

Memory, Data, & Addressing II

CSE 351 Winter 2022

Instructor:

Kyrie Dowling

Teaching Assistants:

Aakash Srazali

Allie Pflieger

Ellis Haker



<http://xkcd.com/138/>

Relevant Course Information

- ❖ hw1 due tonight @ 11:59 pm
- ❖ Lab 0 due tonight @ 11:59 pm
 - *You will revisit the concepts from this program!*
- ❖ hw2 due Wed (6/29), hw3 due Friday
 - Autograded, unlimited tries, no late submissions
- ❖ Lab 1a released today, due a week from Wed (7/06)
 - Pointers in C
 - Last submission graded, can optionally work with a partner
 - One student submits, then add their partner to the submission
 - Short answer “synthesis questions” for after the lab

Late Days

- ❖ You are given **5 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 20% 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

Reading Review

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

- ❖ Questions from the Reading?

64-bit example
(pointers are 64-bits wide)

Review Questions

- ❖

```
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. $\ast(\&p)$
 - E. $ar[1]$
 - F. $\&ar[2]$

Pointer Operators

- ❖ $\&$ = “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!

Assignment in C

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

little-endian

- ❖ A variable is represented by a location
- ❖ Declaration \neq initialization (initially “mystery data”)
- ❖ **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	

Assignment in C

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

little-endian

- ❖ A variable is represented by a location
- ❖ Declaration \neq initialization (initially “mystery data”)
- ❖ **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

- ❖ 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;

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

& = "address of"

* = "dereference"

	0x00	0x01	0x02	0x03	
0x00					
0x04	00	00	00	00	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"

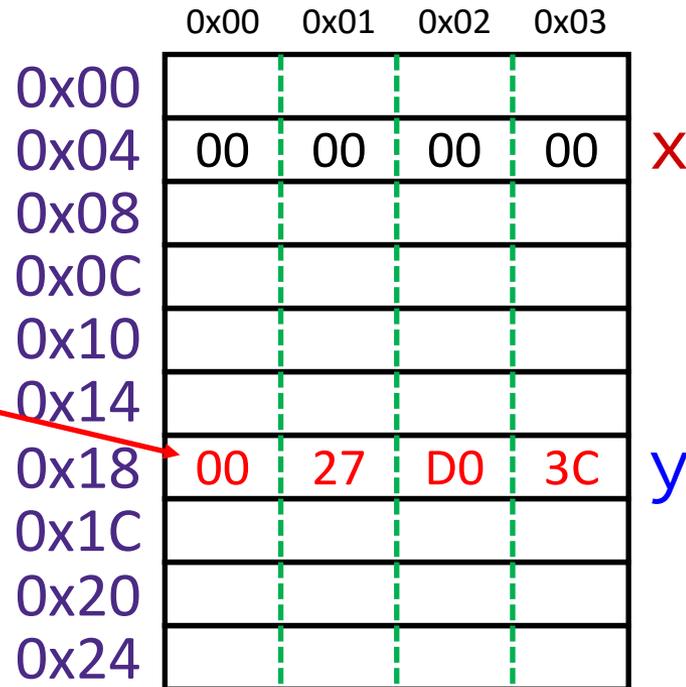
- ❖ left-hand side = right-hand side;
 - LHS must evaluate to a *location*
 - RHS must evaluate to a *value* (could be an address)
 - Store RHS value at LHS location

❖ **int** x, y;

❖ x = 0;

❖ 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

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

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"

	0x00	0x01	0x02	0x03	
0x00					
0x04	03	27	D0	3C	X
0x08					
0x0C					
0x10					
0x14					
0x18	00	27	D0	3C	y
0x1C					
0x20	DE	AD	BE	EF	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*

	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

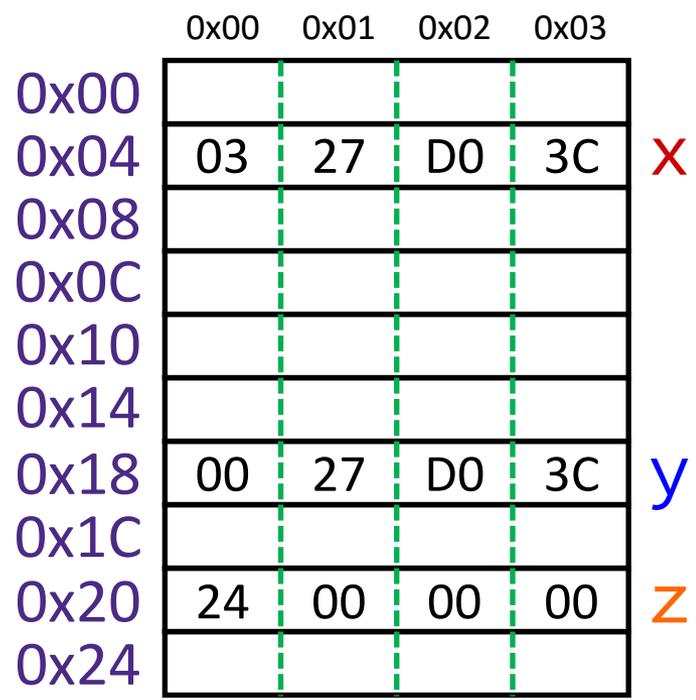
Assignment in C

- ❖ `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;`

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

& = "address of"
* = "dereference"



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 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

	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	

Addresses and Pointers in C (Review)

- ❖ Draw out a box-and-arrow diagram for the result of the following C code:

```
int* ptr;
```

```
int x = 5;
```

```
int y = 2;
```

```
ptr = &x;
```

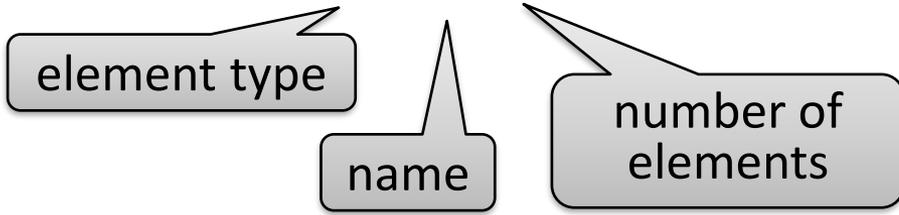
```
y = 1 + *ptr;
```

Arrays in C

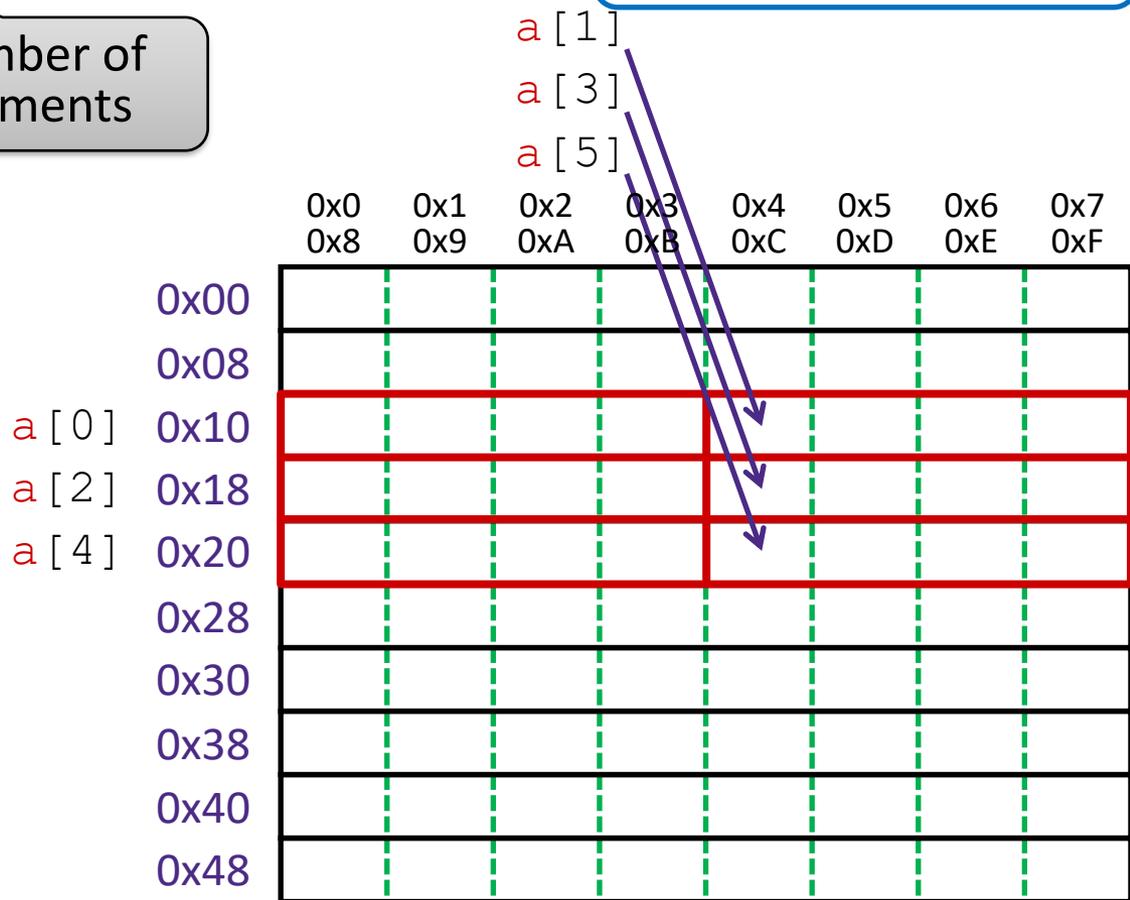
Arrays are adjacent locations in memory storing the same type of data object

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

Declaration: `int a[6];`



64-bit example
(pointers are 64-bits wide)



Arrays in C

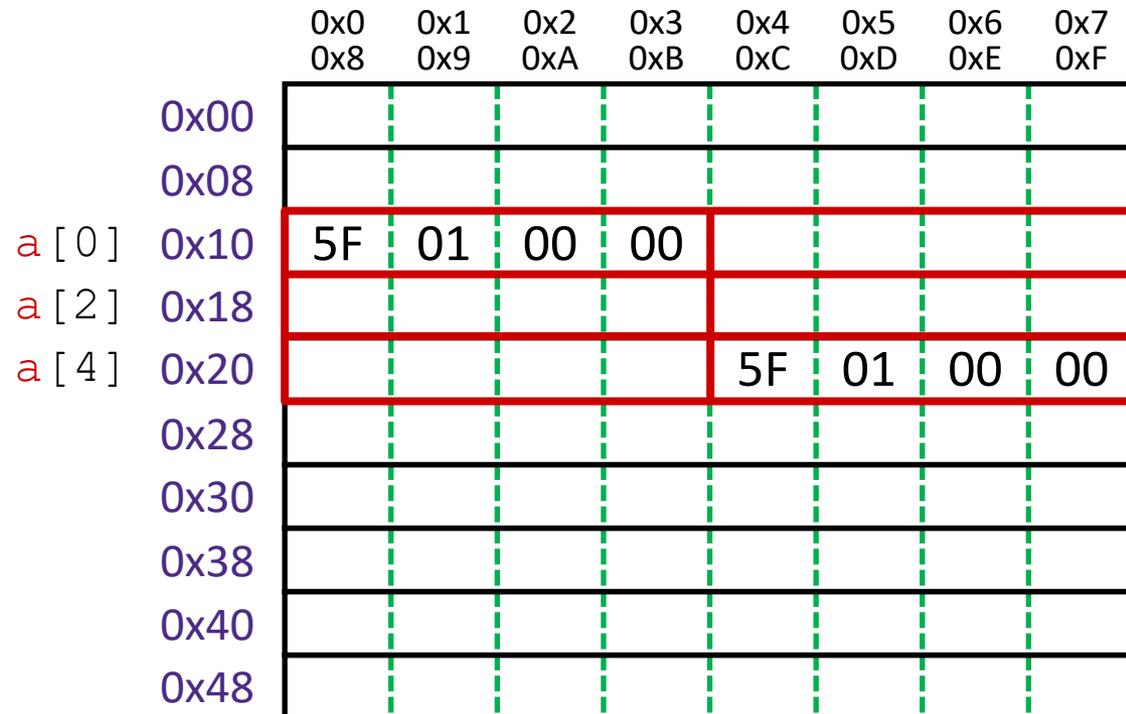
Declaration: `int a[6];`

