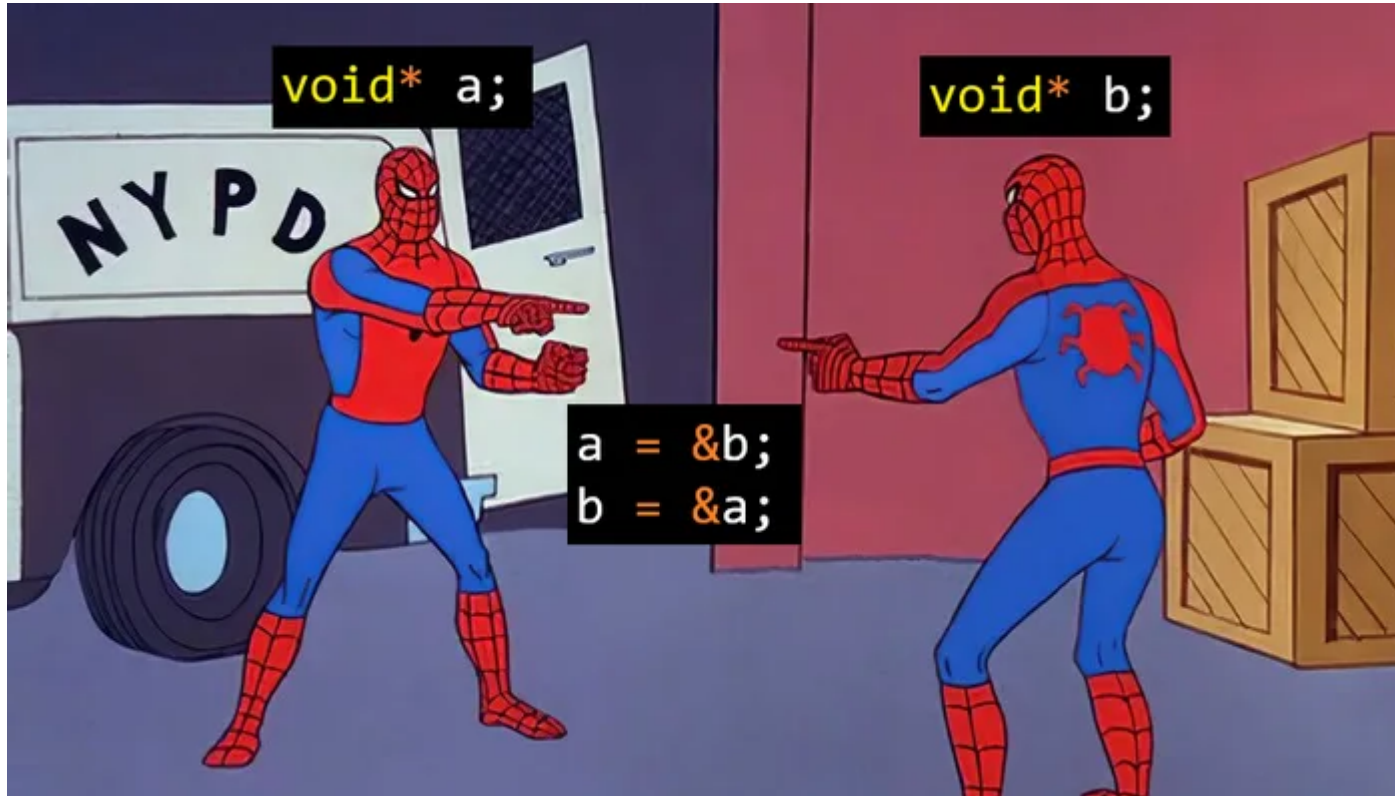


CSCI 2330



Lab0 graded

- Find your grade on krebs in:
- `assignments/lab0/grade.txt`

TODO

- Assignment 1 is assigned (due September 24)

Overview for today

- Pointers
- Arrays

“but”...

- Earlier we saw “but” is logically “and”
- Where could this be relevant?
- Consider xorBits():
- For bits a and b , $a \wedge b$ is 1 whenever:
- a **or** b is 1 **but** not both

- “Pointers can feel quite intimidating when you first see them... If you’re feeling a bit lost... don’t panic! That’s totally normal. To get comfortable with pointers, you’ll need to spend a lot of time toying with code... and drawing lots of diagrams.” – Stanford lecture notes

- “The topic of pointers is no doubt the hardest topic that we will cover/have covered for this class. The basic concepts of pointers take a lot of thinking to understand. This ***will*** be hard stuff...” UC Berkeley lecture notes

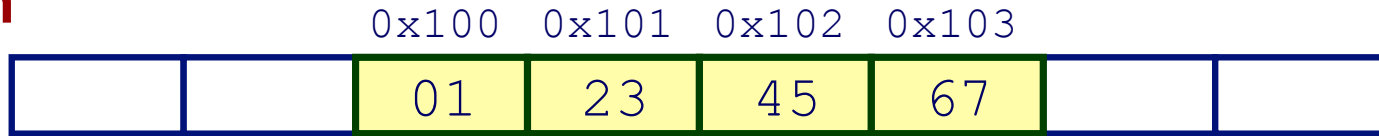
Memory is *byte*-addressable

- **Not** bit-addressable!
- **Not** int-addressable!
- If we want a specific *bit*:
 - Get the entire corresponding byte
 - Use bitwise operations to extract the bit
- How do we store an int?

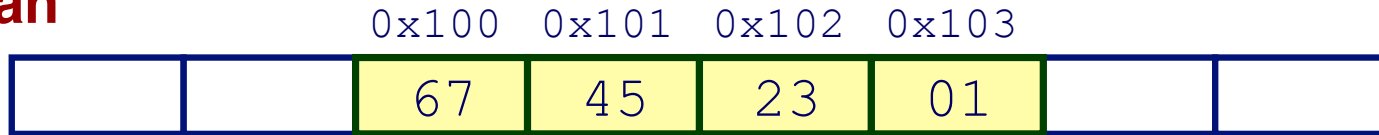
Endianness

```
int x = 0x01234567; // 32-bit num at address 0x100
```

Big Endian



Little Endian



Addresses in C

- Every variable in C has a *value*
- Every variable in C has an *address*

C pointers

- Address – index of a byte of memory
- Pointer – variable storing an address
- `T *p; // declare a pointer p pointing to type T`
- `&x // address of x`
- `*p // dereference p (evaluates to what p points to)`

Creating an address

```
int i = 5;  
double x = 0.25;  
int *p1 = &i; // p1 now points to address of i  
double *p2 = &x; // p2 now points to address of d
```

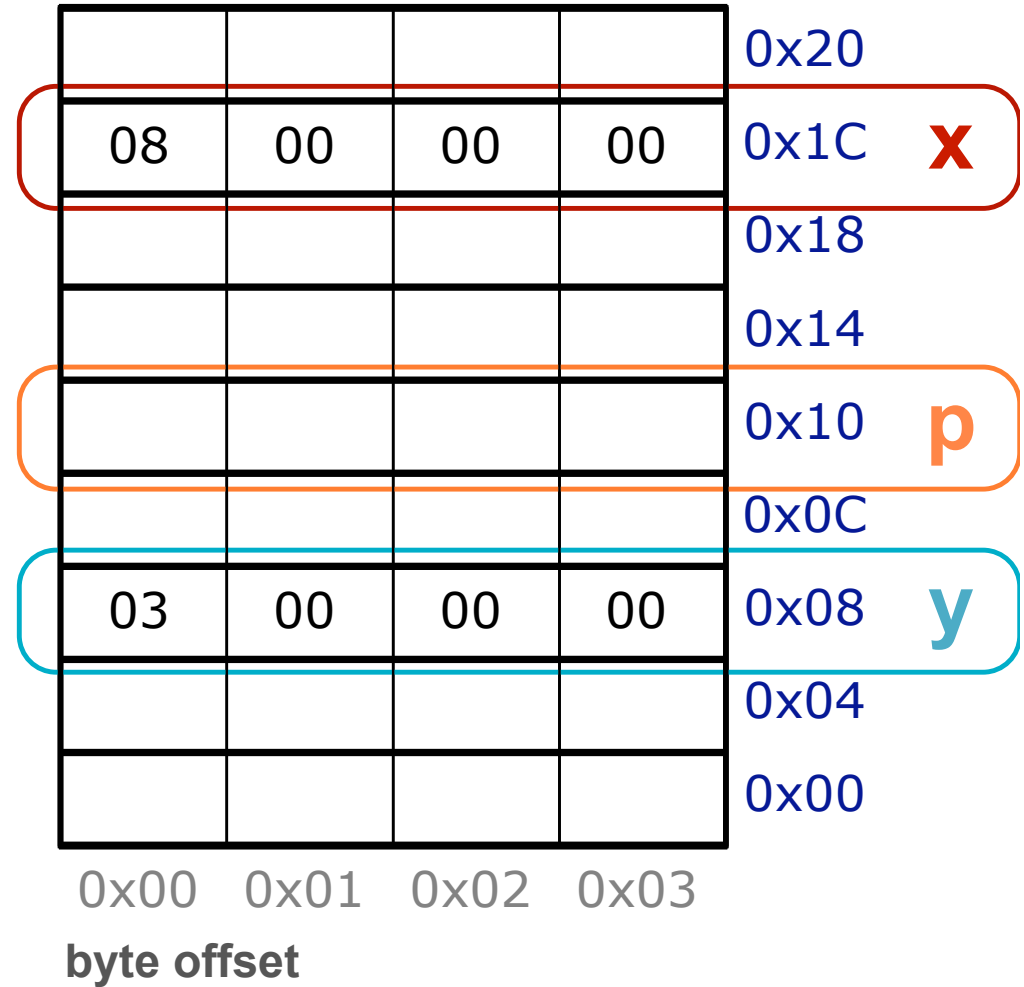
Dereferencing a pointer

```
int i = 5;  
double x = 0.25;  
int *p1 = &i; // p1 now points to address of i  
double *p2 = &x; // p2 now points to address of d  
int j = *p1; // j is now 5  
double y = *p2; // y is now 0.25
```

```
int *p; // p: 0x10
```

```
int x = 8; // x: 0x1C
```

```
int y = 3; // y: 0x08
```

