The Hardware/Software Interface

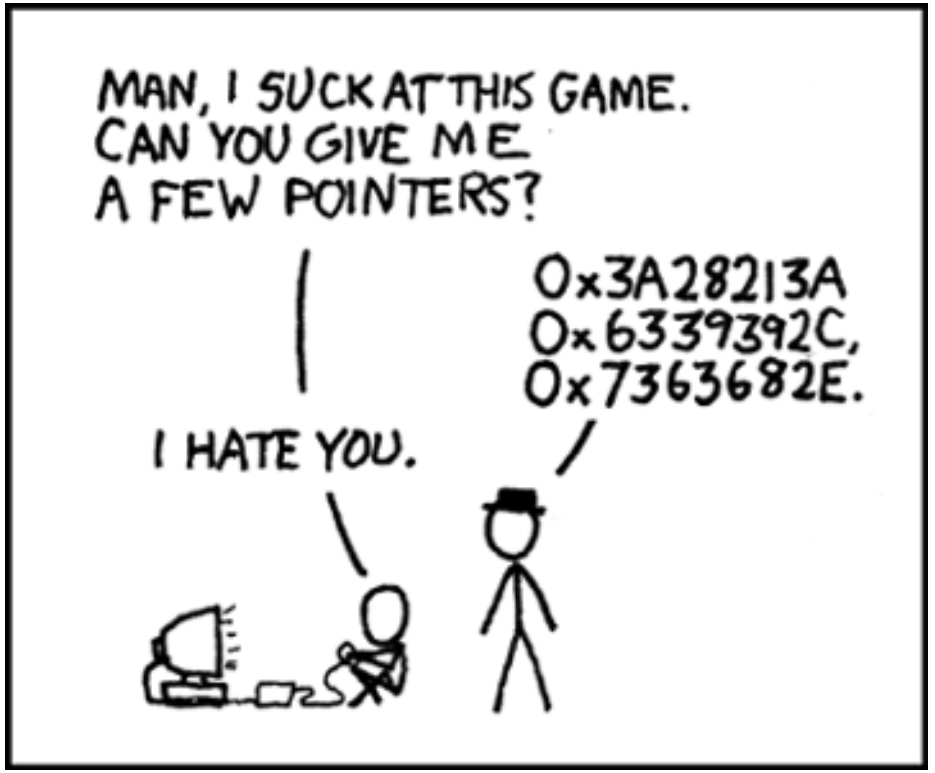
Memory, Data, & Addressing II

Instructors:

Justin Hsia, Amber Hu

Teaching Assistants:

Anthony Mangus
Grace Zhou
Jiuyang Lyu
Kurt Gu
Mendel Carroll
Naama Amiel
Rose Maresh
Violet Monserate
Divya Ramu
Jessie Sun
Kanishka Singh
Liander Rainbolt
Ming Yan
Pollux Chen
Soham Bhosale



http://xkcd.com/138/

Relevant Course Information

- Lab 0 due today @ 11:59 pm
  - *You will revisit the concepts & behavior from this program in future labs!*
- HW1 due tonight, HW2 due Wednesday, HW3 due Friday @ 11:59 pm
  - Autograded, unlimited tries, no late submissions
- Lab 1a released today, due next Monday (10/6)
  - Pointers in C
  - Last submission graded, can optionally work with a partner
    - One student submits, then adds their partner to the submission (*for every submission*)
  - Short answer "synthesis questions" for after the lab

Late Days

- You are given 6 late day tokens for the whole quarter
  - Tokens can only apply to Labs
  - No benefit to having leftover tokens
- 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 10% deduction of your score per day
  - Only late labs are 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

Lecture Outline (1/4)

- **Pointers**
- Pointer Arithmetic
- Arrays in C Introduction
- C “Strings”

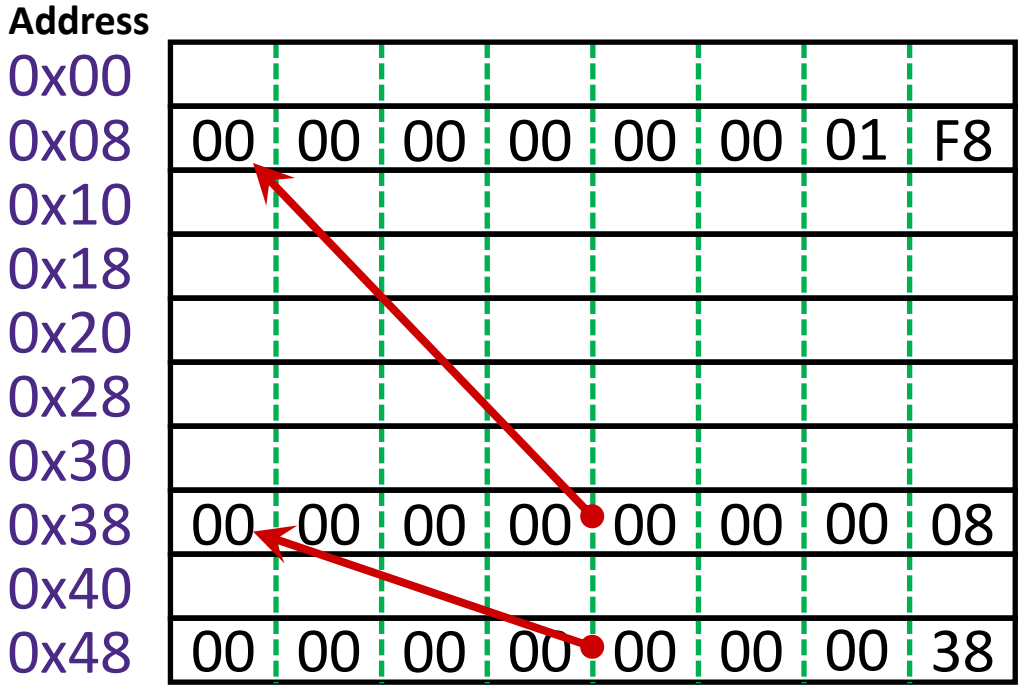
Data Types and Sizes (Revisited)

