Memory, Data, & Addressing II

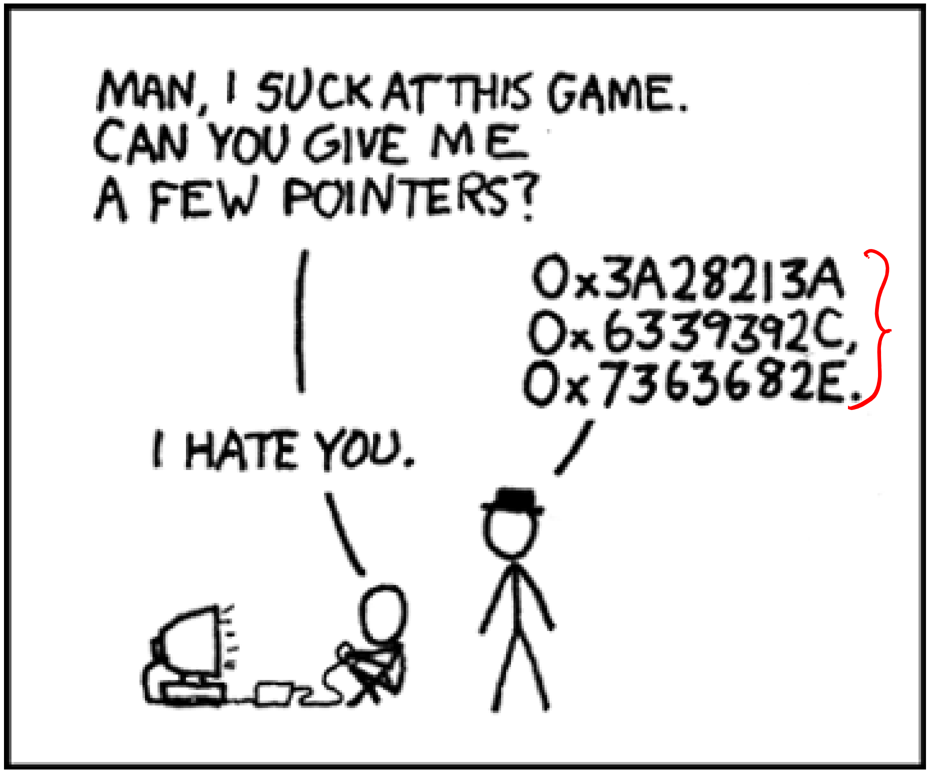
CSE 351 Spring 2020

Instructor:

Ruth Anderson

Teaching Assistants:

Alex Olshanskyy
Rehaan Bhimani
Callum Walker
Chin Yeoh
Diya Joy
Eric Fan
Edan Sneh
Jonathan Chen
Jeffery Tian
Millicent Li
Melissa Birchfield
Porter Jones
Joseph Schafer
Connie Wang
Eddy (Tianyi) Zhou



32-bit addresses

http://xkcd.com/138/

Administrivia

- Assignments – Nothing Due this week!
- Lecture Video to watch before Monday (4/06)
  - Not a full lecture, but part of one to help us retain time in lectures for small group interaction.
  - Posted later today, should be watched before lecture on Monday
- Pre-Course Survey, hw0, hw1, hw2 due Mon (4/06) – 11:59pm
- Lab 0 due Tuesday (4/07) – 11:59pm
- hw3 due Wednesday (4/08) – 11am
  - Autograded, unlimited tries, no late submissions
- Lab 1a released today, due next Monday (4/13)
  - Pointers in C
  - Reminder: last submission graded, *individual* work
- Feedback for today: https://catalyst.uw.edu/webq/survey/rea2000/387932

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
- Late penalty is 20% deduction of your score per day
  - Only late work is 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

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

Addresses and Pointers in C

* is also used with variable declarations

- `&` = "address of" operator
- `*` = "value at address" or "dereference" operator

datatype

equivalent

```
int* ptr;
int *ptr;
```

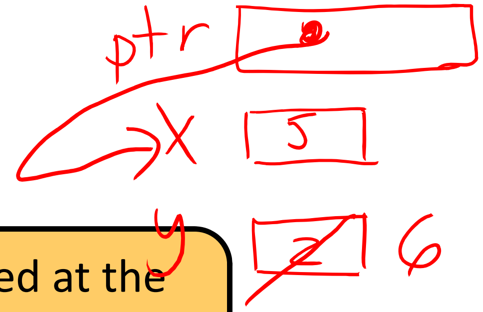
Declares a variable, `ptr`, that is a pointer to (*i.e.* holds the address of) an `int` in memory

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

Declares two variables, `x` and `y`, that hold `int`s, 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)` ?

returns value stored in y (equivalent to just using 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

