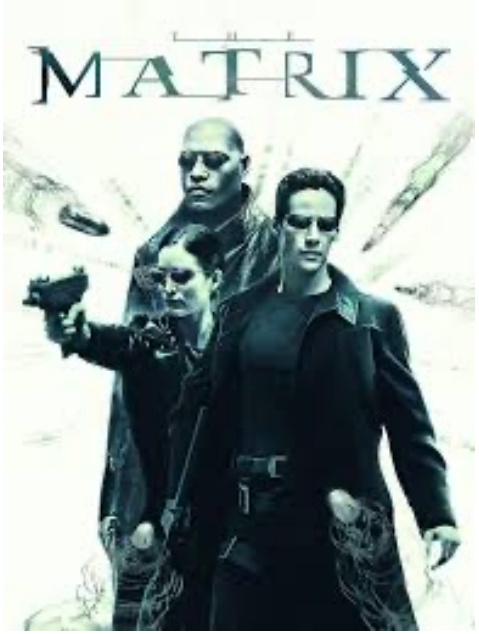
If CMPT 295 was a movie...



And I am... *Morpheus* — I'll hand you the red pill, the tools, and challenges, but you have to master

If it was a game

Arrvindh (TASC 9009)

Alaa (next door)

Why are you here ?

- “Because It’s Required”
- “It Might Be Easy, Right?”
- “I Just Want to Get It Over With”
- “Because Everyone Else Is Doing It”
- “It’s Not Like I Have Anything Better to Do”

How do you get to 300 million petaflops?

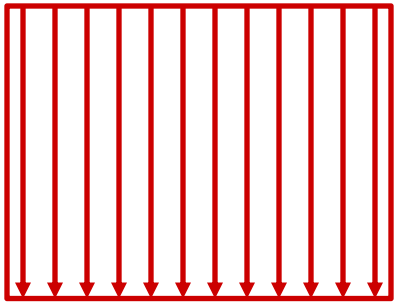
In CMPT 295 you will develop a machine that does 32 Gigaflops i.e,. 32 ops/step running at 1 Ghz frequency (1 billion steps/s)

Training GPT-3 reportedly took **several months**, leveraging a large-scale cluster of thousands of GPUs or TPUs to process its massive dataset and optimize its 175 billion parameters.

Message ChatGPT

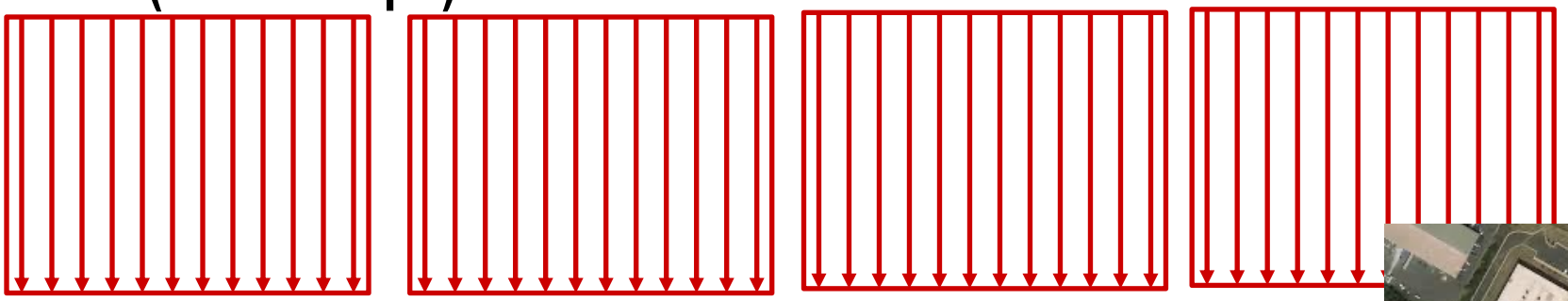
How do you get to 300 million petaflops?

1 chip = x1000 (32 teraflops)



How do you get to 300 million petaflops?

x1000 (32 teraflops)



x1000 machines (32 Petaflop/s) * 60s/m*60m/h* 24h

3 million Petaflops/day

300 million/3million = 100 days

Train: 300 million/3 = 100 days

Learning from first-principles

Logistics

- Assignment 1 will be released
  - Need to complete Lab 0 to be able to access
- Complete Lab 0 (ASAP!)
  - Tomorrow: TAs will begin Lab 1
  - Need it before we can grade anything
- SFU has informed that classes will not be recorded.
  - Class slides will be available as Class.pptx on syllabus page at the end of lecture day (i.e, if you are sick please follow BCCDC instructions)

Abstraction

The process of removing details or attributes in the study of systems to focus attention on details of greater importance.

Abstraction (Levels of Representation/Interpretation)

Python Program

```
def square(num):
    return num * num
```

C Program

```
int square(int num):
    return num * num
```

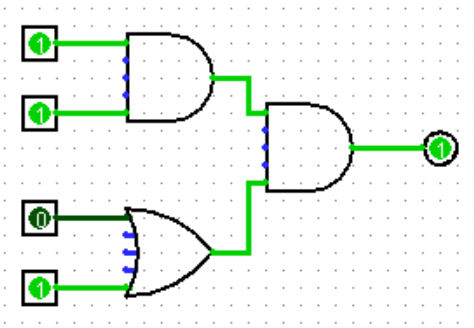
Binary

```
0x00000317
0x00830067
0xff010113
0x00112623
0x00812423
0x01010413
0xfea42a23
.....
```

Assembly

```
square:
addi   sp,sp,-16
sw     ra,12(sp)
sw     s0,8(sp)
addi   s0,sp, 16
sw     a0,-12(s0)
lw     a0,-12(s0)
addi   a1,a0,2
mul    a0,a1,a0
lw     ra,12(sp)
lw     s0,8(sp)
addi   sp,sp,16
ret
```

Logic



Software (Typed) vs Hardware (No Types)

- **Software Variables**
  carry meaning: int, char, pointer.
  Compilers enforce rules (sizes, alignment, valid ops).
  Guides debuggers, checks & optimizations.

Hardware world (untyped)
- Only **voltages/bits** moving through wires: `0/1`.
- Ops or **opcodes have types**, not data
- The **same 32 bits** can be treated as many things.

```
char c, d; int a; int b;
 c = a + b ; // Compiler error/warn
 c = c + d //  Unsupported op
```

Bits: 0100 0001 (0x41)
- → **as int: 65**
- → **as char: 'A'**
- → **as float: nonsense**

Performance Cost of Abstraction

- c = (a + b) OR c = (a * b) + d
- c = a + b OR c = a[i] + b[i]
- c = a[i] + b[i] OR c = a[c[i]] + a[c[i]]

Base Comparison

- Why does all of this matter?
  - *Humans* think about numbers in **base 10**, but *computers* "think" about numbers in **base 2**
  - Binary encoding is what allows computers to do all of the amazing things that they do!

- You should have this table memorized by the end of the 1st class
  - Might as well start now!

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Why different number systems?

- Binary System: foundation of computers (0s and 1s) because it aligns with a transistor's switch state (off/on)
  - All data in computers—text, images, sound—can ultimately be represented as binary sequences.

- Hexadecimal System: compact representation; balance between human-readable and easy conversion to binary
  - `0xAB = 0b10101011 = 0d171.`
- - Memory: hexadecimal matches neatly with word/block sizes in computer architecture (e.g., 8-bit, 16-bit systems).
- Realistic use: Identifying pointer properties.

Numerical Encoding

- **AMAZING FACT: You can represent *anything* countable using numbers!**
  - Need to agree on an encoding
  - Kind of like learning a new language

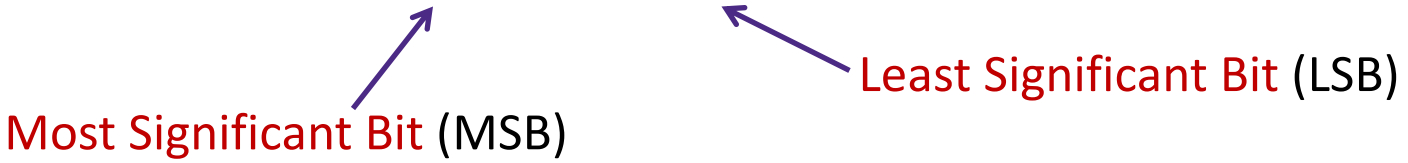
- Examples:
  - Decimal Integers: 0→0b0, 1→0b1, 2→0b10, etc.
  - English Letters: CSE→0x435345, yay→0x796179
  - Emoticons: 😀 0x0, 😞 0x1, 😎 0x2, 😇 0x3, 😈 0x4, 🤷 0x5