- A pointer is a data type that stores an address
  - Address size = word size

C Data Type	Java "Equivalent"	Size in bytes (x86-64)
`char`	`byte`	1
`short`	`short`	2
`int`	`int`	4
`long`		8
`long long`	`long`	8
`float`	`float`	4
`double`	`double`	8
`long double`		16
pointer (`type*`)	reference	8

Addresses and Pointers Example

- *Address* – refers to a location in memory
- *Pointer* – data object that stores/holds an address
- In this example, assume a 64-bit machine using big-endian
  1) Store 504 = 0x1F8 as 8 bytes at addr 0x08
  2) Store pointer pointing to 0x08 at addr 0x38
  3) Store pointer pointing to 0x38 at addr 0x48
     - Pointer to a pointer!
  - Was the original data (504) a pointer?
    - Could be, depending on how you use it



Pointers in C (Review, 1/3)

- Declaration: **`type*`** `ptr;` or **`type`** `*ptr;` (equivalent)
  - Word size (*e.g.*, 8 bytes on a 64-bit machine) to store addresses

- & = "address of" operator
  - ```
    int  q;
    int* p = &q;  // stores address of q in p
    ```

- * = "value at address" or "dereference" operator
  - ```
    int  q = 351;
    int* p = &q;
    int  r = *p;  // store the data pointed at by p in r
    ```

Pointers in C (Review, 2/3)

- Declaration: **`type*`** `ptr;` or **`type`** `*ptr;` (equivalent)
- `&` = "address of" operator
- `*` = "value at address" or "dereference" operator

- Operator confusion
  - The pointer operators are *unary* (*i.e.*, take 1 operand)
  - These operators both have *binary* forms
    - `x & y` is bitwise AND (we'll talk about this next lecture)
    - `x * y` is multiplication
  - `*` is also used as part of the data type in pointer variable declarations – this is NOT an operator in this context!

Pointers in C (Review, 3/3)

- Declaration: **`type*`** `ptr;` or **`type`** `*ptr;` (equivalent)
- & = "address of" operator
- `*` = "value at address" or "dereference" operator

- `NULL` is a constant for a pointer to "nothing"
  - Example: **`int*`** `p = NULL;`
  - Dereferencing `NULL` always results in a runtime error

Box-and-Arrow Diagrams

- Visual representation of C code with pointers
  - Boxes are variables, arrows connect pointers to target (NULL shown as Ø)
  - Useful for planning code and debugging

```
int x = 3, y;
int* p = &x;
int** pp = &p;
y = **pp + 2;
p = NULL;
y = *p - 2;
```

Lecture Outline (2/4)

- Pointers
- **Pointer Arithmetic**
- Arrays in C Introduction
- C “Strings”

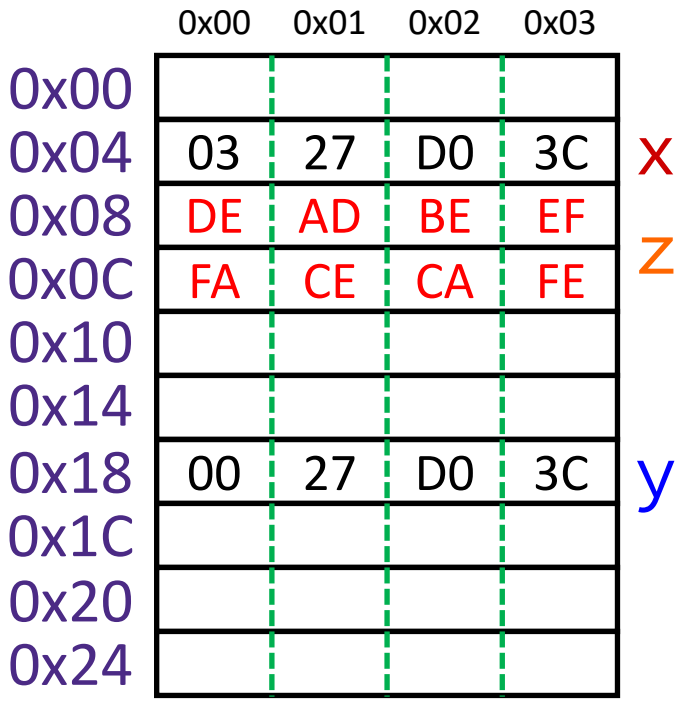
Pointer Arithmetic (Review)

- Pointer arithmetic is arithmetic (*e.g.*, +, –) performed on an expression that represents an address (*e.g.*, a pointer variable name)
  - The effect of the arithmetic operator is scaled by the size of the target type
    - Can consider this a change in units from bytes to the target type
    - Most commonly, adding constants to pointers and subtracting two pointers of the same type
- Examples:
  - For **`int*`** `p1`, `p1=p1+1` will increase its value by 4 (incremented by 1 **`int`**)
  - For **`long*`** `p2`, `p2=p2+1` will increase its value by 8 (incremented by 1 **`long`**)
  - For **`int*`** `p3` and **`int*`** `p4`, `p4-p3` will return the number of ***`int`***`s` between the two addresses
- ⚠️ Not all arithmetic operations are valid on pointers!