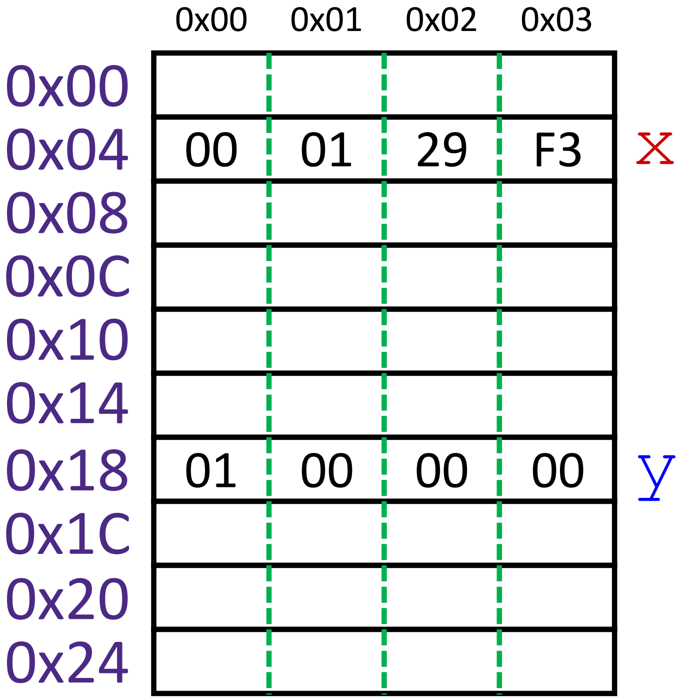
	0x00	0x01	0x02	0x03	
0x00	A7	00	32	00	
0x04	00	01	29	F3	x
0x08	EE	EE	EE	EE	
0x0C	FA	CE	CA	FE	
0x10	26	00	00	00	
0x14	00	00	10	00	
0x18	01	00	00	00	y
0x1C	FF	00	F4	96	
0x20	DE	AD	BE	EF	
0x24	00	00	00	00	

Assignment in C

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

little-endian

- 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;` pad 0x 00 00 00 00

int (4 bytes)

	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

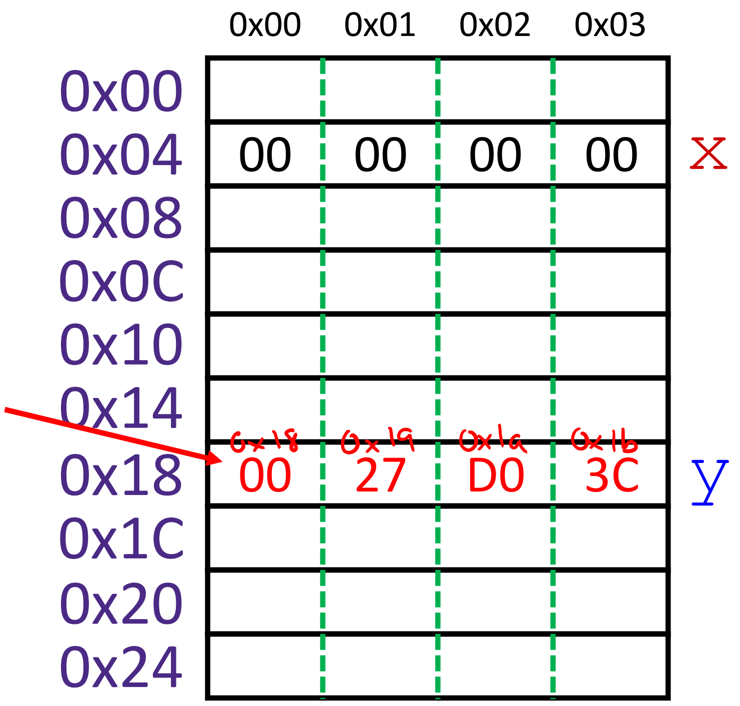
32-bit example
(pointers are 32-bits wide)

`&` = “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;`

least significant byte

little endian!



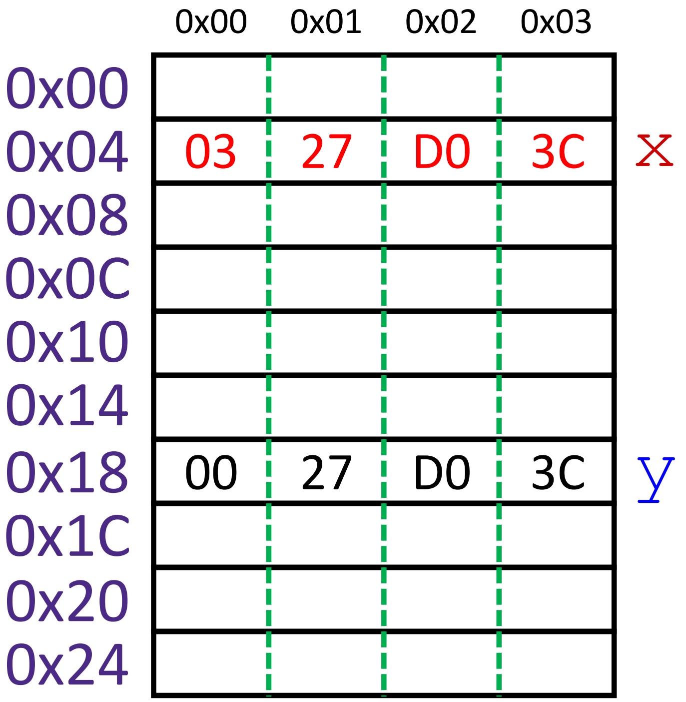
	0x00	0x01	0x02	0x03	
0x00					
0x04	00	00	00	00	x
0x08					
0x0C					
0x10					
0x14					
0x18	0x18 00	0x19 27	0x1a D0	0x1b 3C	y
0x1C					
0x20					
0x24					