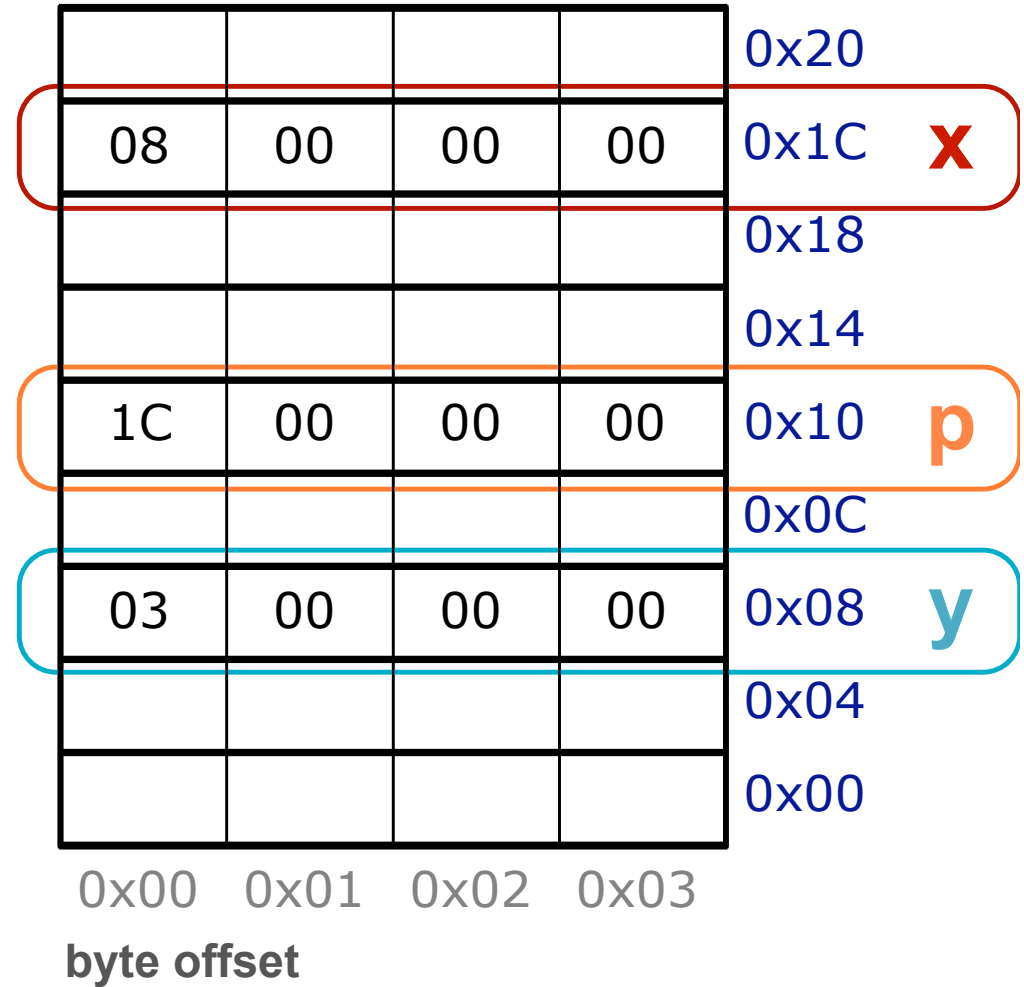


```
int *p; // p: 0x10
```

```
int x = 8; // x: 0x1C
```

```
int y = 3; // y: 0x03
```

```
p = &x;
```



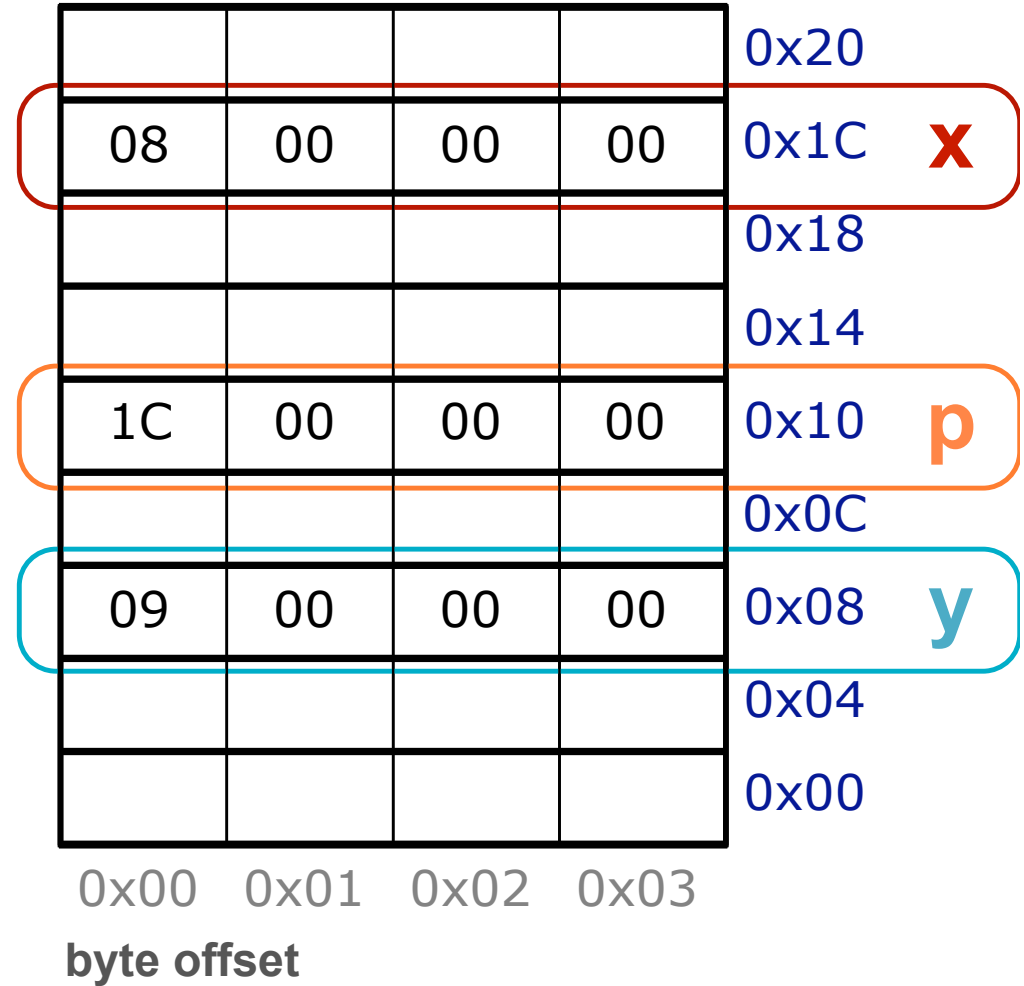
```
int *p; // p: 0x10
```

```
int x = 8; // x: 0x1C
```

```
int y = 3; // y: 0x08
```

```
p = &x;
```

```
y = 1 + *p;
```



```
int *p; // p: 0x10
```

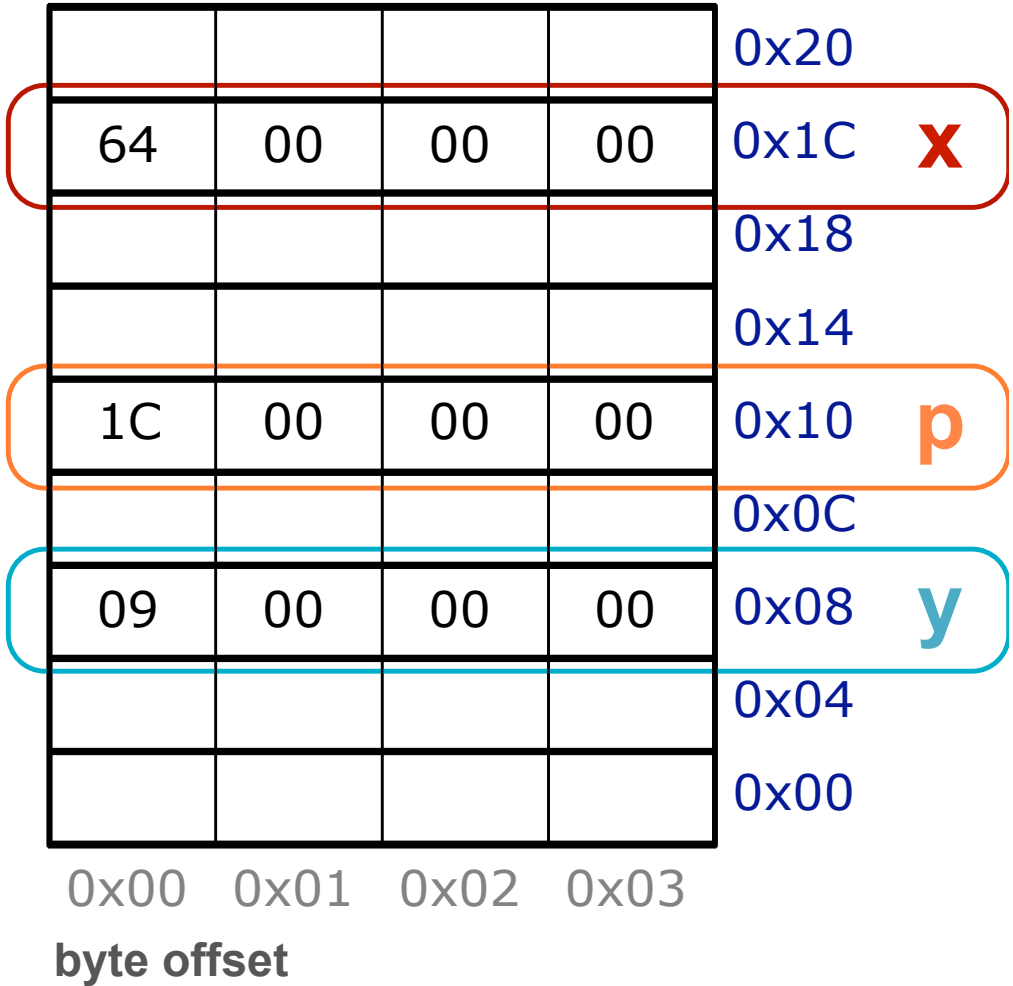
```
int x = 8; // x: 0x1C
```

```
int y = 3; // y: 0x08
```

```
p = &x;
```

```
y = 1 + *p;
```

```
*p = 100;
```



Pointers Exercise

```
int *p; // p: 0x10
```

```
int x = 8; // x: 0x1C
```

```
int y = 3; // y: 0x08
```

```
p = &x;
```

```
y = 1 + *p;
```

```
*p = 100;
```

				0x20
64	00	00	00	0x1C x
				0x18
				0x14
1C	00	00	00	0x10 p
				0x0C
09	00	00	00	0x08 y
				0x04
				0x00
0x00	0x01	0x02	0x03	
byte offset				

Swap function

```
void swap(int *p1, int *p2) {  
    int temp = *p1;  
    *p1 = *p2;  
    *p2 = temp;  
}  
int x = 5;  
int y = 3;  
swap(&x, &y);  
// now x == 3 and y == 5
```

Arrays in C

- Seen a little when discussing strings
- Recall: strings in C are char arrays
- Consecutive storage in C
- E.g., array of 5 4-byte ints is 20 bytes