Memory, Data, & Addressing I

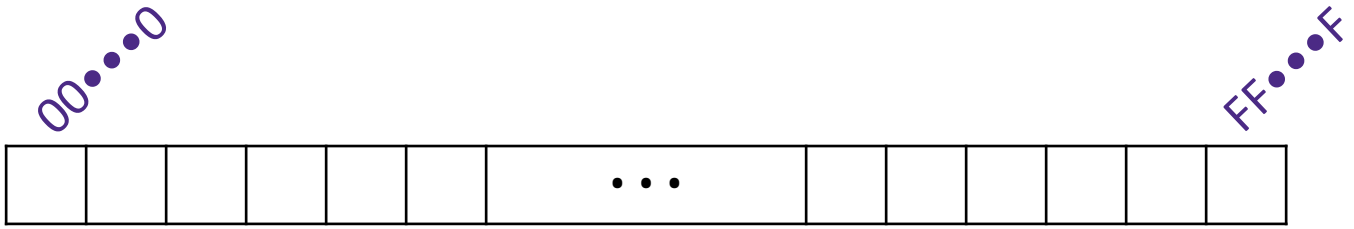
Binary Encoding Additional Details

- Because storage is finite in reality, everything is stored as "fixed" length
  - Data is moved and manipulated in fixed-length chunks
  - Multiple fixed lengths (*e.g.* 1 byte, 4 bytes, 8 bytes)
  - Leading zeros now *must* be included up to "fill out" the fixed length

- Example: the "eight-bit" representation of the number 4 is 0b00000100



Byte-Oriented Memory Organization



- Conceptually, memory is a single, large array of bytes, each with a unique *address* (index)
  - **Each address is just a number represented in *fixed-length* binary**
- Programs refer to bytes in memory by their *addresses*
  - Domain of possible addresses = *address space*
  - **Pointer: We can store addresses as data to "remember" where other data is in memory**

- But not all values fit in a single byte… (*e.g.* 295)
  - Many operations actually use multi-byte values

Peer Instruction Question

- If we choose to use 4-bit addresses, how big is our address space?
  - *i.e.* How much space can we "refer to" using our addresses?

A. 16 bits
B. 16 bytes
C. 4 bits
D. 4 bytes
E. We're lost…

Why is 1KB = 1024 bytes (not 1000 bytes)

- Computers typically organize memory in powers of 2, since addressing grows as a power of 2.
- Consider,
- 1 bit address has two values 0,1 and hence can address a memory with 2 bytes.
- 2 bits address can have values 0,1,2,3 and hence can address memory with 4 bytes.

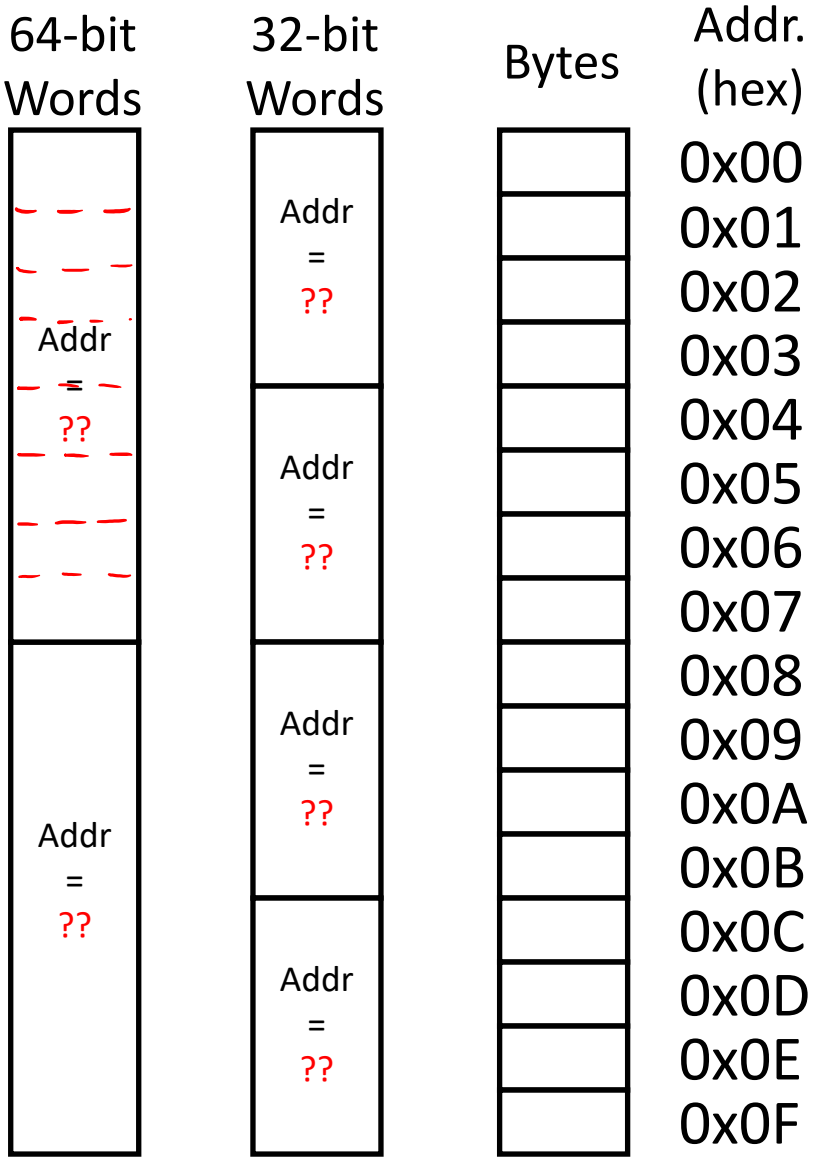
- 10 bits address can have values 0 – 1023 and hence can address memory with 1024 bytes (1KB).

Machine "Words"

- We have *chosen* to tie word size to address size/width
  - word size = address size = register size
  - word size = w bits $\rightarrow 2^w$ addresses
- Current x86 systems use **64-bit (8-byte) words**
  - Potential address space: $\mathbf{2^{64}}$ addresses
    2^{64} bytes $\approx$ **1.8 x 10^{19} bytes**
    = 18 billion billion bytes = 18 EB (exabytes)
  - Actual physical address space: **48 bits**

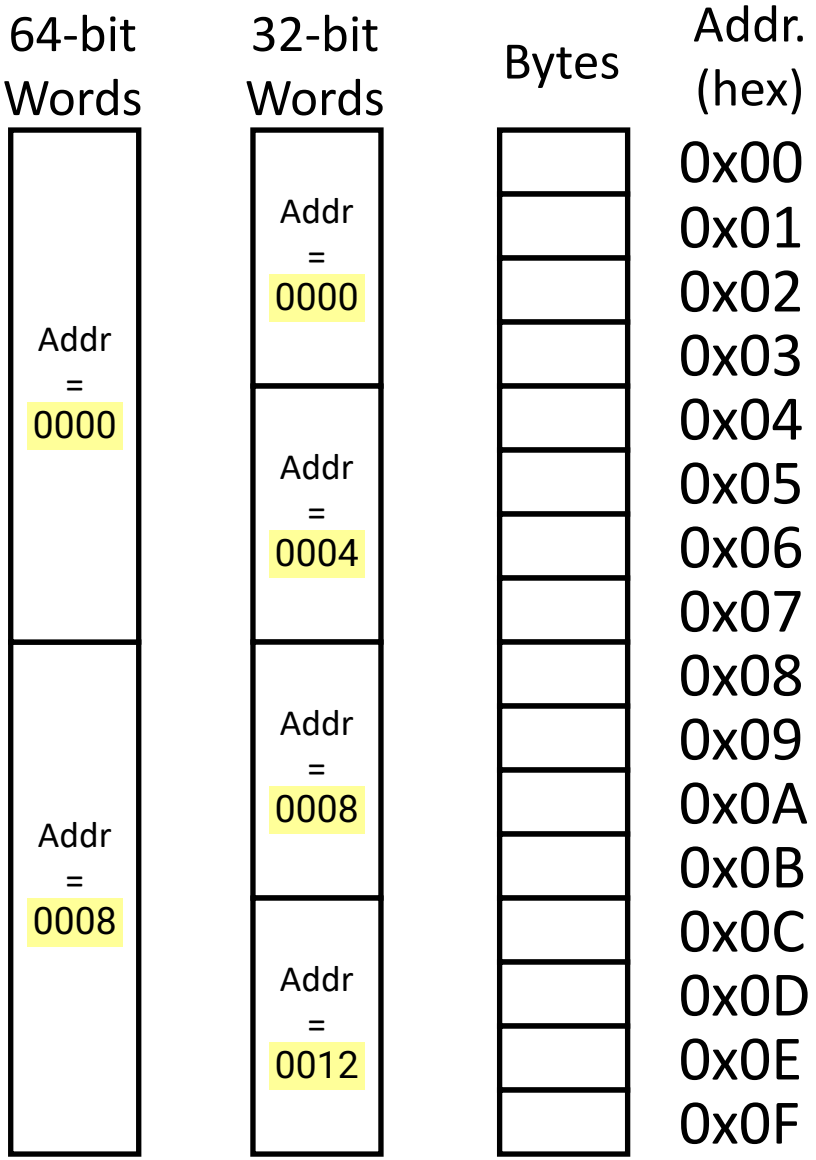
Word-Oriented Memory Organization

- Addresses still specify locations of *bytes* in memory
  - Addresses of successive words differ by word size (in bytes): *e.g.* 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, … 10?



