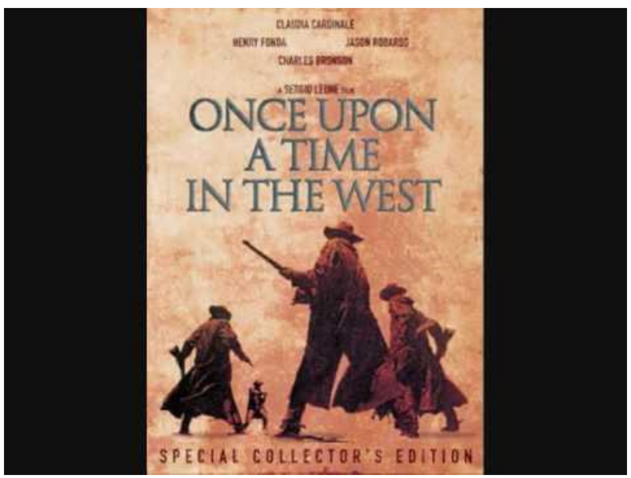
Memory, Data, & Addressing II

CSE 351 Summer, 2021

Instructor:
Mara Kirdani-Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobes
Tim Mandzyuk



Once upon a time in the west, main theme
Ennio Morricone

32-bit addresses

http://xkcd.com/138/

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!

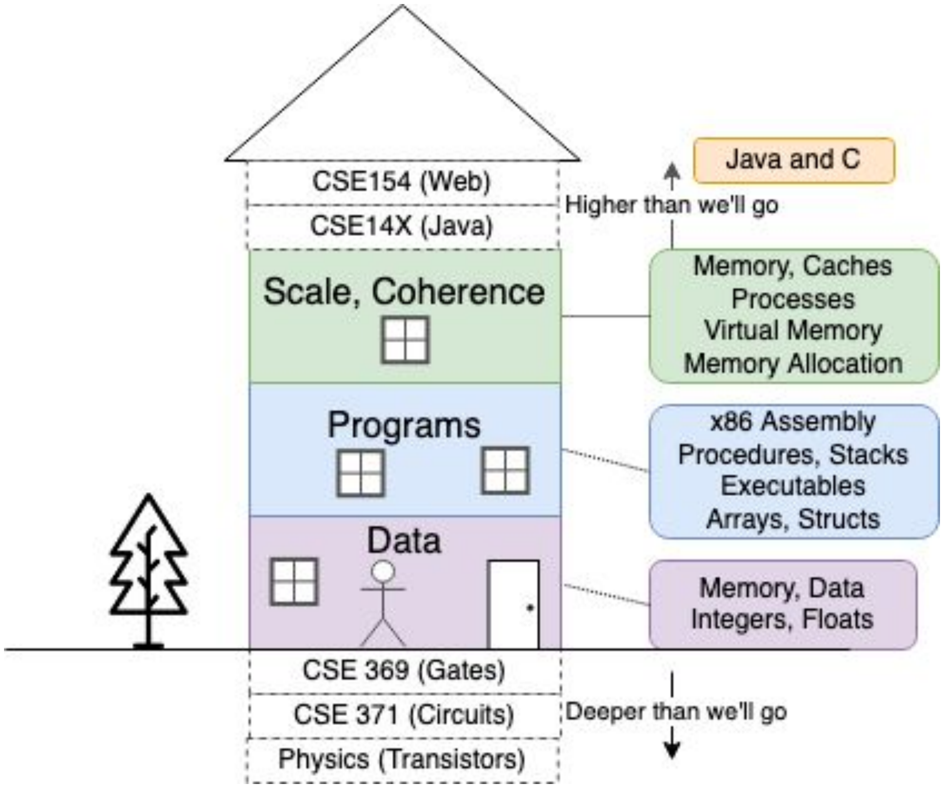
Lab Reflections

- 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

* is also used with variable declarations

```
int* ptr;
```

Declares a variable, ptr, that is a pointer to (*i.e.* holds the address of) an int in memory

```
int x = 5;
int y = 2;
```

Declares two variables, x and y, that hold ints, and *initializes* them to 5 and 2, respectively

```
ptr = &x;
```

Sets ptr to the address of x ("ptr points to x")

```
y = 1 + *ptr;
```

"Dereference ptr"

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

	0x00	0x01	0x02	0x03	
0x00	A7	00	32	00	
0x04	00	01	29	F3	x
0x08	EE	EE	EE	EE	
0x0C	FA	CE	CA	FE	
0x10	26	00	00	00	
0x14	00	00	10	00	
0x18	01	00	00	00	y
0x1C	FF	00	F4	96	
0x20	DE	AD	BE	EF	
0x24	00	00	00	00	

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

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

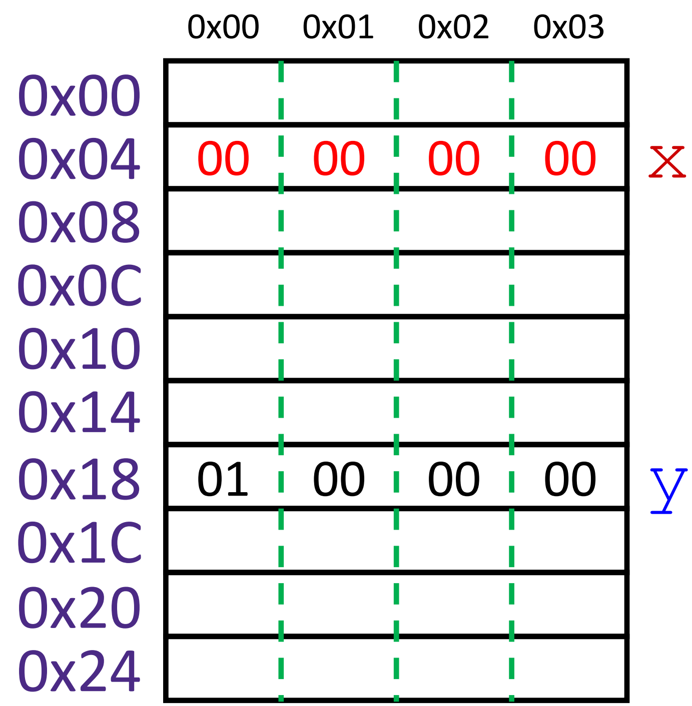
	0x00	0x01	0x02	0x03	
0x00					
0x04	00	01	29	F3	x
0x08					
0x0C					
0x10					
0x14					
0x18	01	00	00	00	y
0x1C					
0x20					
0x24					