Arrays in C

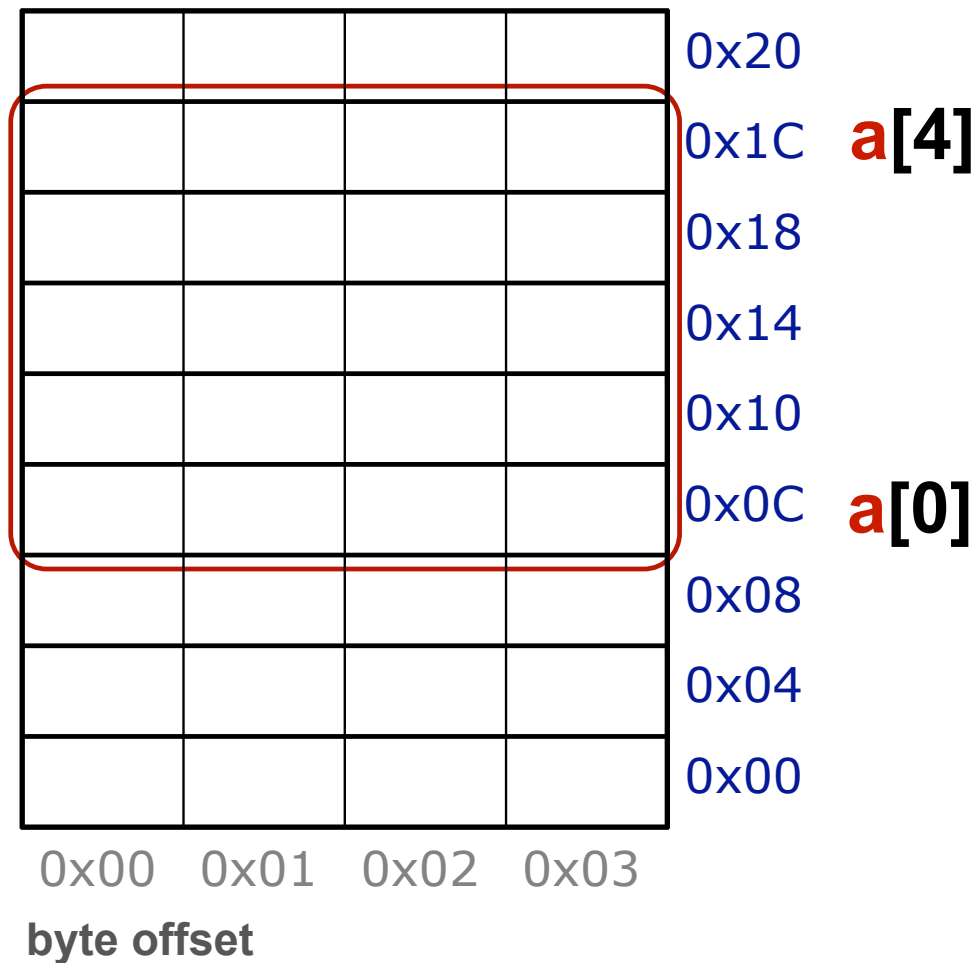
- Creation, setting, getting follow familiar syntax...

```
int a[5]; // creation
```

```
a[0] = 0xFF; // indexing
```

```
a[3] = a[0];
```

```
a.length; // nope!
```



We can keep track of length

- `int a[5]; // only those 5 * 4 == 20 bytes of ints`
- `a.length; // nope`
- `int length = 5;`

Endianness versus arrays

- Endianness applies to the array elements
- Not to the array indexes!
- `int a[2]; a[0] = 0x11223344; a[1] = 0x55667788;`
- 44 33 22 11 88 77 66 55
- **NOT:**
- 88 77 66 55 44 33 22 11

What happens when we are bad?

- What happens to us when we make bad array accesses?

```
int a[5]; // creation
```

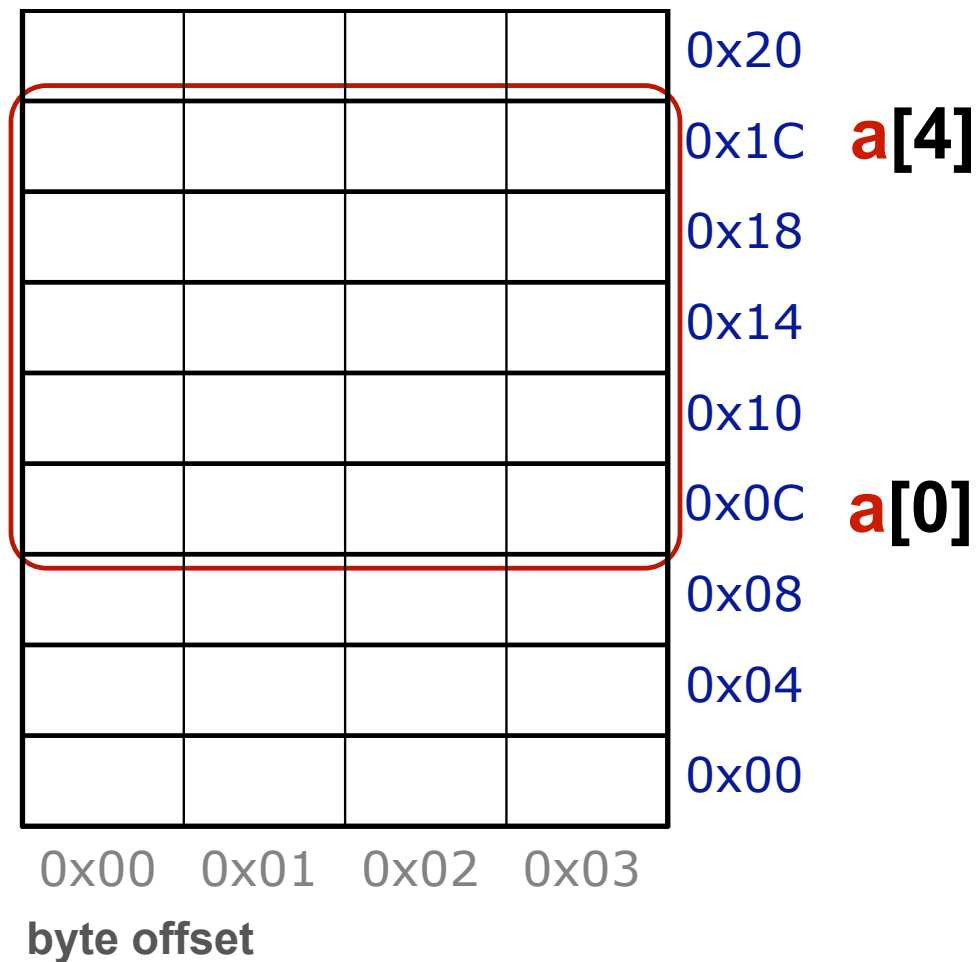
```
a[0] = 0xFF; // indexing
```

```
a[3] = a[0];
```

```
a.length; // nope!
```

```
a[5] = 0xBAD; // uh oh
```

```
a[-1] = 0xBAD; // x2
```



What happens in Java?

- ???

What happens in Java?

- `ArrayIndexOutOfBoundsException`

What happens in Java?

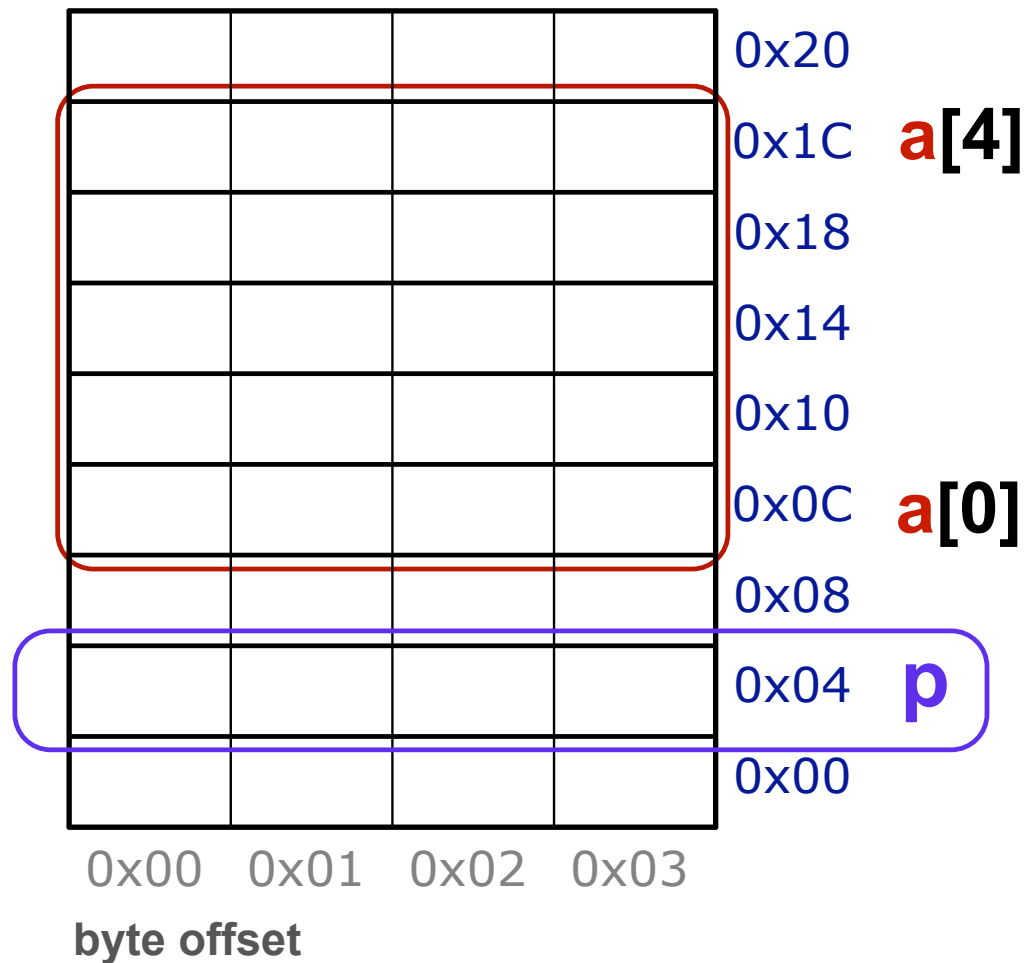
- `ArrayIndexOutOfBoundsException`
- In C?
- Undefined behavior
 - Could read or write unintended places in memory...
 - Program could immediately crash
 - Program could crash later
 - Difficult to debug these issues!

Array to pointer decay

- We can treat arrays as pointers in C
- In fact, any kind of arithmetic on an array causes it to decay into a pointer
- Note: array subscript syntax works on pointers

```
int a[5];
```

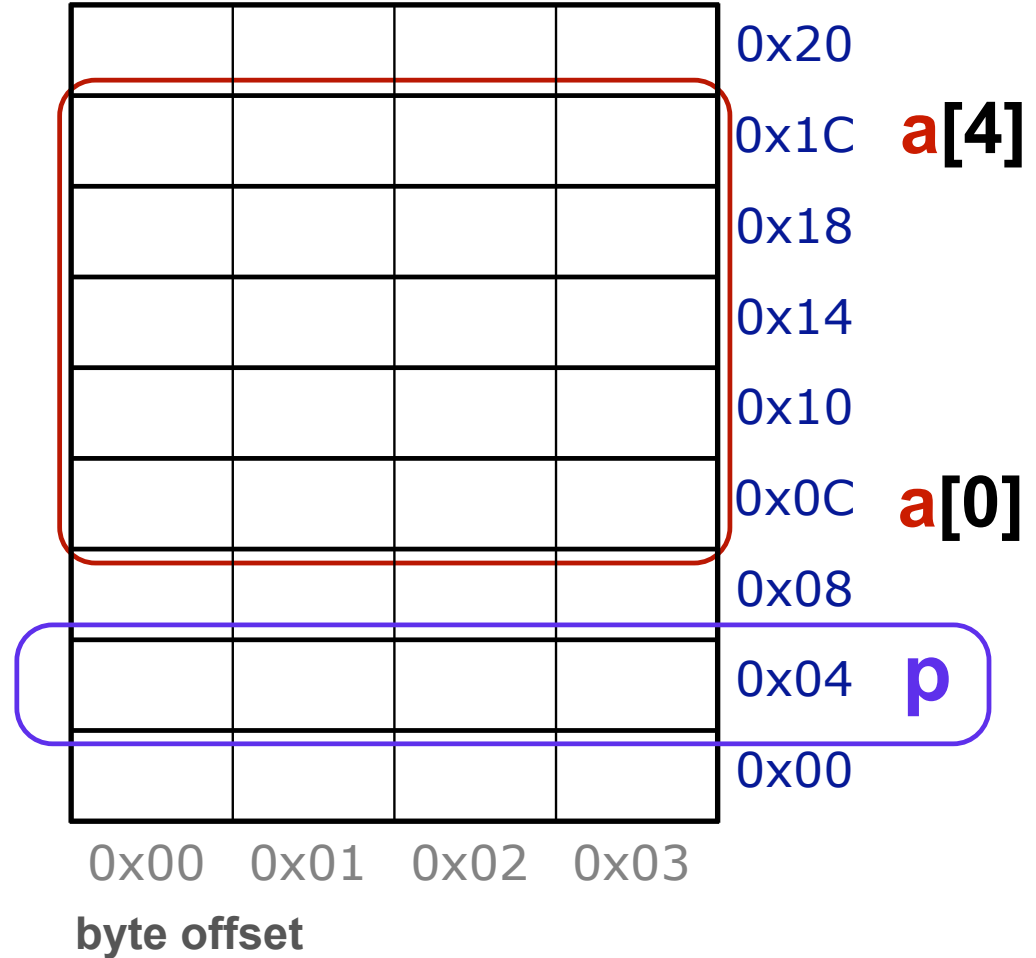
```
int *p;
```



```
int a[5];
```

```
int *p;
```

