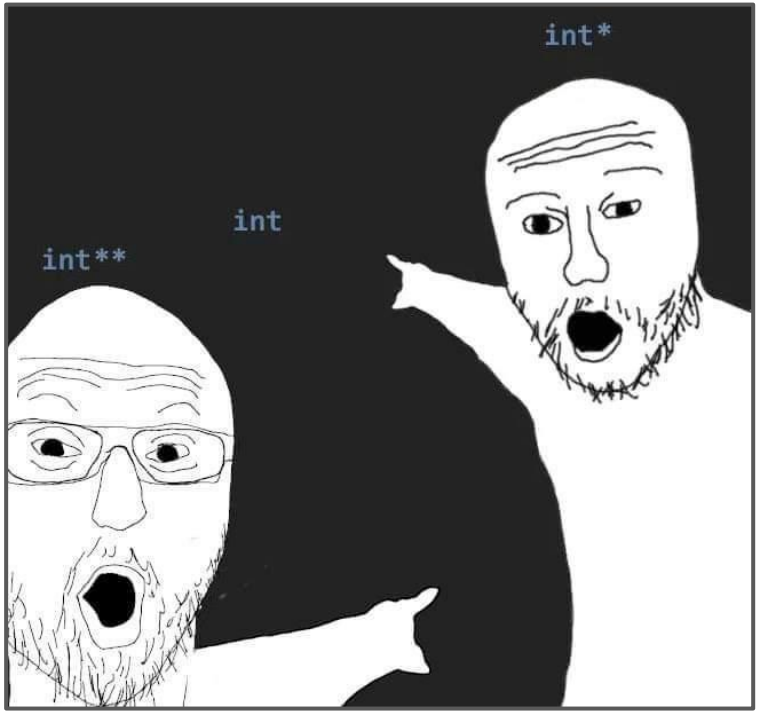
Memory, Data, & Addressing II

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang



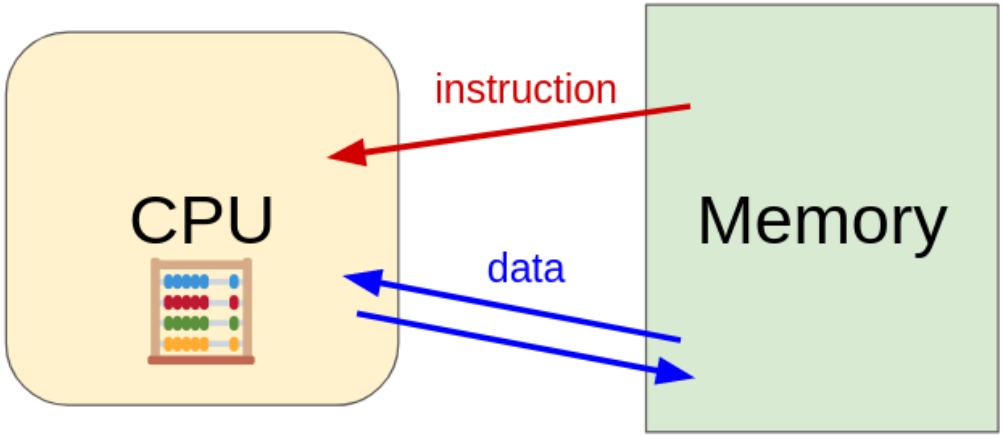
Administrivia

- HW 1 and Lab 0 due tonight (11:59pm)
- Due Wednesday, 6/26:
  - RD 4 (1pm)
  - HW 2 (11:59pm)
- Lab 1a out now!
  - Due 7/3 (late due date 7/5), turn in on Gradescope
  - Can be completed with a partner
    - Turn in *one submission* for the both of you

Reminder: Lab Late Days

- You get 5 free late days for the quarter
  - Late days **only** apply to labs
  - No benefit to having leftover late days
  - If working with a partner, every counts towards both of your late days
- Count lateness in *days* (even if just by a second)
  - Weekends count as 1 day
  - No submissions accepted more than 2 days late
- Once free late days are used up, deduct 10% per day
- Use at your own risk - don’t want to fall to far behind
  - Intended to allow for unexpected circumstances

Recap: CPU and Memory



a. How does the CPU find its data in memory?
b. How are common C types encoded?
c. How can we use C to manipulate data in memory?

today's focus

Lecture Outline

- Assignment in C
  - Memory example
- More pointers
  - Pointer arithmetic
- Arrays
  - Memory example
  - Arrays and pointers
- Strings
- Box-and-arrow diagrams

Review Question

reminder: & = dereference * = address of

In the code on the right, which of the following expressions evaluate to an address?