Assignment in C

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

& = "address of"
* = "dereference"

- 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;`



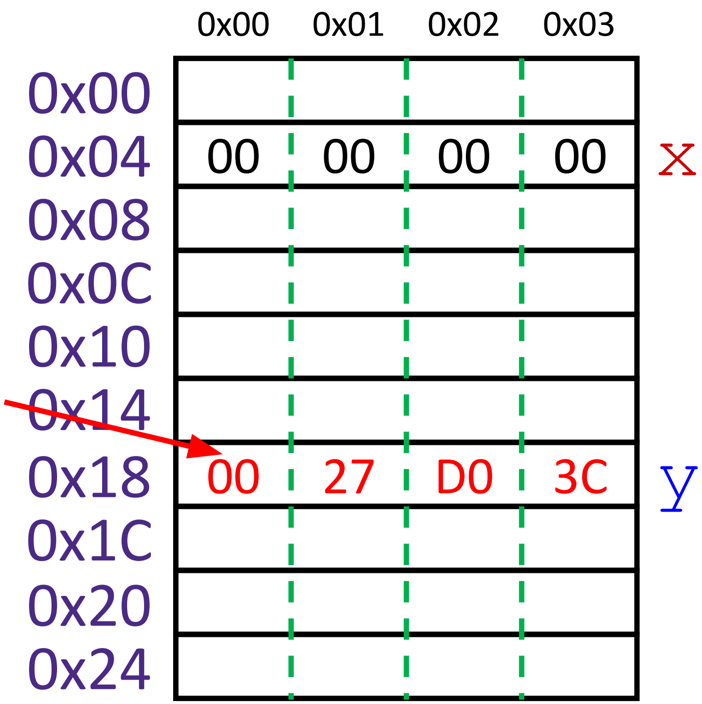
Assignment in C

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

& = "address of"
* = "dereference"

- 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;`

little endian!



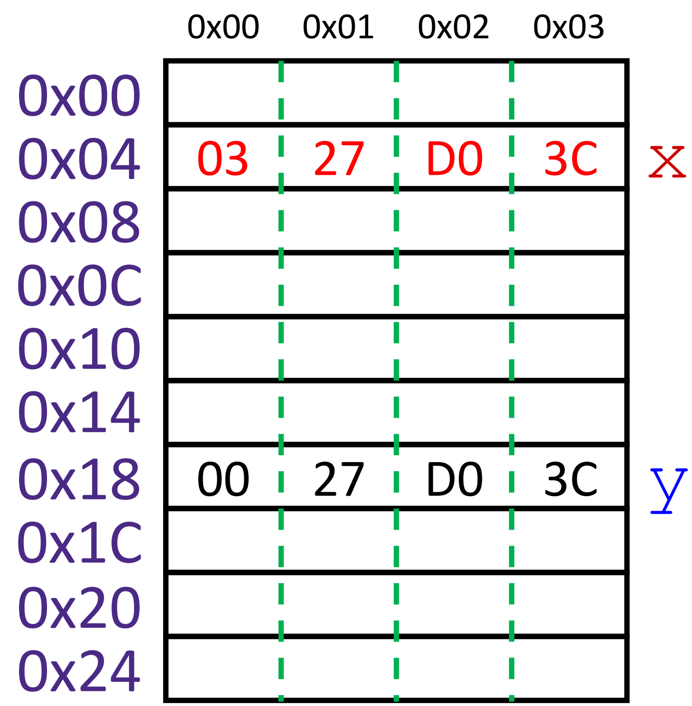
Assignment in C

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

`&` = "address of"

`*` = "dereference"

- 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

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

& = "address of"
* = "dereference"

- 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

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20	FE	ED	AB	BA	z
0x24					

Assignment in C

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

`&` = "address of"
`*` = "dereference"

- 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

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20	24	00	00	00	z
0x24					

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

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

`&` = "address of"
`*` = "dereference"

- `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?

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20	24	00	00	00	z
0x24					

Assignment in C

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

`&` = "address of"
`*` = "dereference"

- `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`

The target of a pointer is also a location

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	x
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20	24	00	00	00	z
0x24	00	27	D0	3C	

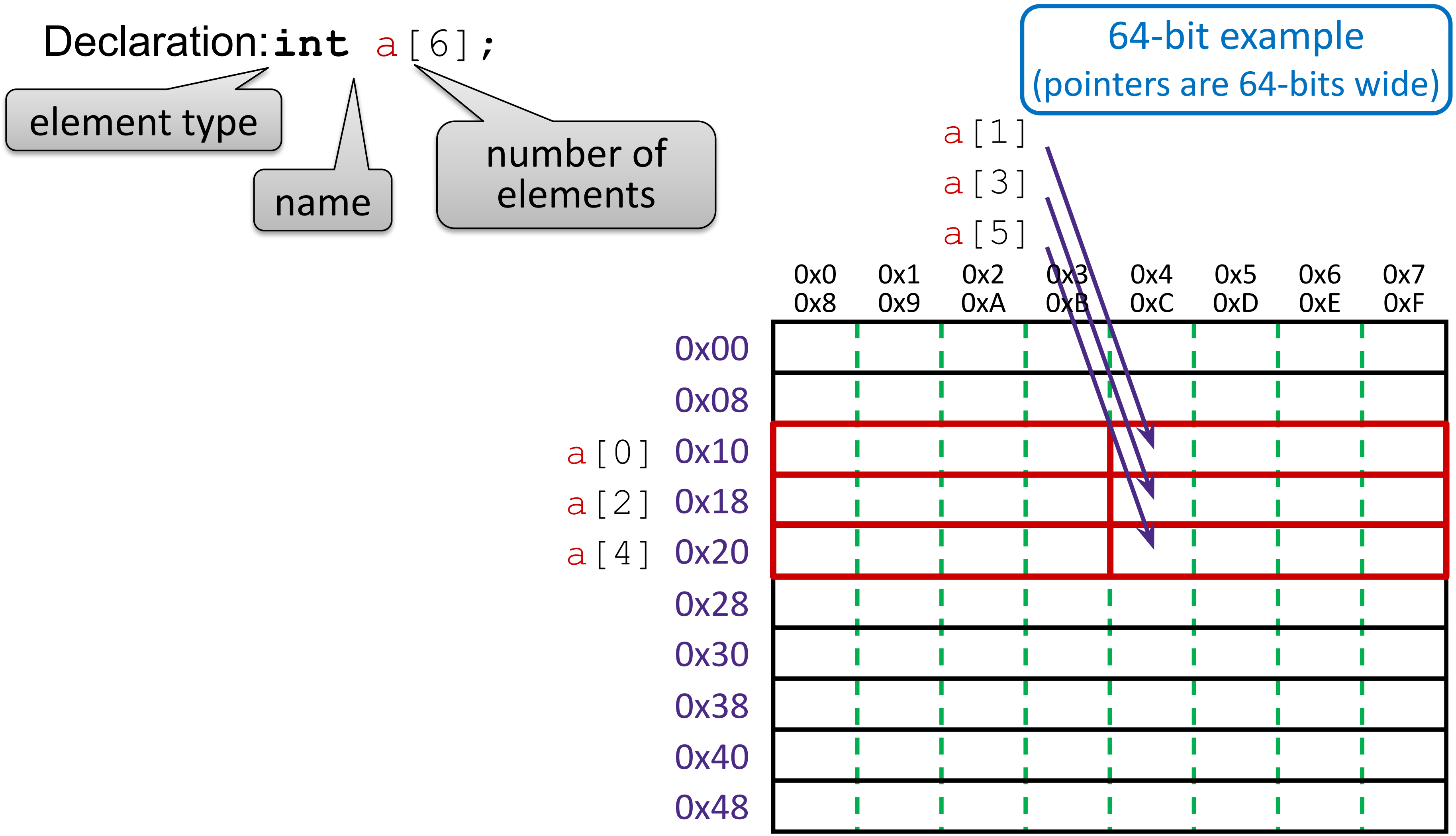
How are we feeling about assignment?

