Review Questions

0) We think of memory as an array of bytes.

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  True
   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? \(2^7 = 128\) bytes
Memory, Data, & Addressing II
CSE 351 Spring 2017

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis

http://xkcd.com/371/
Administrivia

- Start-of-Course survey [Catalyst] due Thursday (3/30)
- Everyone has VM or access to attu? LET US KNOW if not
- Section - Room changes for 9:30 and 10:30 sections
- Lab 0, due Monday (4/3) @ 11:59pm
- Homework 1, due Monday (4/3) @ 11:59pm
- Readings in CSAPP – see schedule
- Office Hours – see schedule – LET US KNOW if you cannot make any of our posted office hours
- Lab 1 – coming soon!
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

int* ptr;

int x = 5;
int y = 2;

ptr = &x;

y = 1 + *ptr;

“Dereference ptr”

What is *(&y) ?

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;”

* is also used with variable declarations
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
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;
```

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

```c
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

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>03 27 D0 3C</td>
</tr>
<tr>
<td>0x04</td>
<td>x</td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>y</td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td>z</td>
</tr>
<tr>
<td>0x24</td>
<td></td>
</tr>
</tbody>
</table>

Declaration: initially contains garbage
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;
x = y + 3;
```

- Get value at y, add 3, store in x

```c
int* z = &y + 3;
```

- Get address of y, “add 3”, store in z

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

& = “address of”
*
= “dereference”

Pointer arithmetic
Pointer Arithmetic

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

```c
int* z = &y + 3;
```
- Get address of `y`, add `3 * sizeof(int)`, store in `z`
  ```c
  &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?

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
</tr>
<tr>
<td>0x01</td>
<td>03</td>
</tr>
<tr>
<td>0x02</td>
<td>27</td>
</tr>
<tr>
<td>0x03</td>
<td>D0</td>
</tr>
<tr>
<td>0x04</td>
<td>3C</td>
</tr>
<tr>
<td>0x05</td>
<td></td>
</tr>
<tr>
<td>0x06</td>
<td></td>
</tr>
<tr>
<td>0x07</td>
<td></td>
</tr>
<tr>
<td>0x08</td>
<td></td>
</tr>
<tr>
<td>0x09</td>
<td></td>
</tr>
<tr>
<td>0x0A</td>
<td></td>
</tr>
<tr>
<td>0x0B</td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
</tr>
<tr>
<td>0x0D</td>
<td></td>
</tr>
<tr>
<td>0x0E</td>
<td></td>
</tr>
<tr>
<td>0x0F</td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
</tr>
<tr>
<td>0x11</td>
<td></td>
</tr>
<tr>
<td>0x12</td>
<td></td>
</tr>
<tr>
<td>0x13</td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
</tr>
<tr>
<td>0x15</td>
<td></td>
</tr>
<tr>
<td>0x16</td>
<td></td>
</tr>
<tr>
<td>0x17</td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
</tr>
<tr>
<td>0x19</td>
<td></td>
</tr>
<tr>
<td>0x1A</td>
<td></td>
</tr>
<tr>
<td>0x1B</td>
<td></td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
</tr>
<tr>
<td>0x1D</td>
<td></td>
</tr>
<tr>
<td>0x1E</td>
<td></td>
</tr>
<tr>
<td>0x1F</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>0x21</td>
<td></td>
</tr>
<tr>
<td>0x22</td>
<td></td>
</tr>
<tr>
<td>0x23</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
</tr>
</tbody>
</table>

& = “address of”
* = “dereference”
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**: `int`
- **name**: `a`
- **number of elements**: `6`

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

- `a[0]` is a name for the array’s address.
- 64-bit example (pointers are 64-bits wide)

```
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48

```
### 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.

---

<table>
<thead>
<tr>
<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>0x0</td>
<td>0x08</td>
<td>0x10</td>
<td>0x18</td>
<td>0x20</td>
<td>0x28</td>
<td>0x30</td>
<td>0x38</td>
</tr>
<tr>
<td>0x40</td>
<td>0x48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- `a[0]`: 5F 01 00 00
- `a[2]`: 01 00 00
- `a[4]`: 5F 01 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;`

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>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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x00</td>
<td>0x08</td>
<td>0x10</td>
<td>0x18</td>
<td>0x20</td>
<td>0x28</td>
<td>0x30</td>
<td>0x38</td>
</tr>
<tr>
<td>AD</td>
<td>OB</td>
<td>00</td>
<td>00</td>
<td>AD</td>
<td>OB</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>5F</td>
<td>01</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>a[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AD</td>
<td>OB</td>
<td>00</td>
<td>00</td>
<td>AD</td>
<td>OB</td>
<td>00</td>
<td>00</td>
</tr>
</tbody>
</table>
```
Arrays in C

**Declaration:**

```c
int a[6];
```

**Indexing:**

```c
a[0] = 0x015f;

a[5] = a[0];
```

**No bounds checking:**

```c
a[6] = 0xBAD;

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

**Pointers:**

```c
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
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

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

Equivalent:

```
p[1] = 0xB;
*p + 1 = 0xB;
p = p + 2;
```
Arrays in C

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

Indexing:  
\[
\begin{align*}
a[0] &= 0x015f; \\
a[5] &= a[0]; \\
\end{align*}
\]

No bounds  
\( a[6] = 0xBAD; \)

checking:  
\( a[-1] = 0xBAD; \)

Pointers:  
\[
\begin{align*}
\text{int }* p; \\
\text{equivalent } &\begin{cases} \\
p = a; \quad &\text{equivalent } \\
p = &\text{\&}a[0]; \\
* p = 0xA; \end{cases} \\
\end{align*}
\]

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

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

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

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

\[
\begin{align*}
p[1] &= 0xB; \\
*(p+1) &= 0xB; \\
p &= p + 2; \\
* p &= a[1] + 1; \\
\end{align*}
\]
Fill in the blanks!

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

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

No bounds \(a[6] = 0xBAD;\)

checking: \(a[-1] = 0xBAD;\)

Pointers: \(\text{int* } p;\)

\(p = a;\)  
\(p = \&a[0];\)  
\(*p = 0xA;\)

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

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

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

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

\(*p = a[1] + 1;\)
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;
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
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 Code</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:** “Life is good” stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>0x4c</td>
<td>Life</td>
</tr>
<tr>
<td>105</td>
<td>0x69</td>
<td>i</td>
</tr>
<tr>
<td>102</td>
<td>0x66</td>
<td>e</td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td>s</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>0x69</td>
<td>is</td>
</tr>
<tr>
<td>115</td>
<td>0x73</td>
<td>g</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>103</td>
<td>0x67</td>
<td>o</td>
</tr>
<tr>
<td>111</td>
<td>0x6f</td>
<td>d</td>
</tr>
<tr>
<td>111</td>
<td>0x6f</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0x64</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0x00</td>
<td>\0</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

```c
char s[6] = "12345";
```

- **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 \t 0x%02x\n", start+i, *(start+i));
    printf("\n");
}
```

**printf directives:**
- `%p` Print pointer
- `	` Tab
- `%x` Print value as hex
- `
` 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

```c
int a = 12345;
0x7fffb7f71dbc 0x39
0x7fffb7f71dbd 0x30
0x7fffb7f71dbe 0x00
0x7fffb7f71dbf 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)