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;`
- `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

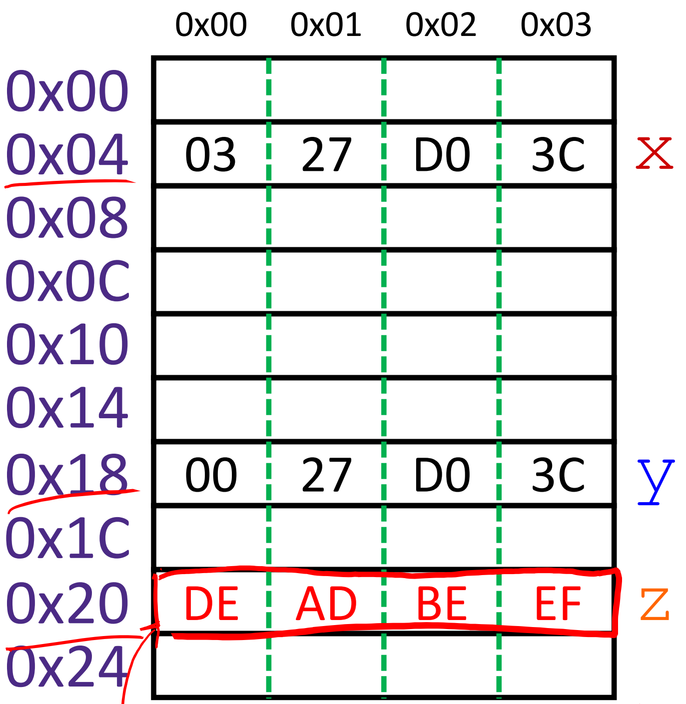
32-bit example
(pointers are 32-bits wide)

`&` = "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;`

pointer to an int

  - `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	DE	AD	BE	EF	z
0x24					

initial value is whatever bits were already there! ("garbage")

Assignment in C

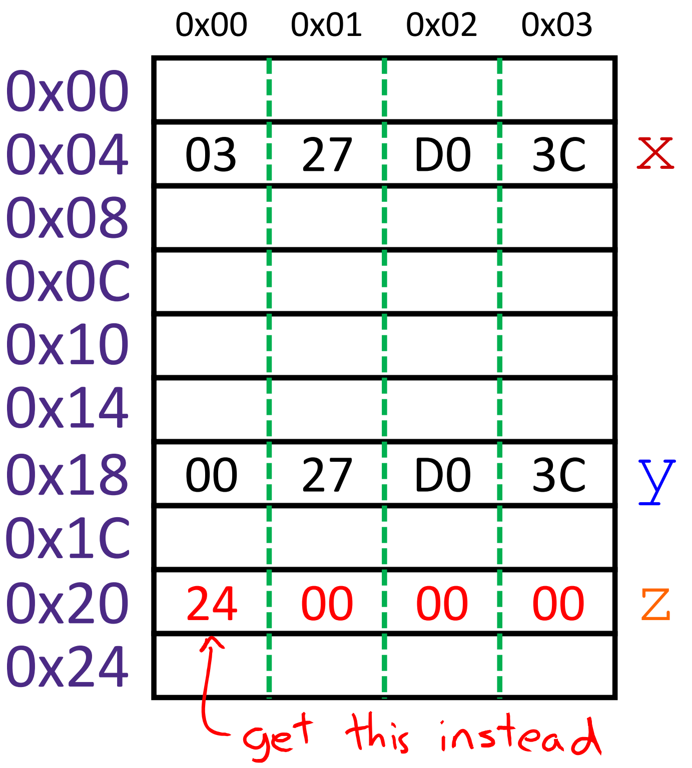
32-bit example
(pointers are 32-bits wide)

& = "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;` // expect 0x1b
  - Get address of `y`, "add 3", store in `z`

0x18

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					

get this instead

Pointer Arithmetic

- Pointer arithmetic is scaled by the size of target type
  - In this example, `sizeof(int)` = 4
- `int* z = &y + 3;`

  4 bytes

  - 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
  - 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 address of `y`, 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					

