>x</td>
<td>0x00</td>
</tr>
<tr>
<td>0x04</td>
<td>0x08</td>
<td>0x0C</td>
<td>0x10</td>
<td>0x14</td>
<td>0x18</td>
</tr>
<tr>
<td>0x1C</td>
<td>0x20</td>
<td>0x24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  (little endian)

Today

- **Brief review of pointers**
- **Arrays and address arithmetic**
- **Strings as arrays**
- **Boolean algebra and bitwise manipulations**
- **Start integer representations**
Assignment in C (review)

- Left-hand-side = right-hand-side;
  - LHS must evaluate to a memory location.
  - RHS must evaluate to a value. (could be an address!)
  - Store RHS value at LHS location.

- int x, y;
- x = 0x0000015F;
- y = 0x3CD02700;
- int* z;
- z = &y;
  // Get address of y, put it in z.
- *z = x;
  // Get value in x, put it in memory
  // at the address z points to.

Arrays in C

Declaration: int a[6];

Arrays are adjacent locations in memory storing the same type of data object.

- a is a name for the array's address,
  not a pointer to the array.

Indexing:
a[0] = 0x015f;
a[5] = a[0];

The address of a[i] is the address of a[0] plus i times the element size in bytes.

No bounds check:
a[6] = 0xBAD;
a[-1] = 0xBAD;
Arrays in C

**Declaration:** int a[6];

**Indexing:**
- a[0] = 0x015f;
- a[5] = a[0];

**No bounds** a[6] = 0xBAf;

**check:** a[-1] = 0xBAf;

**Pointers:**
- int* p;
- p = a;
- p = &a[0];
- *p = 0xA;

Arrays are adjacent locations in memory storing the same type of data object.
a is a name for the array's address, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.

---

Arrays in C

**Declaration:** int a[6];

**Indexing:**
- a[0] = 0x015f;
- a[5] = a[0];

**No bounds** a[6] = 0xBAf;

**check:** a[-1] = 0xBAf;

**Pointers:**
- int* p;
- p = a;
- p = &a[0];
- *p = 0xA;

Arrays are adjacent locations in memory storing the same type of data object.
a is a name for the array's address, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.
**Arrays in C**

**Declaration:** int a[6];

**Indexing:**
- a[0] = 0x015f;
- a[5] = a[0];

**No bounds** a[6] = 0xBAD;
**check:** a[-1] = 0xBAD;

**Pointers:** int* p;
- p = a;
- p = &a[0];
- p = 0xA;
- p[1] = 0xB;

**array indexing = address arithmetic**
Both are scaled by the size of the type.

**Arrays are adjacent locations in memory storing the same type of data object.**
a is a name for the array's address, not a pointer to the array.
The address of a[i] is the address of a[0] plus i times the element size in bytes.

---

**Arrays in C**

**Declaration:** int a[6];

**Indexing:**
- a[0] = 0x015f;
- a[5] = a[0];

**No bounds** a[6] = 0xBAD;
**check:** a[-1] = 0xBAD;

**Pointers:** int* p;
- p = a;
- p = &a[0];
- p = 0xA;
- p[1] = 0xB;

**array indexing = address arithmetic**
Both are scaled by the size of the type.

---

**Arrays in C**

**Declaration:** int a[6];

**Indexing:**
- a[0] = 0x015f;
- a[5] = a[0];

**No bounds** a[6] = 0xBAD;
**check:** a[-1] = 0xBAD;

**Pointers:** int* p;
- p = a;
- p = &a[0];
- p = 0xA;
- p[1] = 0xB;

**array indexing = address arithmetic**
Both are scaled by the size of the type.

---

**Arrays in C**

**Declaration:** int a[6];

**Indexing:**
- a[0] = 0x015f;
- a[5] = a[0];

**No bounds** a[6] = 0xBAD;
**check:** a[-1] = 0xBAD;

**Pointers:** int* p;
- p = a;
- p = &a[0];
- p = 0xA;
- p[1] = 0xB;

**array indexing = address arithmetic**
Both are scaled by the size of the type.
Arrays in C

Declaration:  int a[6];

Indexing:  a[0] = 0x15f;
            a[5] = a[0];

No bounds  a[6] = 0xBAD;
check:  a[-1] = 0xBAD;

Pointers:  int* p;
           p = a;
           p = &a[0];
           *p = 0xA;

array indexing = address arithmetic
Both are scaled by the size of the type.

\*p = a[1] + 1;

Null-terminated Strings

- For example, “Harry Potter” can be stored as a 13-byte array.

<table>
<thead>
<tr>
<th>72 97 114 114 121 32 80 111 116 116 101 114 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harry Potter \0</td>
</tr>
</tbody>
</table>

- Why do we put a 0, or null zero, at the end of the string?
  - Note the special symbol: string[12] = \\"\0\\";

- How do we compute the string length?