Word-Oriented Memory Organization

- Addresses still specify locations of *bytes* in memory
  - Addresses of successive words differ by word size (in bytes): *e.g.* 4 (32-bit) or 8 (64-bit)
  - Address of word 0, 1, … 10?
- Address of word
  = address of *first* byte in word
  - The address of *any* chunk of memory is given by the address of the first byte
  - ***Alignment***



Data Representations

- Sizes of data types (in bytes)

Java Data Type	C Data Type	32-bit	x86-64
`boolean`	`bool`	1	1
`byte`	`char`	1	1
`char`		2	2
`short`	`short int`	2	2
`int`	`int`	4	4
`float`	`float`	4	4
	`long int`	4	8
`double`	`double`	8	8
`long`	`long`	8	8
	`long double`	8	16
`(reference)`	**`pointer *`**	**4**	**8**

address size = word size

To use "`bool`" in C, you must `#include <stdbool.h>`

Memory Alignment

- **Aligned:** Primitive object of K bytes must have an address that is a multiple of K
  - More about alignment later in the course

K	Type
1	`char`
2	`short`
4	`int, float`
8	`long, double,` pointers

- For good memory system performance, data has to be aligned.

Alignment REPL

https://replit.com/@ashriram/Alignment#main.cpp

```
int i_int;
char ch = 'a';
long long int i_long_long;
short s_num = 0xBEEF;
uintptr_t int_ptr = (uintptr_t)&i_int;
printf("Sizeof data : %ld, Address, of
data %p, Addr. Binary)
```

Alignment REPL

```
Sizeof data : 4, Addr: 0x7ffd38f84cac, Addr Binary
00000000000000001111111111110100111000111110000100110010101100
 Sizeof data : 8, Address: 0x7ffd38f84c98, Addr in Binary
00000000000000001111111111110100111000111110000100110010011000
 Sizeof data : 1, Addr: 0x7ffd38f84c97, Addr in Binary
00000000000000001111111111110100111000111110000100110010010111
 Sizeof data : 2, Addr: 0x7ffd38f84c94, Addr in Binary
00000000000000001111111111110100111000111110000100110010010100
```

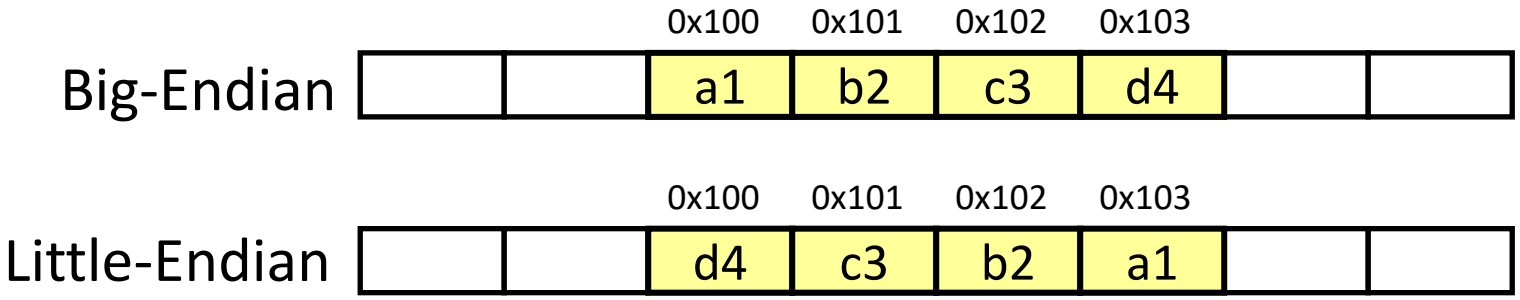
Byte Ordering

- How should bytes within a word be ordered *in memory?*
  - **Example:** store the 4-byte (32-bit) `int`:
    `0x a1 b2 c3 d4`

- By convention, ordering of bytes called *endianness*
  - The two options are big-endian and little-endian
    - In which address does the least significant *byte* go?
    - Based on *Gulliver's Travels*: tribes cut eggs on different sides (big, little)

Byte Ordering

- Big-endian (SPARC, z/Architecture)
  - Least significant byte has highest address
- Little-endian (x86, x86-64, **RISC-V**)
  - Least significant byte has lowest address
- Bi-endian (ARM, PowerPC)
  - Endianness can be specified as big or little
- **Example:** 4-byte data 0xa1b2c3d4 at address 0x100

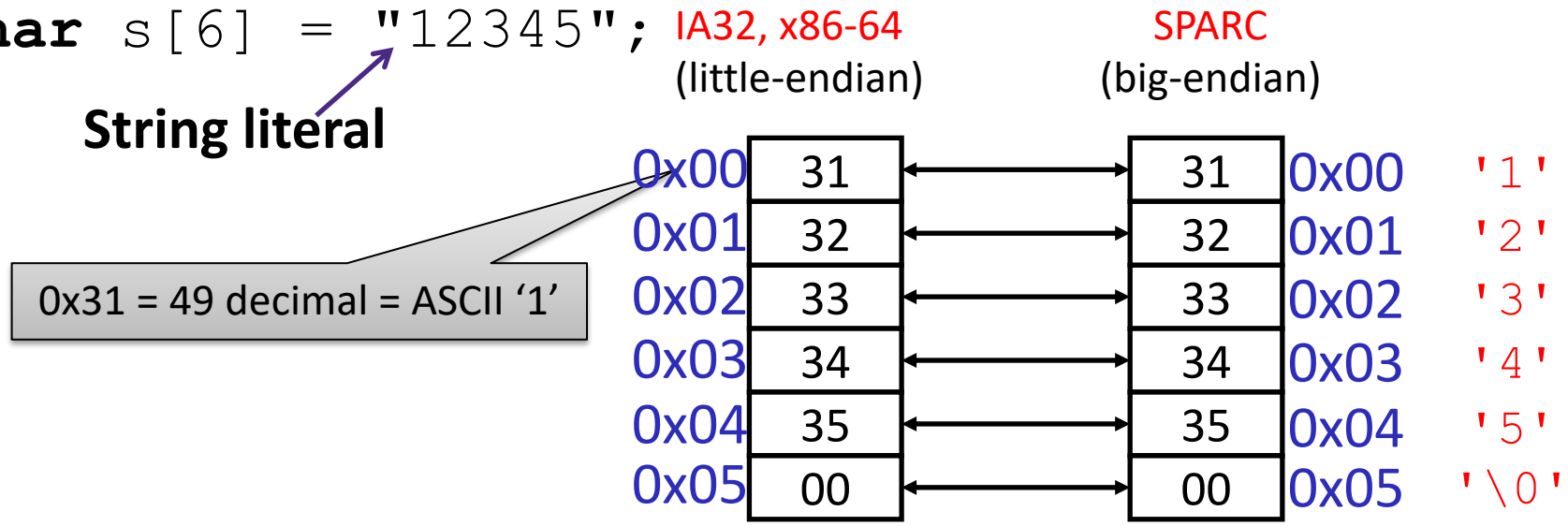


Endianness and Strings

C (char = 1 byte)