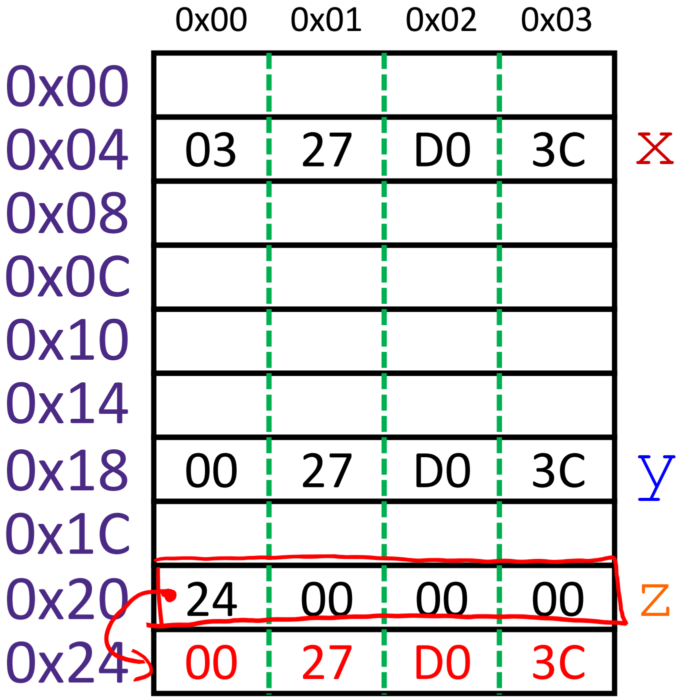
Assignment in C

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 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	

Arrays in C

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

a (array name) returns the array's address

4 bytes each

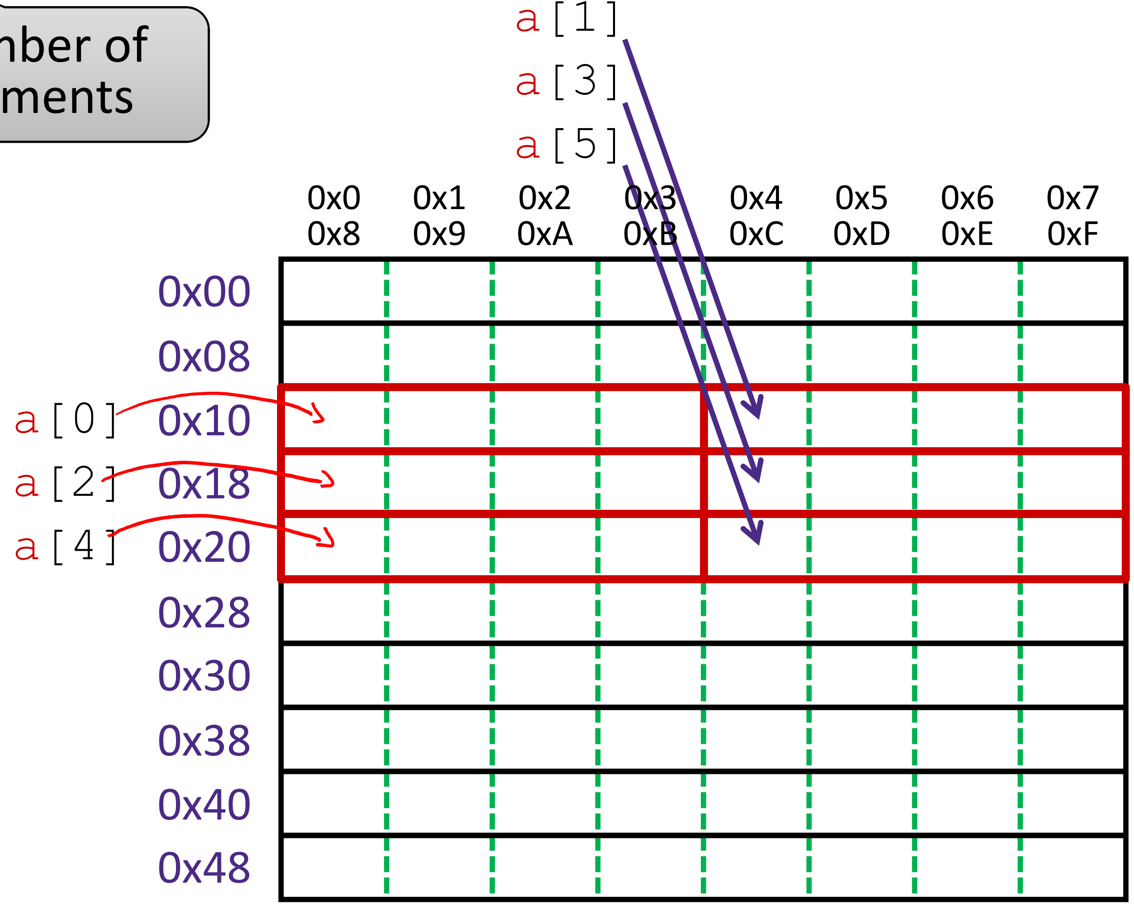
Declaration: **`int`** `a[6];` // &a is 0x10