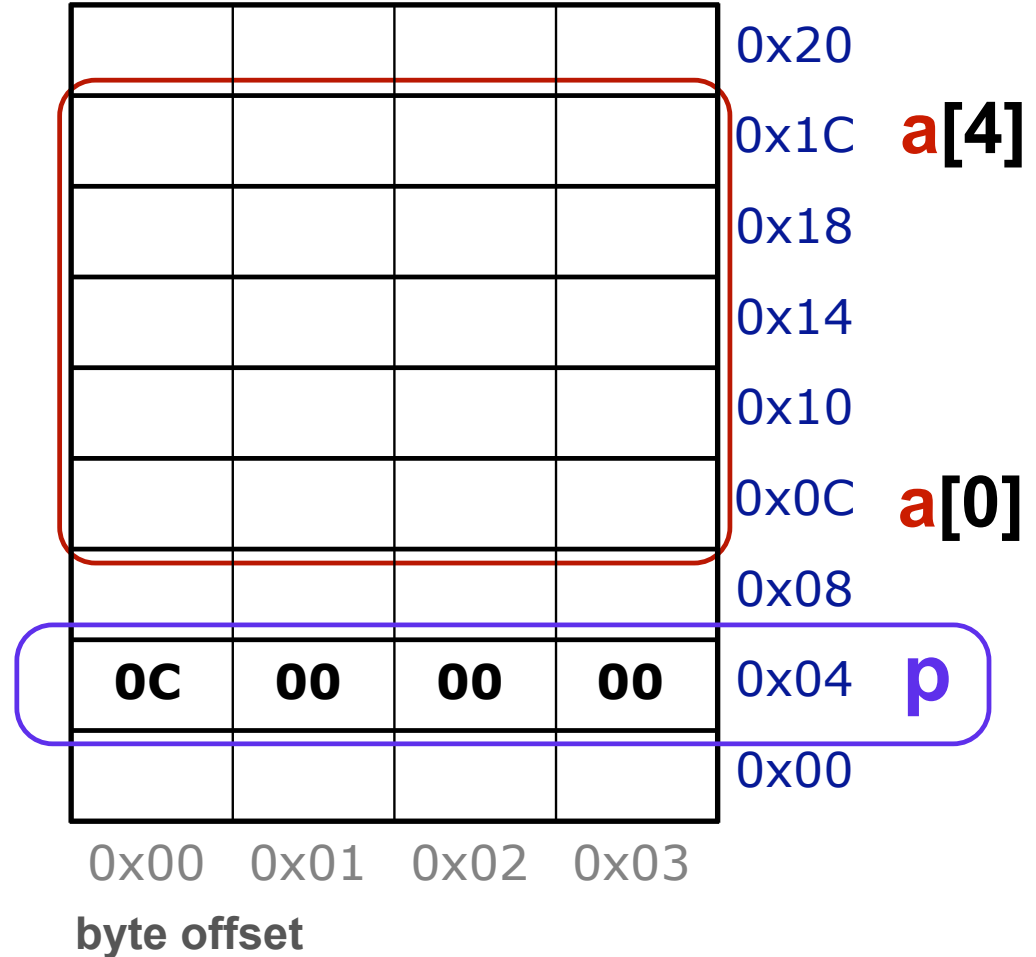
```
p = &a[0];
```



```
int a[5];
```

```
int *p;
```

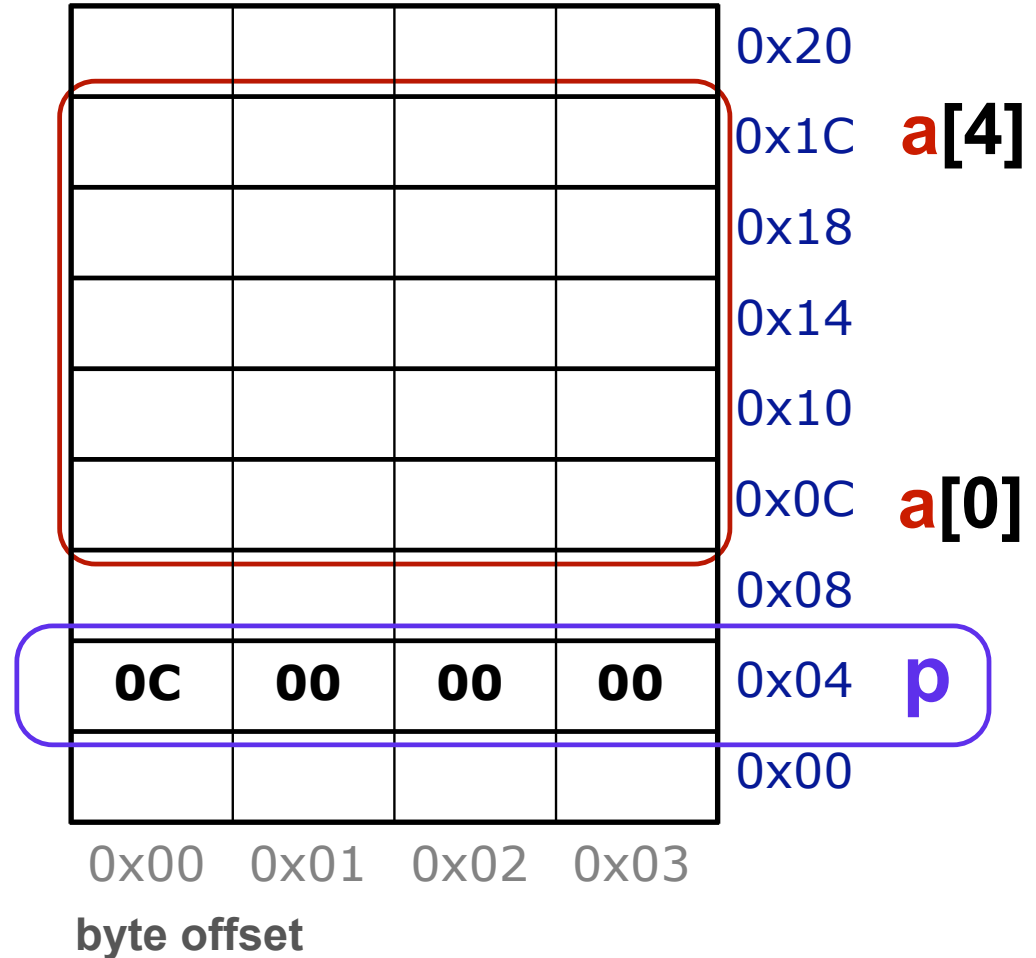
```
p = &a[0];
```



```
int a[5];
```

```
int *p;
```

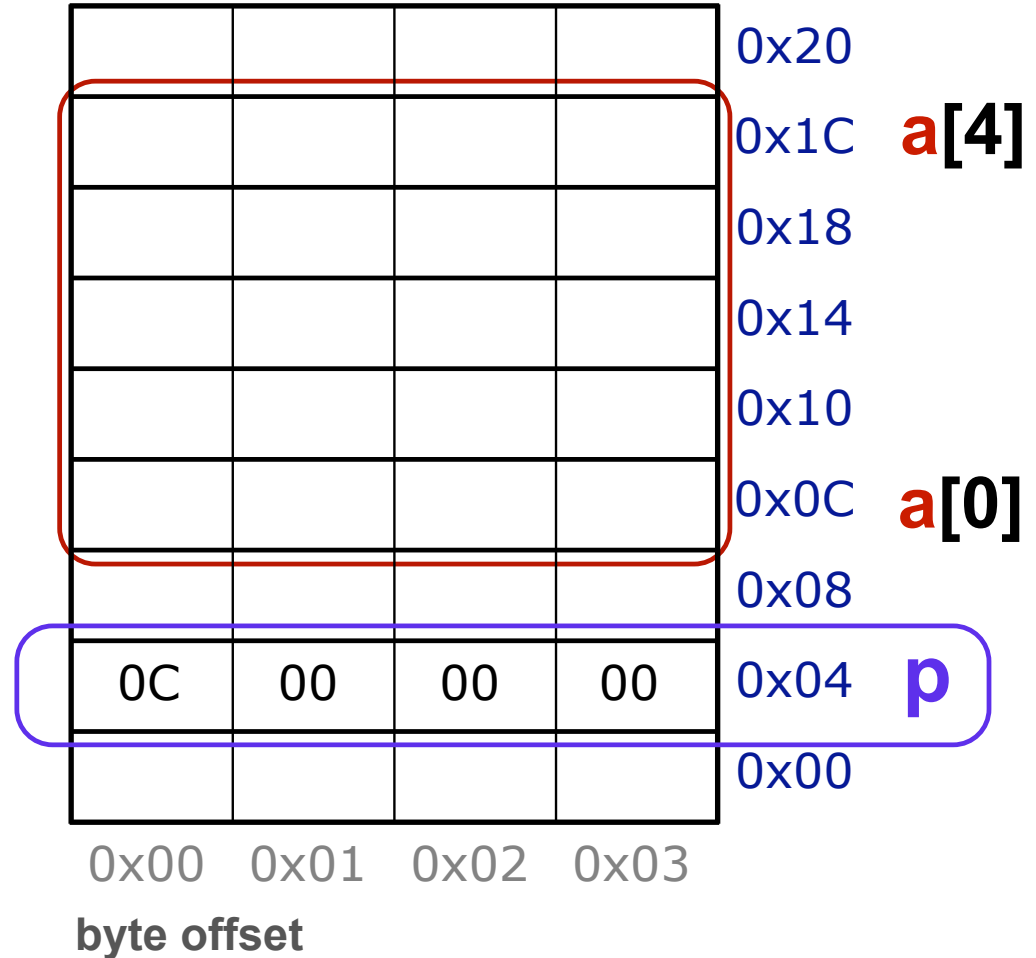
```
p = &a[0]; // or p = a;
```



```
int a[5];
```

```
int *p;
```