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

A) x + 10 int + int = int

B) p + 10 char* + int = char*

C) &x + 10 int* + int = int*

D) *(&p) & and * cancel each other out, end up w/ p, so char*

E) ar[1] int

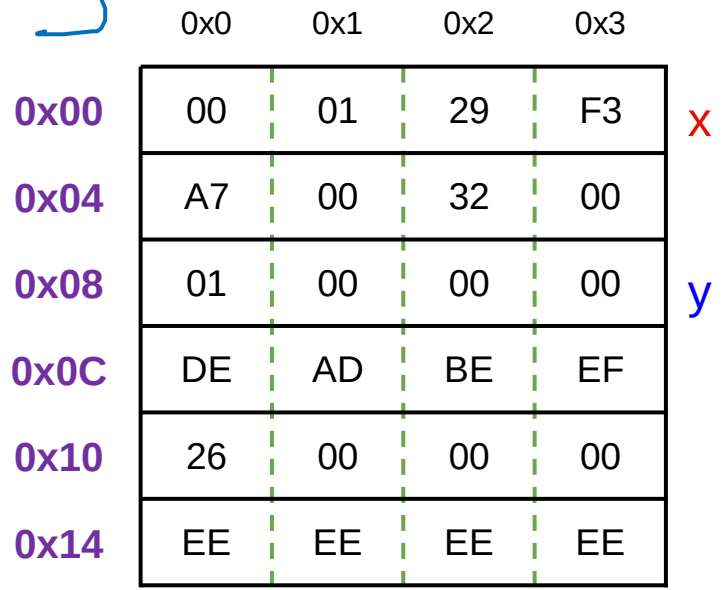
F) &ar[2] int*

Assignment in C

declaring a variable = allocating memory

- A variable is represented by a location
- Declaration ≠ initialization!
  - Initially “garbage” or “mystery data”

32-bit example
(pointers are 4-bytes wide)
little-endian



Example:

```
int x, y;
```

(x is at address `0x00`, y is at address `0x08`)

Assignment in C (pt 2)

- A variable is represented by a location
- Declaration ≠ initialization!
  - Initially “garbage” or “mystery data”

Example:

```
int x, y;
```

(x is at address `0x00`, y is at address `0x08`)

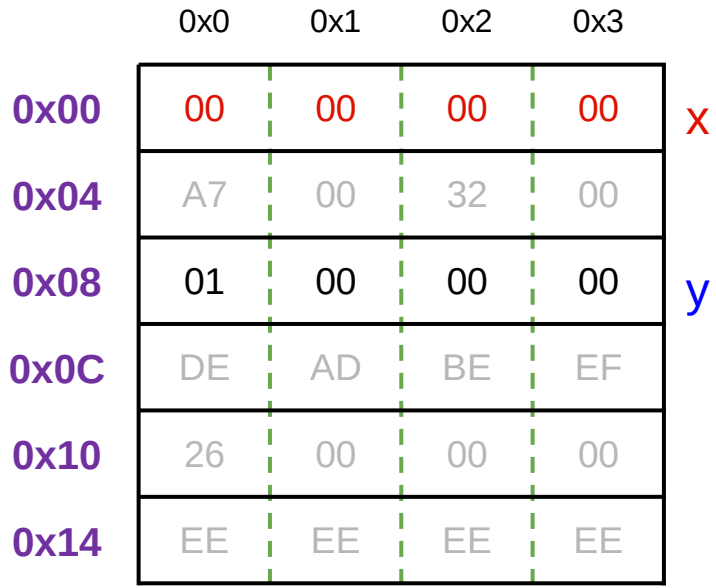
32-bit example
(pointers are 4-bytes wide)
little-endian

	0x0	0x1	0x2	0x3	
0x00	00	01	29	F3	x
0x04	A7	00	32	00	
0x08	01	00	00	00	y
0x0C	DE	AD	BE	EF	
0x10	26	00	00	00	
0x14	EE	EE	EE	EE	

Assignment in C (pt 3)

- `Left hand side = Right hand side`
  - LHS must evaluate to a *location*
  - RHS must evaluate to a *value*
    - Could be an address, or any other data
  - Store RHS value at LHS location

32-bit example
(pointers are 4-bytes wide)
little-endian



Example:

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

Assignment in C (pt 4)