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

String literal



- 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

Show bytes demo

https://replit.com/@ashriram/ShowBytes#main.cpp

show_bytes Execution Example

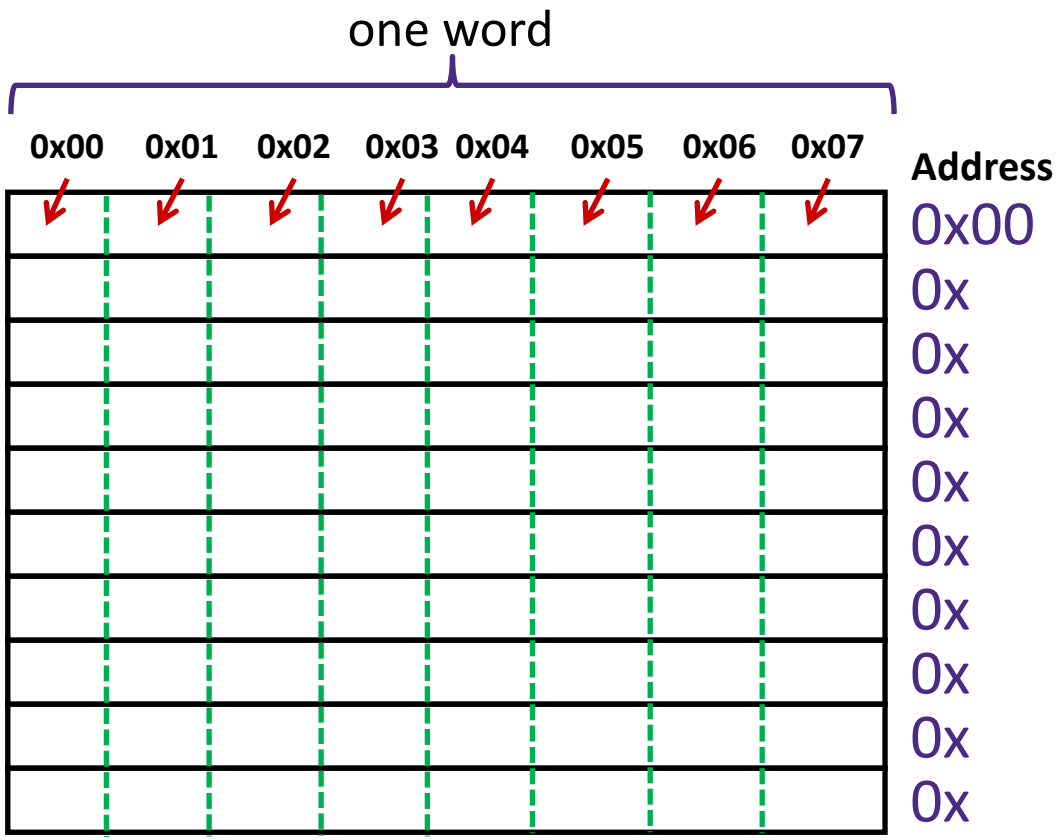
```
int x = 12345; // 0x00003039
printf("int x = %d;\n",x);
show_int(x);  // show_bytes((char *) &x,
sizeof(int));
```

