Review Questions

1) If the word size of a machine is 64-bits, which of the following is usually true? (pick all that apply)
   
   a) 64 bits is the size of a pointer   
      [false]
   
   b) 64 bits is the size of an integer   
      [false]
   
   c) 64 bits is the width of a register   
      [true]

2) (True/False) By looking at the bits stored in memory, I can tell if a particular 4-bytes is being used to represent an integer, floating point number, or instruction

3) If the size of a pointer on a machine is 7 bits, the address space is how many bytes?  \( 2^7 = 128 \) B
Memory, Data, & Addressing II
CSE 351 Autumn 2016

Instructor:
Justin Hsia

Teaching Assistants:
Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun

http://xkcd.com/371/
Administrivia

- **Lab 0 due today @ 5pm**
  - Credit/no credit – we’ll talk about topics in depth later
- **Lab 1 released tomorrow @ 6pm**
  - Some progress due Monday 10/10, Lab 1 due Friday 10/14

**Survey results:**

- **Hoping to get out**
  - Linux/Unix, C, Assembly
  - How a computer works
- **Concerns**
  - Fast-paced & course load
  - Having little background in this area
  - Not familiar with C or Linux
Memory, Data, and Addressing

- Representing information as bits and bytes
- Organizing and addressing data in memory
- Manipulating data in memory using C
- Boolean algebra and bit-level manipulations
Addresses and Pointers in C

- `&` = “address of” operator
- `*` = “value at address” or “dereference” operator

```c
int* ptr;
int x = 5;
int y = 2;
ptr = &x;
y = 1 + *ptr;
```

- Declares a variable, `ptr`, that is a pointer to (i.e. holds the address of) an `int` in memory
- Declares two variables, `x` and `y`, that hold `ints`, and sets them to 5 and 2, respectively
- Sets `ptr` to the address of `x` ("`ptr` points to `x""")
- Sets `y` to “1 plus the value stored at the address held by `ptr`. Because `ptr` points to `x`, this is equivalent to `y=1+x"""

What is `*(&y)`?
Assignment in C

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds "garbage")

```c
int x, y;
```

- `x` is at address 0x04, `y` is at 0x18

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>A7 00 32 00</td>
</tr>
<tr>
<td>0x01</td>
<td>00 01 29 F3</td>
</tr>
<tr>
<td>0x02</td>
<td>EE EE EE EE</td>
</tr>
<tr>
<td>0x03</td>
<td>FA CE CA FE</td>
</tr>
<tr>
<td>0x04</td>
<td>26 00 00 00</td>
</tr>
<tr>
<td>0x05</td>
<td>00 00 10 00</td>
</tr>
<tr>
<td>0x06</td>
<td>01 00 00 00</td>
</tr>
<tr>
<td>0x07</td>
<td>FF 00 F4 96</td>
</tr>
<tr>
<td>0x08</td>
<td>DE AD BE EF</td>
</tr>
<tr>
<td>0x09</td>
<td>00 00 00 00</td>
</tr>
<tr>
<td>0x10</td>
<td>0x00 0x04 x</td>
</tr>
<tr>
<td>0x11</td>
<td>0x08 0x0C</td>
</tr>
<tr>
<td>0x12</td>
<td>0x10 0x14</td>
</tr>
<tr>
<td>0x13</td>
<td>0x18 0x1C</td>
</tr>
<tr>
<td>0x14</td>
<td>0x20 0x24</td>
</tr>
</tbody>
</table>
```
Assignment in C

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds “garbage”)
- `int x, y;`
  - `x` is at address 0x04, `y` is at 0x18

32-bit example (pointers are 32-bits wide)
Assignment in C

- 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 = 0;`

32-bit example (pointers are 32-bits wide)

\& = “address of”
* = “dereference”
Assignment in C

- 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

```c
int x, y;
x = 0;
y = 0x3CD02700;
```

- int x, y;
- x = 0;
- y = 0x3CD02700;

32-bit example (pointers are 32-bits wide)
& = “address of”
* = “dereference”

little endian!
Assignment in C

- 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 = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`

---

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”
Assignment in C

- 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 = 0;
y = 0x3CD02700;
x = y + 3;
```
- Get value at \( y \), add 3, store in \( x \)

```
int* z;
z is at address 0x20
```
Assignment in C

- 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 = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
- int* z = &y + 3;
  - Get address of y, “add 3”, store in z
Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int) = 4`