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

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

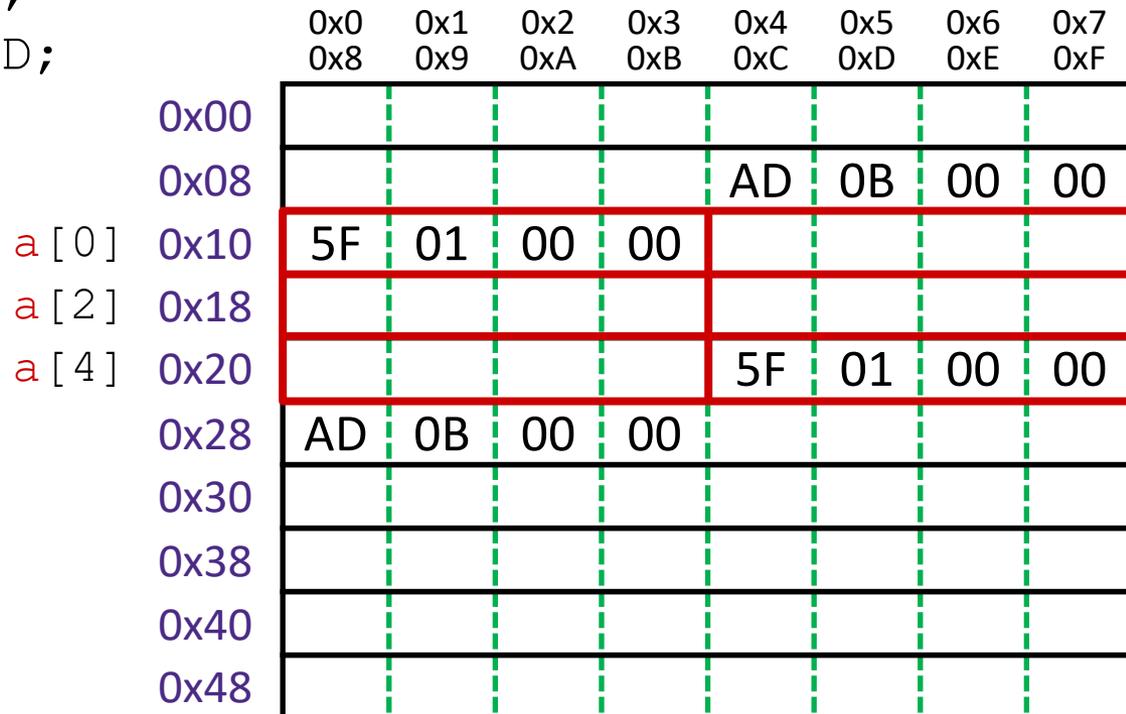
`a` (array name) returns the array's address

`&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;`



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];`

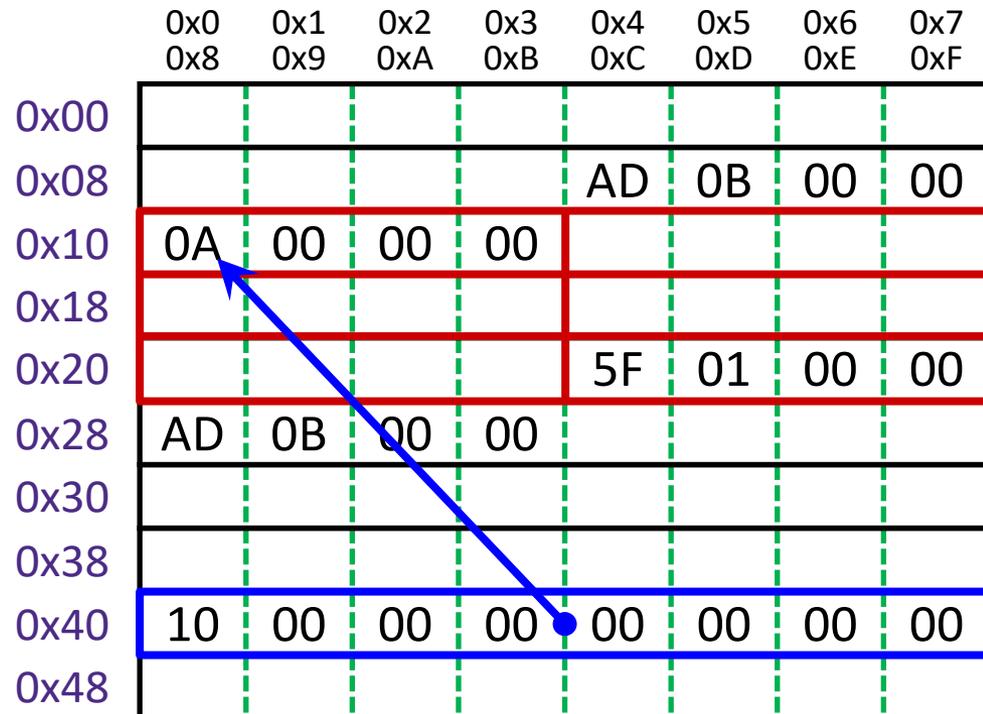
Indexing: `a[0] = 0x015f;`
`a[5] = a[0];`

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

Pointers: `int* p;`

