Once upon a time in the west, main theme

*Ennio Morricone*
Thanks for the feedback!
Syllabus Update

- Originally the syllabus had grades for doing the section worksheets…
  - I’m just giving everyone 100%

- CAVEAT:
  - Assess how you feel about the material!
  - Attend section (and/or watch recordings) if you feel that more review would be helpful!
  - Err on attending; people tend to be pretty bad at assessing their knowledge.
It’s about to get really hot in Seattle...
Gentle Reminders!

- Lab 0 & hw1 due Tonight (6/25) – 8pm
- hw2 due Monday (6/28) – 10am
- hw3 due Wednesday (6/30) – 10am
- Lab 1a released today, due a week from today (7/2)
  - Pointers in C
  - Reminder: last submission graded, *individual* work
Gentle and Loving Reminders!

- Lab 1a’s released!
  - Workflow:
    1) Edit `pointer.c`
    2) Run the Makefile (`make`) and check for compiler errors & warnings
    3) Run `ptest` (`./ptest`) and check for correct behavior
    4) Run rule/syntax checker (`python dlc.py`) and check output
  - Due Friday 7/2 at 8pm
    - Lab 1b will be released next week, due 7/9
    - Structured so you shouldn’t need to work over the holiday
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
- The late penalty for using more than 7 late days is a 20% deduction of your score per excess day
  - Only late work is eligible for penalties
- Intended to allow for unexpected circumstances
Emoji!

How are you feeling about Lab 0?
It's due today!
Emoji!
How are you feeling about Unit Summary 1?
We’re here to help!

- If you’re lost or don’t know where to start, come ask for help!
Lab Reflections

- All subsequent labs (after Lab 0) have a “reflection” portion
  - The Reflection questions can be found on the lab specs and are intended to be done after you finish the lab
  - You will type up your responses in a .txt file for submission on Gradescope
  - We’ll read these and give feedback

- We’re using these as a way to check your understanding of what’s going on in lab
First Floor: Data

- How do we represent data (strings, numbers) computationally?
- What limits exist? Why?
- What values were encoded into data representations?
- What was prioritized? Why?

- Today: Memory & C!
Breakouts!
What have you heard about C?
How do you feel about C?
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
Learning Objectives

- At the end of this lecture, you should be able to...
  - Access and manipulate memory in C, using the * and & operators
  - Declare arrays in C, and draw the array layout in memory using a “box-and-arrow” diagram
  - Convert between array and pointer representations
  - Explain why pointer arithmetic is dangerous
  - View data representations with show_bytes()
  - Explain the values of the C language, and some of their origins
Reading Review

- Terminology:
  - address-of operator (&), dereference operator (*), NULL
  - box-and-arrow memory diagrams
  - pointer arithmetic, arrays
  - C string, null character, string literal
Review Questions

- `int x = 351;
  char *p = &x;
  int ar[3];`

- How much space does the variable `p` take up?
  - 1 byte
  - 2 bytes
  - 4 bytes
  - 8 bytes
  - Help!!

- Which of the following expressions evaluate to an address?
  - `x + 10`
  - `p + 10`
  - `&x + 10`
  - `*(&p)`
  - `ar[1]`
  - `&ar[2]`
  - Help!!
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)*?
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”)

```c
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 location
  - RHS must evaluate to a value (could be an address)
  - Store RHS value at LHS location

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

<table>
<thead>
<tr>
<th>0x00</th>
<th>0x01</th>
<th>0x02</th>
<th>0x03</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>*</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>03</td>
<td>27</td>
<td>D0</td>
<td>3C</td>
</tr>
<tr>
<td>0x00</td>
<td>0x04</td>
<td>0x08</td>
<td>0x0C</td>
</tr>
<tr>
<td>0x10</td>
<td>0x14</td>
<td>0x18</td>
<td>0x1C</td>
</tr>
<tr>
<td>0x20</td>
<td>0x24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\& = “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;`
  - Get address of `y`, “add 3”, store in `z`
Pointer Arithmetic

- Pointer arith 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!!
  - Very easy to make mistakes, if you’re not careful!
  - 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 `y`'s address, 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 `y`’s address, add `12`, store in `z`
- `*z = y;`
  - Get value of `y`, put in address stored in `z`
How are we feeling about assignment?
Arrays in C

Declaration:

```c
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` (array name) returns the array’s address.