Arrays in C

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

a (array name) returns the array's address

Declaration: `int a[6];`



Arrays in C

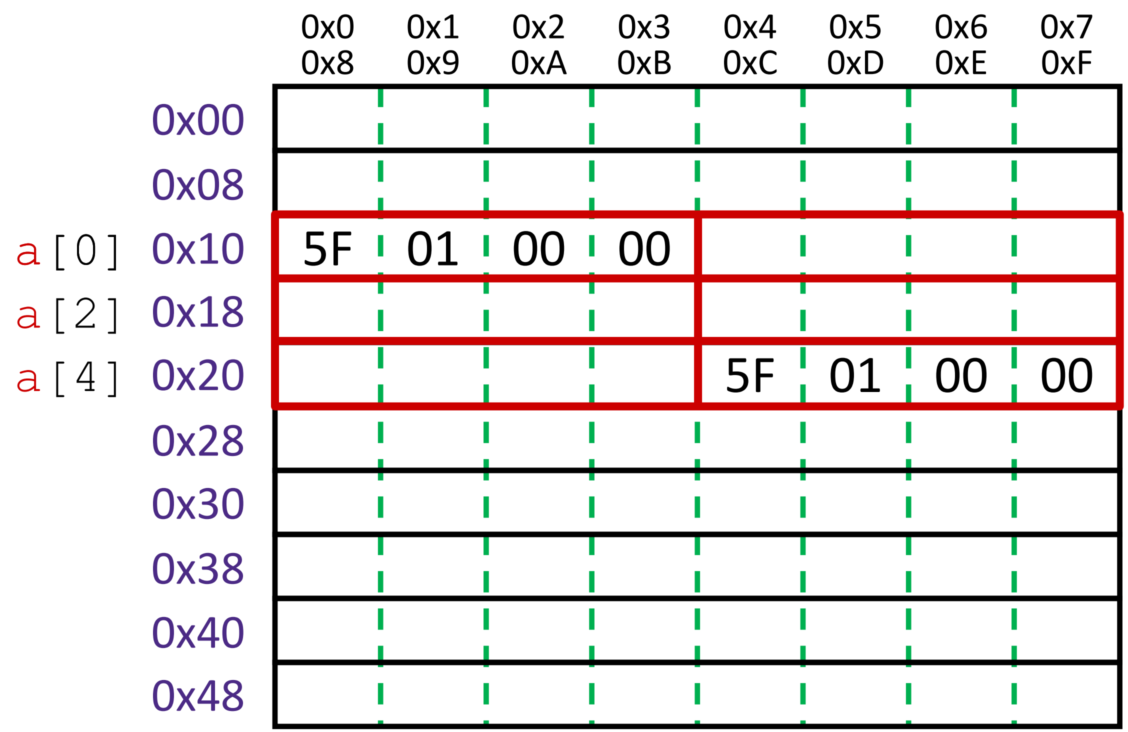
Arrays are adjacent locations in memory storing the same type of data object

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

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

Declaration: `int a[6];`

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



Arrays in C

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

Declaration: `int a[6];`

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

No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

		0x0 0x8	0x1 0x9	0x2 0xA	0x3 0xB	0x4 0xC	0x5 0xD	0x6 0xE	0x7 0xF
	0x00								
	0x08					AD	0B	00	00
a[0]	0x10	5F	01	00	00				
a[2]	0x18								
a[4]	0x20					5F	01	00	00
	0x28	AD	0B	00	00				
	0x30								
	0x38								
	0x40								
	0x48								