Assignment Example (Revisited, 1/3)

- Syntax: left-hand side (LHS) = right-hand side (RHS);
- Effect: store *value* of RHS into the *location* given by LHS

- Example: Little-endian, *partial* state of memory
  - `int x = 0, y = 0x3CD02700;`
  - `x = y + 3;`
  - `int* z;  // at address 0x08`

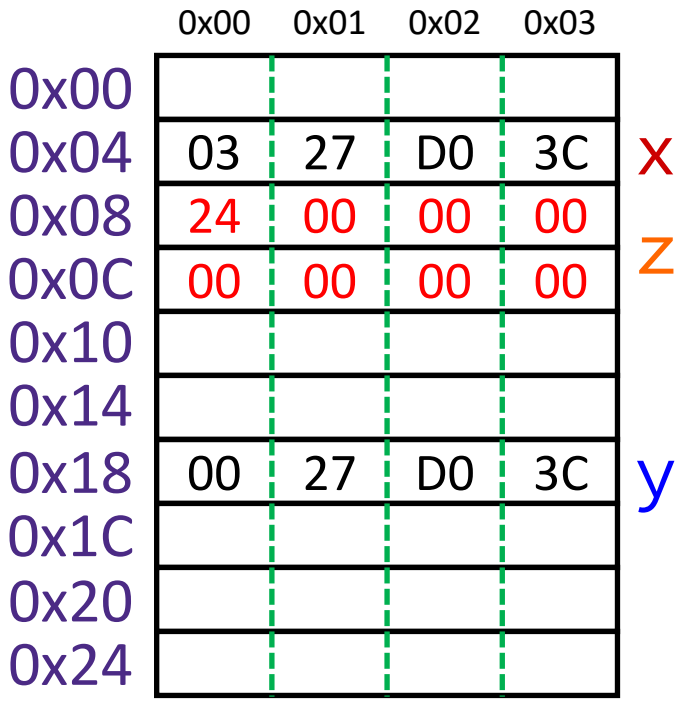


Assignment Example (Revisited, 2/3)

- Syntax: left-hand side (LHS) = right-hand side (RHS);
- Effect: store *value* of RHS into the *location* given by LHS

- Example: Little-endian, *partial* state of memory
  - `int x = 0, y = 0x3CD02700;`
  - `x = y + 3;`
  - `int* z;`
  - `z = &y + 3;`
    - Get address of y, "add 3", store in z

Pointer arithmetic

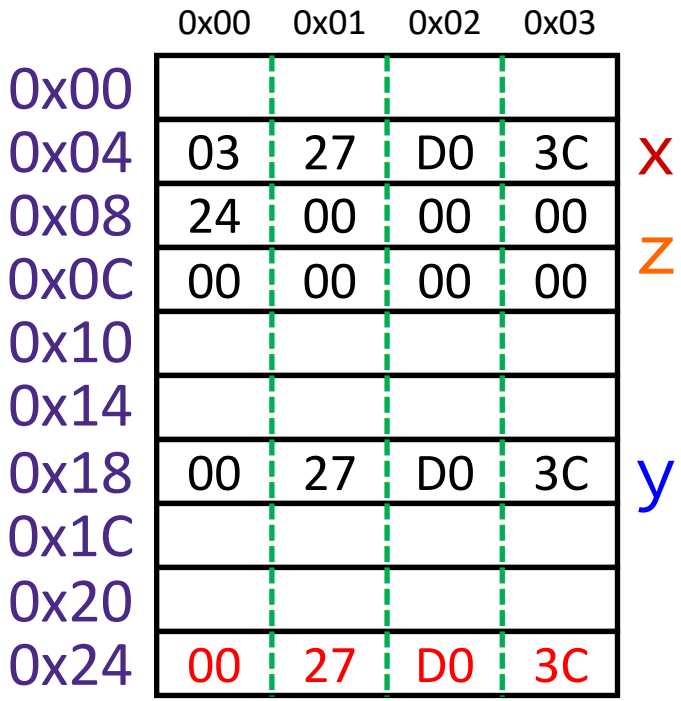


Assignment Example (Revisited, 3/3)

- Syntax: left-hand side (LHS) = right-hand side (RHS);
- Effect: store *value* of RHS into the *location* given by LHS

- Example: Little-endian, *partial* state of memory
  - `int x = 0, y = 0x3CD02700;`
  - `x = y + 3;`
  - `int* z;`
  - `z = &y + 3;`
    - Get address of y, 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



Pointer Arithmetic Warning

- Pointer arithmetic is arithmetic (*e.g.*, +, –) performed on an expression that represents an address (*e.g.*, a pointer variable name)
  - The effect of the arithmetic operator is scaled by the size of the target type
    - Can consider this a change in units from bytes to the target type
    - Most commonly, adding constants to pointers and subtracting two pointers of the same type

- ⚠️ Pointer arithmetic can be dangerous and can easily lead to bad memory accesses if you are not careful!
  - Be careful with data types and *casting*
    - Example: For `int* p`, the expression `(short*)p + 1` will actually scale by 2 instead of 4 because we are now treating the value in `p` as if it was a `short*`

Polling Questions (1/2)

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

- How much space does the variable p take up?

  A. 1 byte

  B. 2 bytes

  C. 4 bytes

  D. 8 bytes

- Which of the following expressions evaluate to an address?

  A. `x + 10`

  B. `p + 10`

  C. `&x + 10`

  D. `*(&p)`

  E. `ar[1]`

  F. `&ar[2]`
  ```

Aside: Java References

- In Java, everything that is not a primitive data type is an *object*
  - An object variable is actually a *"reference"* – a restricted pointer