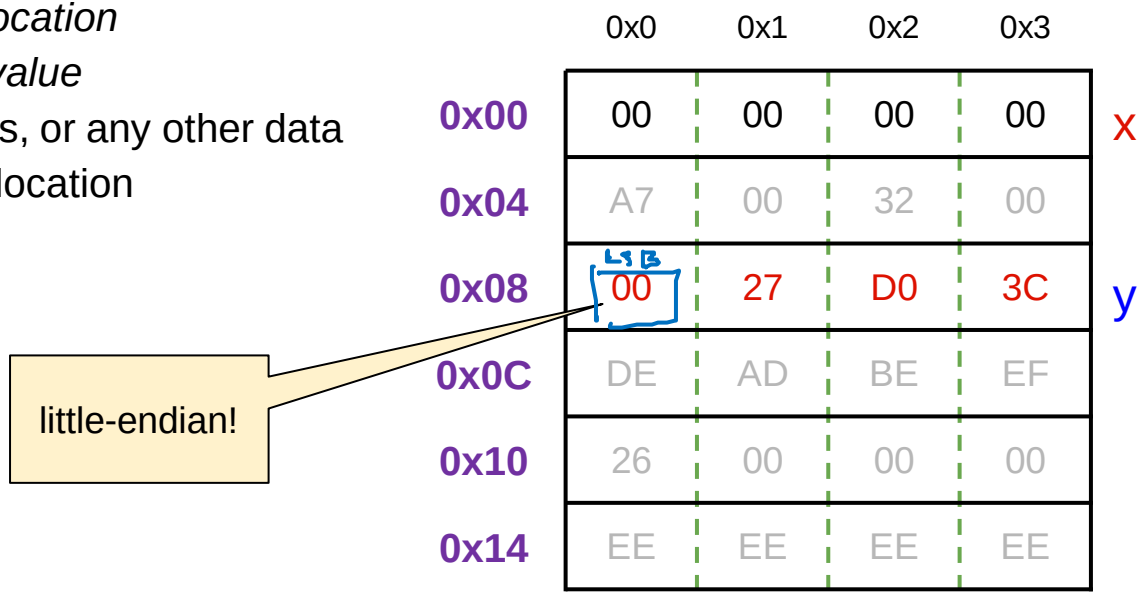
- `Left hand side = Right hand side`
  - LHS must evaluate to a *location*
  - RHS must evaluate to a *value*
    - Could be an address, or any other data
  - Store RHS value at LHS location

Example:

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

LSB

32-bit example
(pointers are 4-bytes wide)
little-endian



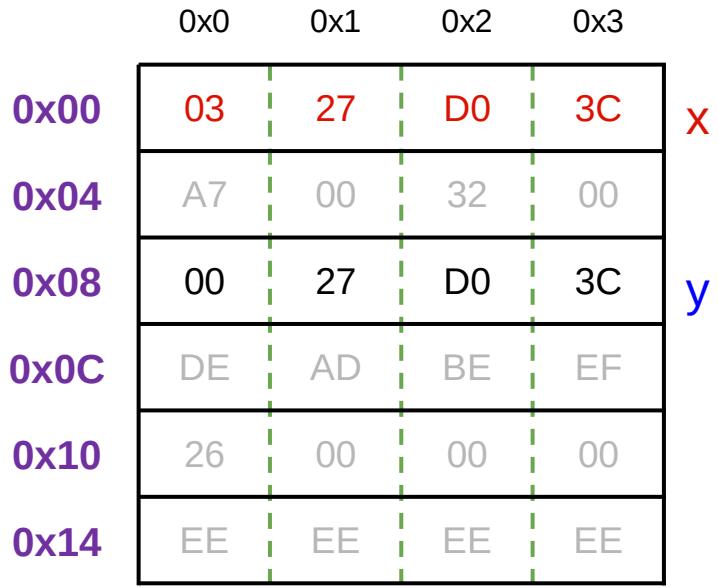
Assignment in C (pt 6)

- `Left hand side = Right hand side`
  - LHS must evaluate to a *location*
  - RHS must evaluate to a *value*
    - Could be an address, or any other data
  - Store RHS value at LHS location

Example:

```
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
```
(compute y+3, then store into x)

32-bit example
(pointers are 4-bytes wide)
little-endian



Assignment in C (pt 7)

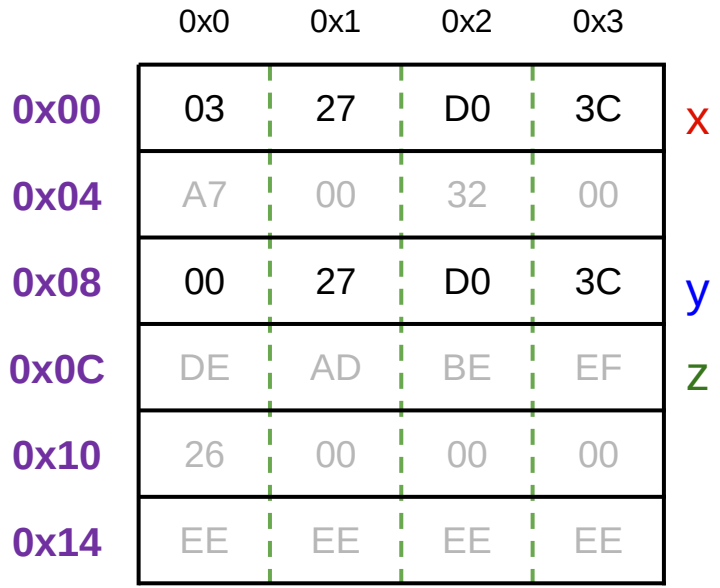
32-bit example
(pointers are 4-bytes wide)
little-endian

Example:

```
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
int* z = &y + 3;
```

(z stored at 0x0C)

(compute &y+3, and store into z?) – Nope!