element type

name

number of elements

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



Arrays in C

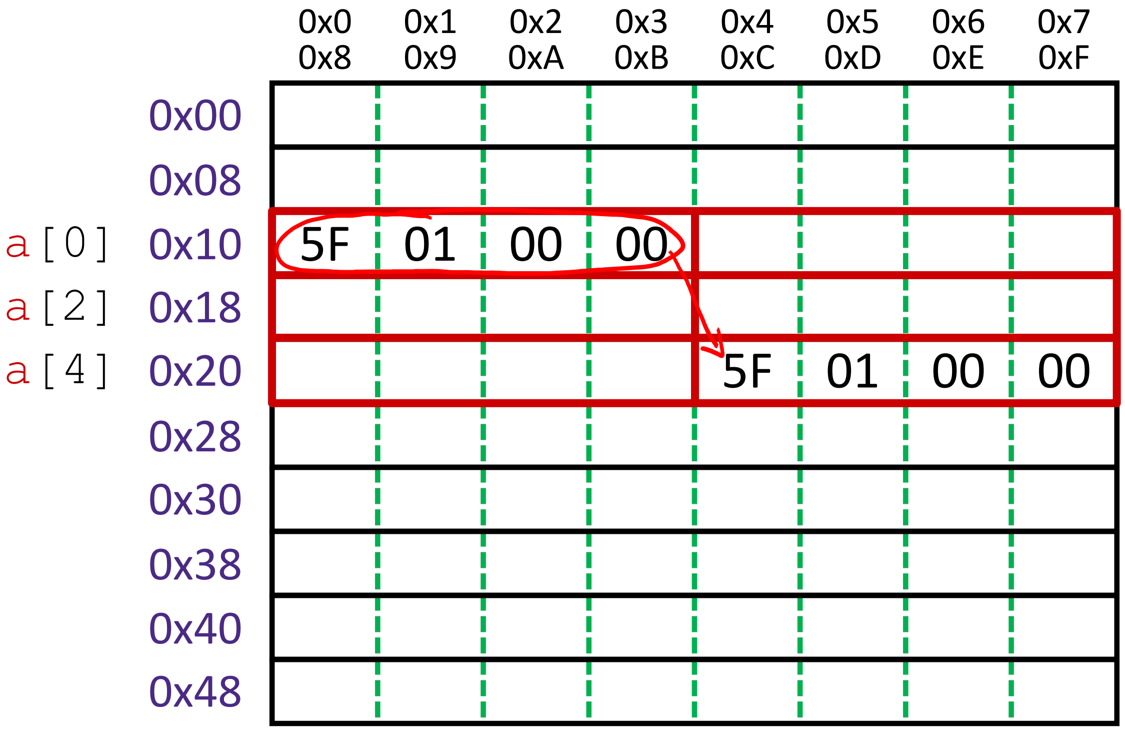
Arrays are adjacent locations in memory storing the same type of data object

`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

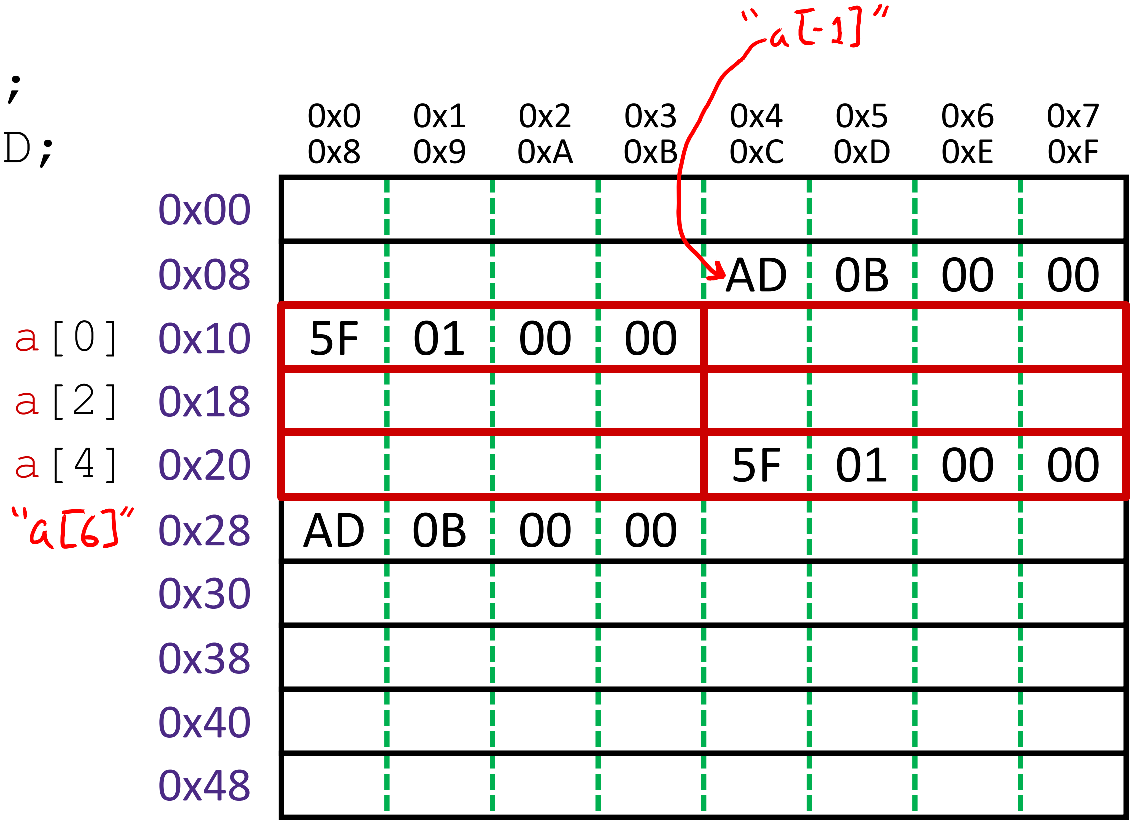
`a` (array name) returns the array's address