Representing strings

- A C-style string is represented by an array of bytes (char).
  - Elements are one-byte ASCII codes for each character.
  - ASCII = American Standard Code for Information Interchange

Endianness and Strings

C (char = 1 byte)
char s[6] = "12345";

<table>
<thead>
<tr>
<th>32 space 48 0 64 @ 80 P 96 , 112 p</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

Java (char = 2 bytes)
String s = "123";
(not all of the string representation is shown)

IA32, x86-64     SPARC
| 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 |
|---------------|---------------|
| 31 32 00 00   32 32 00 00   33 33 00 00   34 34 00 00   35 35 00 00   36 36 00 00   37 37 00 00   38 38 00 00   39 39 00 00   40 40 00 00   41 41 00 00   42 42 00 00   43 43 00 00   44 44 00 00   45 45 00 00   46 46 00 00   47 47 00 00 |

- Byte ordering (endianness) is not an issue for 1-byte values.
  - Arrays are not values; elements are values; chars are single bytes.

- Unicode characters – up to 4 bytes/character
  - ASCII codes still work (just add leading zeros).
  - Unicode can support the many characters in all languages in the world.
  - Java and C have libraries for Unicode (Java commonly uses 2 bytes/char)
Examining Data Representations

- Code to print byte representation of data
  - Any data type can be treated as a byte array by casting it to char.
  - C has unchecked casts. << DANGER >>

```c
typedef char byte; // size of char == 1 byte

void show_bytes(byte* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%.2x\n", start+i, *(start+i));
}
```

```c
void show_int (int x) {
    show_bytes( (byte *) &x, sizeof(int));
}
```

show_bytes Execution Example

```c
int a = 12345; // represented as 0x00003039
printf("int a = %d\n", a);
show_int(a); // show_bytes((pointer) &a, sizeof(int));
```

Result (Linux):

```
int a = 12345;
0x11ffffcb8 0x39
0x11ffffcb9 0x30
0x11ffffcba 0x00
0x11ffffcbb 0x00
```

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0
  - AND: A&B = 1 when both A is 1 and B is 1
  - OR: A|B = 1 when either A is 1 or B is 1
  - XOR: A^B = 1 when either A is 1 or B is 1, but not both
  - NOT: ~A = 1 when A is 0 and vice-versa
  - DeMorgan’s Law: ~(A | B) = ~A & ~B

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0 1</th>
<th>1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>^</th>
<th>0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1</td>
</tr>
<tr>
<td>1</td>
<td>0 1</td>
</tr>
</tbody>
</table>

General Boolean Algebras

- Operate on bit vectors
  - Operations applied bitwise
    - `&` 0101001 & 0101010 = 0101001
    - `|` 0101001 | 0101010 = 0101010
    - `^` 0101001 ^ 0101010 = 0000000
    - `~` ~0101010 = 1010101

- All of the properties of Boolean algebra apply

- How does this relate to set operations?
Representing & Manipulating Sets

- **Representation**
  - A $w$-bit vector represents subsets of $\{0, ..., w-1\}$
  - $a_j = 1$ iff $j \in A$

  - $01101001$ \{3, 5, 6\}
  - $01010101$ \{2, 4, 6\}

- **Operations**
  - & Intersection $01000001$ \{0, 6\}
  - | Union $01111101$ \{0, 2, 3, 4, 5, 6\}
  - ^ Symmetric difference $00111100$ \{2, 3, 4, 5\}
  - ~ Complement $10101010$ \{1, 3, 5, 7\}

Bit-Level Operations in C

- **& | ^ ~**
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors

- **Examples (char data type)**
  - $\sim 0x41 \rightarrow 0xBE$
    - $01000001 \rightarrow 10111110$
  - $\sim 0x00 \rightarrow 0xFF$
    - $00000000 \rightarrow 11111111$
  - $0x69 \& 0x55 \rightarrow 0x41$
    - $01101001 \& 01010101 \rightarrow 01000000$
  - $0x69 \mid 0x55 \rightarrow 0x7D$
    - $01101001 \mid 01010101 \rightarrow 01111101$

- **Many bit-twiddling puzzles in Lab 1**

Contrast: Logic Operations in C

- **Contrast to logical operators**
  - $\&\& \mid\mid \:$!
    - $0$ is “False”
    - Anything nonzero is “True”
    - Always return 0 or 1
    - Early termination a.k.a. short-circuit evaluation

- **Examples (char data type)**
  - $!0x41 \rightarrow 0x00$
  - $!0x00 \rightarrow 0x01$
  - $!!0x41 \rightarrow 0x01$
  - $0x69 \&\& 0x55 \rightarrow 0x01$
  - $0x69 \mid\mid 0x55 \rightarrow 0x01$
  - $p \&\& *p++$ (avoids null pointer access, null pointer = 0x00000000)