```
class Record { ... }
Record x = new Record();
```

- Reference restrictions:
  - No pointer arithmetic, just reassignment
    - Reassignment must adhere to rules set by typing system (*e.g.*, inheritance)
  - References can only be "dereferenced" in ways that match class definition
    - *e.g.*, calling a method, accessing a field in object
- All higher-level languages use pointers/addresses under the hood, but likely abstracted away from the programmer

Lecture Outline (3/4)

- Pointers
- Pointer Arithmetic
- **Arrays in C Introduction**
- C “Strings”

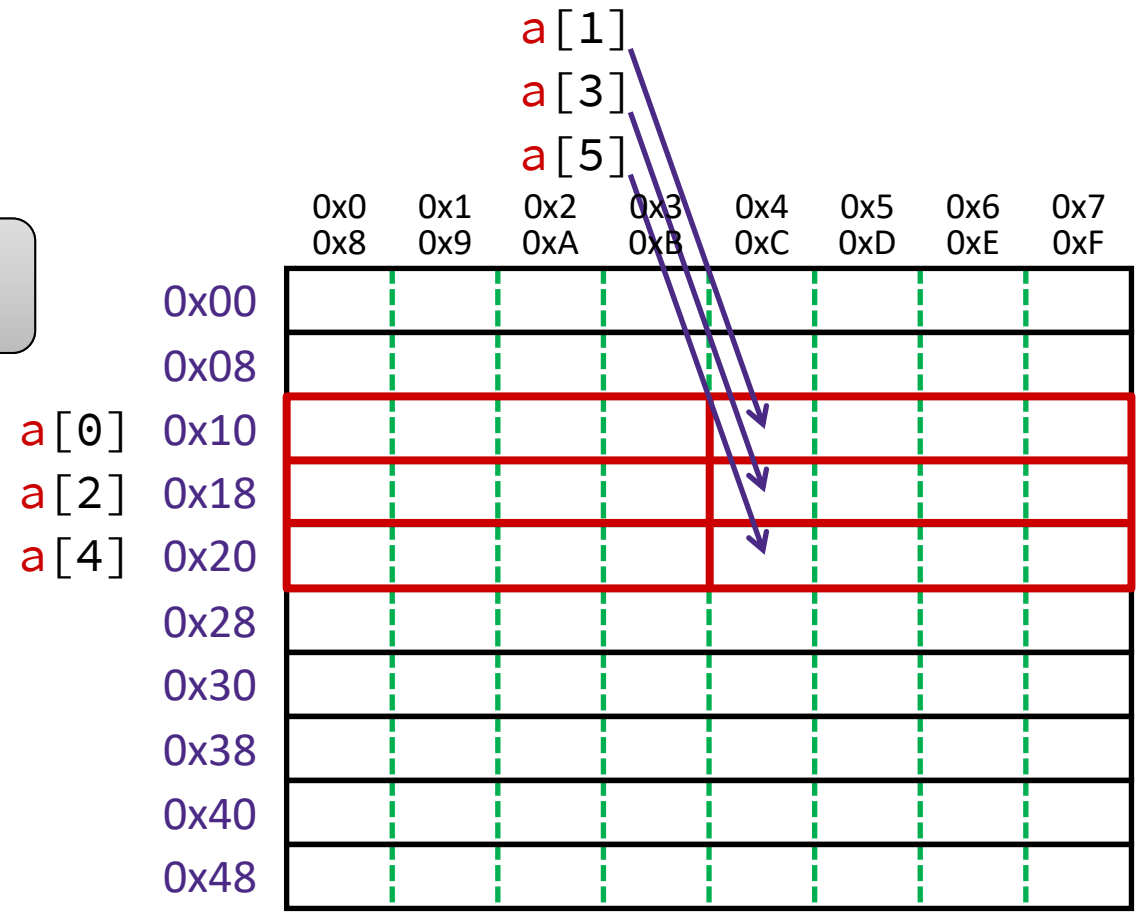
Arrays in C: Declaration (Review)

- **Arrays** are adjacent locations in memory storing the same type of data object
  - Declaration: `int a[6];`

element type

name

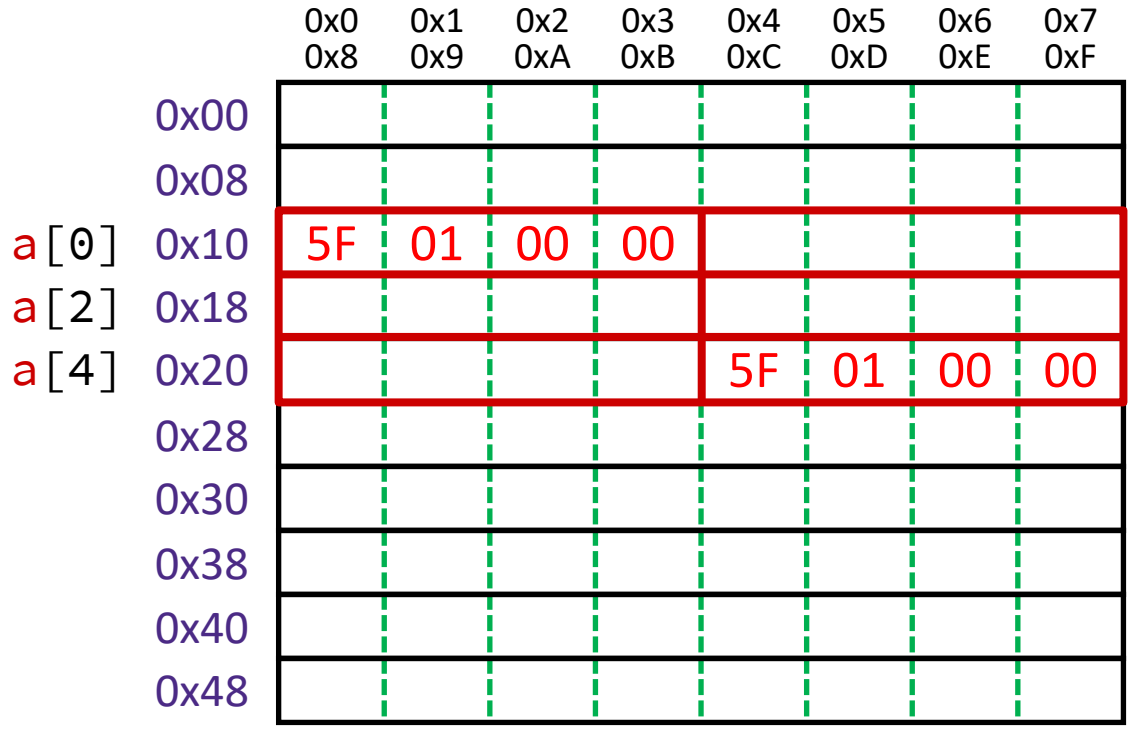
number of elements



Arrays in C: Indexing (Review)

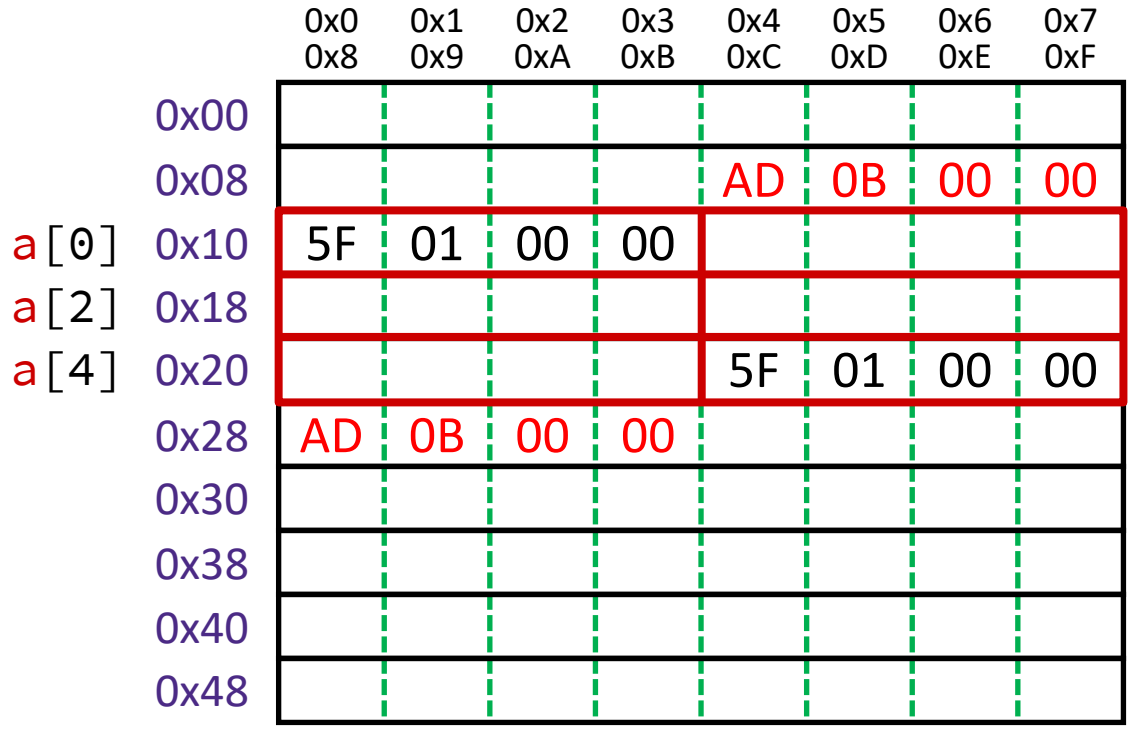
- **Arrays** are adjacent locations in memory storing the same type of data object
  - Declaration: `int a[6];`
  - Indexing:

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



Arrays in C: Lack of Bounds Checking (Review)

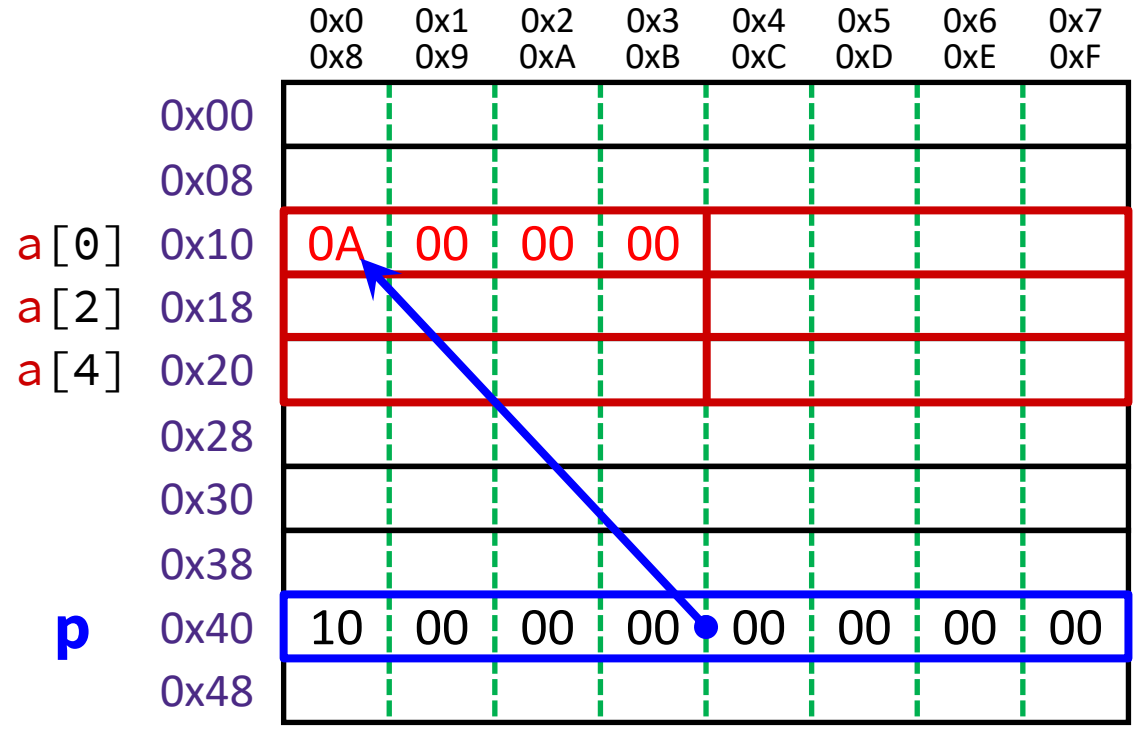
- **Arrays** are adjacent locations in memory storing the same type of data object
  - Declaration: `int a[6];`
  - Indexing:
    ```
