Memory, Data, & Addressing II
CSE 351 Spring 2020

Instructor:
Ruth Anderson

Teaching Assistants:
Alex Olshanskyy
Rehaan Bhimani
Callum Walker
Chin Yeoh
Diya Joy
Eric Fan
Edan Sneh
Jonathan Chen
Jeffery Tian
Millicent Li
Melissa Birchfield
Porter Jones
Joseph Schafer
Connie Wang
Eddy (Tianyi) Zhou

http://xkcd.com/138/
Administrivia

- Assignments – Nothing **Due** this week!
- Lecture Video to watch before Monday (4/06)
  - Not a full lecture, but part of one to help us retain time in lectures for small group interaction.
  - Posted later today, should be watched before lecture on Monday
- Pre-Course Survey, hw0, hw1, hw2 due Mon (4/06) – 11:59pm
- Lab 0 due Tuesday (4/07) – 11:59pm
- hw3 due Wednesday (4/08) – **11am**
  - Autograded, unlimited tries, no late submissions
- Lab 1a released today, due next Monday (4/13)
  - Pointers in C
  - Reminder: last submission graded, *individual* work
- Feedback for today: [https://catalyst.uw.edu/webq/survey/rea2000/387932](https://catalyst.uw.edu/webq/survey/rea2000/387932)
Late Days

- You are given 7 late days for the whole quarter
  - Late days can only apply to Labs & Unit Summaries
  - No benefit to having leftover late days
- 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 work is 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
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

```
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`;

```
*(&y)
```

- What is \*(&y) ?
- Returns value stored in `y` (equivalent to just using `y`.)

* is also used with variable declarations
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
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

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

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

```c
int x, y;

x = 0;
```

& = “address of”
* = “dereference”

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

- `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`
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;
x = 0;
y = 0x3CD02700;
x = y + 3;

int* z;
```

- Get value at y, add 3, store in x
- z is at address 0x20
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;
x = 0;
y = 0x3CD02700;
x = y + 3;
```

- Get value at \( y \), add 3, store in \( x \)

```c
int* z = &y + 3; // expect 0x1b
```

- 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`

```
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)

- Element type
- Element name
- Number of elements

4 bytes each
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

\[ \text{Address of } a[0] + i \times \text{Element Size} \]

```
0x00 0x08 0x10 0x18 0x20 0x28 0x30 0x38 0x40 0x48
0x15F 0x15F 0x5FBAD 0x5FBAD 0xBAD 0xBAD
```

**Diagram:**
- `a[0]` is at `0x15F`
- `a[2]` is at `0x18F` (0x15F + 0x008)
- `a[4]` is at `0x20F` (0x15F + 0x010)
- `a[6]` is at `0x28F` (0x15F + 0x018)
- `a[-1]` is at `0xBAD`
Arrays in C

Declaration: `int a[6];`

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

No bounds checking:
- `a[6] = 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

```
   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
   AD  OB  00  00  0A  00  00  00  5F  01  00  00
   00  00  00  00  00  00  00  00  10  00  00  00
```

Equivalent:
- `p` 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:
- `0x10 + 2 → 0x18`
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)

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

- Vote at [http://pollev.com/rea](http://pollev.com/rea)

```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

| ASCII | Tell | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| 32    | space| 48 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 33    | !    | 49 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 34    | "    | 50 | 2 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 35    | #    | 51 | 3 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 36    | $    | 52 | 4 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 37    | %    | 53 | 5 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 38    | &    | 54 | 6 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 39    | '    | 55 | 7 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 40    | (    | 56 | 8 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 41    | )    | 57 | 9 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 42    | *    | 58 | 0 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 43    | +    | 59 | ; |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 44    | ,    | 60 | < |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 45    | -    | 61 | = |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 46    | .    | 62 | > |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 47    | /    | 63 | ? |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

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></td>
</tr>
<tr>
<td>110</td>
<td>0x6E</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td></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>
</tbody>
</table>
  | 0       | 0x00 | \\

  - 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

---

**Hexadecimal Values:***

<table>
<thead>
<tr>
<th>IA32, x86-64</th>
<th>SPARC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(little-endian)</td>
<td>(big-endian)</td>
</tr>
<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>

- **Hexadecimal Values:**
  - 31 (49 decimal) = ASCII '1'
  - 0x31 = 49 decimal = ASCII '1'

- **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	0x%.2x\n", start+i, *(start+i));
    printf("\n");
}
```

printf directives:
- %p  Print pointer
- \t  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; // 0x000003039
printf("int x = %d;\n", x);
show_int(x);   // show_bytes((char *) &x, sizeof(int));
```

الف: 

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

```
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)