Assignment in C (pt 8)

32-bit example
(pointers are 4-bytes wide)
little-endian

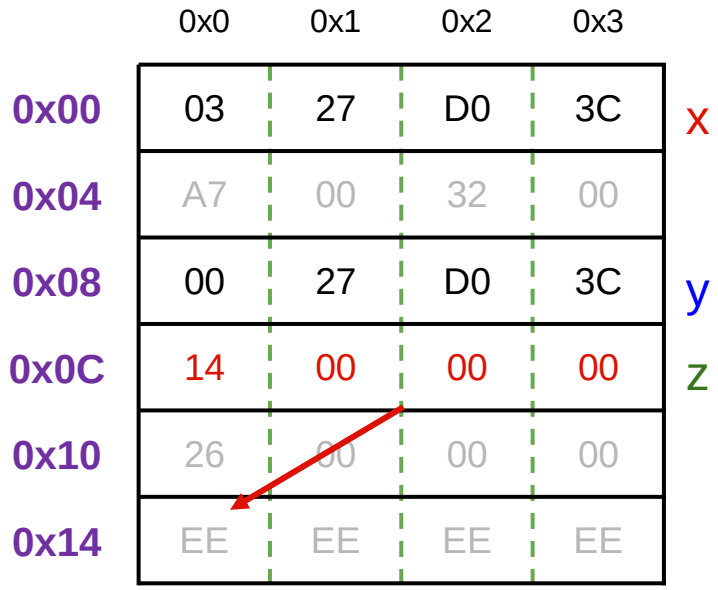
Example:

```
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
int* z = &y + 3;
```

(z stored at 0x0C)

(compute &y+**12**, and store into z)

Pointer arithmetic!



Pointer Arithmetic (Review)

- Pointer arithmetic is scaled by the size of the target type
  - In this example `sizeof(int)` = 4
- `int* z = &y + 3;`
  - Get address of `y`, add `3*sizeof(int)`, then store result in `z`
  - &y = 0x08 = 8_{10}
  - 3*4=12
  - 8+12 =20 = 0x14 (1*16 + 4)
- **Pointer arithmetic can be dangerous!**
  - Can easily lead to bad memory accesses
  - Be especially careful with casting

"set z to point 3 ints ahead of y's location"

commonly used for arrays

Assignment in C (pt 8)

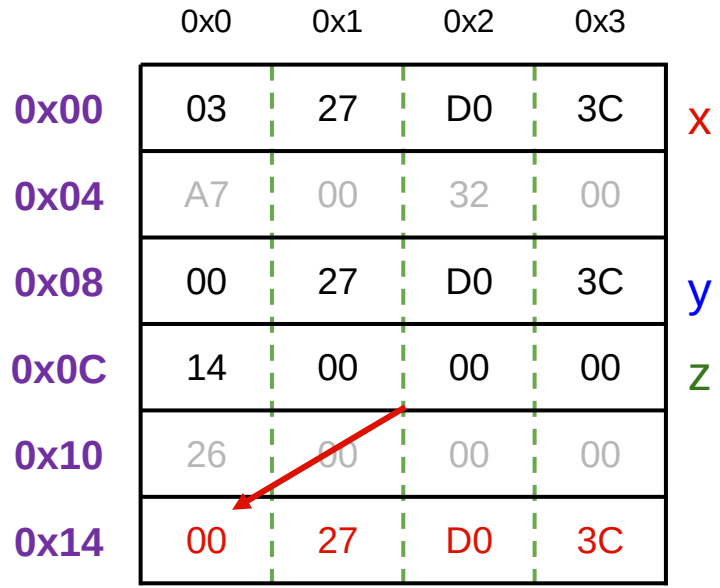
32-bit example
(pointers are 4-bytes wide)
little-endian

Example:

```
int x, y;
x = 0;
y = 0x3CD02700;
x = y + 3;
int* z = &y + 3;
```

The dereference of a pointer is also a location

`*z = y;` (get value at y, put in address stored in z)



Arrays in C (Review)

- Declaration: `type name[size];`
  - Example: `int a[6];`
  - Can also use initialization list: `int a[] = {2, 4, 6, 8, 10};`
- Indexing using brackets
  - i.e. `a[n]` gets the n^{th} element of the array, counting from 0
- Stored as a single **contiguous** chunk of memory
  - Total size = size of data * number of elements
  - Variable name (a) evaluates to the *starting address*
  - Endianness only applies within a *single element*. Elements always stored in increasing order