    a[0] = 0x15F;
    a[5] = a[0];
    ```
  - No bounds checking:
    ```
    a[6] = 0xBAD;
    a[-1] = a[6];
    ```



Arrays and Pointers in C Example (1/4)

- Pointers are very handy when using arrays:
  - Using the name of an array in an expression evaluates to the array's address

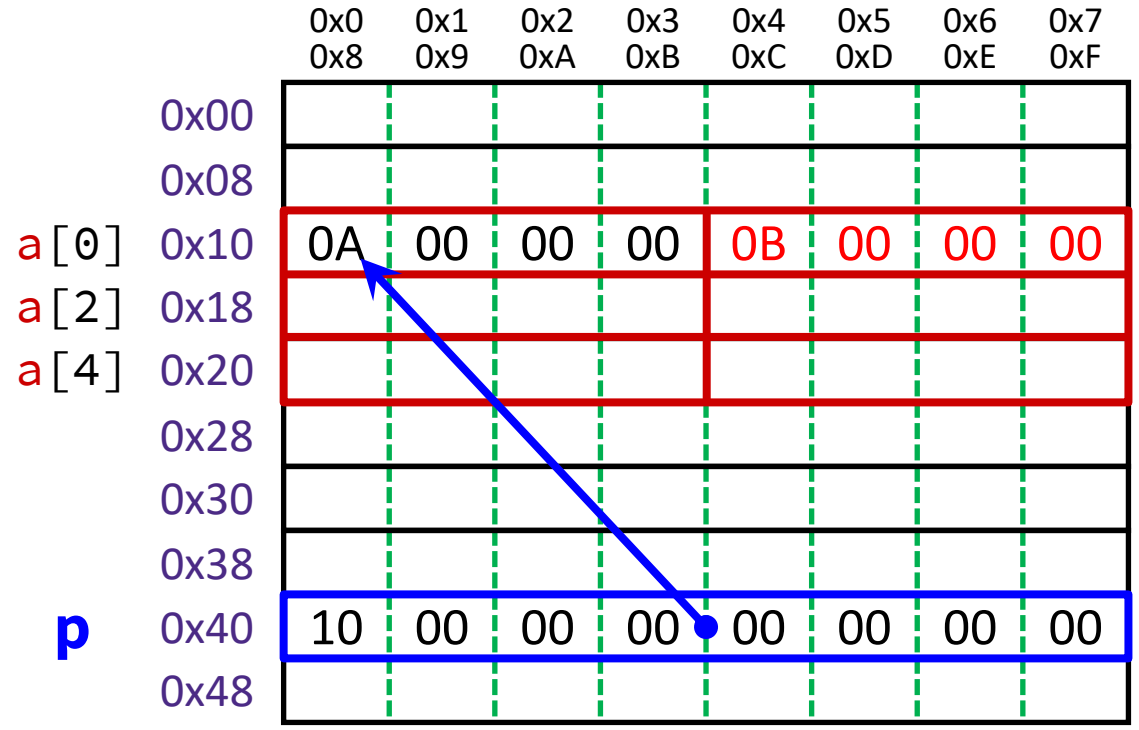
- Examples:
  - `int a[6];`
  - `int* p = a;  // or &a[0];`
  - `*p = 0xA;`



Arrays and Pointers in C Example (2/4)

- Pointers are very handy when using arrays:
  - Using the name of an array in an expression evaluates to the array's address
  - `a[i]` is actually `*(a + i)` and `&a[i]` is equivalent to `a+i`

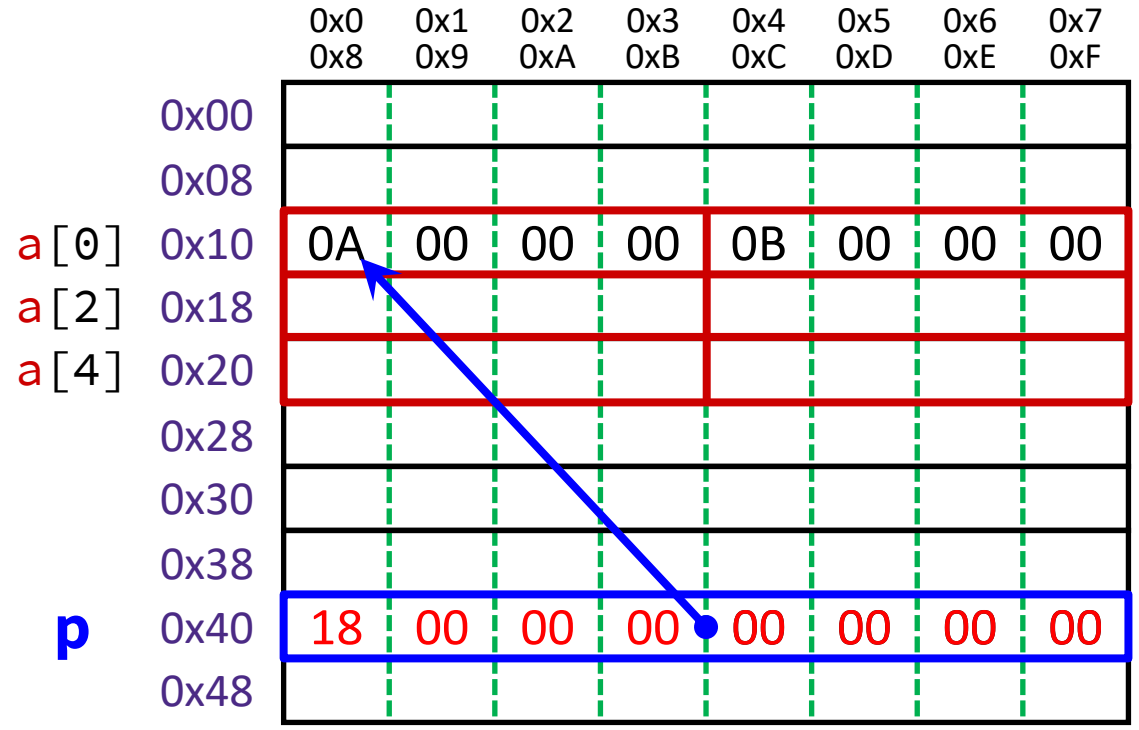
- Examples:
  - `int a[6];`
  - `int* p = a;  // or &a[0];`
  - `*p = 0xA;`
  - `p[1] = 0xB;  // or *(p+1)`



Arrays and Pointers in C Example (3/4)

- Pointers are very handy when using arrays:
  - Using the name of an array in an expression evaluates to the array's address
  - `a[i]` is actually `*(a + i)` and `&a[i]` is equivalent to `a+i`

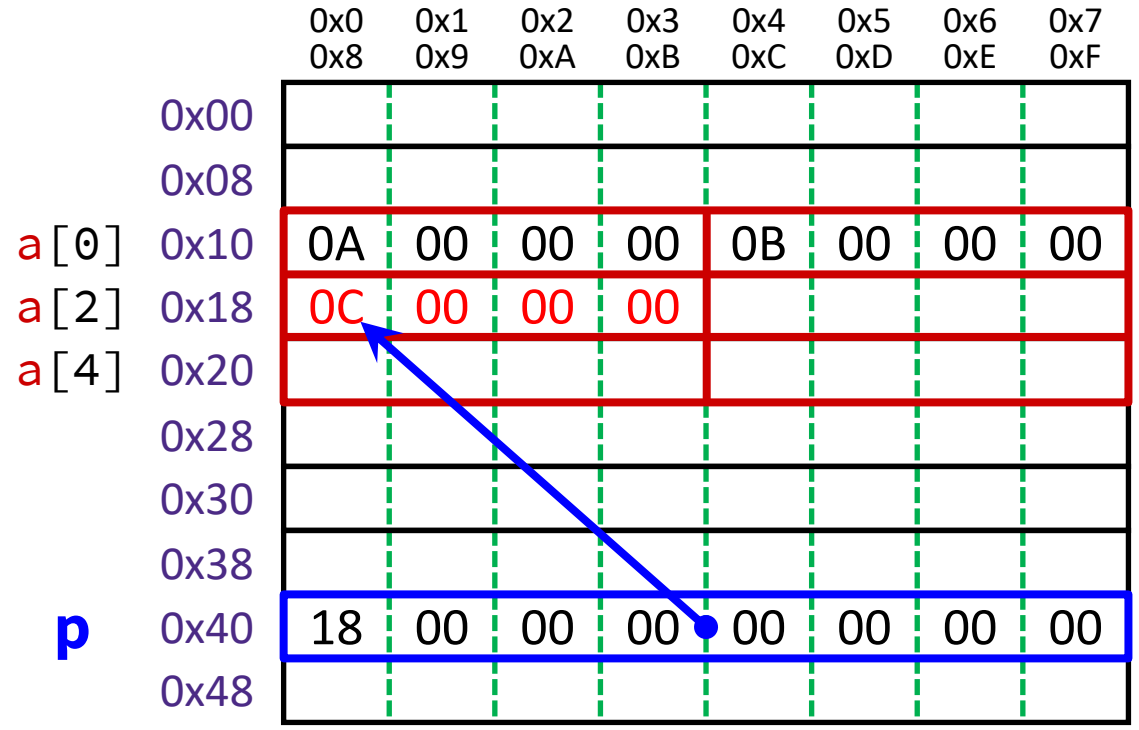
- Examples:
  - `int a[6];`
  - `int* p = a;  // or &a[0];`
  - `*p = 0xA;`
  - `p[1] = 0xB;  // or *(p+1)`
  - `p = p + 2;`



Arrays and Pointers in C Example (4/4)

- Pointers are very handy when using arrays:
  - Using the name of an array in an expression evaluates to the array's address
  - `a[i]` is actually `*(a + i)` and `&a[i]` is equivalent to `a+i`

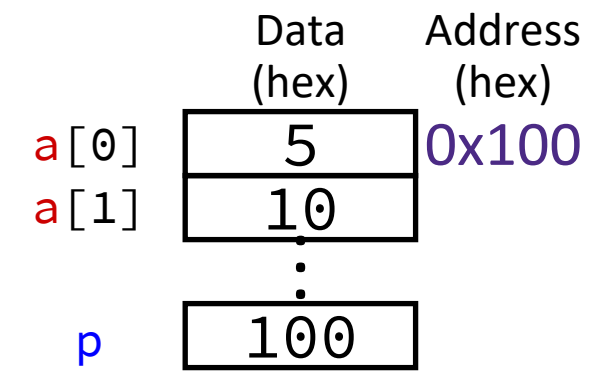
- Examples:
  - `int a[6];`
  - `int* p = a;  // or &a[0];`
  - `*p = 0xA;`
  - `p[1] = 0xB;  // or *(p+1)`
  - `p = p + 2;`
  - `*p = a[1] + 1;`



Polling Questions (2/2)

- The variable values after Line 3 executes are shown on the right. What are they after Line 5?

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



	p	a[0]	a[1]
(A)	0x101	0x5	0x11
(B)	0x104	0x5	0x11
(C)	0x101	0x6	0x10
(D)	0x104	0x6	0x10

Lecture Outline (4/4)

- Pointers
- Pointer Arithmetic
- Arrays in C Introduction
- **C "Strings"**

Representing Strings: ASCII

- C-style string stored as an array of bytes (**`char*`**)
  - No "`String`" keyword, unlike Java
  - Elements are one-byte *ASCII codes* for each character
    - Characters in C indicated with single quotes (*e.g.*, `'3'`) and evaluate to decimal constants

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

Representing Strings: Null Character (Review)

- C-style string stored as an array of bytes (**`char*`**)
  - No "`String`" keyword, unlike Java
  - Elements are one-byte ASCII codes for each character
    - Characters in C indicated with single quotes (*e.g.*, `'3'`) and evaluate to decimal constants