- `int* z = &y + 3;`
  - Get address of `y`, add `3* sizeof(int)`, store in `z`
  - `&y = 0x18 = 1*16^1 + 8*16^0 = 24`
  - `24 + 3*(4) = 36 = 2*16^1 + 4*16^0 = 0x24`

- Pointer arithmetic can be dangerous!
  - Can easily lead to bad memory accesses
  - Be careful with data types and *casting*


Assignment in C

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int* z = &y + 3;`
  - Get address of `y`, add 12, store in `z`
- `*z = y;`
  - What does this do?
Assignment in C

- int x, y;
- x = 0;
- y = 0x3CD02700;
- x = y + 3;
  - Get value at y, add 3, store in x
- int* z = &y + 3;
  - Get address of y, add 12, store in z
- *z = y;
  - Get value of y, put in address stored in z

32-bit example (pointers are 32-bits wide)

& = “address of”
* = “dereference”

The target of a pointer is also a memory location
Arrays in C

Declaration: `int a[6];`

- **element type**
- **name**
- **number of elements**

Arrays are adjacent locations in memory storing the same type of data object. `a` is a name for the array’s address.

64-bit example (pointers are 64-bits wide)
## Arrays in C

Declaration: `int a[6];`

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

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

- `a` is a name for the array’s address.
- The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

![Array Memory Layout](image)
Arrays in C

Declaration: `int a[6];`

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

No bounds `a[6] = 0xBAD;`

checking: `a[-1] = 0xBAD;`

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

- `a` is a name for the array’s address
- The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes

<table>
<thead>
<tr>
<th>Index</th>
<th>0x0</th>
<th>0x1</th>
<th>0x2</th>
<th>0x3</th>
<th>0x4</th>
<th>0x5</th>
<th>0x6</th>
<th>0x7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AD</td>
<td>0B</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48
Arrays in C

Declaration: \( \text{int } a[6]; \)

Indexing: \( a[0] = 0x015f; \)  
\( a[5] = a[0]; \)

No bounds \( a[6] = 0x\text{BAD}; \)  
checking: \( a[-1] = 0x\text{BAD}; \)

Pointers: \( \text{int }* p; \)
\( p = a; \)
\( p = \&a[0]; \)
\( *p = 0xA; \)
\( \text{a[4]} \)

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

\( a \) is a name for the array’s address

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;`
checking:  `a[-1] = 0xBAD;`

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

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

```
0x00 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
0x00 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
OA 00 00 00 0B 00 00 00 00
5F 01 00 00 00 00 00 00 00
AD 0B 00 00 00 00 00 00 00
```
Arrays in C

Declaration: `int a[6];`

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

No bounds `a[6] = 0xBAD;`

checking: `a[-1] = 0xBAD;`

Pointers: `int* p;`
"equivalent"

```
p = a;
p = &a[0];
*p = 0xA;
```

array indexing = address arithmetic (both scaled by the size of the type)

```
p[1] = 0xB;
*(p+1) = 0xB;
p = p + 2;
```

`*p = a[1] + 1;`

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

`a` is a name for the array’s address

The address of `a[i]` is the address of `a[0]` plus `i` times the element size in bytes
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?

1 void main() {
2  int a[] = {5,10};
3  int * p = a;
4  p = p + 1;
5  *p = *p + 1;
6  }

(A) 101 10 5 10 then 101 11 5 11
(B) 104 10 5 10 then 104 11 5 11
(C) 100 6 6 10 then 101 6 6 10
(D) 100 6 6 10 then 104 6 6 10

http://PollEv.com/justinh
Representing strings

- C-style string stored as an array of bytes (char *)
  - Elements are one-byte ASCII codes for each character
  - No “String” keyword, unlike Java

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>space</td>
</tr>
<tr>
<td>33</td>
<td>!</td>
</tr>
<tr>
<td>34</td>
<td>&quot;</td>
</tr>
<tr>
<td>35</td>
<td>#</td>
</tr>
<tr>
<td>36</td>
<td>$</td>
</tr>
<tr>
<td>37</td>
<td>%</td>
</tr>
<tr>
<td>38</td>
<td>&amp;</td>
</tr>
<tr>
<td>39</td>
<td>’</td>
</tr>
<tr>
<td>40</td>
<td>(</td>
</tr>
<tr>
<td>41</td>
<td>)</td>
</tr>
<tr>
<td>42</td>
<td>*</td>
</tr>
<tr>
<td>43</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>,</td>
</tr>
<tr>
<td>45</td>
<td>-</td>
</tr>
<tr>
<td>46</td>
<td>.</td>
</tr>
<tr>
<td>47</td>
<td>/</td>
</tr>
</tbody>
</table>