- 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
```

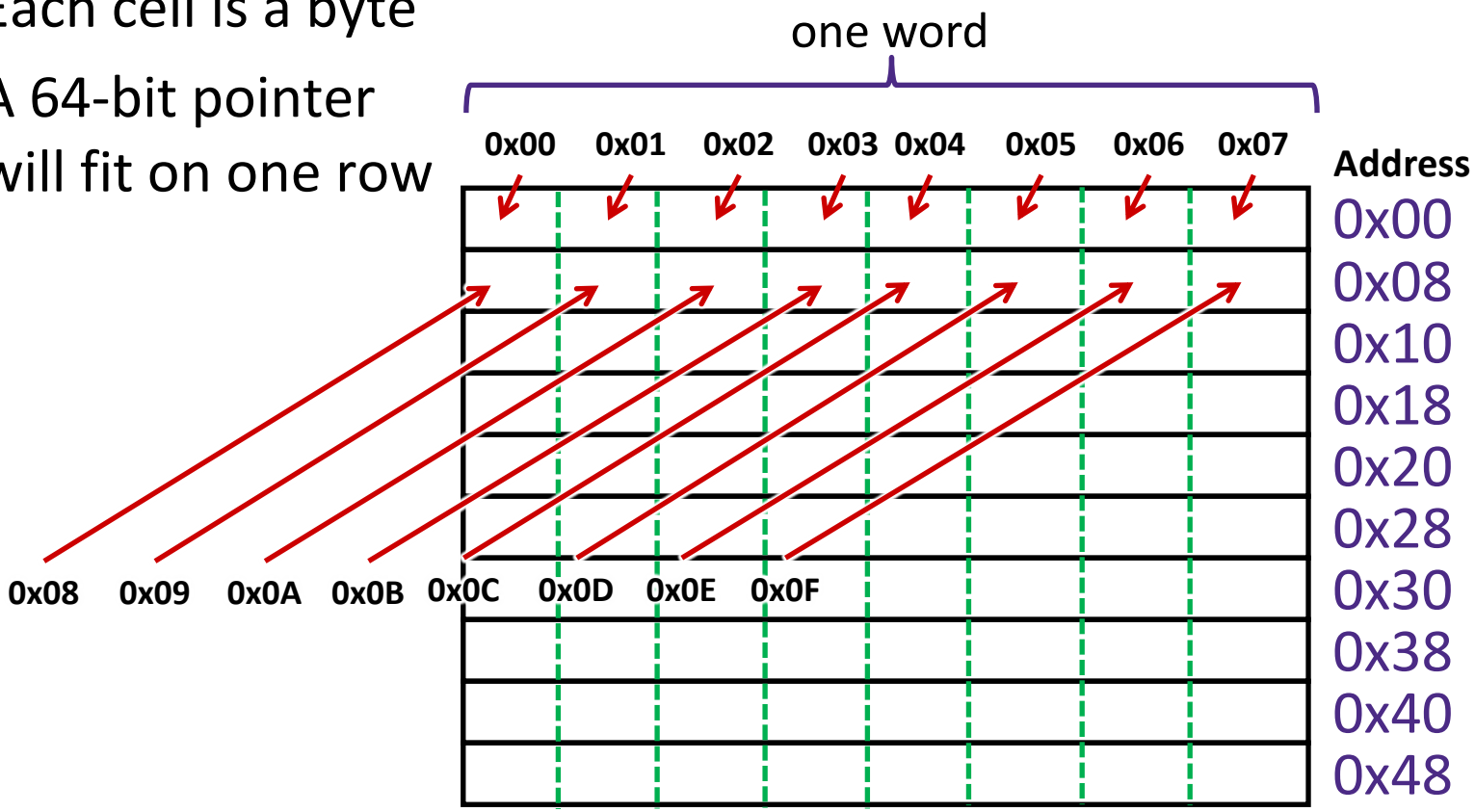
A Picture of Memory (64-bit view)

- A "64-bit (8-byte) word-aligned" view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - A 64-bit pointer will fit on one row



A Picture of Memory (64-bit view)

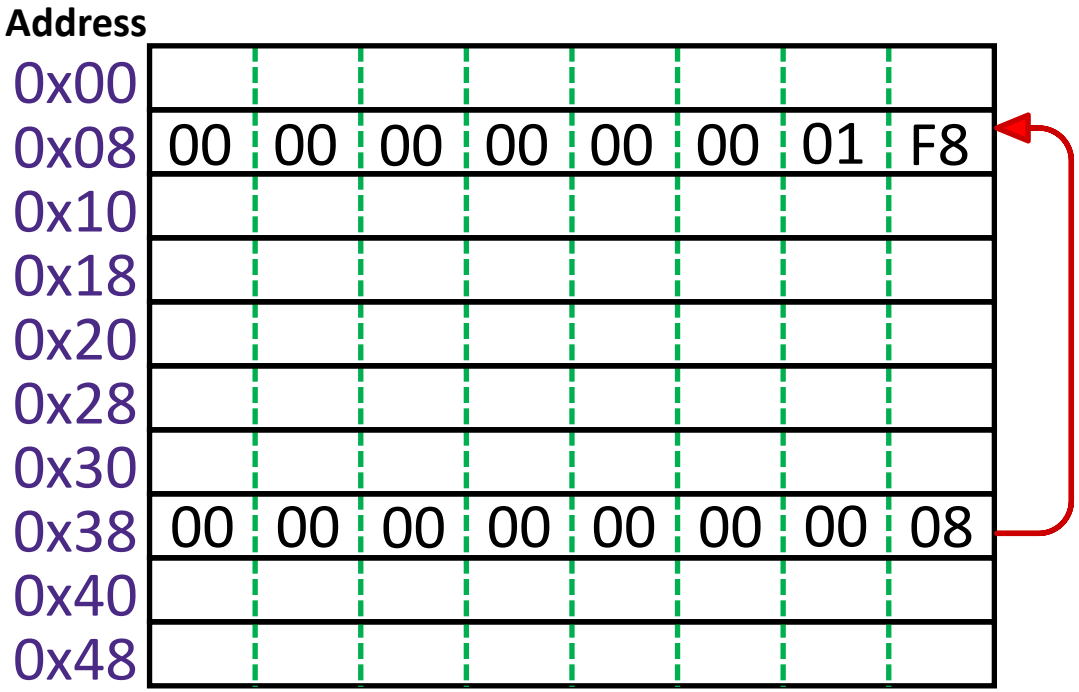
- A "64-bit (8-byte) word-aligned" view of memory:
  - In this type of picture, each row is composed of 8 bytes
  - Each cell is a byte
  - A 64-bit pointer will fit on one row



Addresses and Pointers

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

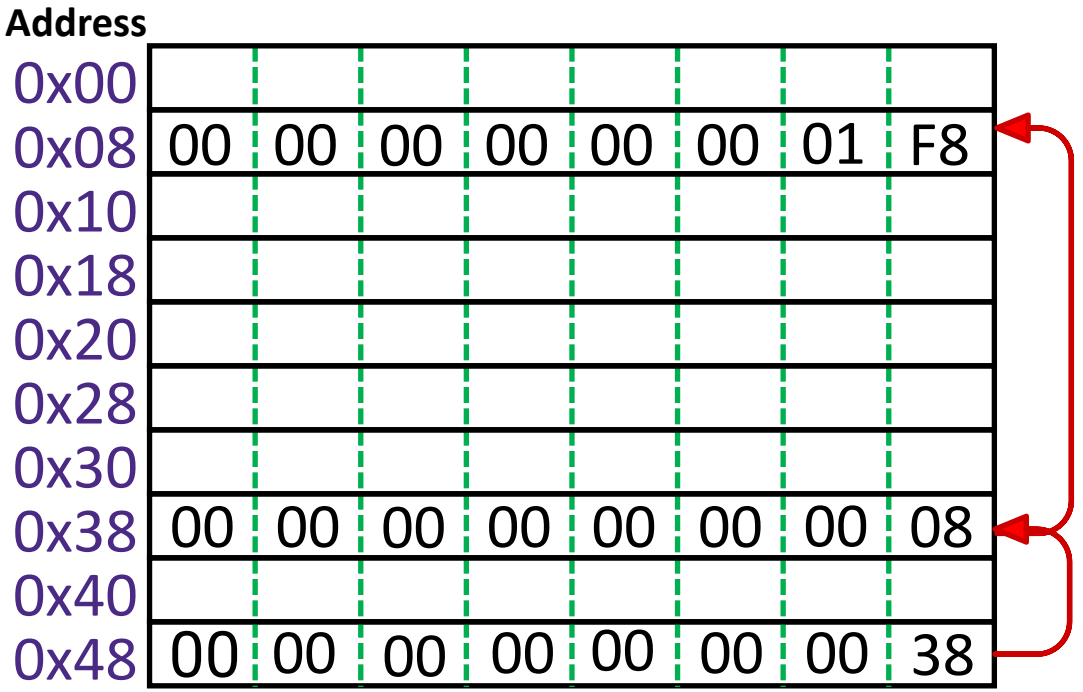
- An *address* is a location in memory
- A *pointer* is a data object that holds an address
  - Address can point to *any* data
- Value 504 stored at address 0x08
  - $504_{10} = 1F8_{16}$
    = 0x 00 … 00 01 F8
- Pointer stored at 0x38 points to address 0x08



Addresses and Pointers

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

- An *address* is a location in memory
- A *pointer* is a data object that holds an address
  - Address can point to *any* data
- Pointer stored at 0x48 points to address 0x38
  - Pointer to a pointer!
- Is the data stored at 0x08 a pointer?
  - Could be, depending on how you use it



Memory, Data, & Addressing II

Arrays in C

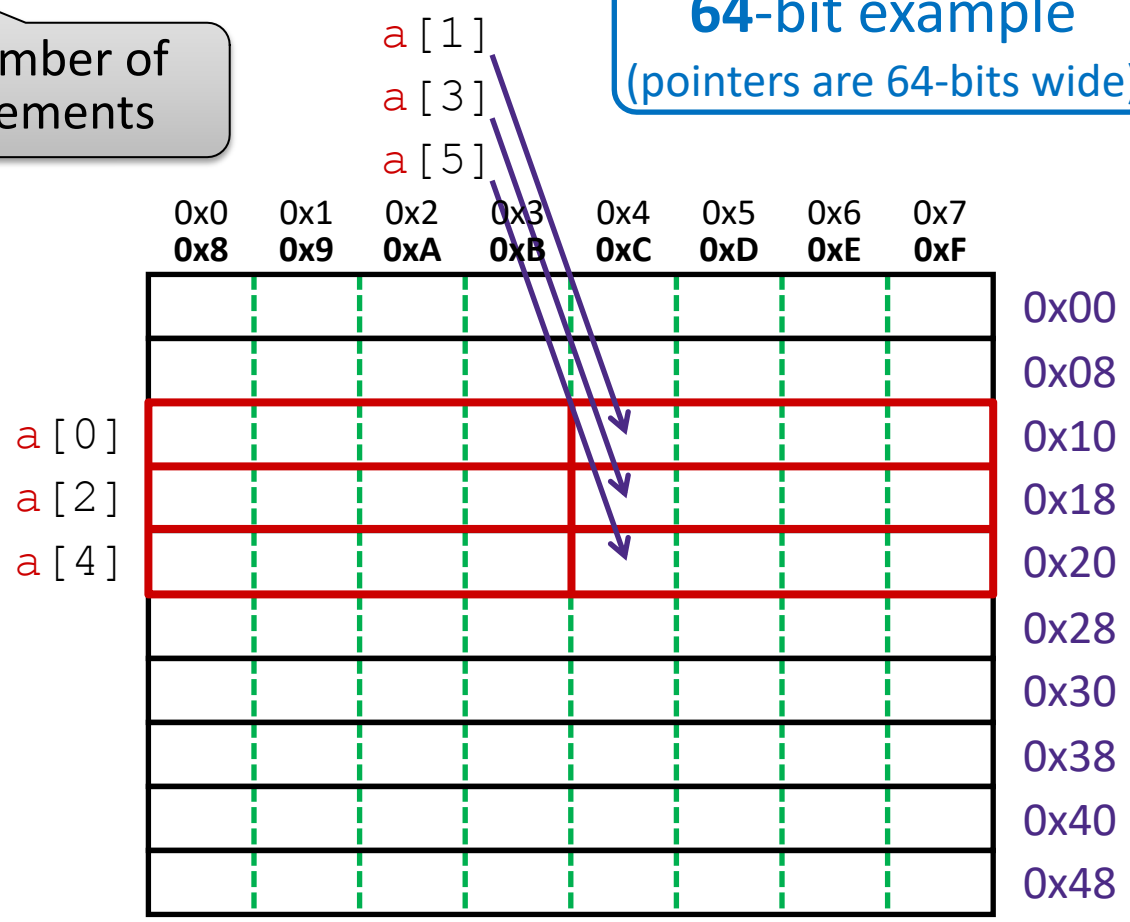
Arrays are adjacent locations in memory storing the same type of data object

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

Declaration: `int a[6];`

- element type
- name
- number of elements

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



Arrays Basics

- **Pitfall:** An array in C does not know its own length, and its bounds are not checked!
  - We can accidentally access off the end of an array
  - We must pass the array **and its size** to any procedure that is going to manipulate it
- Mistakes with array bounds cause *segmentation faults* and *bus errors*
  - Be careful! These are VERY difficult to find (You'll learn how to debug these in lab)

Arrays in C

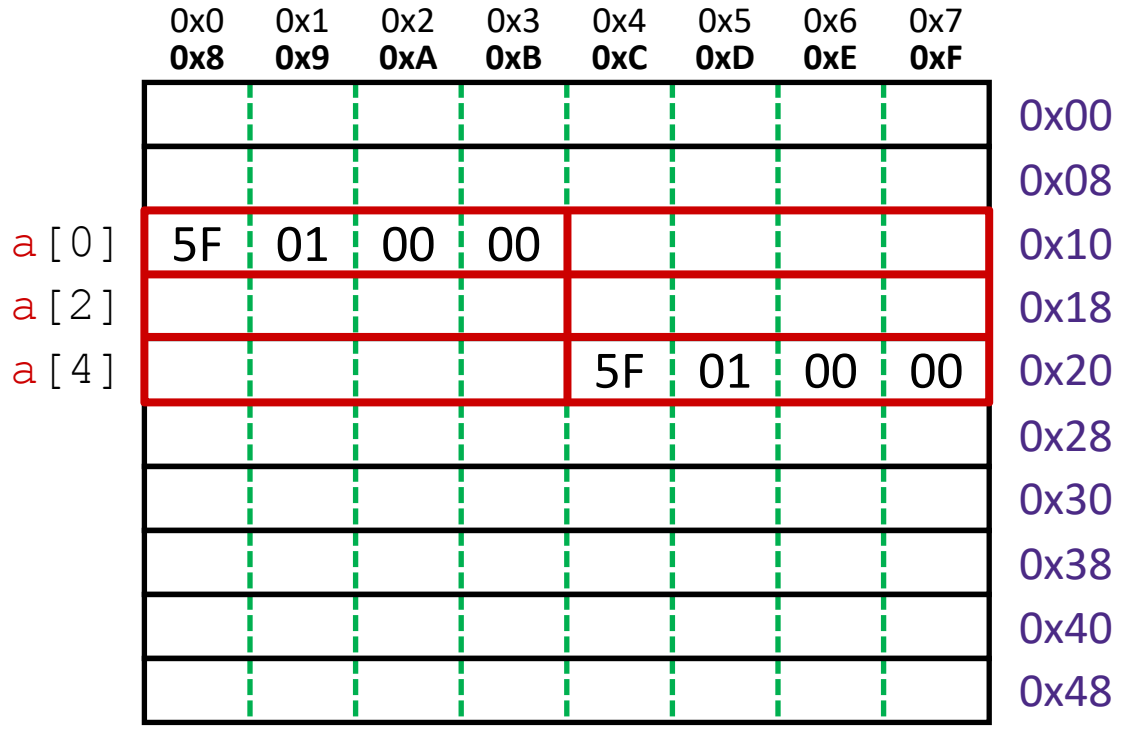
Declaration: `int a[6];`

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

Declaration: `int a[6];`

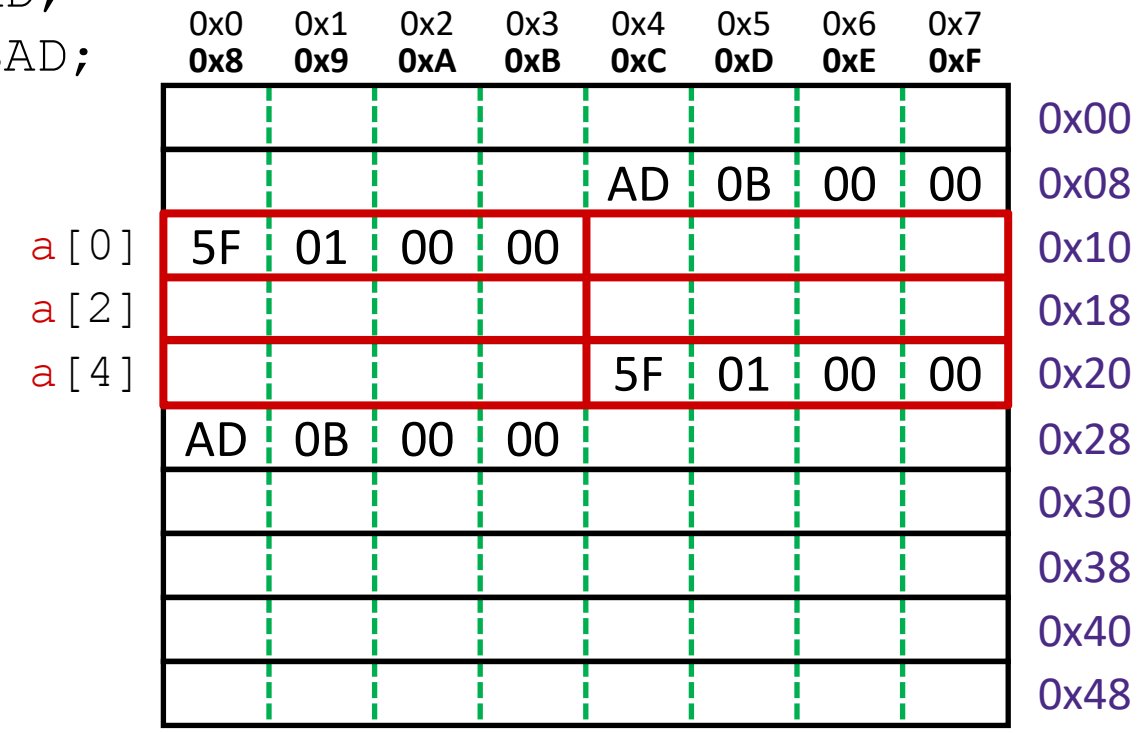
Indexing:
```
a[0] = 0x015f;
a[5] = a[0];
```

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

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

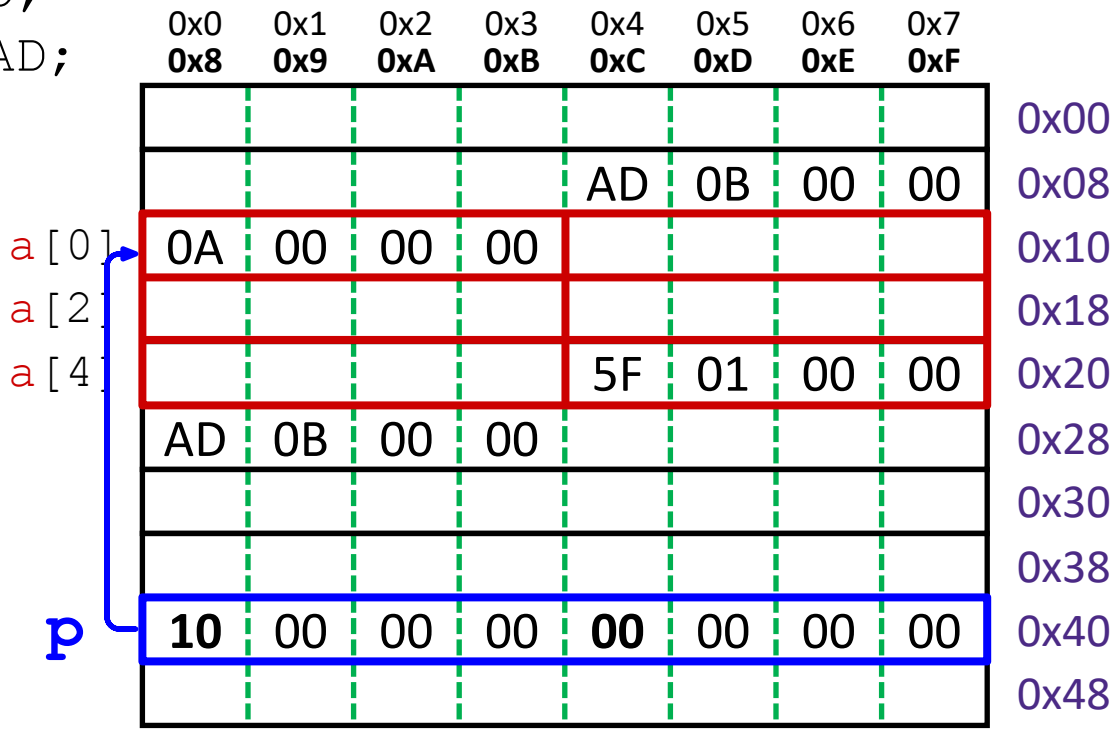
Declaration: `int a[6];`

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

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

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];
```