Arrays in C

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

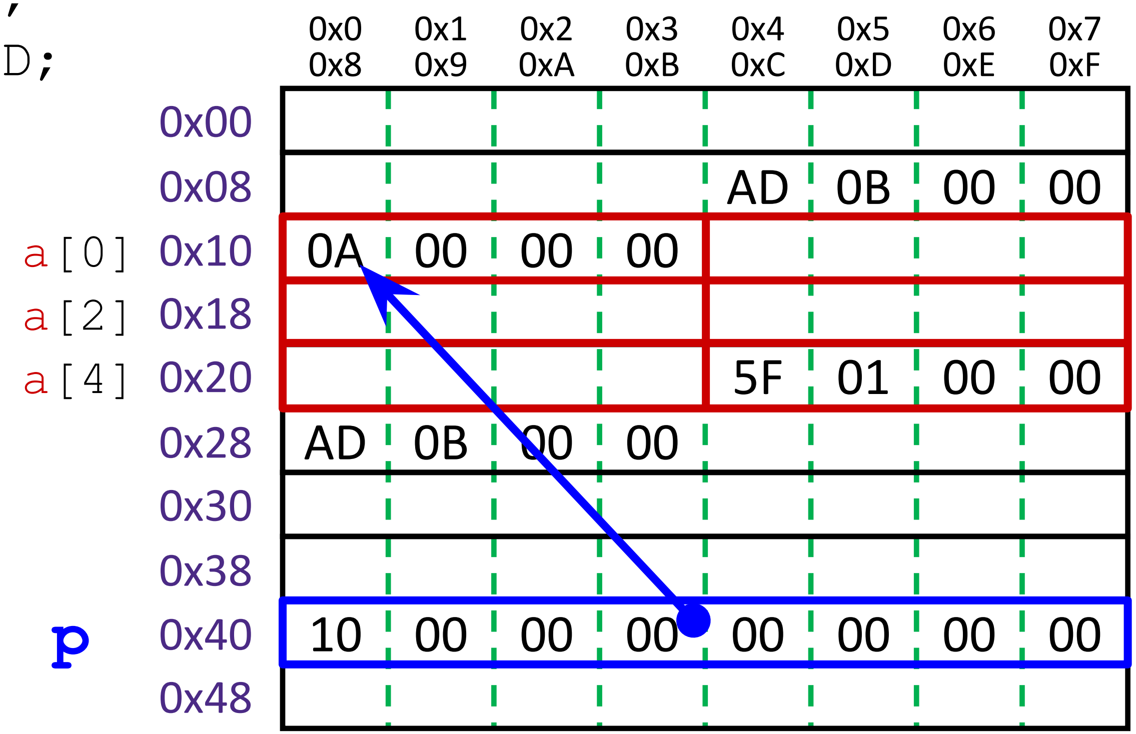
Declaration: `int a[6];`

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

No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers: `int* p;`

equivalent
```
p = a;
p = &a[0];
```
```
*p = 0xA;
```



Arrays in C

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

Declaration: `int a[6];`

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

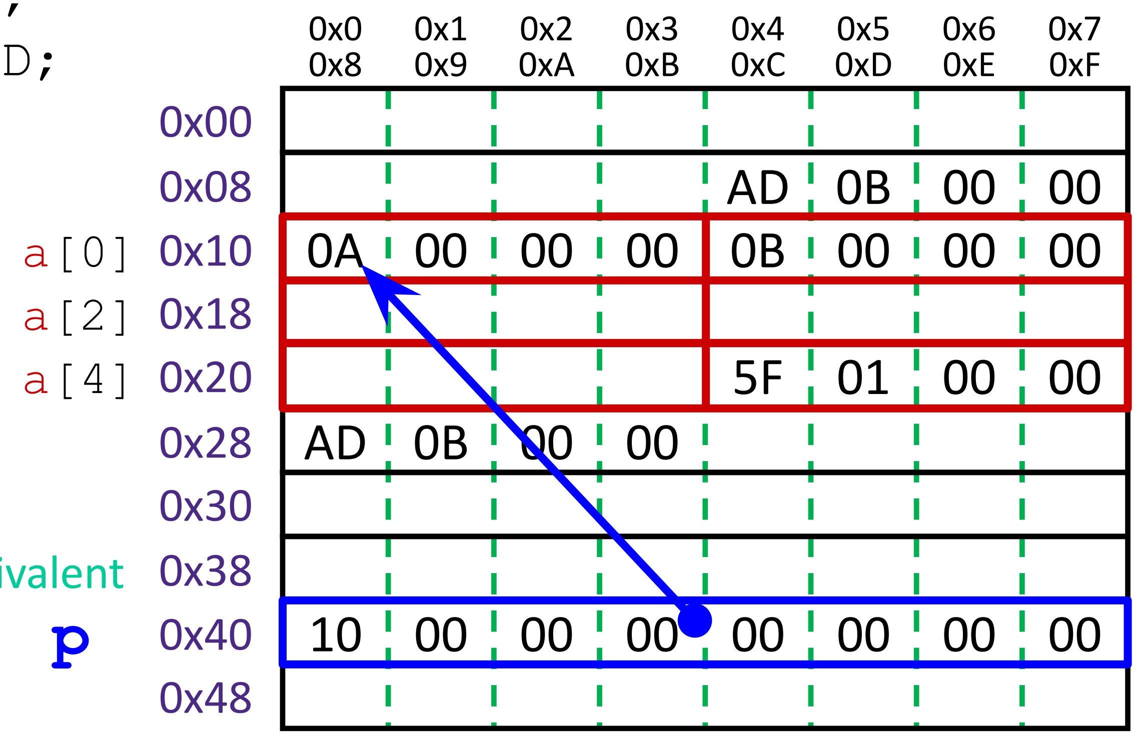
No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:
```
int* p;
```
equivalent {
```
p = a;
p = &a[0];
```
}
```
*p = 0xA;
```

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

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

```
p = p + 2;
```



Arrays in C

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

Declaration: `int a[6];`

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

No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers: `int* p;`

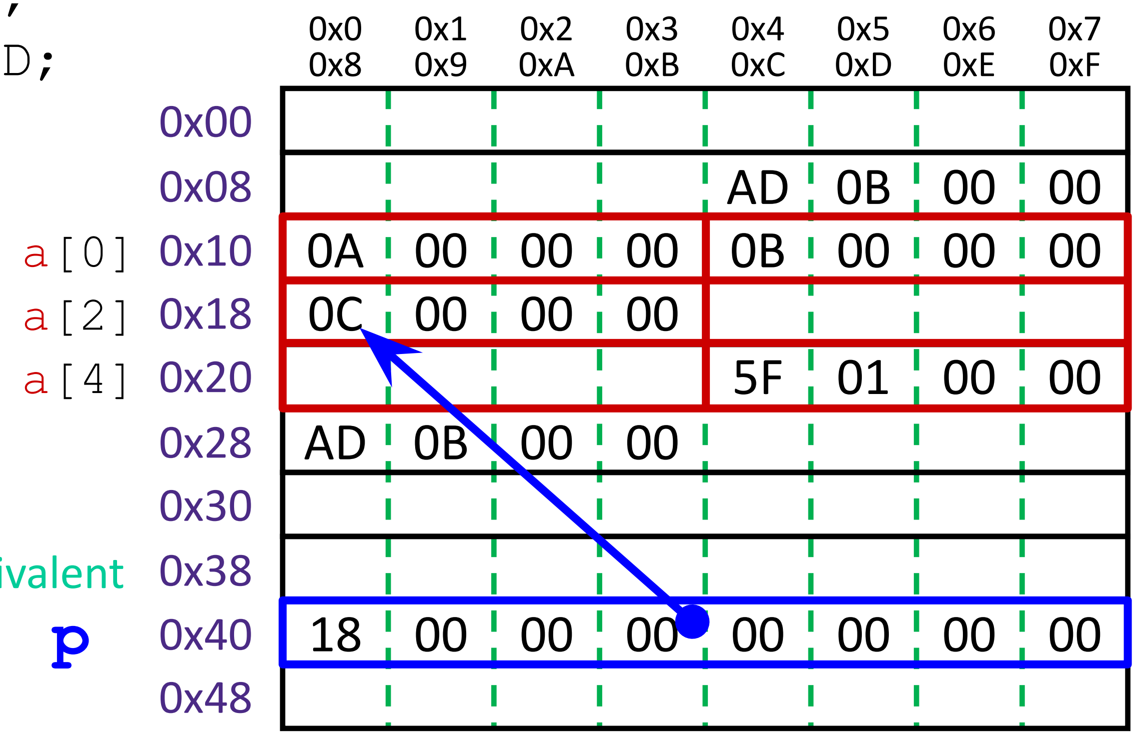
equivalent {
```
p = a;
p = &a[0];
```
}
```
*p = 0xA;
```