**64-bit example**

(pointers are 64-bits wide)

<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

![Memory Diagram]

- `0x00`
- `0x08`
- `0x10`
- `0x18`
- `0x20`
- `0x28`
- `0x30`
- `0x38`
- `0x40`
- `0x48`
- `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;```

```equiv```lent: ```
p = a;

p = &a[0];

*p = 0xA;
```n
ter
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 \[
  \begin{aligned}
  p &= a; \\
  p &= &a[0]; \\
  *p &= 0xA;
  \end{aligned}
  \]

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

- `p[1] = 0xBB;`
- `*(p+1) = 0xBB;`
- `p = p + 2;`

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:**\[\textbf{int} \ a[6];\]

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

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

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

**Pointers:** \[\textbf{int}* \ p;\]

Equivalent:
\[
\begin{align*}
    p &= a; \\
    p &= &\&a[0]; \\
    *p &= 0xA;
\end{align*}
\]

Array indexing = address arithmetic (both scaled by the size of the type)
\[
\begin{align*}
    p[1] &= 0xB; \\
    *(p+1) &= 0xB;
\end{align*}
\]

\[p = p + 2;\]

\[*p = a[1] + 1;\]
How are we feeling about arrays?
Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5? (😊 == Help!)

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

<table>
<thead>
<tr>
<th>Address (decimal)</th>
<th>Data (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>a[0] 5, a[1] 10</td>
</tr>
<tr>
<td>0x10</td>
<td>p 0x10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address (decimal)</th>
<th>Data (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x11</td>
<td>a[0] 5, a[1] 10</td>
</tr>
<tr>
<td>0x14</td>
<td>a[0] 5, a[1] 10</td>
</tr>
<tr>
<td>0x10</td>
<td>a[0] 5, a[1] 10</td>
</tr>
</tbody>
</table>

- Line 3: `int* p = a;` assigns the address of `a[0]` to `p`.
  - After Line 3: `0x10` (address of `a[0]`), `5` (value of `a[0]`), `10` (value of `a[1]`).
- Line 4: `p = p + 1;` updates `p` to point to `a[1]`.
- Line 5: `*p = *p + 1;` updates the value at `p` (which is now `a[1]`) by 1.
Representing strings

- C-style string stored a byte array (**char***)
  - Elements are one-byte **ASCII** codes for each character
  - No “String” keyword, unlike Java
Null-Terminated Strings

- **Ex:** “Ice Creamery” stored as a 13-byte array

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>0x49</td>
<td>Ice</td>
</tr>
<tr>
<td>99</td>
<td>0x63</td>
<td>c</td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td>e</td>
</tr>
<tr>
<td>32</td>
<td>0x20</td>
<td>Cre</td>
</tr>
<tr>
<td>67</td>
<td>0x43</td>
<td>ame</td>
</tr>
<tr>
<td>114</td>
<td>0x72</td>
<td>rye</td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td>y</td>
</tr>
<tr>
<td>97</td>
<td>0x61</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td>0x6d</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>0x65</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>0x72</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>0x79</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")
  - Need to remember when allocating memory!
  - Note that '0' ≠ '\0' (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";`

String literal

- **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
  - No need to order bytes with just one byte!

<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’`
- `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
- `\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

```c
int x = 12345;
0x7ffffff7f71dbc 0x39
0x7ffffff7f71dbd 0x30
0x7ffffff7f71dbf 0x00
```
Summary

- Assignment in C: Location = Value
- 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 with array-like structures in memory
  - Be careful – 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)
What values were embedded in the C language?
C language (1978)

- Created in 1972, “standardized” in 1978
  - Goal of writing Unix (precursor to Linux/OSX)
  - Different time, performance/resource limits
- Explicit Goals:
  - Portability, performance (better than B, it’s C!)
What have you heard about C?
How do you feel about C?
Principles of C, viewed today