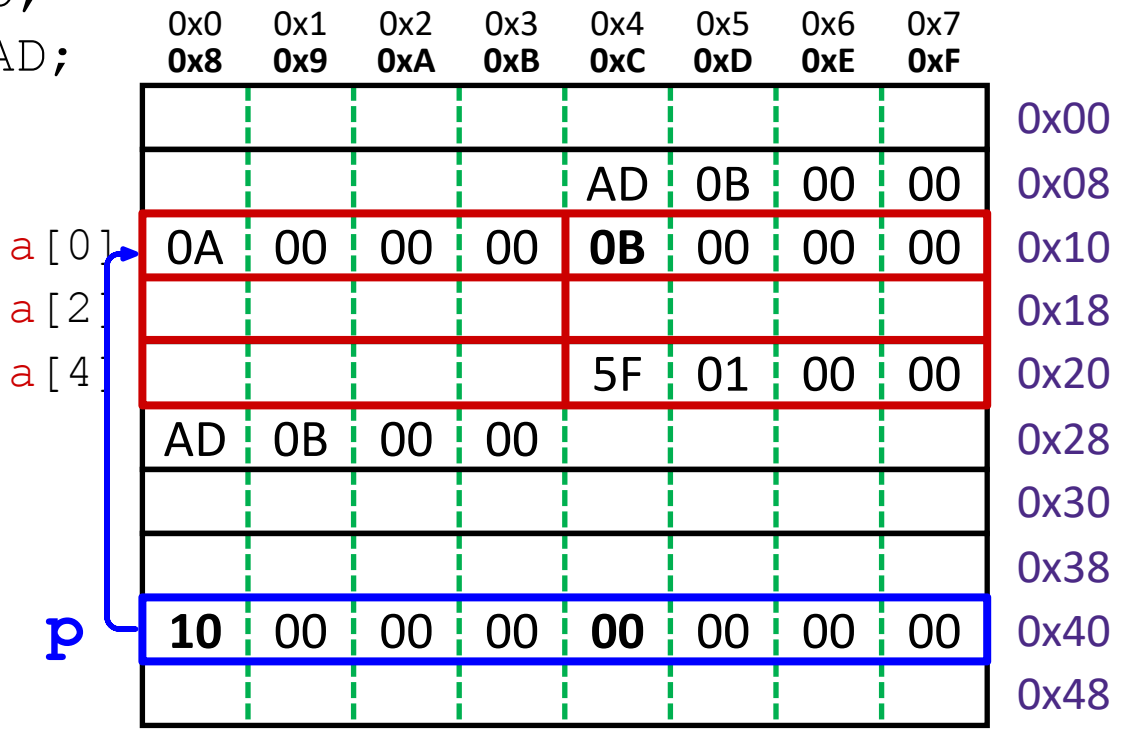
No bounds checking:
```
a[6] = 0xBAD;
a[-1] = 0xBAD;
```