syntax is very similar to Java

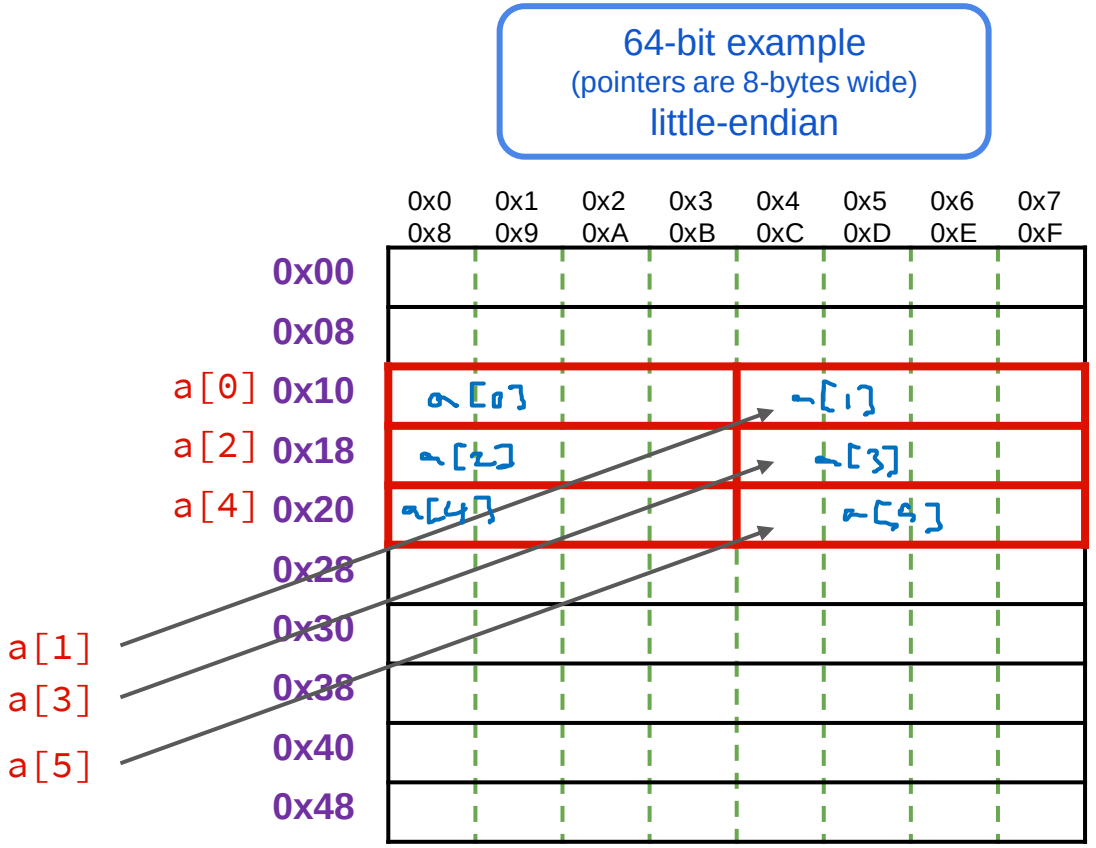
not exactly the same as a pointer!

more like a constant

ex: bytes w/in a[0] would be stored according to endianness, but a[0] always comes before a[1], regardless of endianness!

Arrays in C Example

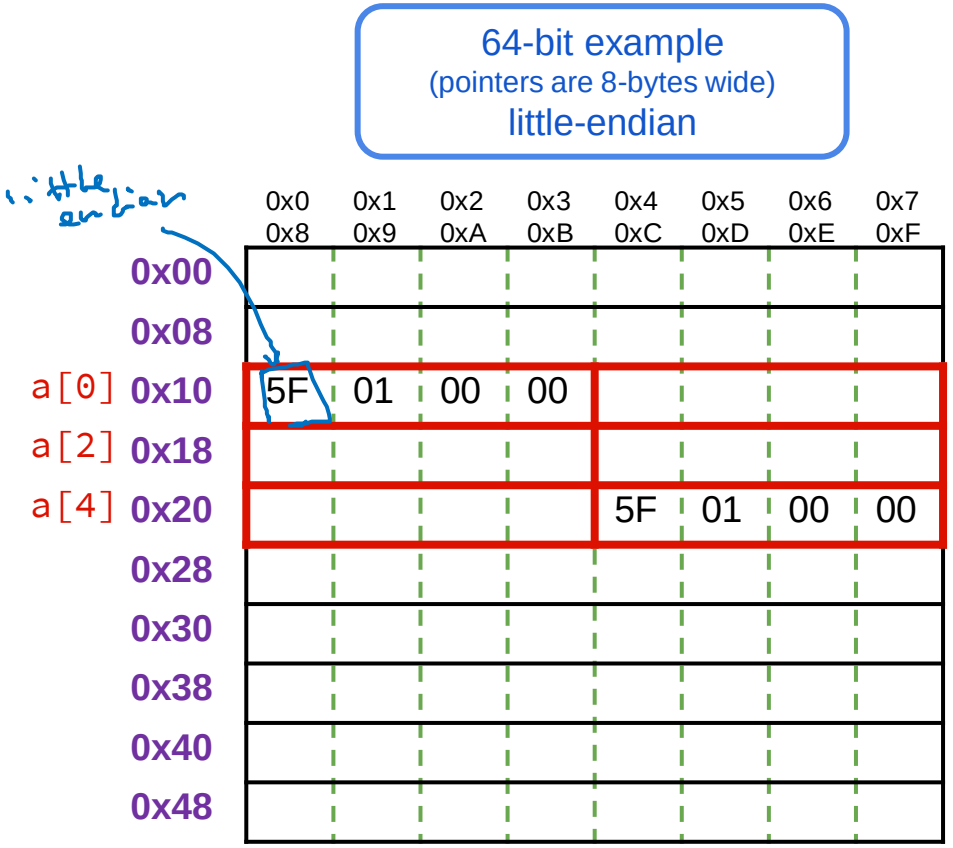
Declaration: `int a[6];`



Arrays in C Example (pt 2)

