Memory, Data, & Addressing II
CSE 351 Autumn 2019

Instructor:
Justin Hsia

Teaching Assistants:
Andrew Hu
Antonio Castelli
Cosmo Wang
Diya Joy
Ivy Yu
Kaelin Laundry
Maurice Montag
Melissa Birchfield
Millicent Li
Suraj Jagadeesh

http://xkcd.com/138/
Administrivia

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

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

- Lab 1a released today, due next Monday (10/7)
  - Pointers in C
  - Reminder: last submission graded, *individual* work
Late Days

- You are given 5 late day tokens 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 (32 bits = 4 bytes)
   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?
   \[ 2^6 \text{ bytes of data} = 64 \text{ B} \]
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;
```

* is also used with variable declarations

- 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)*?  

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

8 bytes
4 bytes
4 bytes
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

<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>A7 00 32 00</td>
</tr>
<tr>
<td>0x04</td>
<td>00 01 29 F3</td>
</tr>
<tr>
<td>0x08</td>
<td>EE EE EE EE</td>
</tr>
<tr>
<td>0x0C</td>
<td>FA CE CA FE</td>
</tr>
<tr>
<td>0x10</td>
<td>26 00 00 00</td>
</tr>
<tr>
<td>0x14</td>
<td>00 00 10 00</td>
</tr>
<tr>
<td>0x18</td>
<td>01 00 00 00</td>
</tr>
<tr>
<td>0x1C</td>
<td>FF 00 F4 96</td>
</tr>
<tr>
<td>0x20</td>
<td>DE AD BE EF</td>
</tr>
<tr>
<td>0x24</td>
<td>00 00 00 00</td>
</tr>
</tbody>
</table>

Current state of memory
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

```
0x00 0x01 0x02 0x03
0x04 00 01 29 F3
0x08 00 00 00 00
0x0C
0x10
0x14
0x18 01 00 00 00
0x1C
0x20
0x24
```

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

- \texttt{int x, y;}
- \texttt{x = 0;}
- \texttt{\&x = 0x00}

\texttt{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

- \textbf{int} \ x, \ y;
- \textbf{x} = 0;
- \textbf{y} = 0x3CD02700;

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`

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`
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`
- `z = y;`
  - Get value of `y`, put in address stored in `z`

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

`&` = “address of”
`*` = “dereference”
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
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;`
- `p = a;`
- `p = &a[0];`
- `*p = 0xA;`
- `p[1] = 0xB;`
- `*(p+1) = 0xB;`
- `p = p + 2;`
- `a[6] = 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

**Diagram:**
- Array indices and memory layout
- Pointer arithmetic
- Array indexing = address arithmetic (both scaled by the size of the type)

**Formula:**
- `a + 2 * sizeof(int) = 0x18`
Arrays in C

Declaration:  \texttt{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:  \texttt{int* p;}

\[
\begin{align*}
\text{equivalent} & \\
\text{p} & = a; & a[0] & \\
\text{p} & = \&a[0]; & a[2] & \\
*\text{p} & = 0xA; & a[4] & \\
p[1] & = 0xB; & \\
*(p+1) & = 0xB; & \\
p & = p + 2; & \\
\end{align*}
\]

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

\[
\begin{align*}
\text{equivalent} & \\
\text{p} & = a[1] + 1; & \\
\end{align*}
\]

\text{store at } 0x18  
\( *p = 0x\text{B} + 1 = 0xC \)

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
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?


```c
void main() {
    int a[] = {5, 10};
    int* p = a;
    p = p + 1; // sizeof(int) = 4
    *p = *p + 1;
}
```

<table>
<thead>
<tr>
<th></th>
<th>After Line 4</th>
<th>After Line 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>*p</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>a[0]</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>a[1]</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

- (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
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>Donald</td>
<td>al</td>
<td>d</td>
<td>T</td>
<td>r</td>
<td>u</td>
<td>m</td>
<td>p</td>
<td>\0</td>
<td></td>
<td></td>
<td></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 \neq \\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

---

<table>
<thead>
<tr>
<th>IA32, x86-64 (little-endian)</th>
<th>SPARC (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 31</td>
<td>0x00 31</td>
</tr>
<tr>
<td>0x01 32</td>
<td>0x01 32</td>
</tr>
<tr>
<td>0x02 33</td>
<td>0x02 33</td>
</tr>
<tr>
<td>0x03 34</td>
<td>0x03 34</td>
</tr>
<tr>
<td>0x04 35</td>
<td>0x04 35</td>
</tr>
<tr>
<td>0x05 00</td>
<td>0x05 00</td>
</tr>
</tbody>
</table>

0x31 = 49 decimal = ASCII ‘1’
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"); // format string
}

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

```c
int x = 12345; // 0x00003039
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;
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)