`&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;`



"a[-1]"

		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
"a[6]"	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];`

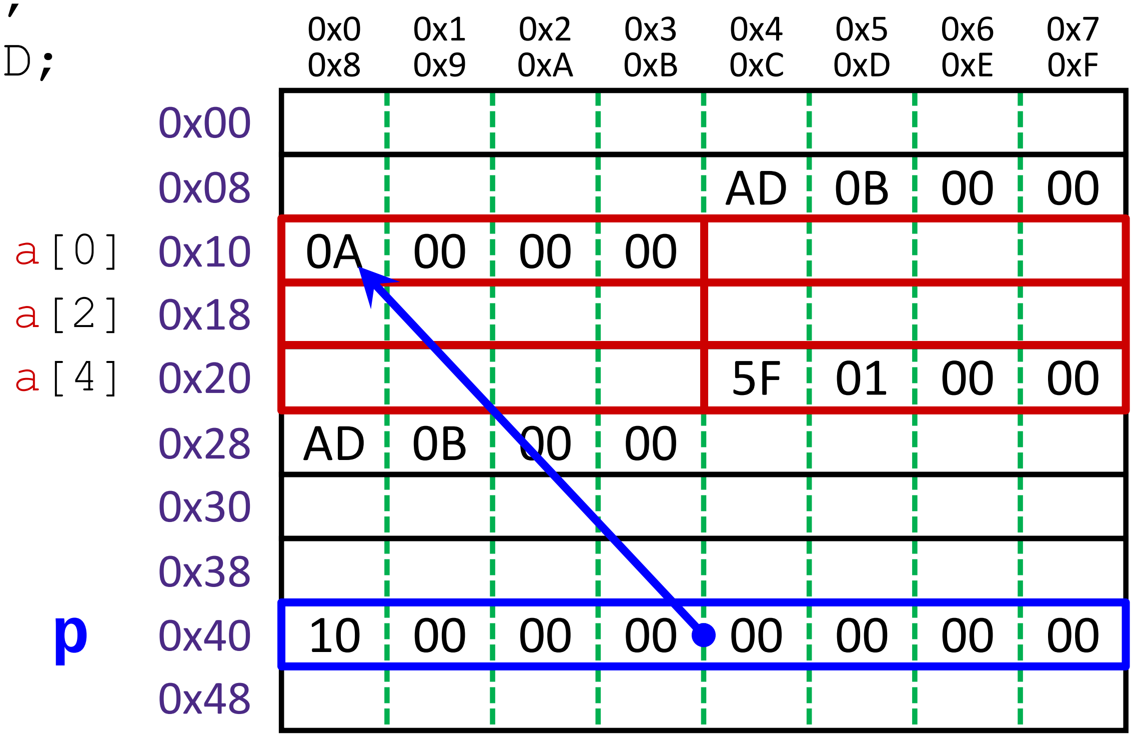
Indexing: `a[0] = 0x015f;`
`a[5] = a[0];`

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

Pointers: `int* p;`

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

`*p = 0xA;`



		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	0A	00	00	00				
a[2]	0x18								
a[4]	0x20					5F	01	00	00
	0x28	AD	0B	00	00				
	0x30								
	0x38								
p	0x40	10	00	00	00	00	00	00	00
	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;`