Pointers:
```
int* p;
```

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

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

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

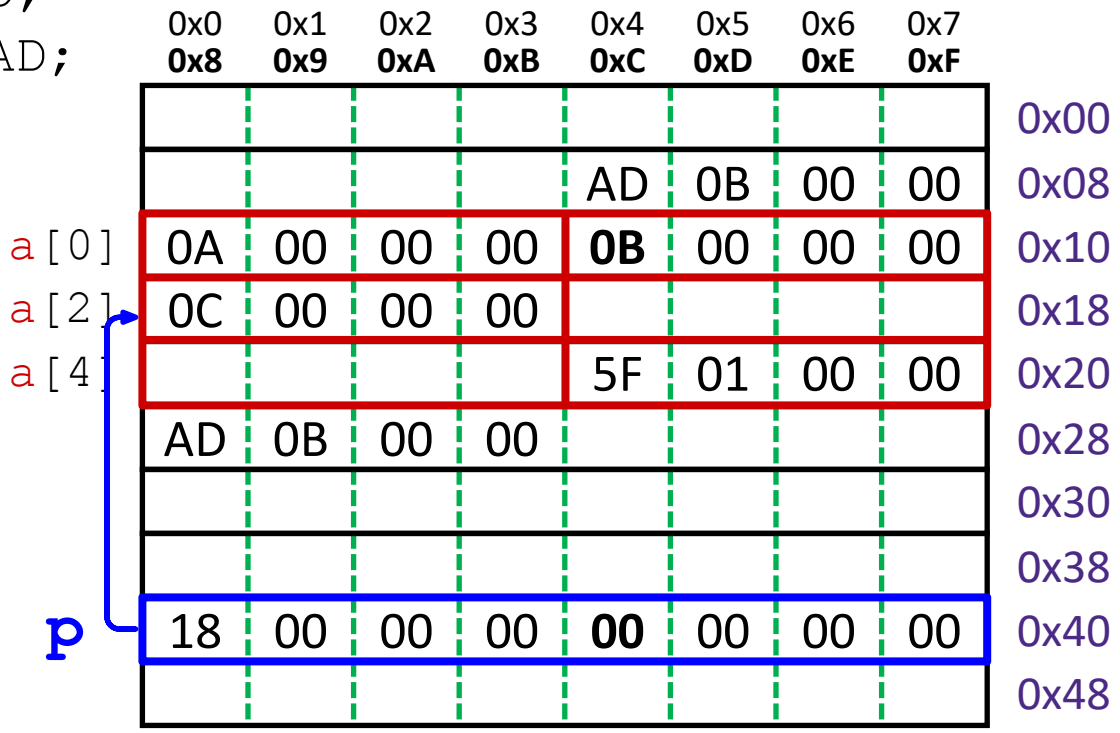
Pointers: `int* p;`

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

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

equivalent:
```
p[1] = 0xB;
*(p+1) = 0xB;
```
```
p = p + 2;
```

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



Pointer Arithmetic

- Pointer arithmetic 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
  - Be careful with data types and *casting*

Assignment in C

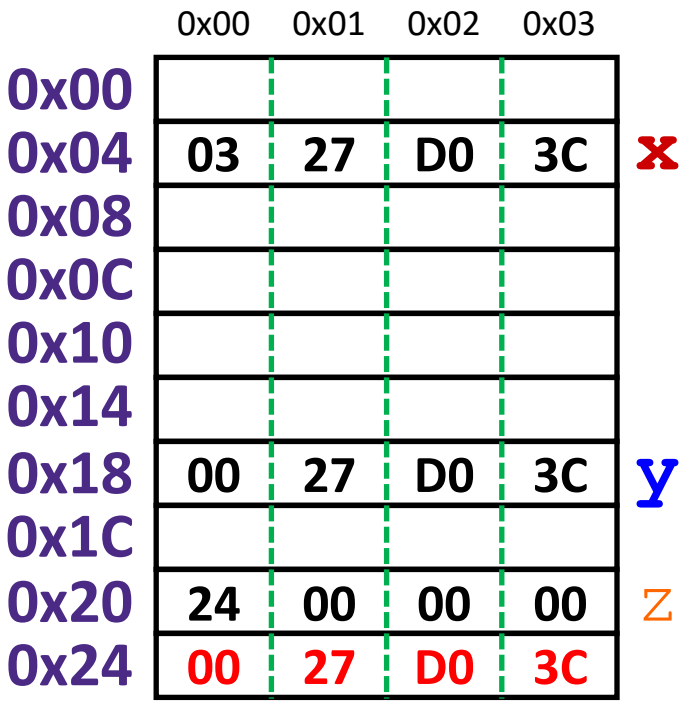
32-bit example
(pointers are **32**-bits wide)

`&` = "**a**ddress 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`

The target of a pointer is also a location

- `*z = y;`
  - Get value of `y`, put in address stored in `z`

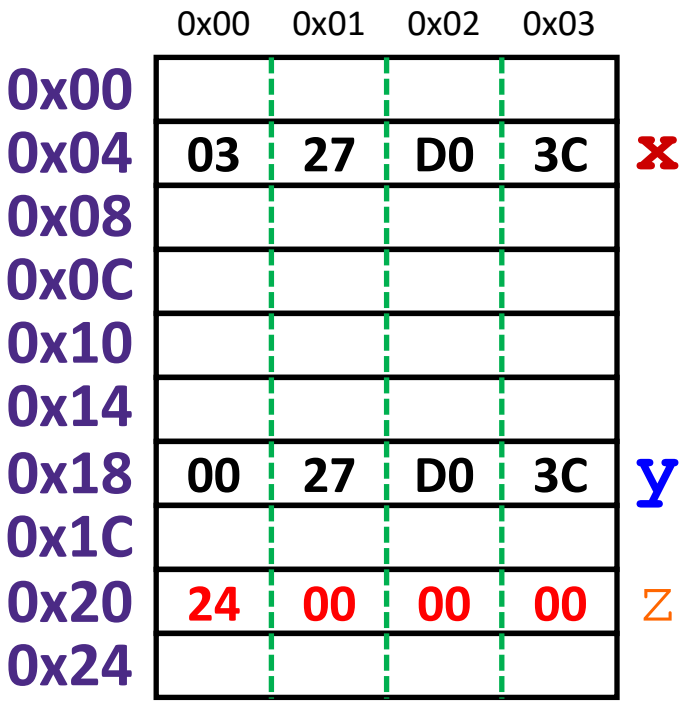


Assignment in C

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

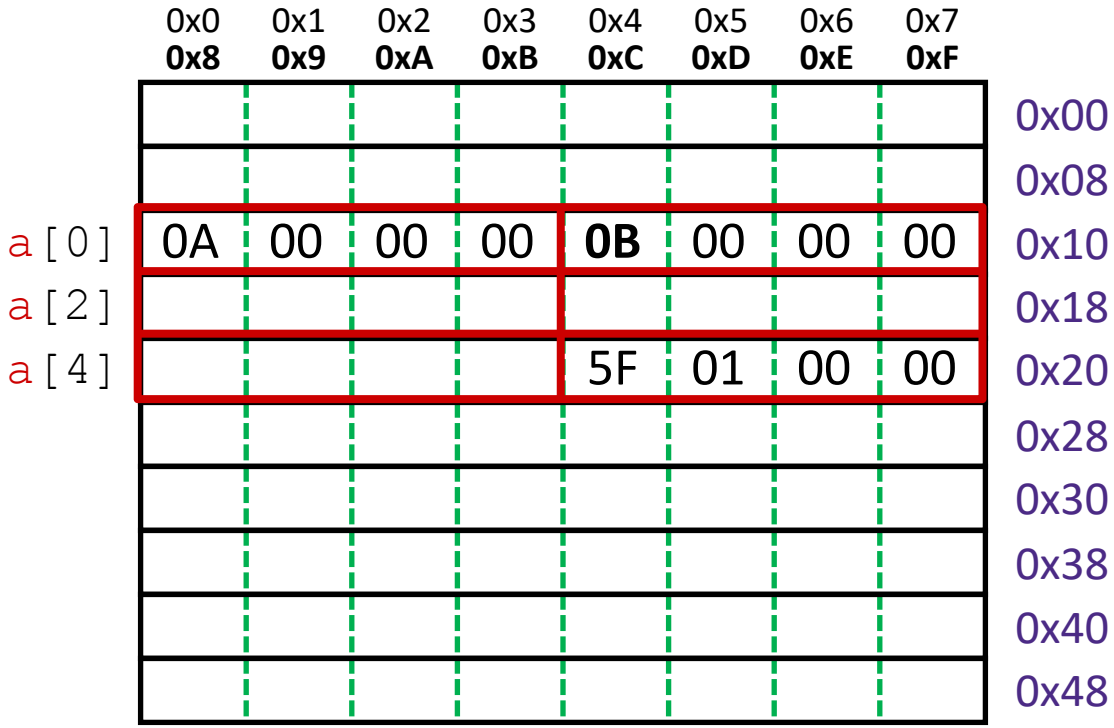
`&` = "**a**ddress of"
`*` = "dereference"

