Memory, Data, & Addressing II
CSE 351 Winter 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Jonathan Chen
Justin Johnson
Porter Jones
Josie Lee
Jeffery Tian
Callum Walker
Eddy (Tianyi) Zhou

http://xkcd.com/138/
Administrivia

- Lab 0 due today @ 11:59 pm
  - *You will be revisiting this program throughout this class!*

- hw2 due Monday, hw3 due Wednesday @ 11:00 am
  - Autograded, unlimited tries, no late submissions

- Lab 1a released today, due next Friday (1/17)
  - Pointers in C
  - Reminder: last submission graded, *individual* work
Late Days

- You are given 5 late lab days for the whole quarter
  - Tokens can only apply to Labs
  - No benefit to having leftover tokens
- Count lateness in days (even if just by a second)
  - Special: weekends count as one day
  - No submissions accepted more than two days late
- Late penalty is 20% deduction of your score per day
  - Only late labs are eligible for penalties
  - Penalties applied at end of quarter to maximize your grade
- Use at own risk – don’t want to fall too far behind
  - Intended to allow for unexpected circumstances
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 [T]
   b) 64 bits is the size of an integer [F]
   c) 64 bits is the width of a register [T]

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 6 bits, the address space is how many bytes?
   - Represent $2^6$ things $\rightarrow 2^6$ addresses $\rightarrow 2^6$ bytes of data [64 B]
Memory, Data, and Addressing

- Representing information as bits and bytes
  - Binary, hexadecimal, fixed-widths
- Organizing and addressing data in memory
  - Memory is a byte-addressable array
  - Machine “word” size = address size = register size
  - Endianness – ordering bytes in memory
- Manipulating data in memory using C
  - Assignment
  - Pointers, pointer arithmetic, and arrays
- 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 initializes 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)`?

* is also used with variable declarations.

returns value stored in `y` (equivalent to just using `y`)
Assignment in C

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

```
int x, y;

x is at address 0x04, y is at 0x18
```
### Assignment in C

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

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

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

#### 32-bit example
(pointers are 32-bits wide)  
little-endian

<table>
<thead>
<tr>
<th>Address</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x04</td>
<td>00</td>
<td>01</td>
<td>29</td>
<td>F3</td>
</tr>
<tr>
<td>0x08</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`x` at 0x04 = 00 01 29 F3  
`y` at 0x18 = 01 00 00 00
Assignment in C

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

```c
int x, y;
int (4 bytes)
x = 0;
```

<table>
<thead>
<tr>
<th>Location</th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>y</td>
<td>01</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>y</td>
</tr>
</tbody>
</table>
Assignment in C

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

- `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 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 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”
Assignment in C

- left-hand side = right-hand side;
  - LHS must evaluate to a 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; // expect 0x1b`
  - Get address of y, “add 3”, store in z

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

<table>
<thead>
<tr>
<th></th>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0x00</td>
<td>0x04</td>
<td>0x08</td>
<td>0x0C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pointer arithmetic
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?

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

`&` = “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`
  - The target of a pointer is also a location
- `*z = y;`
  - Get value of `y`, put in address stored in `z`
Arrays in C

Declaration: `int a[6];` // `&a` is 0x10

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

`a` (array name) returns 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` (array name) returns the array’s address.  
`&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;`

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

- `a` (array name) returns the array’s address.
- `&a[i]` is the address of `a[0]` plus `i` times the element size in bytes.

```
0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7
0x8 0x9 0xA 0xB 0xC 0xD 0xE 0xF
```

```
0x00 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
```

```
0x00
0x08
0x09
0x0A
0x0B
0x0C
0x0D
0x0E
0x0F
```

```
0x00
0x08
0x09
0x0A
0x0B
0x0C
0x0D
0x0E
0x0F
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
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` (array name) returns the array’s address
- `&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;`  
- Equivalent: 
  - `p = a;`
  - `p = &a[0];`
  - `*p = 0xA;`

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

- `a` (array name) returns the array’s address
- `&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;`

Pointer arithmetic: 
- `p = p + 2;`
- `p[1] = 0x18;`
- `p = p + 2;`
- `p = p + 2;`
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` (array name) returns the array’s address
- `&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?

- Vote at [http://polleve.com/reax](http://polleve.com/reax)

```c
void main()
{
    int a[] = {5, 10};
    int* p = a;
    p = p + 1;
    *p = *p + 1;
}
```

- (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</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>Hex</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>0x44</td>
<td>Donald</td>
</tr>
<tr>
<td>111</td>
<td>0x6F</td>
<td>a</td>
</tr>
<tr>
<td>110</td>
<td>0x6E</td>
<td>l</td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td>d</td>
</tr>
<tr>
<td>108</td>
<td>0x6C</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>0x64</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>0x54</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>0x72</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>0x75</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>0x6D</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td>0x70</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

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

C (char = 1 byte)
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
- `
`  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 x = 12345; // 0x000003039
printf("int x = %d;\n", x);
show_int(x);    // show_bytes((char *) &x, sizeof(int));
```

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

```c
int x = 12345;
0x7ffffff71dbc 0x39
0x7ffffff71dbd 0x30
0x7ffffff71dbe 0x00
0x7ffffff71dbf 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)
Assignment in C - Handout

- left-hand side = right-hand side;
  - LHS must evaluate to a location
  - RHS must evaluate to a value
  - Store RHS value at LHS location

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

x = 0;
y = 0x3CD02700;
x = y + 3;
*z = y;
```

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

& = “address of”
*
= “dereference”
Arrays in C - Handout

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

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

`a` (array name) returns the array’s address

`&a[i]` is the address of `a[0]` plus `i` times the element size in bytes