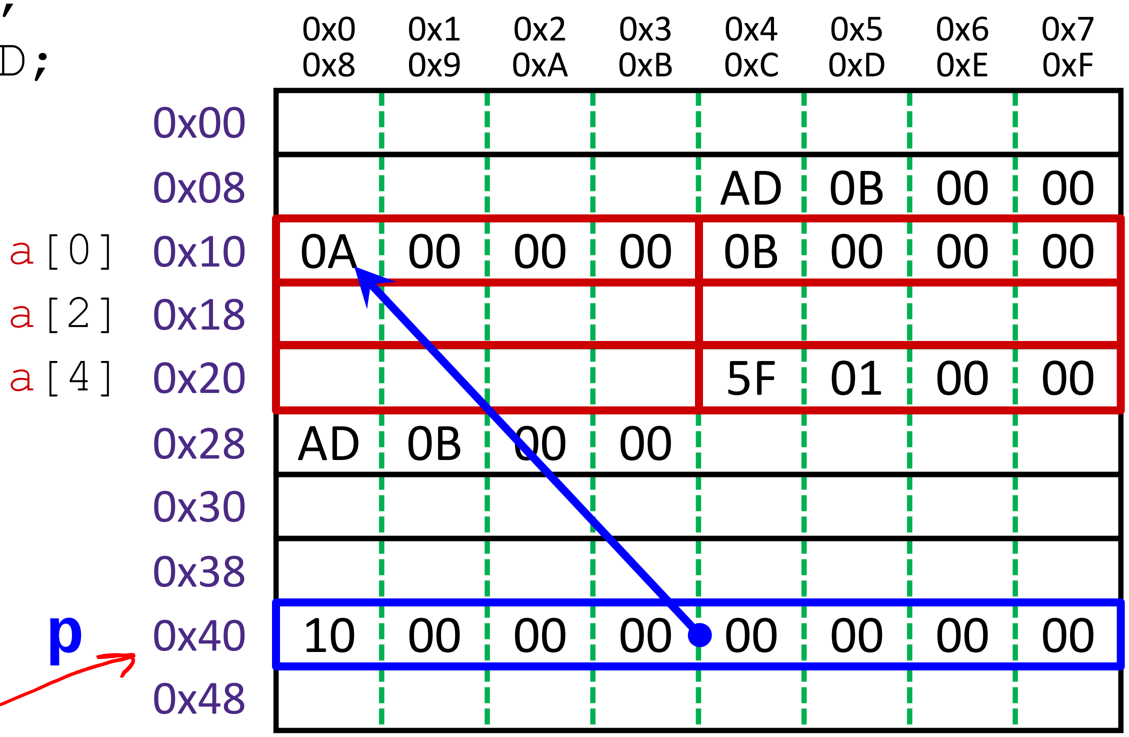
Pointers: `int* p;`

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

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

equivalent {
`p[1] = 0xB;`
`*(p+1) = 0xB;`
}
pointer arithmetic: 0x10 + 1 → 0x14
`p = p + 2;`
0x10+2 → $\boxed{0x18}$

p[i] ⟺ *(p + i)



	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	0A	00	00	00	0B	00	00	00
a[2] 0x18								
a[4] 0x20					5F	01	00	00
0x28	AD	0B	00	00				
0x30								
0x38								
p 0x40	10	00	00	00	00	00	00	00
0x48								

a + 2*sizeof(int) = 0x18

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)

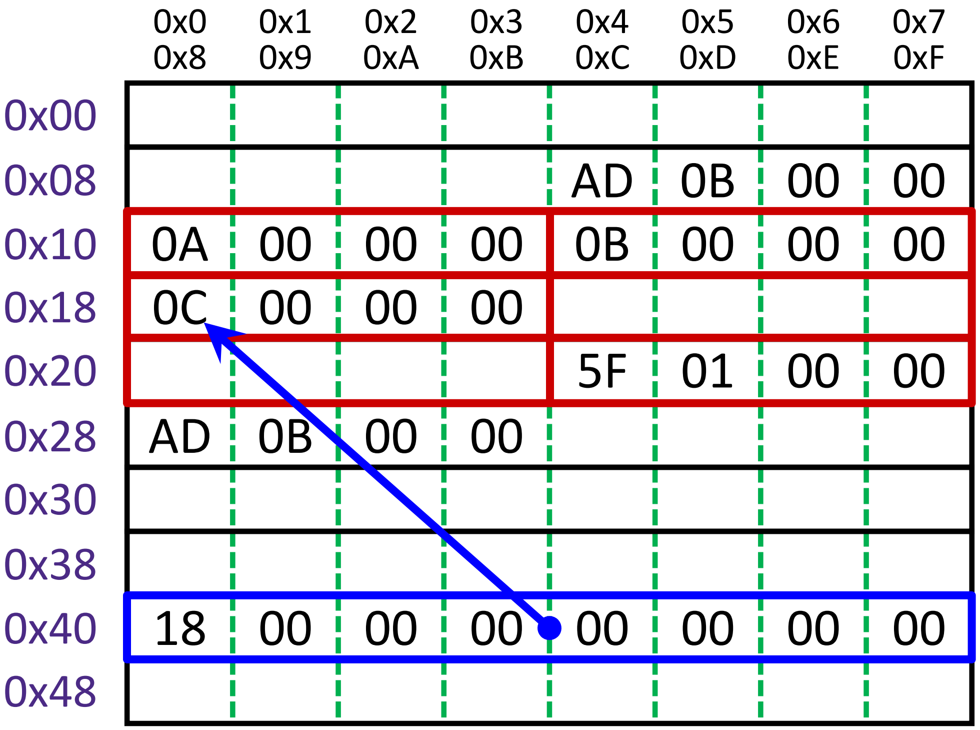
equivalent {
`p[1] = 0xB;`
`*(p+1) = 0xB;`
}