Declaration: `int a[6];`

Indexing:
```
a[0] = 0x15F;
a[5] = a[0];
```



Arrays in C Example (pt 3)

Declaration: `int a[6];`

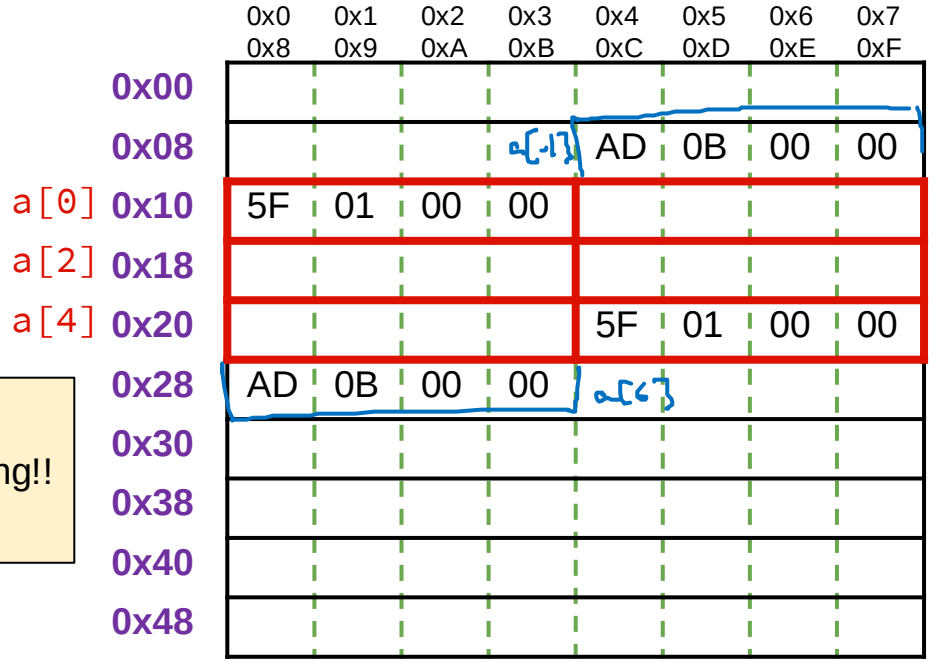
Indexing:

```
a[0] = 0x15F;
a[5] = a[0];
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

calculates where these indices would be if they existed

No bounds checking!!

64-bit example
(pointers are 8-bytes wide)
little-endian



Arrays in C Example (pt 4)

Declaration: `int a[6];`

Indexing:
```
a[0] = 0x15F;
a[5] = a[0];
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

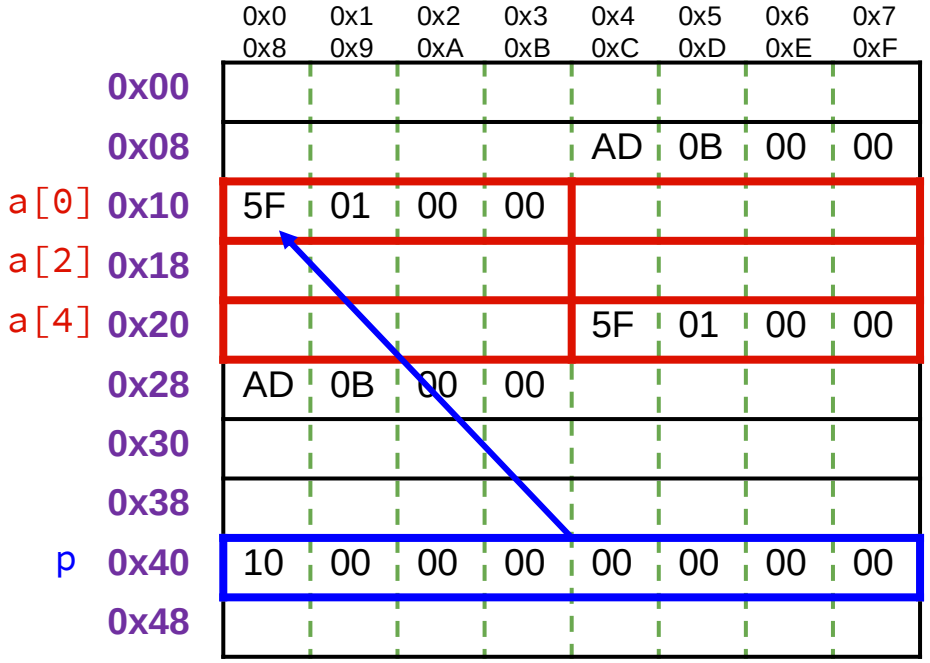
Pointers: `int* p;`

```
p = a;
p = &a[0];
p = 0x10;
```

a evaluates to starting address

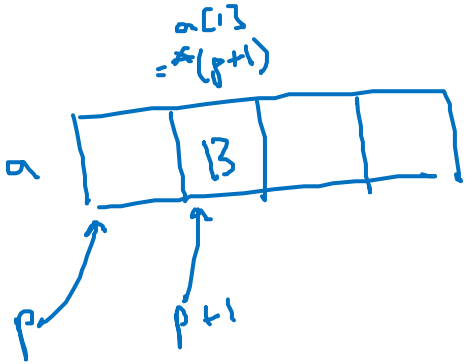
equivalent

64-bit example
(pointers are 8-bytes wide)
little-endian



Arrays and Pointer Arithmetic

- Both are scaled by the size of the type, so they can be used interchangeably!
  - Array indexing = pointer arithmetic + dereference