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

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

```
p = p + 2;
```

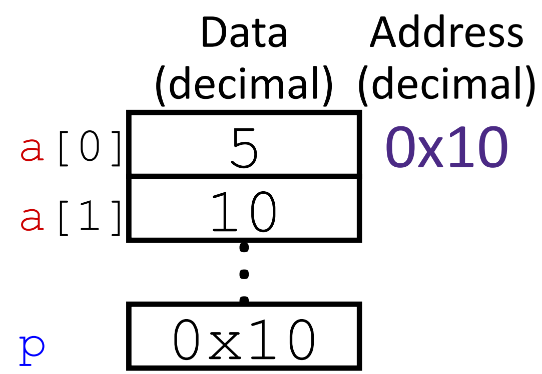
```
*p = a[1] + 1;
```



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!)

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



p	*p	a[0]	a[1]		p	*p	a[0]	a[1]
0x11	10	5	10	then	0x11	11	5	11
0x14	10	5	10	then	0x14	11	5	11
0x10	6	6	10	then	0x11	6	6	10
0x10	**6**	**6**	**10**	**then**	**0x14**	**6**	**6**	**10**

Representing strings

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

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	I	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

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

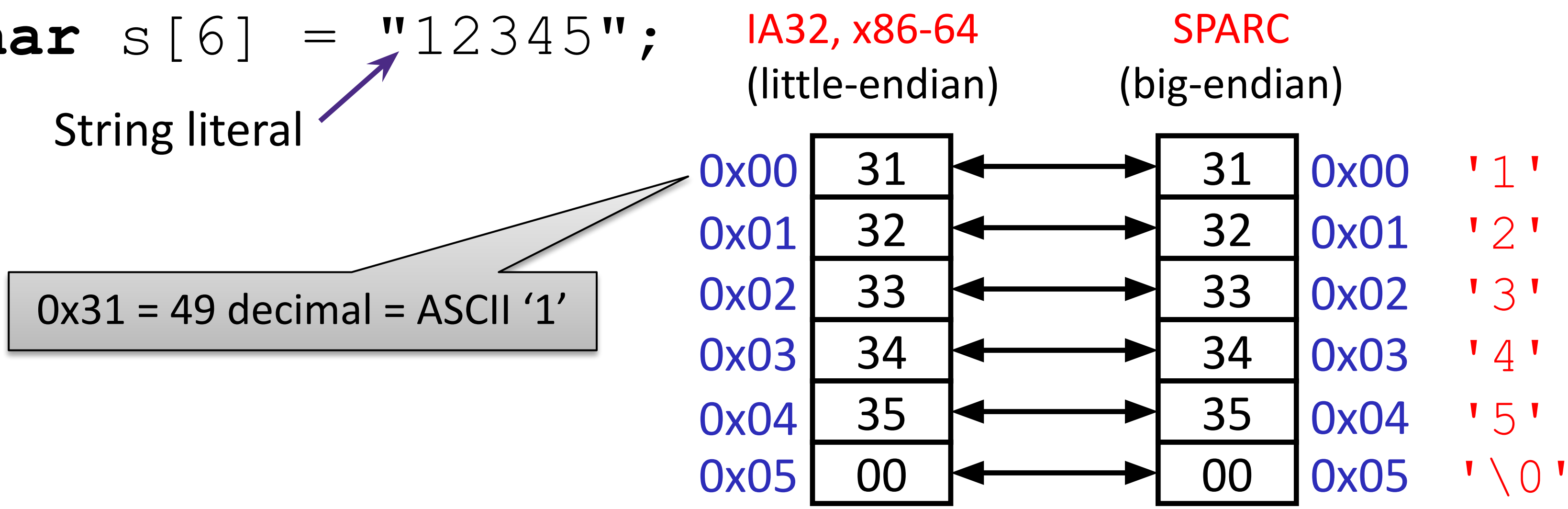
Decimal:	73	99	101	32	67	114	101	97	109	101	114	121	0
Hex:	0x49	0x63	0x65	0x20	0x43	0x72	0x65	0x61	0x6d	0x65	0x72	0x79	0x00
Text:	I	c	e		C	r	e	a	m	e	r	y	\0

- 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

C (char = 1 byte)

```
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
  - No need to order bytes with just one 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 !!

```
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 !!

```
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

```
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

```
int x = 12345;
0x7fffb7f71dbc   0x39
0x7fffb7f71dbd   0x30
0x7fffb7f71dbe   0x00
0x7fffb7f71dbf   0x00
```

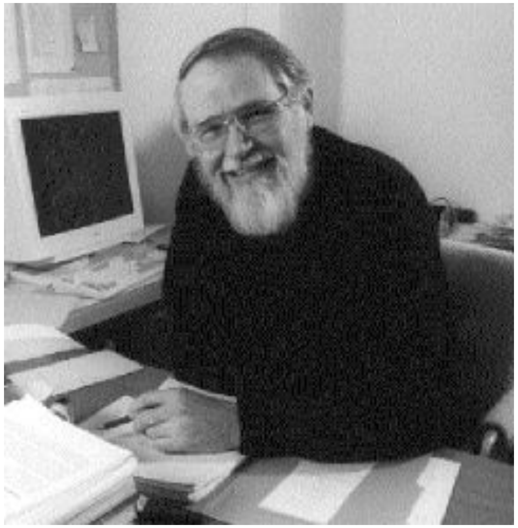
Summary

- 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
  - `&A[2] === A + 2`
  - 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!