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

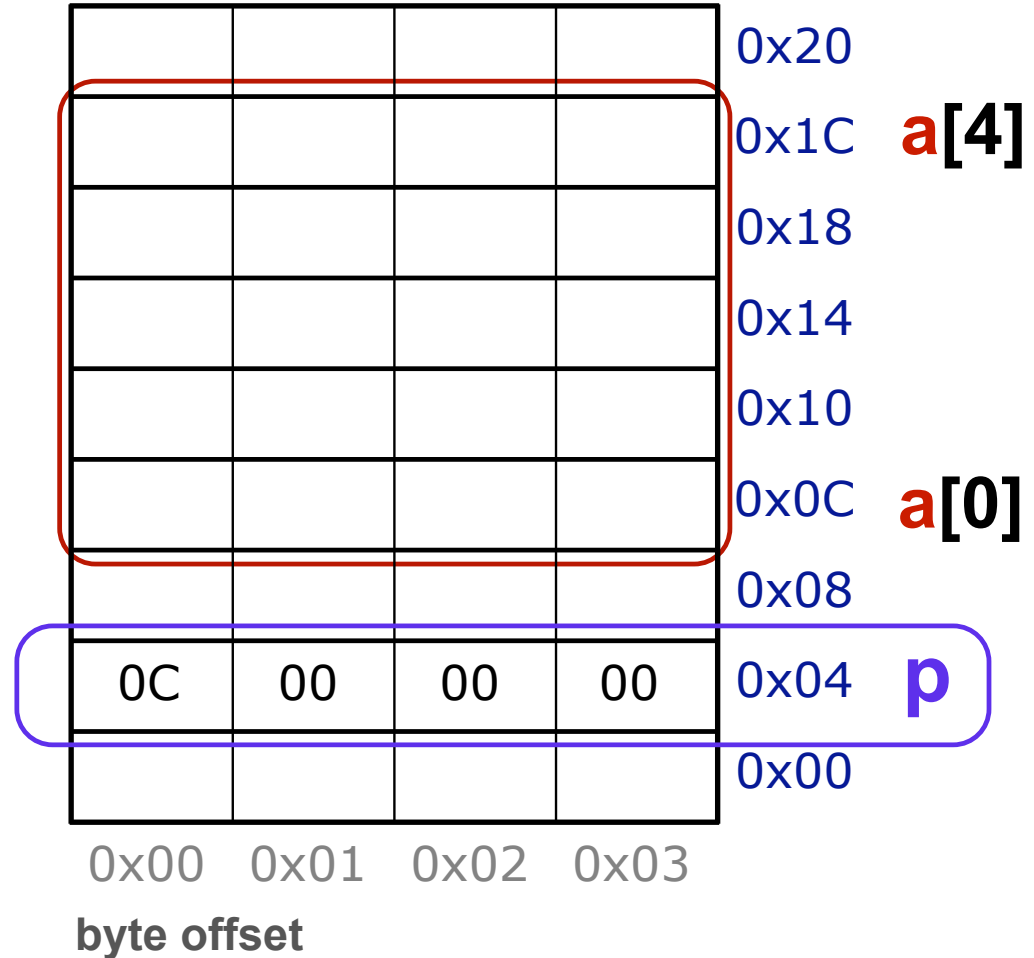


```
int a[5];
```

```
int *p;
```

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

```
*p = 0xA;
```

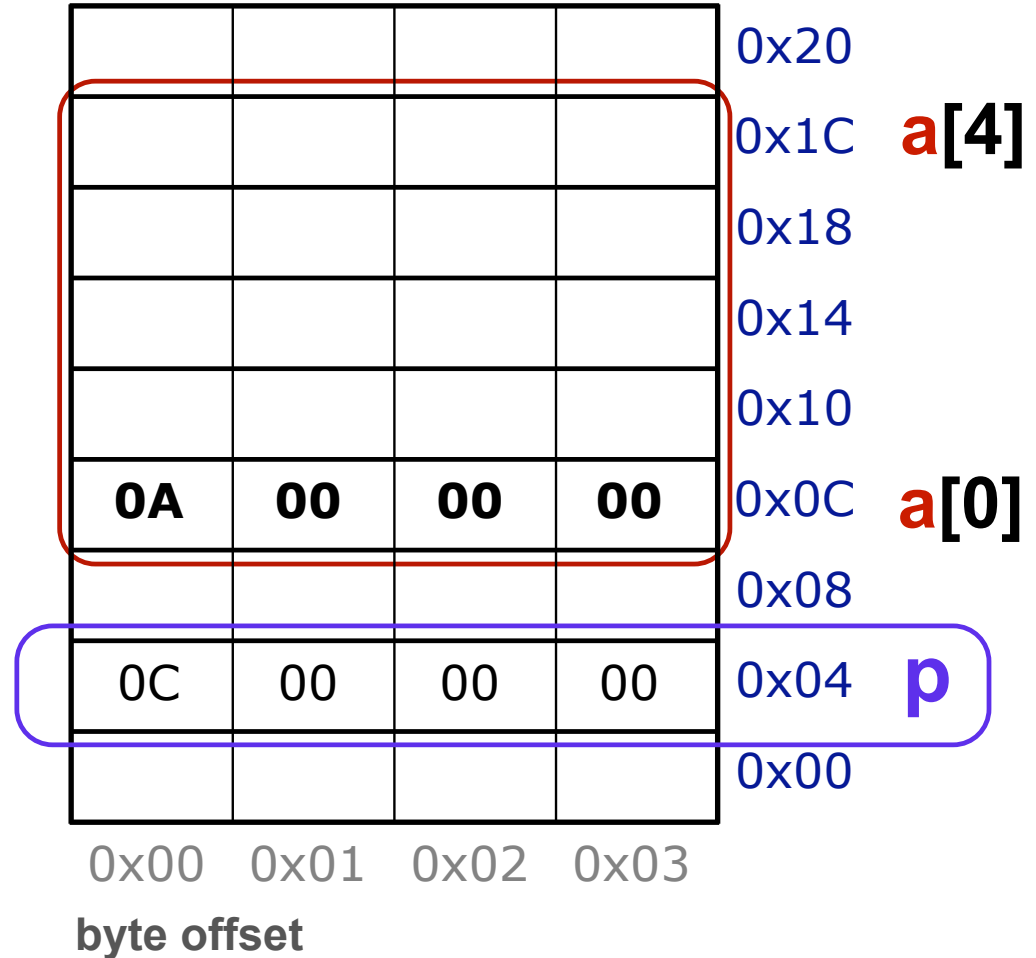


```
int a[5];
```

```
int *p;
```

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

```
*p = 0xA;
```



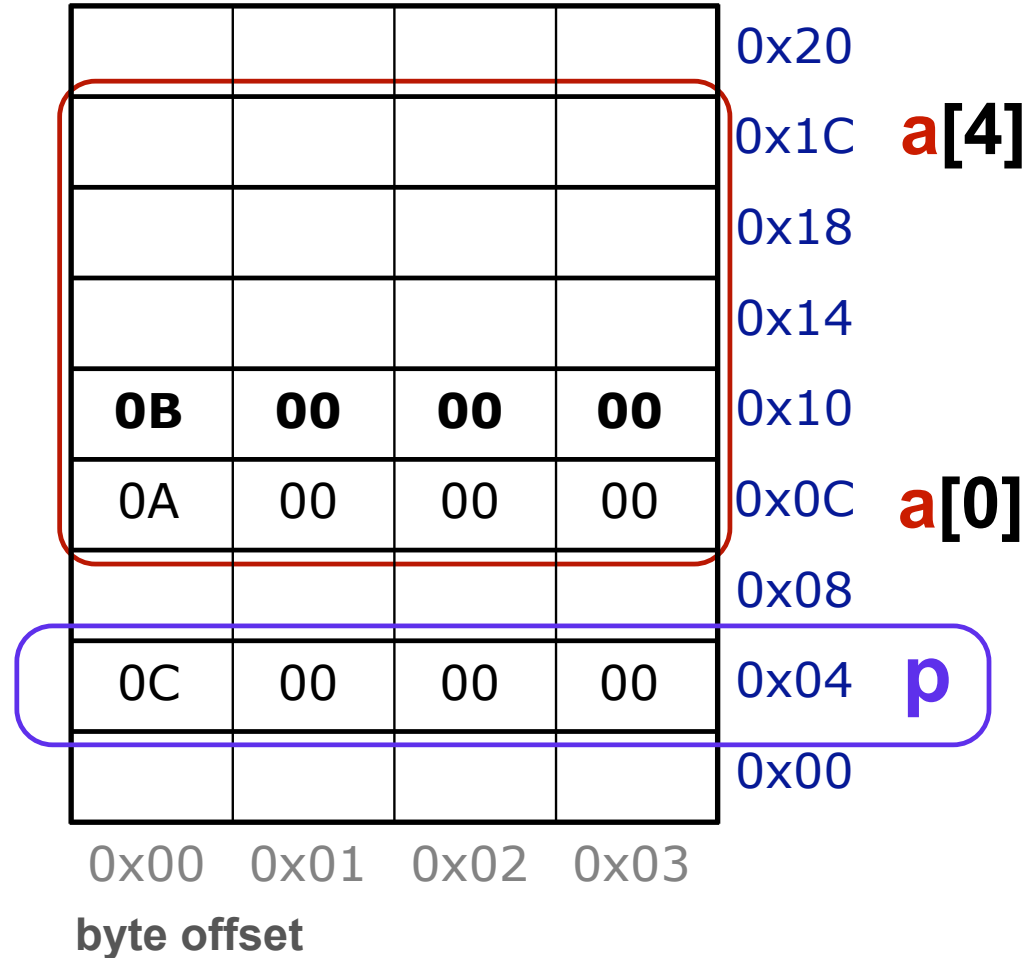
```
int a[5];
```

```
int *p;
```

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

```
*p = 0xA;
```

```
a[1] = 0xB; // using a
```



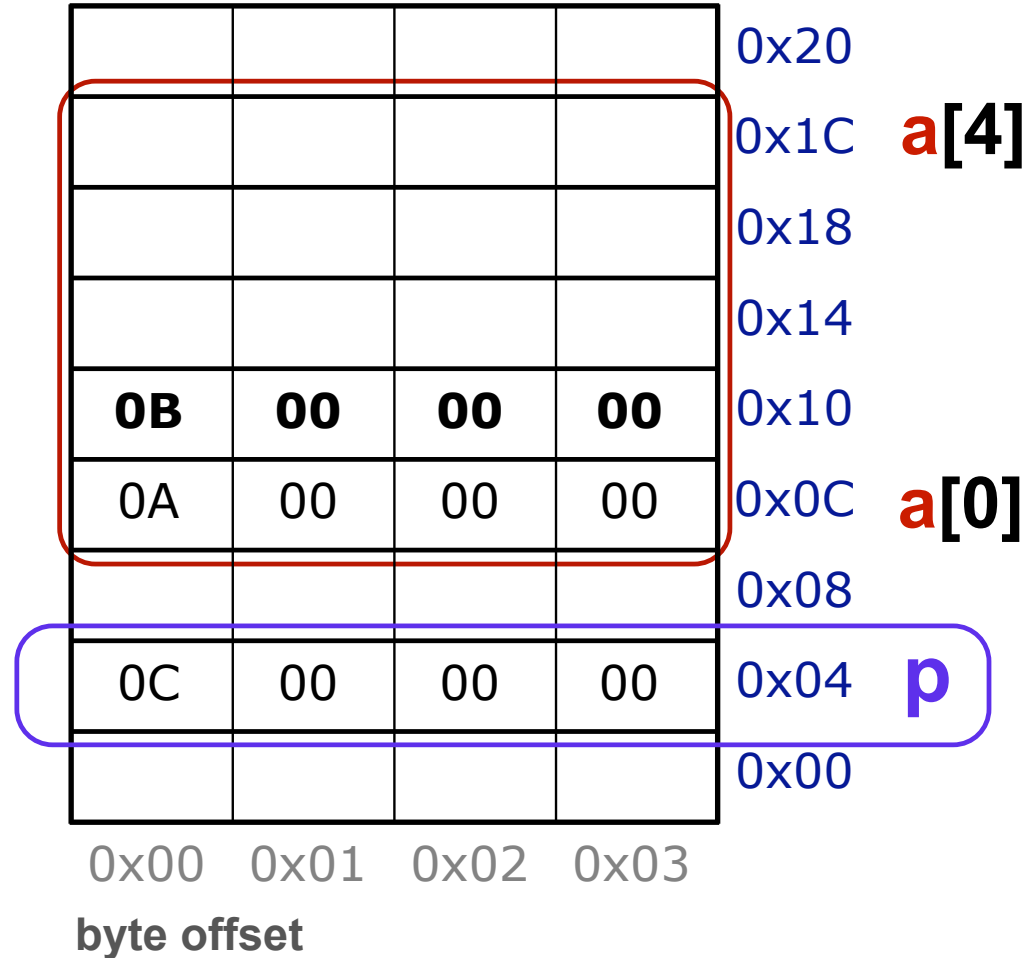
```
int a[5];
```

```
int *p;
```

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

```
*p = 0xA;
```

```
p[1] = 0xB; // using p
```