equivalent $\left\{ \begin{array}{l} p = a; \\ p = \&a[0]; \\ *p = 0xA; \end{array} \right.$ `a[0]`
`a[2]`
`a[4]`

`p`



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;`

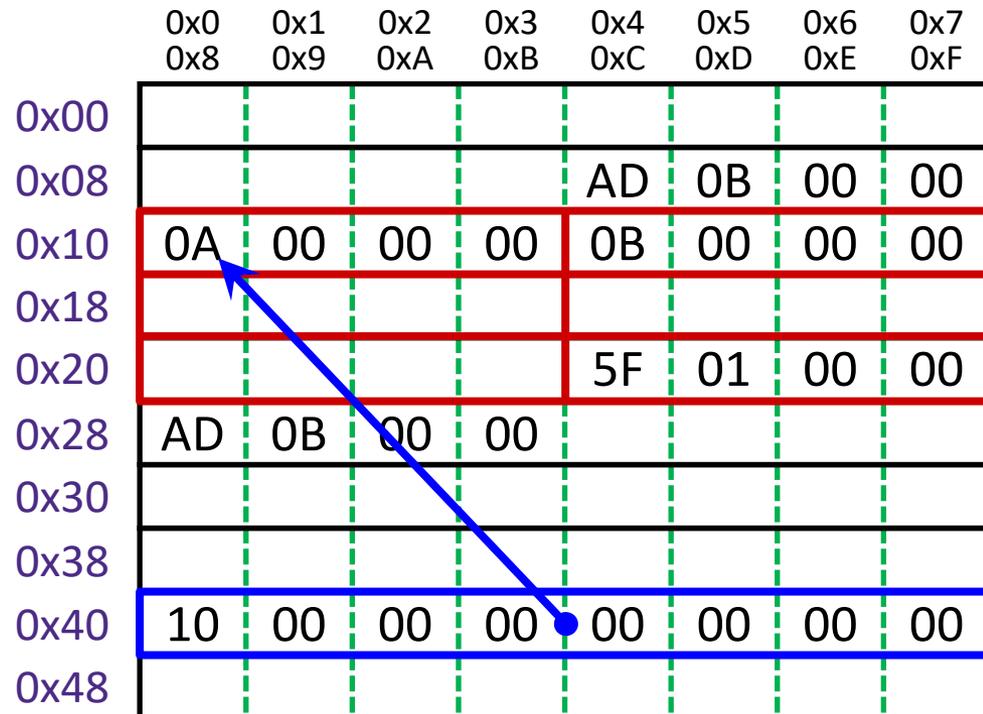
equivalent $\left\{ \begin{array}{l} p = a; \\ p = \&a[0]; \\ *p = 0xA; \end{array} \right.$

`a[0]`
`a[2]`
`a[4]`

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

equivalent $\left\{ \begin{array}{l} p[1] = 0xB; \\ *(p+1) = 0xB; \\ p = p + 2; \end{array} \right.$

`p`



Arrays in C