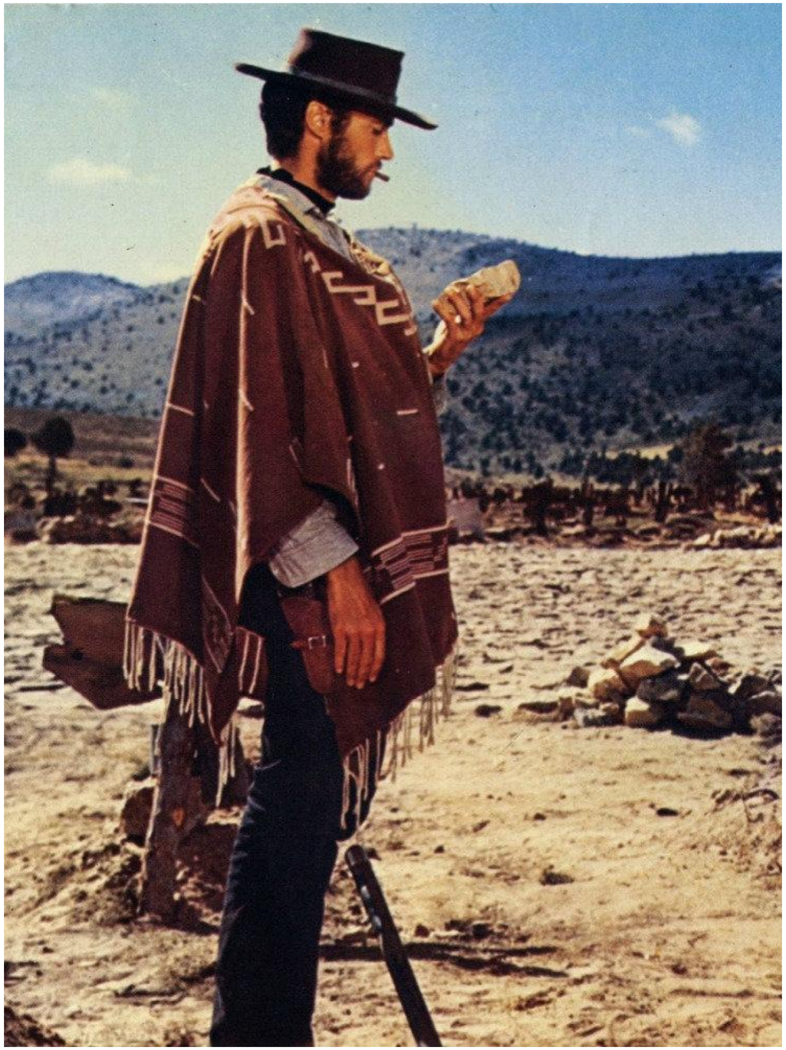
Wranglers in the Wild West

- 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.

      ** BOOOOOOOOOOOM **

   ****  You have died  ****

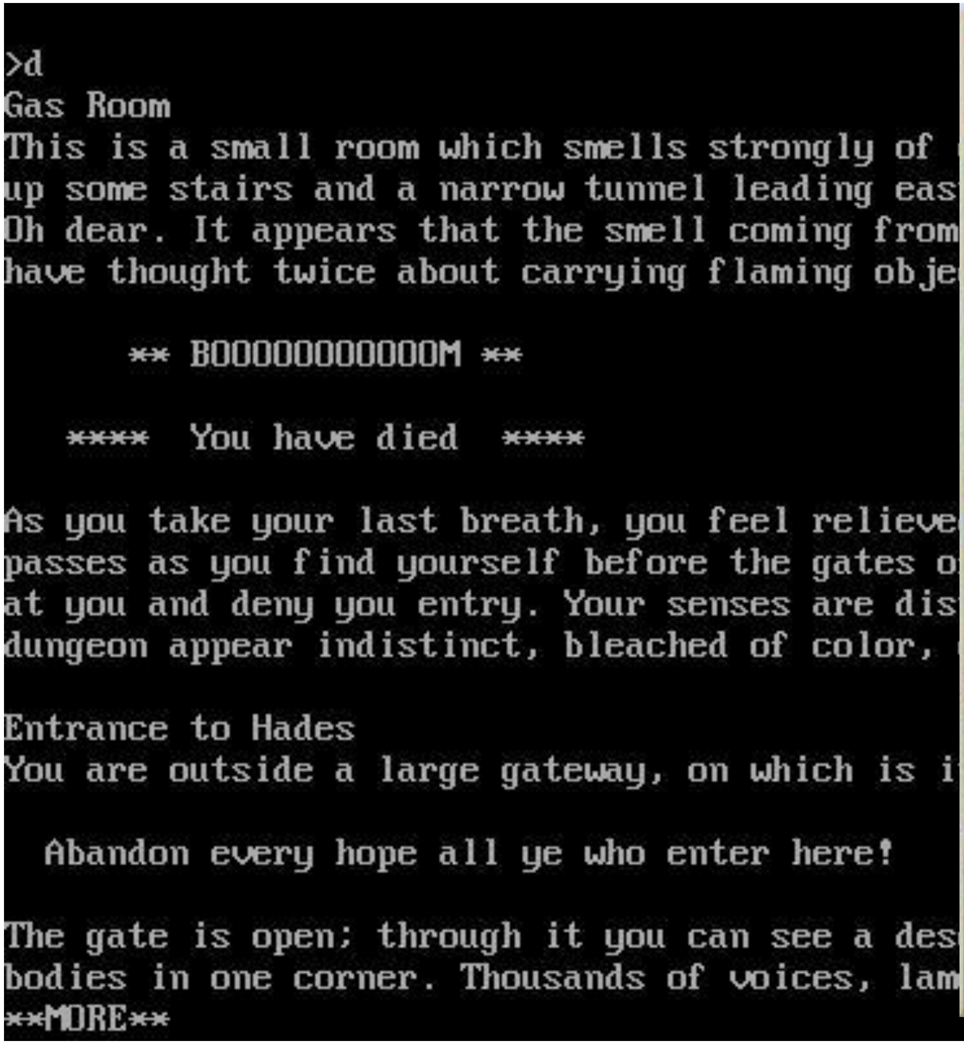
As you take your last breath, you feel relieved of your burdens. The feeling
passes as you find yourself before the gates of Hell, where the spirits jeer
at you and deny you entry. Your senses are disturbed. The objects in the
dungeon appear indistinct, bleached of color, even unreal.