Pointer arithmetic

- Treat the address like an integer and add to get a new address
- `int a[5]; // 0x0c`
- `int *p = a; // p == 0x0c`
- `p = p + 1; // p == 0x0d?`

Pointer arithmetic

- Treat the pointer like a number and add to get the index
- `int a[5]; // 0x0c`
- `int *p = a; // p == 0x0c`
- `p = p + 1; // p == 0x0d 0x10`
- **C automatically scales arithmetic by size of the type being pointed to (int is 4 bytes)**

Pointer arithmetic

- `long *p1 = a1 + 1; // adds 8 to actual address`
- `double *p2 = a2 + 1; // adds 8 to actual address`
- `short *p3 = a3 + 1; // adds 2 to actual address`
- `char *p4 = a4 + 1; // adds 1 to actual address`

Equivalence

- Note:
- `int a[5];`
- `&a[b] == a + b`

More C syntactic sugar

- `a[b]` is syntactic sugar for `*(a + b)`
- Note also: like integer arithmetic, pointer addition is commutative: `*(a + b) == *(b + a)`

So we have...

- We also have $\&a[b] == a + b$
- Because: $\&a[b] == \&*(a + b) == a + b$
- Note: $\&\&\&x == x$
- (A “ $\&*$ ” cancels itself out.)

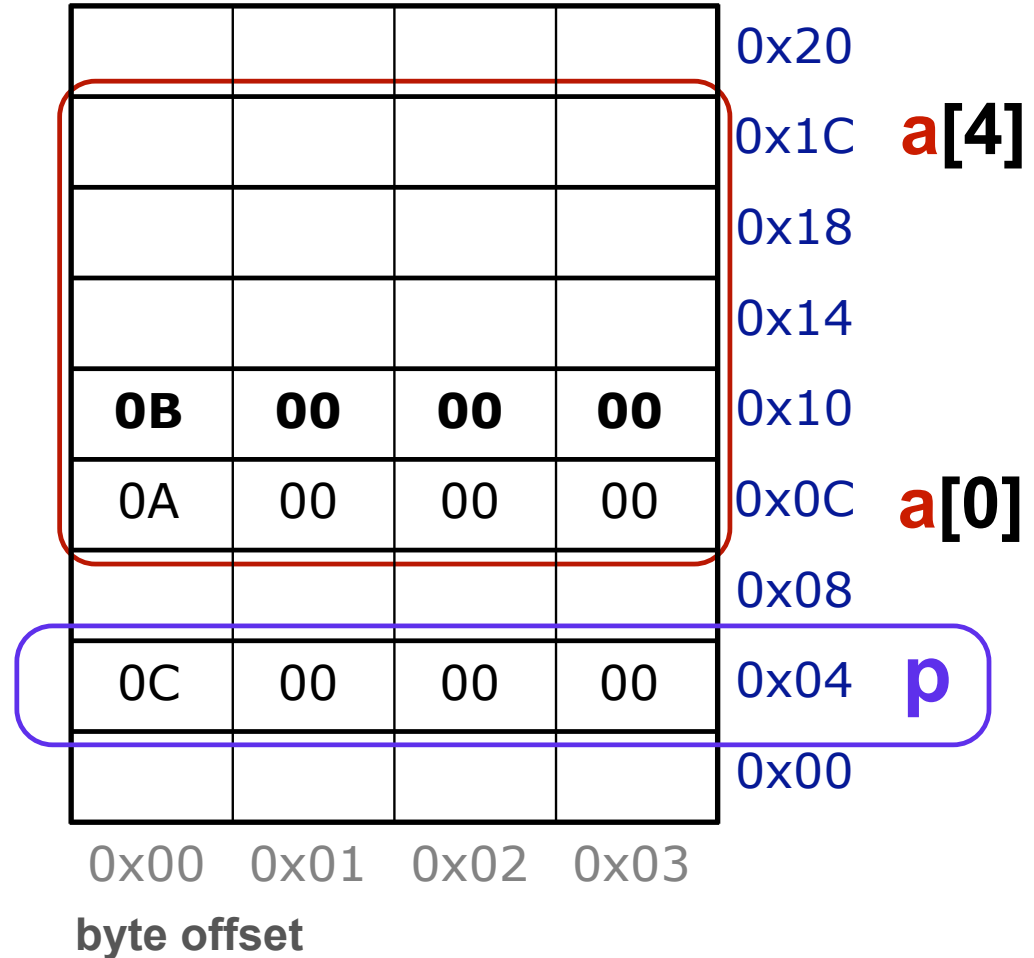
```
int a[5];
```

```
int *p;
```

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

```
*p = 0xA;
```

```
p[1] = 0xB; // using p
```



```
int a[5];
```

```
int *p;
```