- Example: all of the following are equivalent
  - `a[1] = 0xB`
  - `*(p+1) = 0xB`
  - `p[1] = 0xB`
  - `*(a+1) = 0xB`

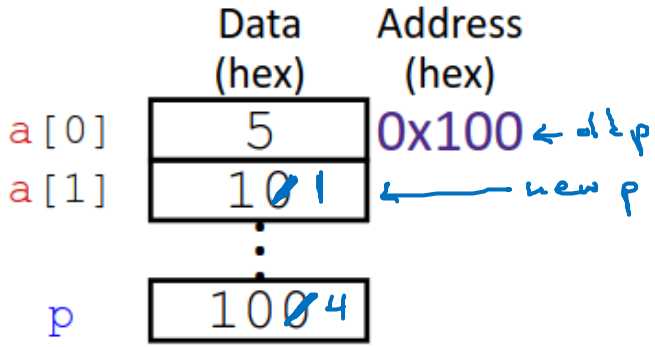
"p" and "a" have the same value, can use interchangeably

Polling Question

Consider the following code:

```
1 void main() {
2       int a[] = {0x5, 0x10};
3       int* p = a;
4       p = p + 1;
5       *p = *p + 1;
6 }
```

The variable values just *after* line 3 executes are shown in the diagram on the right. What are they after line 5?



	p	a[0]	a[1]
A)	0x101	0x05	0x11
B)	0x104	0x05	0x11
C)	0x101	0x06	0x10
D)	0x104	0x06	0x10

Representing Strings (Review)

- In C, stored as an array of bytes (**`char*`**)
  - No `String` keyword, unlike in Java
  - One-byte ASCII codes for each character (English-only)

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Representing Strings (Review) (pt 2)

"null terminator"

- Last character followed by a null character to mark the end of the string
  - 0 byte, often written as `'\0'`

Example: `"Sam is cool!";` ← when you declare a string w/ " ", implicitly allocates 1 extra byte for null char

Decimal	83	97	109	32	105	115	32	99	111	111	108	33	0
Hex	0x53	0x61	0x6D	0x20	0x69	0x73	0x20	0x63	0x6F	0x6F	0x6C	0x21	0x00
Text	'S'	'a'	'm'	' '	'i'	's'	' '	'c'	'o'	'o'	'l'	'!'	'\0'

Endianness and Strings

- Endianness doesn't apply to single-byte values (like `chars`)
- As with all arrays, element 0 is stored at the lowest address *regardless of endianness*

aka: we don't have to care about endianness

Example: `"12345";`

1st char = lowest address, null terminator = highest address

0x00	0x01	0x02	0x03	0x04	0x05
0x31	0x32	0x33	0x34	0x35	0x00
'1'	'2'	'3'	'4'	'5'	'\0'

Examining Data Representations

- `show_bytes` prints out the byte representation of the data at `start`
  - Tread any data as a byte array by casting it to `char*`
    - C has unchecked casts. !! Danger !!

prints in format:
<addr> 0x<data>

```
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++){
        printf("%p\t0x%.2hhX\n", start+i, *(start+i));
    printf("\n");
}
```

= start[i]

- `printf` directives: `%p` = print pointer `\t` = tab
  `%.2hh` = print value as char in hex, padding to 2 digits `\n` = new line

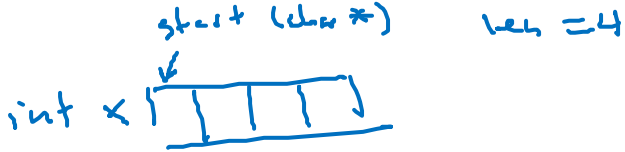
Examining Data Representations (pt 2)

this code trusts that len is correct!