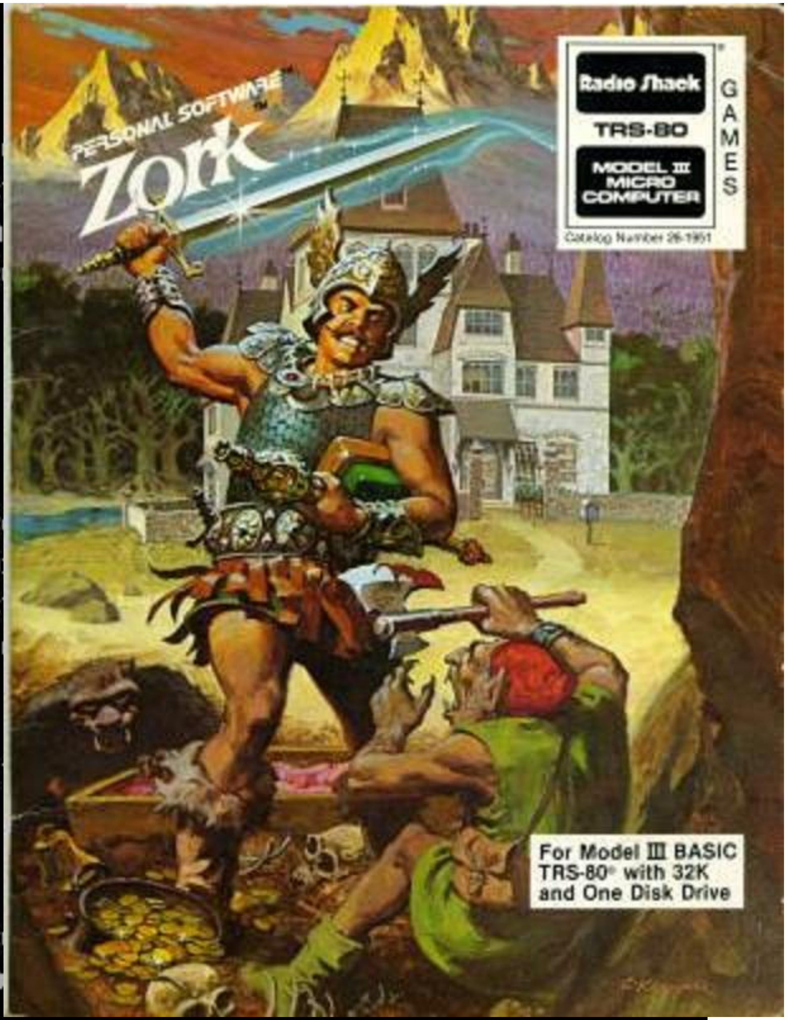
Entrance to Hades
You are outside a large gateway, on which is inscribed

  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

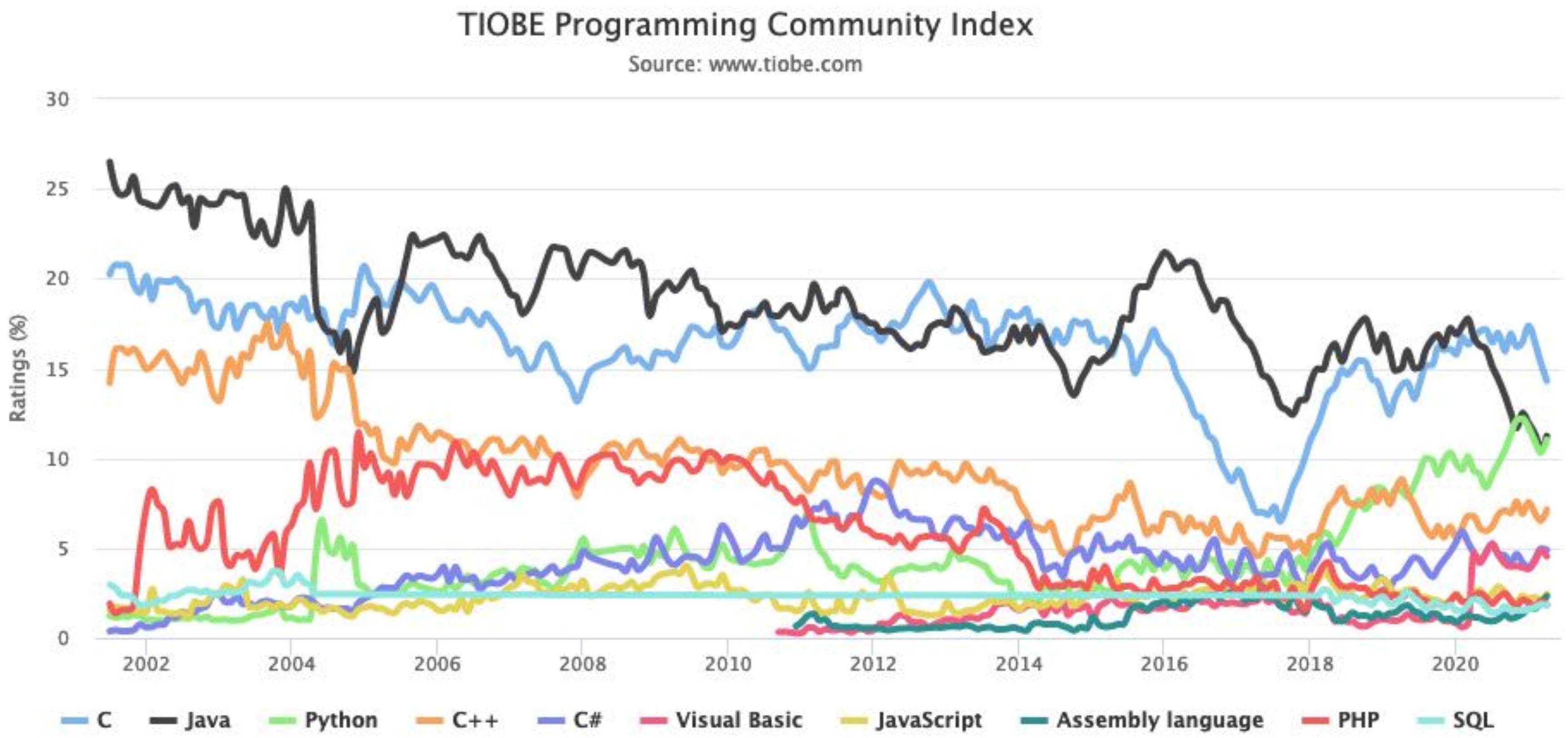




Principles of C, viewed today

- **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?