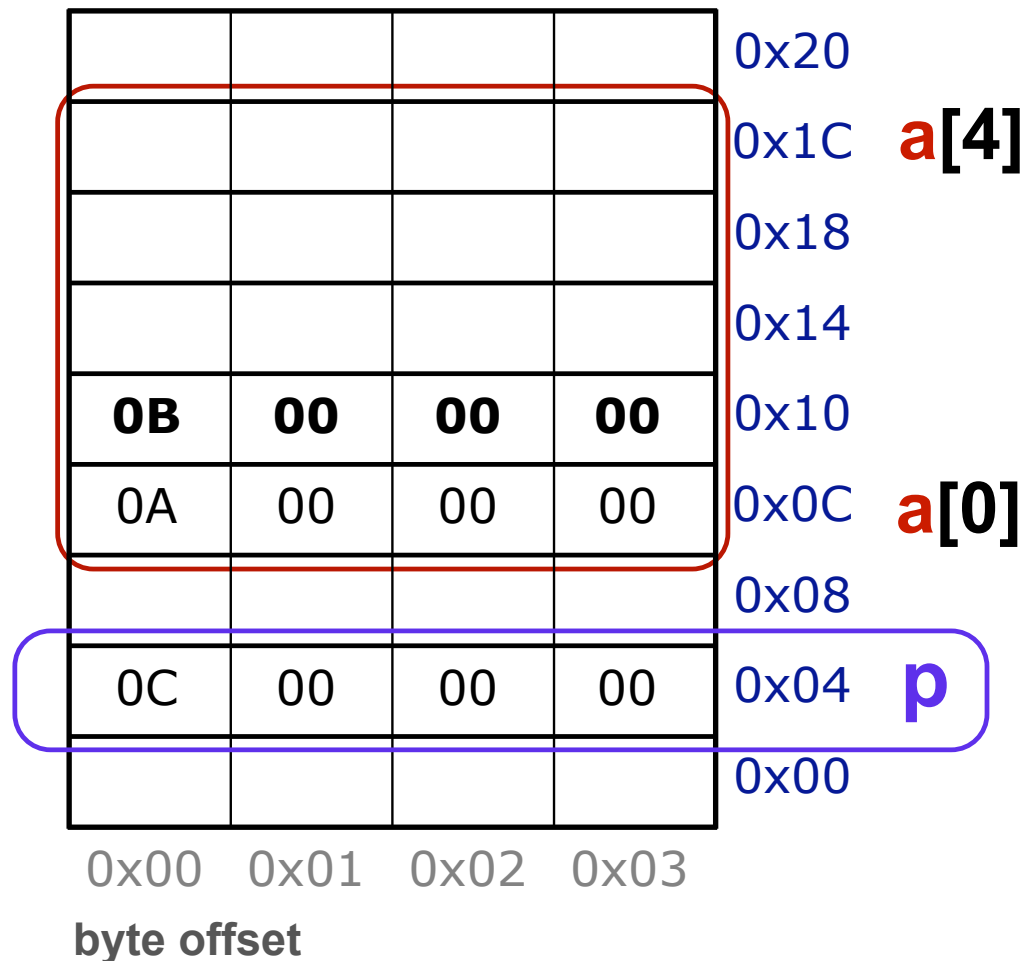
```
p = a; // &a[0]
```

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

 **Pointer
Arithmetic!**

(scaled by size of type)



```
int a[5];
```

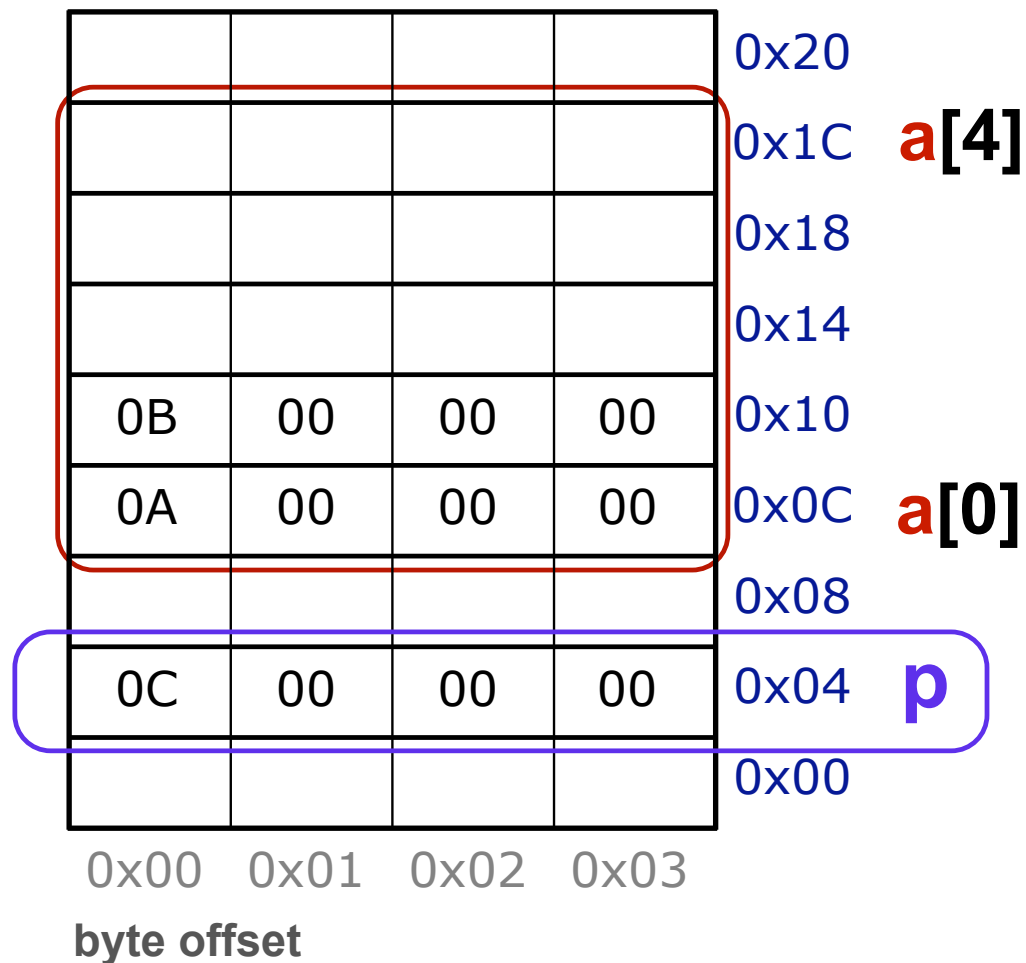
```
int *p;
```

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

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```



```
int a[5];
```

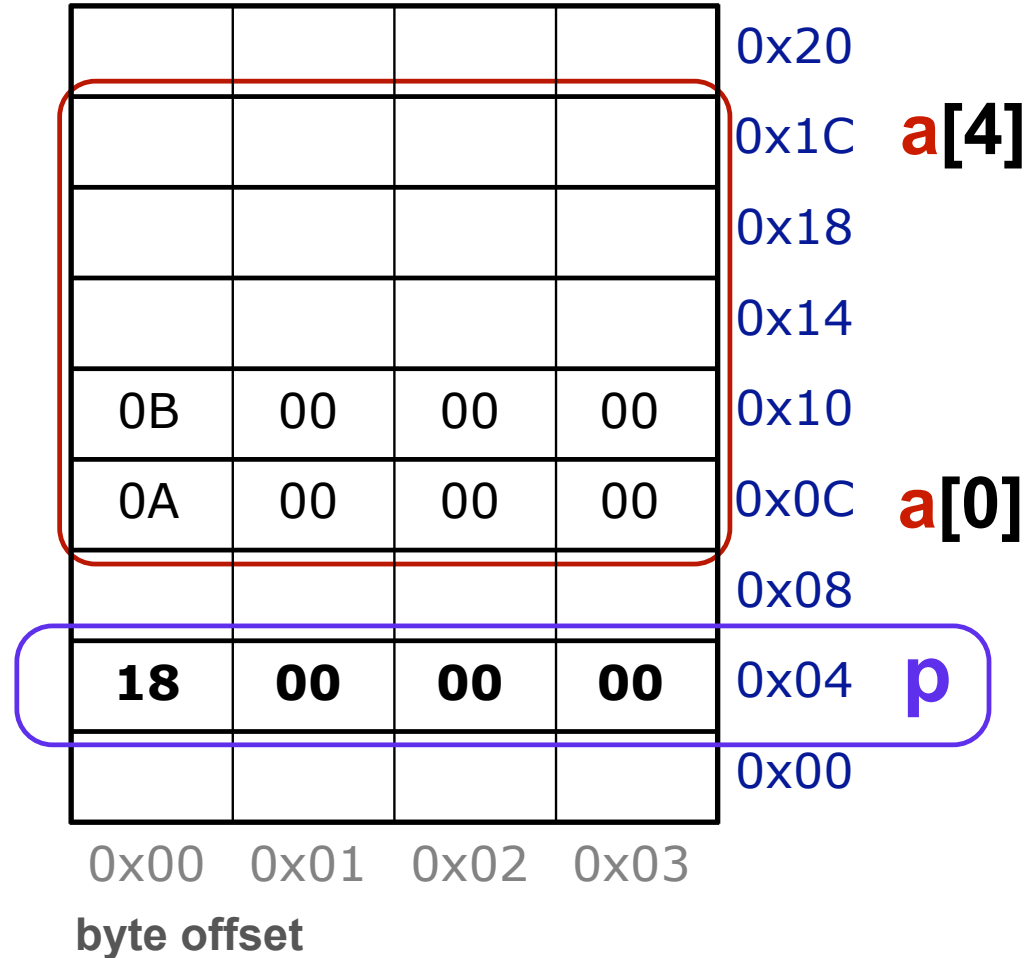
```
int *p;
```

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

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```



```
int a[5];
```

```
int *p;
```

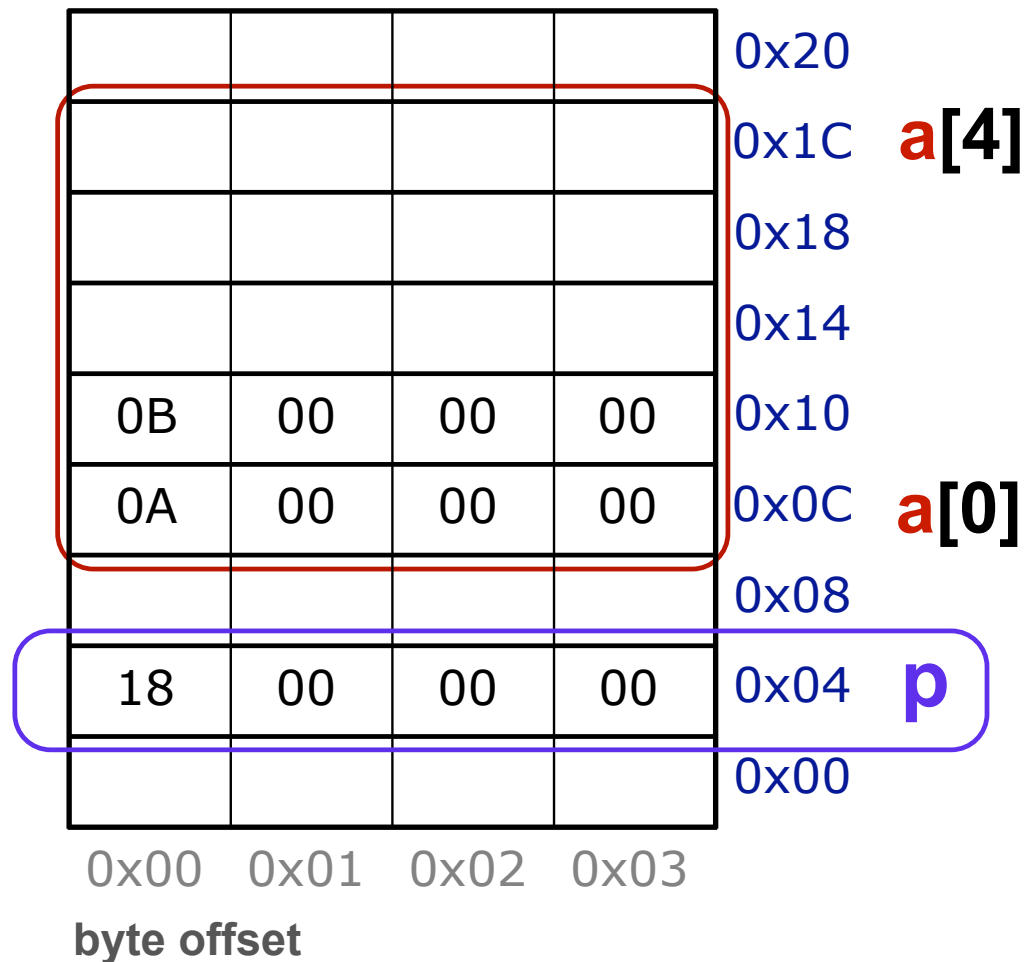
```
p = a; // &a[0]
```

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```

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



```
int a[5];
```

```
int *p;
```

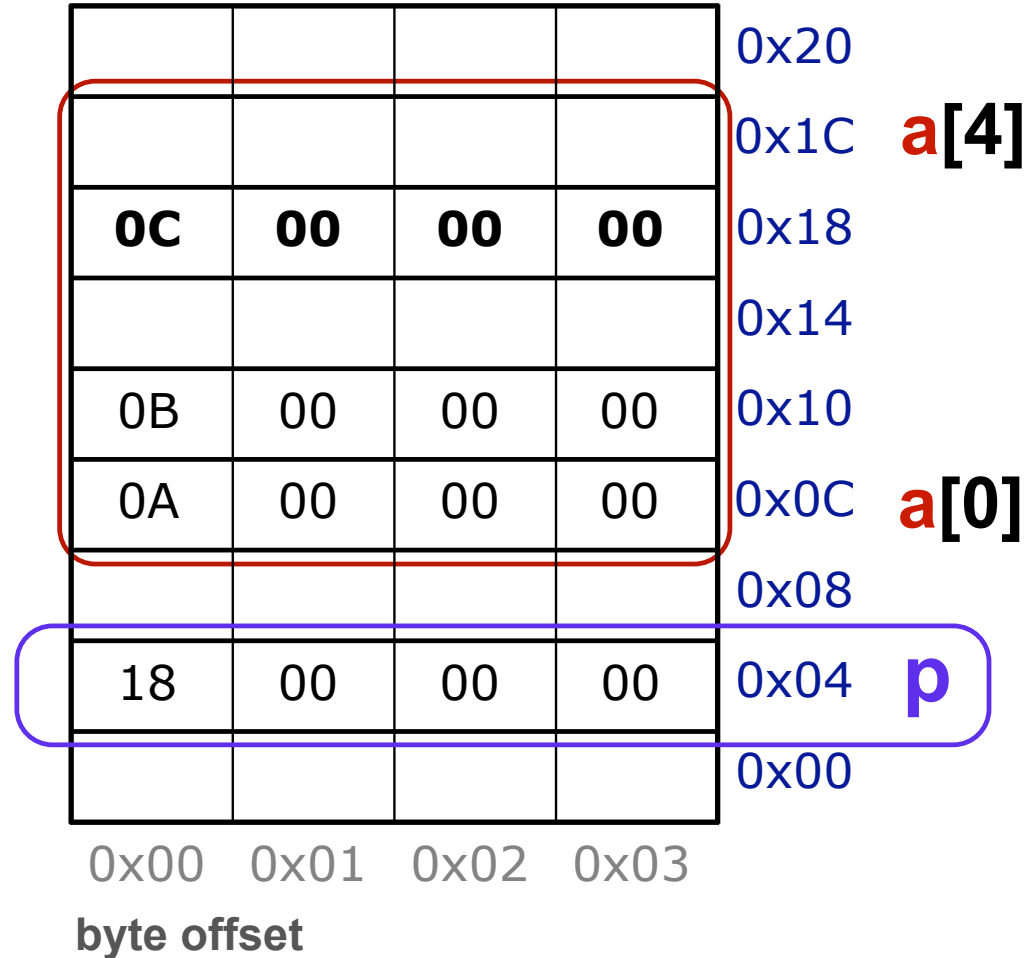
```
p = a; // &a[0]
```

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```

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



```
int a[5];
```

```
int *p;
```