`p = p + 2;`

store at 0x18 →

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

0xB + 1 = 0xC



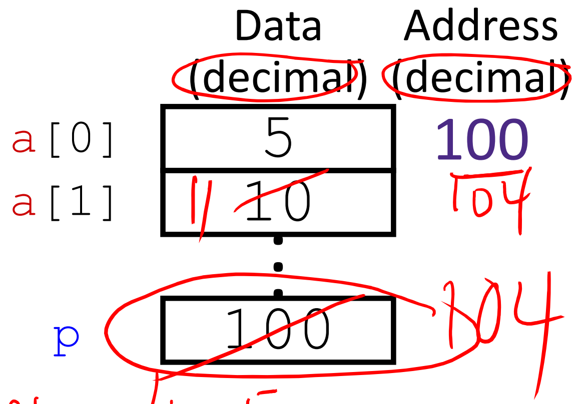
		0x0 0x8	0x1 0x9	0x2 0xA	0x3 0xB	0x4 0xC	0x5 0xD	0x6 0xE	0x7 0xF
	0x00								
	0x08					AD	0B	00	00
a[0]	0x10	0A	00	00	00	0B	00	00	00
a[2]	0x18	0C	00	00	00				
a[4]	0x20					5F	01	00	00
	0x28	AD	0B	00	00				
	0x30								
	0x38								
p	0x40	18	00	00	00	00	00	00	00
	0x48								

(no pointer arithmetic)

Question: The variable values after Line 3 executes are shown on the right. What are they after Line 4 & 5?

- Vote at http://pollev.com/rea

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



	after line 4					after line 5			
	p	***p**	**a[0]**	**a[1]**		**p**	***p**	**a[0]**	**a[1]**
(A)	101	10	5	10	then	101	11	5	11
→ **(B)**	104	10	5	10	then	104	11	5	11
(C)	100	6	6	10	then	101	6	6	10
(D)	100	6	6	10	then	104	6	6	10

Representing strings

- C-style string stored as an array of bytes (`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

6 chars 1 char 5 chars

- **Example:** "Donald Trump" stored as a 13-byte array

Decimal:	68	111	110	97	108	100	32	84	114	117	109	112	0
Hex:	0x44	0x6F	0x6E	0x61	0x6C	0x64	0x20	0x54	0x72	0x75	0x6D	0x70	0x00
Text:	D	o	n	a	l	d		T	r	u	m	p	\0

13 bytes total!

- Last character followed by a 0 byte (`'\0'`) (a.k.a. "null terminator")
  - Must take into account when allocating space in memory
  - Note that `'0'` ≠ `'\0'` (*i.e.* character 0 has non-zero value)

48 0

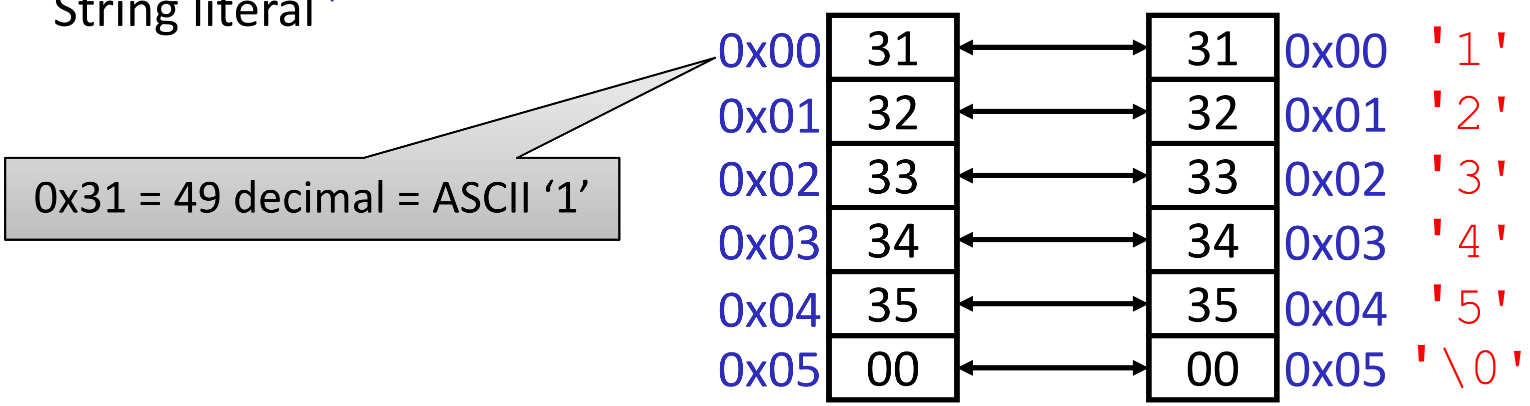
- How do we compute the length of a string?
  - Traverse array until null terminator encountered

Endianness and Strings

C (char = 1 byte)

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

String literal



0x31 = 49 decimal = ASCII ‘1’

	IA32, x86-64 (little-endian)	SPARC (big-endian)		
0x00	31	31	0x00	'1'
0x01	32	32	0x01	'2'
0x02	33	33	0x02	'3'
0x03	34	34	0x03	'4'
0x04	35	35	0x04	'5'
0x05	00	00	0x05	'\0'

- 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
  - 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");
}
```

format string

pointer arithmetic on char*

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

int*

4 bytes

"cast" (treat as)

`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 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)