Memory, Data, & Addressing II
CSE 351 Winter 2018

Instructor:
Mark Wyse

Teaching Assistants:
Kevin Bi
Parker DeWilde
Emily Furst
Sarah House
Waylon Huang
Vinny Palaniappan

http://xkcd.com/138/
Administrivia

- Lab 0 due today @ 11:59pm
  - You will be revisiting this program throughout this class!
- Homework 1 due Wednesday
  - Reminder: autograded, 20 tries, no late submissions
- Lab 1 released today
  - Prelim due Jan. 15
  - Due Jan. 19
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

- A pointer is a variable that holds an address
- Pointers are declared similarly to other variables in C
  - Type (e.g., `int *`)
  - Name (e.g., `ptr`)
  - Declaration, Initialization, Assignment
- Type is specified using one (or more) `*` after some type T
  - `int *ptr;`
  - `struct Scores *s;`
  - `double **dPtr;`
- Operators
  - `& = “address of” operator`
  - `* = “dereference” operator, or “value at address”`
Assignment in C

- A variable is represented by a memory location
- Declaration ≠ initialization (initially holds “garbage”)
- Left-Hand Side = Right-Hand Side
  - = operator
  - LHS is a memory location
  - RHS is a value (could be an address)
Assignment in C

- `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

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

- `&` = “address of”
- `*` = “dereference”

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

Little endian!
Assignment in C

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

- **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

### Example Memory Layout

<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></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>00 27 D0 3C</td>
</tr>
<tr>
<td>0x1C</td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
</tr>
<tr>
<td>0x24</td>
<td></td>
</tr>
</tbody>
</table>

---

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 = &x;`
  - `&x = 0x00000004`
Assignment in C

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
  - Get value at `y`, add 3, store in `x`
- `int *z = &x;`
  - `&x = 0x00000004`

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 = &x;
  - &x = 0x00000004
- int y = *z + 1;

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 = &x;`
  - `&x = 0x00000004`
- `int y = *z + 1;`
  - `y = 0x3CD02704`

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

& = “address of”
*
= “dereference”
### Pointer Arithmetic in C

1. `int x, y;`
2. `x = 0;`
3. `y = 0x3CD02700;`
4. `x = y + 3;`
   - Get value at `y`, add 3, store in `x`
5. `int *z;`
   - `z` is at address 0x20

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

- `&` = “address of”
- `*` = “dereference”

```
0x00 0x01 0x02 0x03
0x00 03 27 D0 3C
0x04 0x08 0x0C 0x10 0x14 0x18 0x1C 0x20 0x24

X

Y

Z
```
Pointer Arithmetic 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 3”, store in `z`

**32-bit example**
* & = “address of”
  * = “dereference”

```
0x00 0x01 0x02 0x03
0x00 03 27 D0 3C
0x04 0x08 0x0C 0x10 0x14 0x18 0x1C 0x20 0x24
DE AD BE EF
```
Pointer Arithmetic in C

❖ Pointer arithmetic is scaled by the size of the pointer’s target data 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*
Pointer Arithmetic 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`

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;`
  - 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`
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` is a name for the array’s address.

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

Declaration: \texttt{int a[6];}

Indexing: \texttt{a[0] = 0x015f;}
\texttt{a[5] = a[0];}

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

\texttt{a} is a name for the array’s address

The address of \texttt{a[i]} is the address of \texttt{a[0]} plus \texttt{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` 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;`
  - equivalent:
    - `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;`