- 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

int a[6];



a = &a[0]

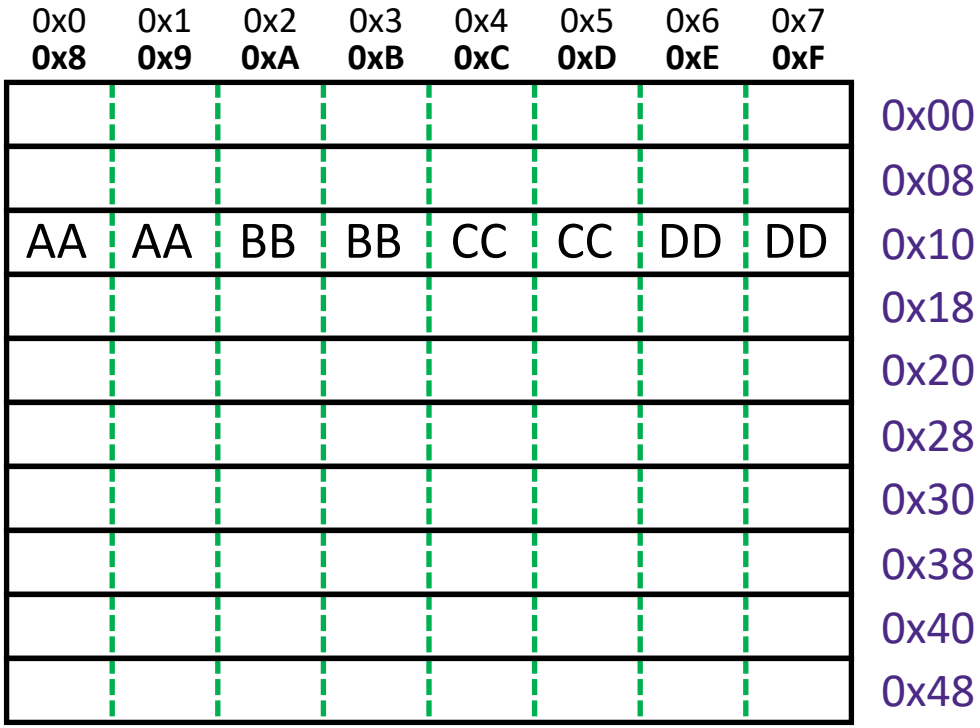
a+i = a[i] e.g., *(a+i) = 5 is same as a[i] = 5

Pointer Cast Example

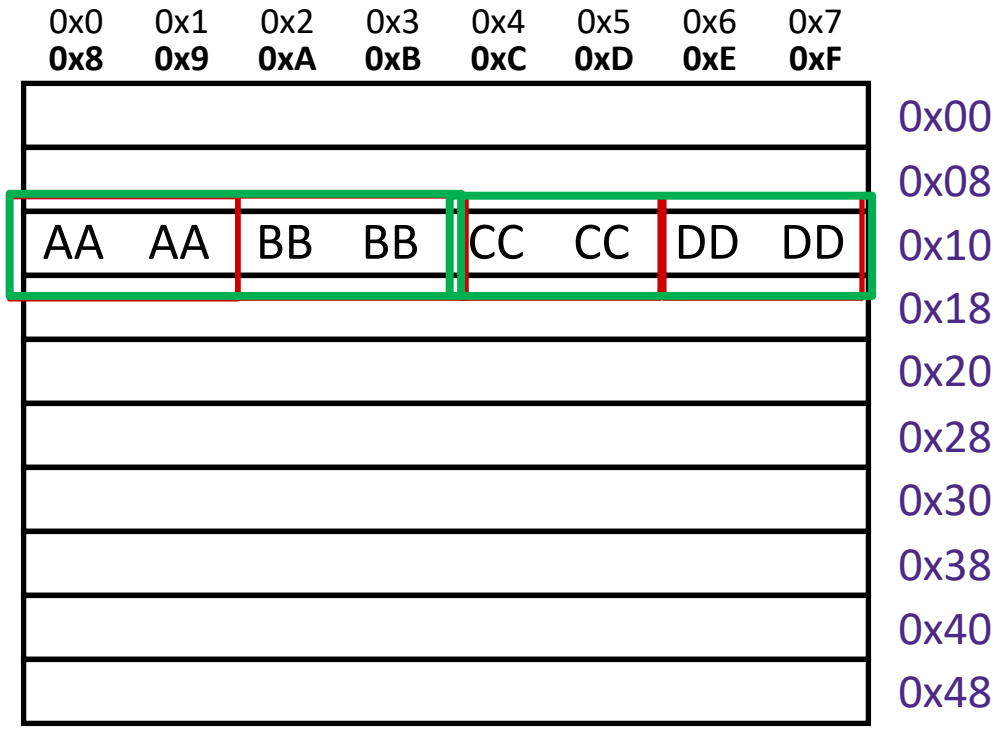
- [PTYPE]* ptr = variable+ offset
  - PTYPE controls how many bytes are read by *ptr
  - Variable type controls scaling factor k for offset
  - Ptr is the integer value variable + offset*sizeof(type)

```
short array[4] = {0xAAAA,0xBBBB,0xCCCC,0xDDDD};
short *s_ptr = (short*)&array;
int   *i_ptr =   (int*)&array;

for(int i = 0; i < 4; i++)
printf("\n [%p], 0x%hx ", s_ptr+i, s_ptr[i]);

for (int i = 0; i < 2; i++)
printf("\n [%p], 0x%lx ", i_ptr+i, i_ptr[i]);
```

short array[4] = {0xAAAA,0xBBBB,0xCCCC,0xDDDD}



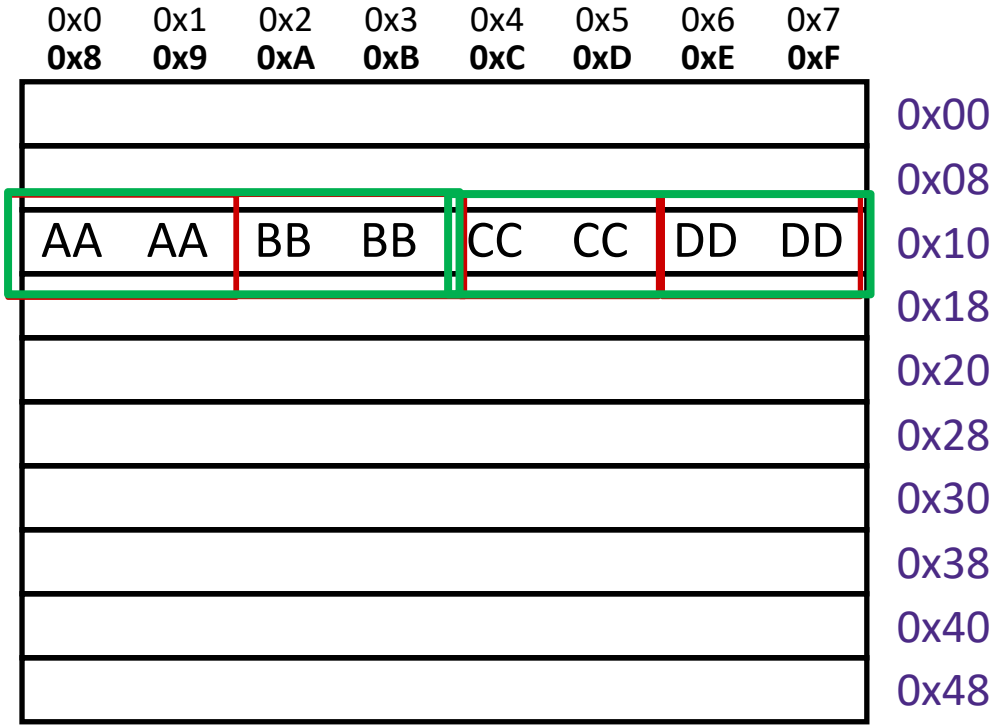
short array[4] = {0xAAAA,0xBBBB,0xCCCC,0xDDDD}



short *s_ptr = &array

int *i_ptr = &array

Pointer Cast Example



// s_ptr: as a short
[0x16f716f00], 0xaaaa
[0x16f716f02], 0xbbbb
[0x16f716f04], 0xcccc
[0x16f716f06], 0xdddd

// i_ptr: as a int
[0x16f716f00], 0xbbbbaaaa
[0x16f716f04], 0xddddcccc

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

Null-Terminated Strings

- **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

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

61996