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!

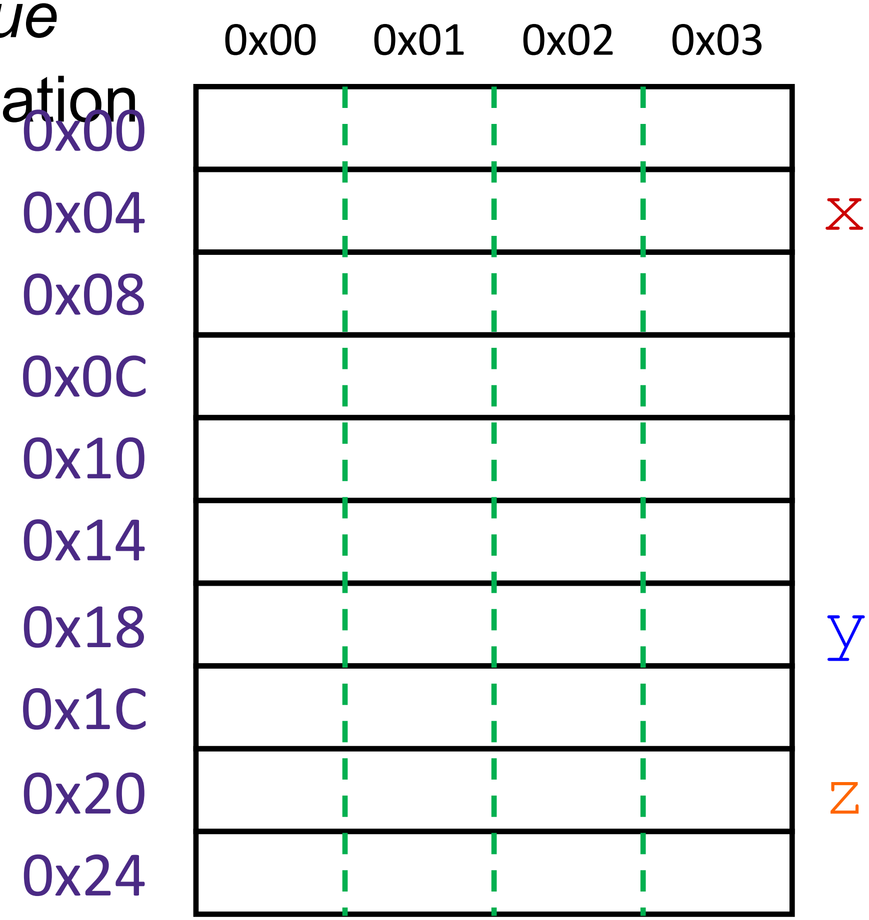
Assignment in C - Handout

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

& = "address of"
* = "dereference"

- 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 - Hando

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

Declaration: `int a[6];`

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

No bounds checking: `a[6] = 0xBAD;`
`a[-1] = 0xBAD;`

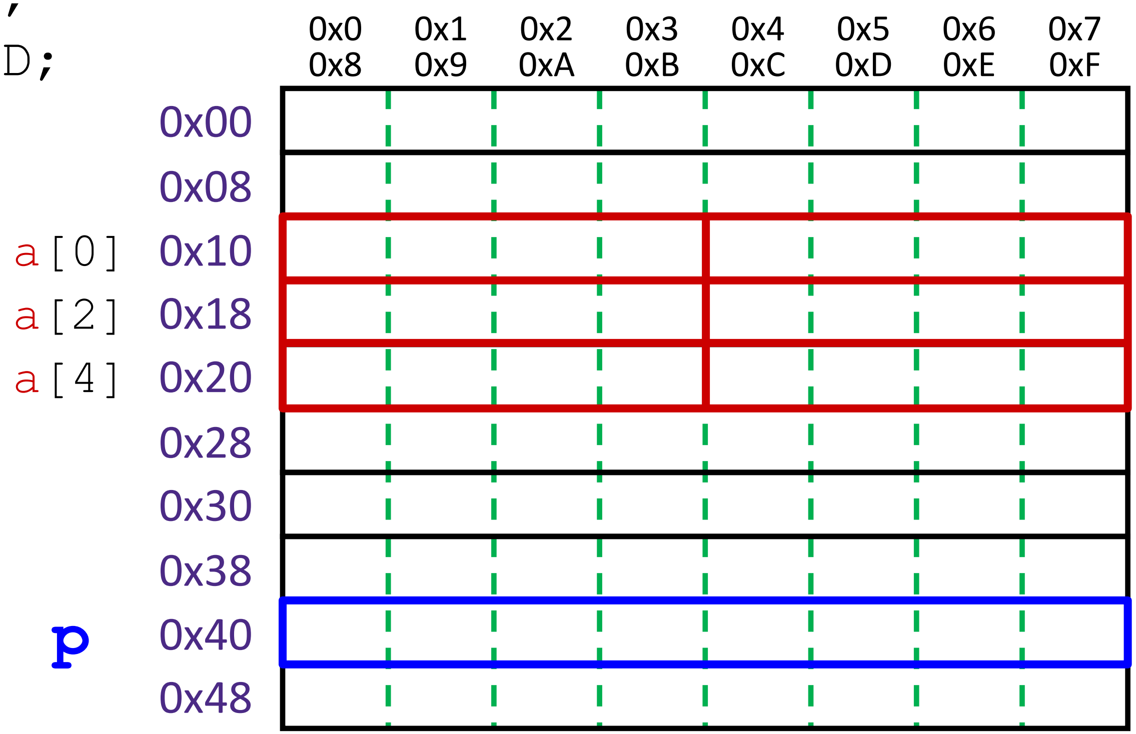
Pointers: `int* p;`

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;`

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?