  - Last character followed by a 0 byte (`'\0'`, the *null character*)
    - Note that `'0'` ≠ `'\0'`
    - Example: **`char`** `str[] = "hi, you";` contains the following values

Decimal:	104	105	44	32	121	111	117	0
Hex:	0x68	0x69	0x2C	0x20	0x79	0x6F	0x75	0x00
Text:	`'h'`	`'i'`	`','`	`' '`	`'y'`	`'o'`	`'u'`	`'\0'`

Representing Strings: Literals (Review)

- C-style string stored as an array of bytes (**`char*`**)
  - No "`String`" keyword, unlike Java
  - Elements are one-byte ASCII codes for each character
    - Characters in C indicated with single quotes (*e.g.*, `'3'`) and evaluate to decimal constants
  - Last character followed by a 0 byte (`'\0'`, the null character)
    - Note that `'0'` ≠ `'\0'`
    - Example: **`char`** `str[] = "hi, you";` contains `{'h','i',' ','y','o','u','\0'}`

- A ***string literal*** (or *string constant*) indicated by double quotes (*e.g.*, `"hi, you"`) and automatically stored as a char array in memory
  - Space for `'\0'` included
  - Cannot be manipulated (need to copy into another char array first)

Printing Strings

- To print a string, use `printf` with the format specifier `%s`
  - Argument is the address of the string
  - Prints out the characters until it finds the first null character

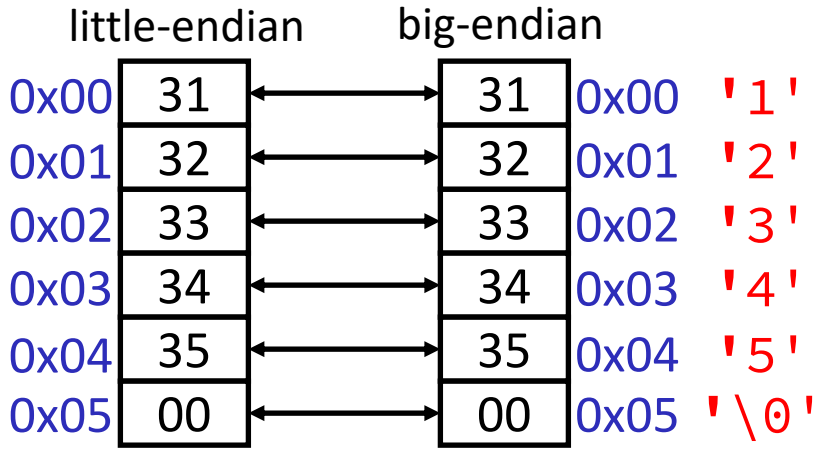
- Example:

```
// Curious about these numbers? See asciitable.com.
char str[] = {89, 111, 33, 0, 121, 111, 117};
// notice that it stops at ^
printf("%s\n", str);
```

Endianness and Strings

- Byte ordering (endianness) is not an issue for 1-byte values
  - The whole array does not constitute a single value
  - Individual elements are values; **char**s are single bytes

- Example:
  - `char s[6] = "12345";`



Examining Data Representations: Code

- Code to print byte representation of data
  - Treat any data type as a *byte array* by **casting** its address 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%.2hhX\n", start+i, *(start+i));
  printf("\n");
}
```

- `printf` legend:
  - Special characters: `\t` = Tab, `\n` = newline
  - Format specifiers: `%p` = pointer,
    `%.2hhX` = 1 byte (hh) in hex (X), padding to 2 digits (`.2`)

Examining Data Representations: Usage

- Code to print byte representation of data
  - Treat any data type as a *byte array* by **casting** its address 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%.2hhX\n", start+i, *(start+i));
  printf("\n");
}
```

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

`show_bytes` Execution Example

```
int x = 123456;  // 0x00 01 E2 40
printf("int x = %d;\n", x);
show_int(x);     // show_bytes((char *) &x, sizeof(int));
```

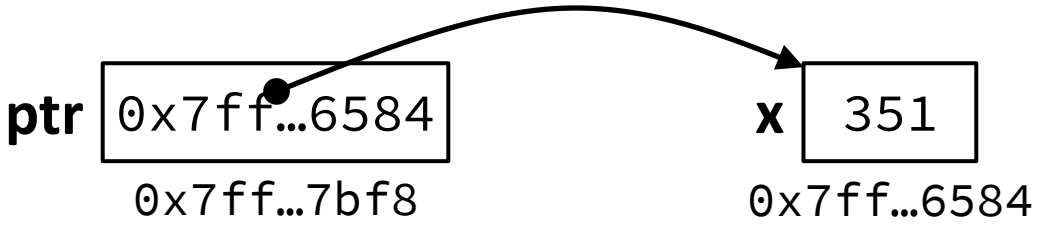
- Result (Linux x86-64):
  - **Note:** The addresses will change on each run (try it!), but fall in same general range

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

Summary (1/2)

- Pointers are data objects that hold addresses
  - Type of pointer determines size of thing being pointed at, which could be another pointer
  - `&` = "address of" operator
  - `*` = "value at address" or "dereference" operator
  - `NULL` is a constant for a pointer to "nothing"

- Can visualize using box-and-arrow diagrams:



Summary (2/2)

- Arrays are adjacent locations in memory storing the same type of data
  - Strings are null-terminated arrays of characters (ASCII)
- Pointer arithmetic scales by size of target type
  - Convenient when accessing array-like structures in memory: `a[i]` ↔ `*(a + i)`
  - Be careful when using – particularly when *casting* variables