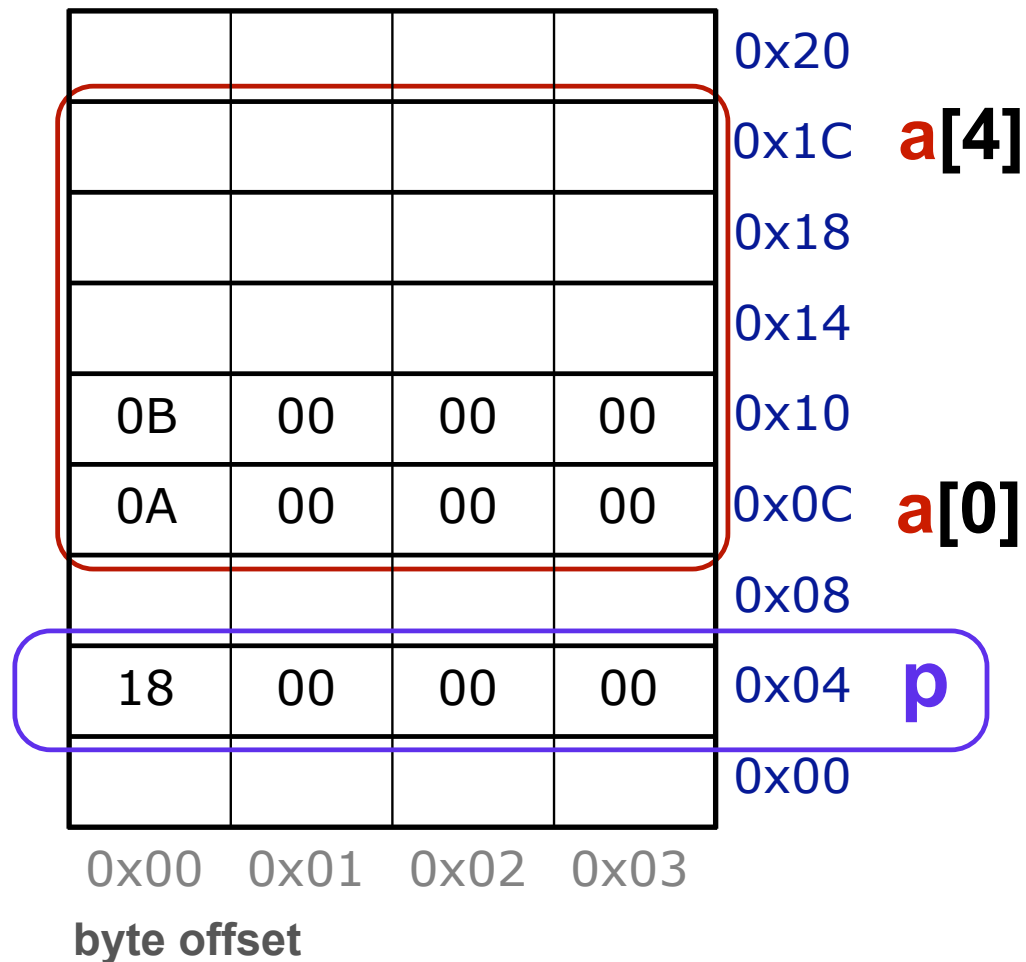
```
p = a; // &a[0]
```

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```

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



More C syntactic sugar

- `a[b]` is syntactic sugar for `*(a + b)`
- Note also: like integer arithmetic, pointer addition is commutative: `*(a + b) == *(b + a)`

```
int a[5];
```

```
int *p;
```

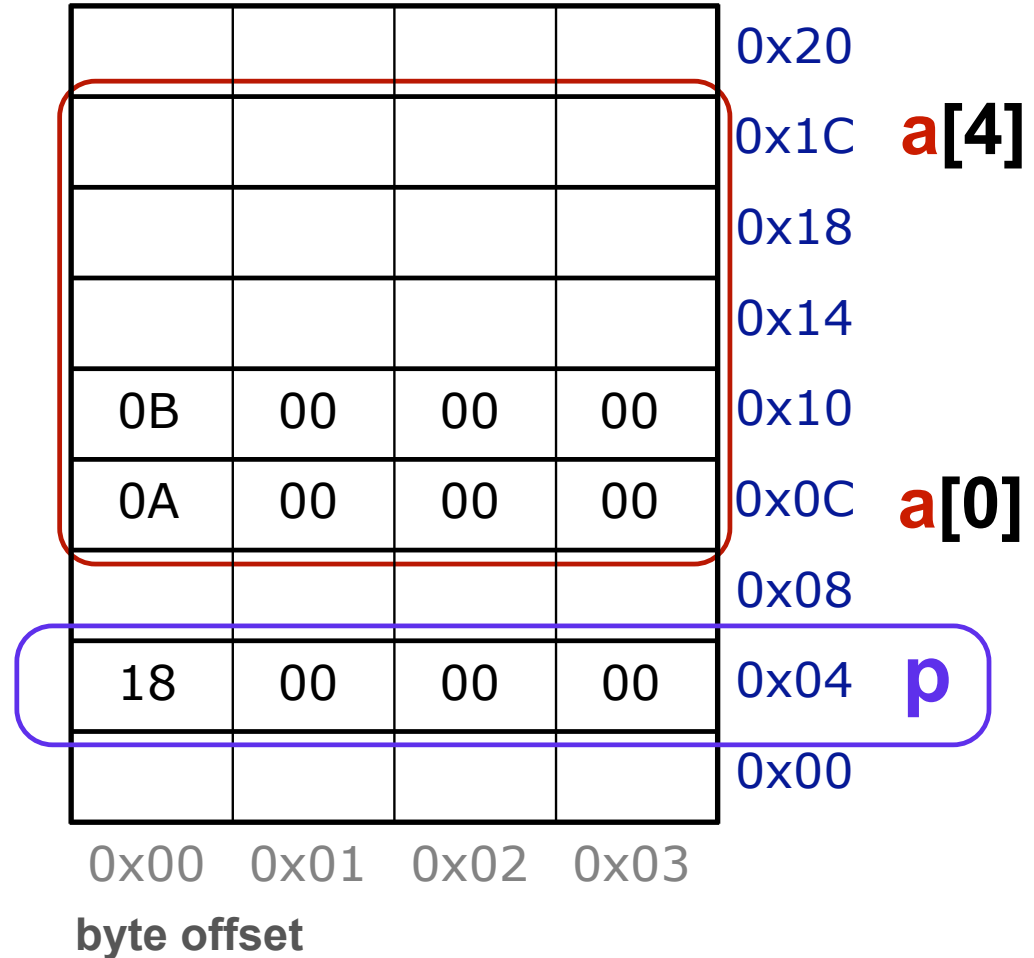
```
p = a; // &a[0]
```

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```

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



```
int a[5];
```

```
int *p;
```

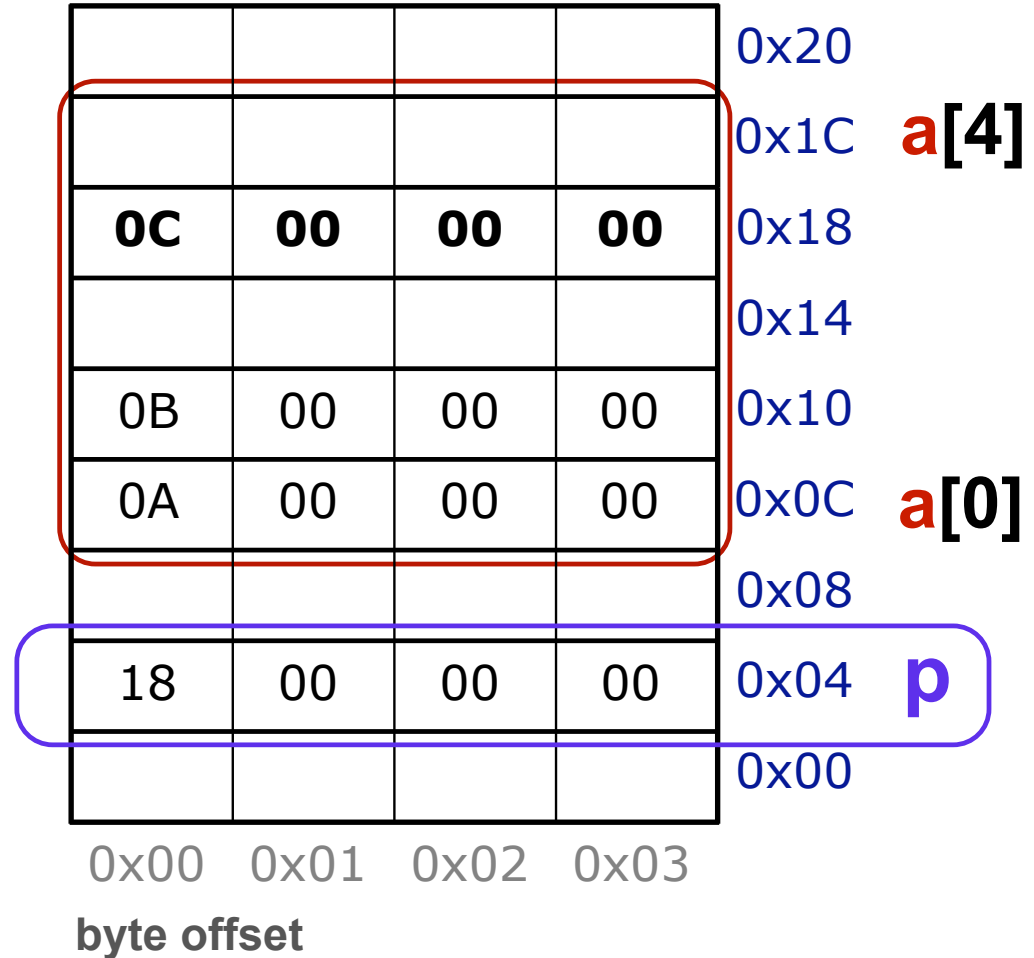
```
p = a; // &a[0]
```

```
*p = 0xA;
```

```
*(p + 1) = 0xB;
```

```
p = p + 3;
```

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



Undefined behavior in C

- E.g., bitshifting a number by its size or more
- `int x = ...; // 32-bit int`
- `(x >> 35) // !?!?`
- Possible behavior:
 - ???

Undefined behavior in C

- E.g., bitshifting a number by its size or more
- `int x = ...; // 32-bit int`
- `(x >> 35) // !?!?`
- Possible behavior:
 - Shift by $\min(32, 35) = 32$
 - Shift by $35 \% 32 = 3$
 - Crash
 - **Anything else actually**

C – a massively portable language



Hardware behavior diverges

- C cannot guarantee how a program will behave across every machine
- Certain behavior is classified as “undefined”

Examples of undefined behavior

- Bitshifting a 32-bit number by 32 or more
 - Keep this in mind for upcoming Lab 1!
- Signed integer overflow/underflow
 - But not unsigned!
 - This is because there are some rare machines that are not two's complement 🙄

Undefined behavior is illegal

- It's worse than just YOLOing it
- You are **not allowed** to have undefined behavior in your program
- This can allow the C compiler to make some...
“interesting” optimizations

Undefined behavior is illegal

```
int flag = some_function();  
if (flag) {  
    defined behavior  
    undefined behavior  
} else {  
    defined behavior  
    defined behavior  
}
```

Undefined behavior is illegal

```
int flag = some_function(); 0;
```

```
if (flag) {
```

```
—defined behavior
```

```
—undefined behavior
```

```
} else {
```

```
    defined behavior
```

```
    defined behavior
```

```
}
```

Undefined behavior is illegal

```
int flag = some_function(); 0;
```

```
if (flag) {
```

```
—defined behavior
```

```
—undefined behavior
```

```
} else {
```

```
    defined behavior
```

```
    defined behavior
```

```
}
```

Compiler: “Jeff would
NEVER write code
with undefined
behavior!”

What the krebs *machine* does

- For shifting more than the width, raw machine code would take the mod approach
- Shift by $35 \% 32 = 3$
- **But you are still not guaranteed that C will do this**

Undefined behavior summary

- Undefined behavior can induce strange behavior even outside of where it is
- Watch out for bit shifts of 32 or more in Lab 1
- Just avoid undefined behavior in C
- Java and Python don't have undefined behavior

TODO

- Assignment 1 is assigned (due September 24)