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 $\left\{ \begin{array}{l} p = a; \\ p = \&a[0]; \\ *p = 0xA; \end{array} \right.$

`a[0]`
`a[2]`
`a[4]`

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

equivalent $\left\{ \begin{array}{l} p[1] = 0xB; \\ *(p+1) = 0xB; \\ p = p + 2; \end{array} \right.$

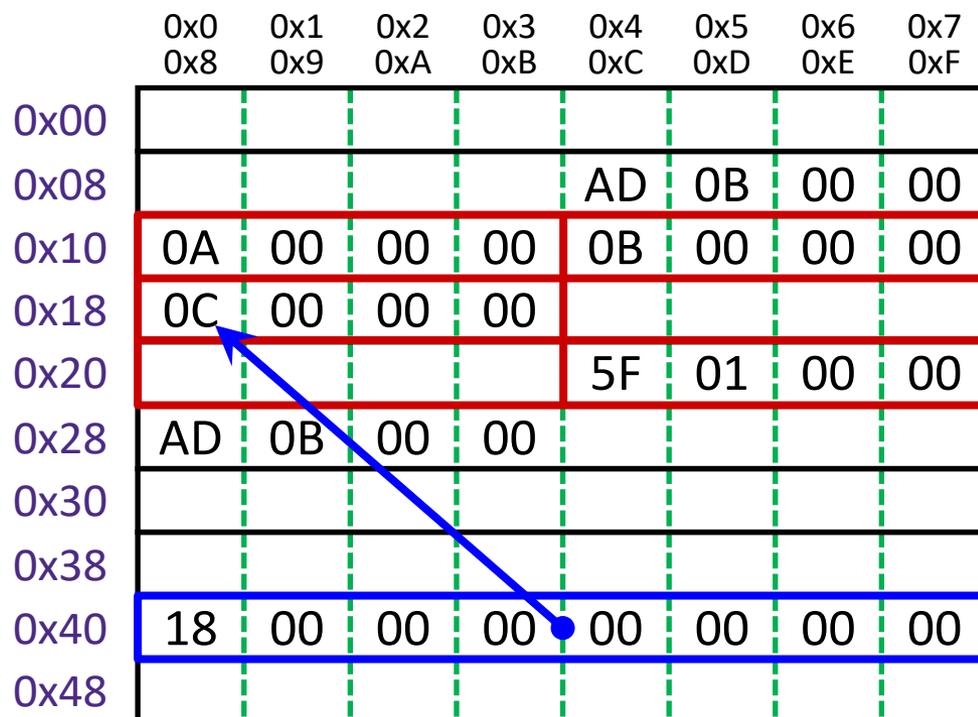
`p`

`*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



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

- Vote in Ed!