- “Since C is relatively small, it can be described in small space, and learned quickly.”
- "Only the bare essentials”
- “Close to the hardware”
- “Shows what’s really happening”
- “No one to help you”
- “You’re on your own”
- “I know what I’m doing, get out of my way”
Principles of C, viewed today

- **Minimalist:**
  - "Since C is relatively small, it can be described in small space, and learned quickly."
  - "Only the bare essentials"

- **Rugged:**
  - "Close to the hardware"
  - "Shows what’s really happening"

- **Individualistic**
  - "No one to help you"
  - "You’re on your own"
  - "I know what I’m doing, get out of my way"
Minimalism, Rugged, Individualistic... Wranglers!
Wranglers in the Wild West

- American Frontierism (~1800 – 1890)
  - Vast expansion westward, from original 13 east coast colonies to pacific ocean

- Manifest Destiny
  - Burgeoning theory that White Americans were “destined” to connect from coast to coast
  - Cultural phenomenon, Indigenous genocide
Manifest Destiny

John Gast, *American Progress*, 1872
Immortalized in Popular Culture
Replicated in Computing Culture
Replicated in Computing Culture

>d
Gas Room
This is a small room which smells strongly of coal gas. There is a short climb up some stairs and a narrow tunnel leading east.
Oh dear. It appears that the smell coming from this room was coal gas. I would have thought twice about carrying flaming objects in here.

** B00000000000M **

***** You have died *****

As you take your last breath, you feel relieved of your burdens. The feeling passes as you find yourself before the gates of Hell, where the spirits jeer at you and deny you entry. Your senses are disturbed. The objects in the dungeon appear indistinct, bleached of color, even unreal.

Entrance to Hades
You are outside a large gateway, on which is inscribed

    Abandon every hope all ye who enter here!

The gate is open; through it you can see a desolation, with a pile of mangled bodies in one corner. Thousands of voices, lamenting some hideous fate, can be

**MORE**
Replicated in Computing Culture

>d
Gas Room
This is a small room which smells strongly of decay.
Go up some stairs and a narrow tunnel leading east.
Oh dear. It appears that the smell coming from
the other rooms have thought twice about carrying flaming objects.

** BODDDDDDDDDDOOM **

**** You have died ****

As you take your last breath, you feel relief.
You pass the gate as you find yourself before the gates of
Hades. You are outside a large gateway, on which is inscribed:

Abandon every hope all ye who enter here!

The gate is open; through it you can see a desolate scene.
Thou hast brought death to a thousand bodies in one corner. Thousands of voices, lamenting:

**MORE**
Principles of C, viewed today

- Minimalist, Rugged, Individualist
  - Cowboys, explorers, adventurers
- I’d argue “hegemonically masculine”
  - Or, what most men in 1970s unconsciously embodied
  - See “real programmers use__/write in __”
- It’s also colonialist!
  - ASCII emphasizes English over other languages
  - C emphasizes
- Apparently Minimalist == Easy to learn?
- The 1970s were a weird time…
Why are we still talking about it?

TIOBE Programming Community Index
Source: www.tiobe.com
Takeaways

- **C: Minimalistic, Rugged, Individualistic**
  - Embodied what was culturally valued at the time!
  - Frontierism! Moon landing was 1969!

- Explore the digital frontier!
  - Only carry the essentials!
  - american frontierism!
  - Manifest destiny (1800 – 1890), colonialism, genocide
  - Glorified in popular culture: westerns, video games

- K&R didn’t mean to do harm!
  - But, they didn’t question the values glorified by society
Ideology: You don’t even need to ask
Contrast with: “Best”, “better”, “more important”
“We shape our tools, and thereafter, our tools shape us”

1967

“Reification”, if you want a single word. To make the abstract concrete.

Computing is a tool, but a tool built by a distinctly non-neutral society! We’ve always had values!
C’s like camping!
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

- `int x, y;`
- `x = 0;`
- `y = 0x3CD02700;`
- `x = y + 3;`
- `int* z = &y + 3;`
- `*z = y;`
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;`
- `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
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
   b) 64 bits is the size of an integer
   c) 64 bits is the width of a register

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?