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

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

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

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

0x00  AD  0B  00  00
0x08  0A  00  00  00  0B  00  00  00
0x10  0C  00  00  00  0D  00  00  00
0x18  5F  01  00  00  0E  00  00  00
0x20  AD  0B  00  00  0F  00  00  00
0x28  00  00  00  00  00  00  00  00
0x30  00  00  00  00  00  00  00  00
0x38  00  00  00  00  00  00  00  00
0x40  00  00  00  00  00  00  00  00
0x48  00  00  00  00  00  00  00  00
```

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?

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

<table>
<thead>
<tr>
<th></th>
<th>Data (decimal)</th>
<th>Address (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>a[1]</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>100</td>
<td></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

| ASCII | \n|---|---|---|---|---|---|---|---|---|---|---|---|
| 32 | space | 48 | 0 | 64 | @ | 80 | P | 96 | ` | 112 | p |
| 33 | ! | 49 | 1 | 65 | A | 81 | Q | 97 | a | 113 | q |
| 34 | " | 50 | 2 | 66 | B | 82 | R | 98 | b | 114 | r |
| 35 | # | 51 | 3 | 67 | C | 83 | S | 99 | c | 115 | s |
| 36 | $ | 52 | 4 | 68 | D | 84 | T | 100 | d | 116 | t |
| 37 | % | 53 | 5 | 69 | E | 85 | U | 101 | e | 117 | u |
| 38 | & | 54 | 6 | 70 | F | 86 | V | 102 | f | 118 | v |
| 39 | ’ | 55 | 7 | 71 | G | 87 | W | 103 | g | 119 | w |
| 40 | ( | 56 | 8 | 72 | H | 88 | X | 104 | h | 120 | x |
| 41 | ) | 57 | 9 | 73 | I | 89 | Y | 105 | l | 121 | y |
| 42 | * | 58 | : | 74 | J | 90 | Z | 106 | j | 122 | z |
| 43 | + | 59 | ; | 75 | K | 91 | [ | 107 | k | 123 | { |
| 44 | , | 60 | < | 76 | L | 92 | \ | 108 | l | 124 | | |
| 45 | - | 61 | = | 77 | M | 93 | ] | 109 | m | 125 | } |
| 46 | . | 62 | > | 78 | N | 94 | ^ | 110 | n | 126 | ~ |
| 47 | / | 63 | ? | 79 | O | 95 | _ | 111 | o | 127 | del |

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

❖ **Example:** “Luke and Leia” stored as a 14-byte array

<table>
<thead>
<tr>
<th>Decimal:</th>
<th>76</th>
<th>117</th>
<th>107</th>
<th>101</th>
<th>32</th>
<th>97</th>
<th>110</th>
<th>100</th>
<th>32</th>
<th>76</th>
<th>101</th>
<th>105</th>
<th>97</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Hex:</em></td>
<td>0x4c</td>
<td>0x75</td>
<td>0x6b</td>
<td>0x65</td>
<td>0x20</td>
<td>0x61</td>
<td>0x6e</td>
<td>0x64</td>
<td>0x20</td>
<td>0x4c</td>
<td>0x65</td>
<td>0x69</td>
<td>0x61</td>
<td>0x00</td>
</tr>
<tr>
<td><em>Text:</em></td>
<td>Luke</td>
<td>and</td>
<td>and</td>
<td>Leia</td>
<td>\0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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’ ≠ ‘\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

**String literal**

<table>
<thead>
<tr>
<th>IA32, x86-64 (little-endian)</th>
<th>SPARC (big-endian)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 31</td>
<td>31 0x00 '1'</td>
</tr>
<tr>
<td>0x01 32</td>
<td>32 0x01 '2'</td>
</tr>
<tr>
<td>0x02 33</td>
<td>33 0x02 '3'</td>
</tr>
<tr>
<td>0x03 34</td>
<td>34 0x03 '4'</td>
</tr>
<tr>
<td>0x04 35</td>
<td>35 0x04 '5'</td>
</tr>
<tr>
<td>0x05 00</td>
<td>00 0x05 '\0'</td>
</tr>
</tbody>
</table>

0x31 = 49 decimal = ASCII ‘1’
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)
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
- `\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");
}

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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fffb7f71dbc</td>
<td>0x39</td>
</tr>
<tr>
<td>0x7fffb7f71dbd</td>
<td>0x30</td>
</tr>
<tr>
<td>0x7fffb7f71dbe</td>
<td>0x00</td>
</tr>
<tr>
<td>0x7fffb7f71dbf</td>
<td>0x00</td>
</tr>
</tbody>
</table>