```

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

	Data (hex)	Address (hex)
a[0]	5	0x100
a[1]	10	
	⋮	
p	100	

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

Representing strings (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

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 (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
 - Last character followed by a 0 byte (`' \ 0 '`) (a.k.a. the **null character**)

<i>Decimal:</i>	83	117	110	32	105	115	32	102	117	110	33	33	0
<i>Hex:</i>	0x53	0x75	0x6E	0x20	0x69	0x73	0x20	0x66	0x75	0x6E	0x21	0x21	0x00
<i>Text:</i>	'S'	'u'	'n'	' '	'i'	's'	' '	'f'	'u'	'n'	'!'	'!'	'\0'

Endianness and Strings

C (char = 1 byte)

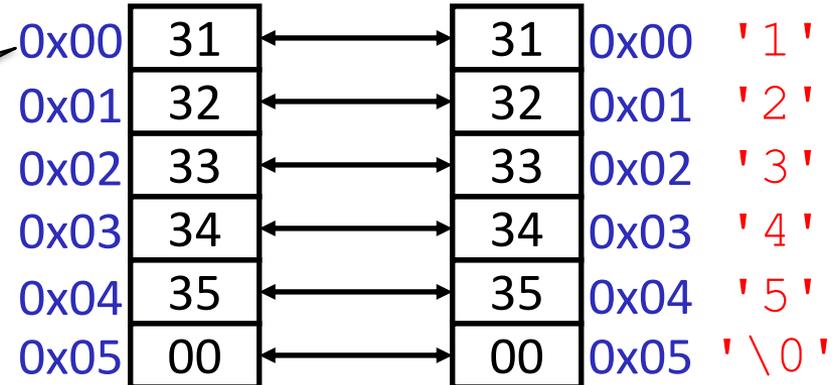
```
char s[6] = "12345";
```

String literal

0x31 = 49 decimal = ASCII '1'

IA32, x86-64
(little-endian)

SPARC
(big-endian)



- ❖ 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

Examining Data Representations

❖ 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` directives:

- `%p` Print pointer
- `\t` Tab
- `%.2hhX` Print value as char (hh) in hex (X), padding to 2 digits (.2)
- `\n` New line

Examining Data Representations

❖ 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

- ❖ 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)