ASCII: American Standard Code for Information Interchange
Null-Terminated Strings

- **Example:** “Donald Trump” stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th>68</th>
<th>111</th>
<th>110</th>
<th>97</th>
<th>108</th>
<th>100</th>
<th>32</th>
<th>84</th>
<th>114</th>
<th>117</th>
<th>109</th>
<th>112</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex:</td>
<td>0x44</td>
<td>0x6F</td>
<td>0x6E</td>
<td>0x61</td>
<td>0x6C</td>
<td>0x64</td>
<td>0x20</td>
<td>0x54</td>
<td>0x72</td>
<td>0x75</td>
<td>0x6D</td>
<td>0x70</td>
<td>0x00</td>
</tr>
<tr>
<td>Text:</td>
<td>D</td>
<td>o</td>
<td>n</td>
<td>a</td>
<td>l</td>
<td>d</td>
<td>T</td>
<td>r</td>
<td>u</td>
<td>m</td>
<td>p</td>
<td>\0</td>
<td></td>
</tr>
</tbody>
</table>

- Last character followed by a 0 byte (‘\0’) (a.k.a. “null terminator”)
  - Must take into account when allocating space in memory
  - Note that ‘0’ ≠ ‘\0’ (i.e. character 0 has non-zero value)

- How do we compute the length of a string?
  - Traverse array until null terminator encountered
Endianness and Strings

\[
\text{char } s[6] = "12345";
\]

String literal

IA32, x86-64 (little endian)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>31</td>
<td>31</td>
<td>0x00</td>
</tr>
<tr>
<td>0x01</td>
<td>32</td>
<td>32</td>
<td>0x01</td>
</tr>
<tr>
<td>0x02</td>
<td>33</td>
<td>33</td>
<td>0x02</td>
</tr>
<tr>
<td>0x03</td>
<td>34</td>
<td>34</td>
<td>0x03</td>
</tr>
<tr>
<td>0x04</td>
<td>35</td>
<td>35</td>
<td>0x04</td>
</tr>
<tr>
<td>0x05</td>
<td>00</td>
<td>00</td>
<td>0x05</td>
</tr>
</tbody>
</table>

SPARC (big endian)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>31</td>
<td>31</td>
<td>0x00</td>
</tr>
<tr>
<td>0x01</td>
<td>32</td>
<td>32</td>
<td>0x01</td>
</tr>
<tr>
<td>0x02</td>
<td>33</td>
<td>33</td>
<td>0x02</td>
</tr>
<tr>
<td>0x03</td>
<td>34</td>
<td>34</td>
<td>0x03</td>
</tr>
<tr>
<td>0x04</td>
<td>35</td>
<td>35</td>
<td>0x04</td>
</tr>
<tr>
<td>0x05</td>
<td>00</td>
<td>00</td>
<td>0x05</td>
</tr>
</tbody>
</table>

- Byte ordering (endianness) is not an issue for 1-byte values
  - The whole array does not constitute a single value
  - Individual 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
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

**printf directives:**
- `%p`  Print pointer
- `	`  Tab
- `%x`  Print value as hex
- `\n`  New line
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
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++)
        printf("%p\t0x%.2x\n", start+i, *(start+i));
    printf("\n");
}

void show_int(int x) {
    show_bytes( (char*) &x, sizeof(int));
}
```
show_bytes Execution Example

```c
int a = 12345; // 0x00003039
printf("int a = 12345;\n");
show_int(a); // show_bytes((char *) &a, sizeof(int));
```

- **Result (Linux x86-64):**
  - **Note:** The addresses will change on each run (try it!), but fall in same general range

```
int a = 12345;
0x7ffffff7f71dbc 0x39
0x7ffffff7f71dbd 0x30
0x7ffffff7f71dbe 0x00
0x7ffffff7f71dbf 0x00
```
Summary

- Assignment in C results in value being put in memory location
- Pointer is a C representation of a data address
  - & = “address of” operator
  - * = “value at address” or “dereference” operator
- Pointer arithmetic scales by size of target type
  - Convenient when accessing array-like structures in memory
  - Be careful when using – particularly when casting variables
- Arrays are adjacent locations in memory storing the same type of data object
  - Strings are null-terminated arrays of characters (ASCII)