DANGER

```
void show_bytes(char* start, int len) {
    int i;
    for (i = 0; i < len; i++){
        printf("%p\t0x%.2hhX\n", start+i, *(start+i));
    printf("\n");
}
```

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



`show_bytes` **Execution Example**

```
int main() {
    int x = 123456; // 0x00 01 E2 40
    printf("int x = %d;\n", x);
    show_int(x);
}
```

→ "int x = 123456;"

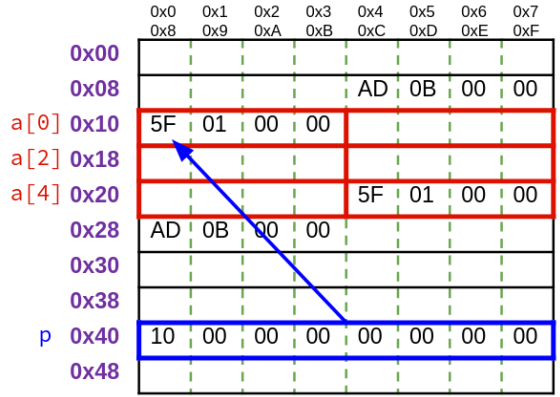
- Result (Linux x86-64)
  - **Note:** addresses will change with each run (try it!), but fall in the same general range

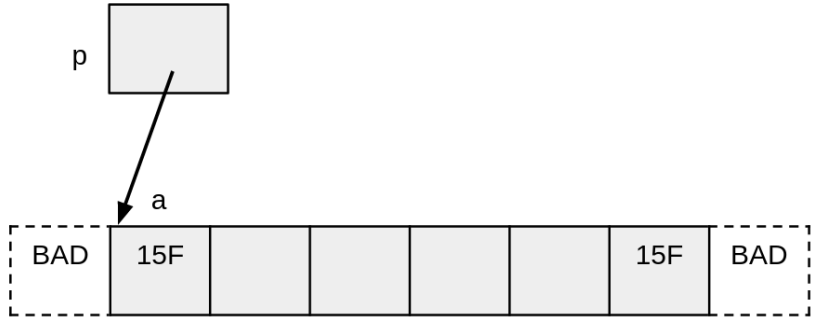
```
int x = 123456;
0x7fffb245549c  0x40
0x7fffb245549d  0xE2
0x7fffb245549e  0x01
0x7fffb245549f  0x00
```

we know this machine is little endian because LSB is at lowest address

Box-and-Arrow Diagrams (Review)

- Simplified way of drawing out memory
- Each variable is a box
  - If relevant, the address may be written above the box
- If a pointer points to another variable, draw an arrow between them
  - If relevant, may also write the address inside the pointer's box

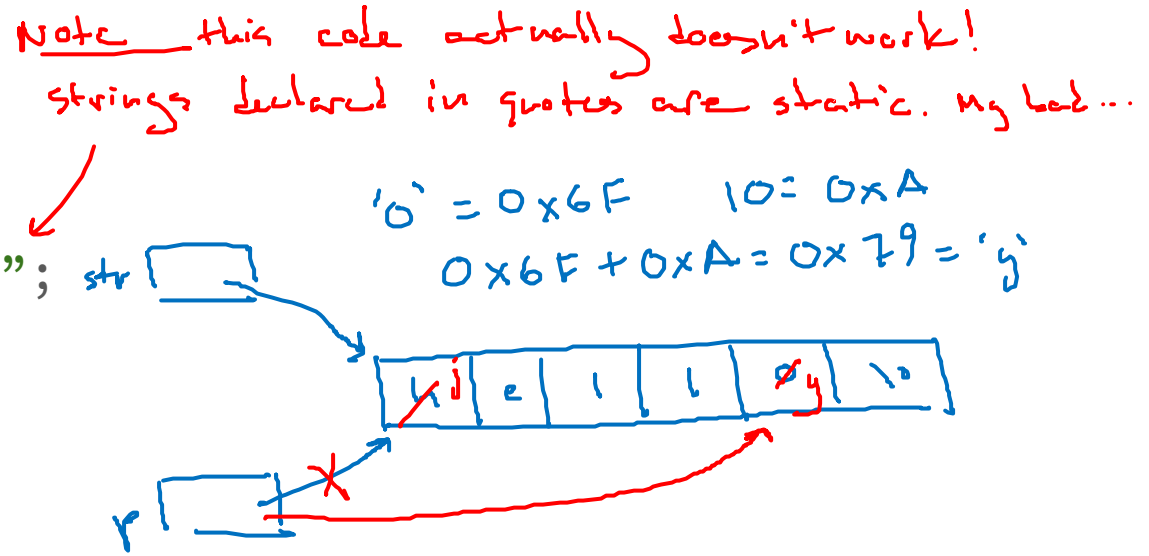




Polling Question

Consider the following code:

```
char* str = “hello”;
char* p = str;
*str = ‘j’;
p += 4;
*p = *p + 10;
```



After it’s finished, what will the value of `str` be? (Hint: draw a memory diagram!) You may want to look up an ASCII encoding chart.

Summary

- Variables are represented by locations in memory
  - Assignment in C stores a value in that location
- **Pointer arithmetic** scales by the size of the target type
  - Convenient when scanning array-like structures in memory
  - Be careful when using! Particularly when casting to another pointer type
- **Arrays** are stored as **contiguous** blocks of memory
  - Unlike Java, C does not do bounds checking on arrays!!
  - Array indexing is equivalent to doing pointer arithmetic, then dereferencing the result
  - **Strings** are **null-terminated** arrays of ASCII characters
    - Unlike Java, no dedicated String type