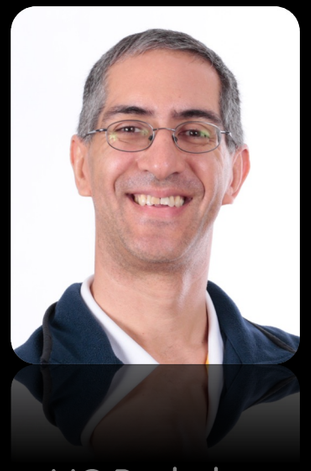


UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

C Pointers, Arrays, and Strings

Agenda

continued from last time

More C Features

- More C Features
- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings
- Endianness, Word Alignment

The C bool: True or False?

- Boolean is **not** a primitive type in C! Instead:
- FALSE:
  - `0` (integer, i.e., all bits are 0)
  - `NULL` (pointer) (more later)
- TRUE:
  - **Everything else!**
  - Note: Same is true in Python.
- Nowadays: true and false are provided by `stdbool.h`.

```
if (42) {
  printf("meaning of life\n");
}
```

A. meaning of life
B. (nothing)

More Typing: Typedefs and Structs

- typedef allows you to define new types.

```
typedef uint8_t BYTE;
BYTE b1, b2;
```

- structs are structured groups of variables, e.g.,

```
typedef struct {
  int length_in_seconds;
  int year_recorded;
} SONG;

SONG song1;
song1.length_in_seconds  =  213;
song1.year_recorded      = 1994;

SONG song2;
song2.length_in_seconds  =  248;
song2.year_recorded      = 1988;
```

SONG is an alias for `typedef struct {int length_in_seconds; int year_recorded; }`

Dot notation: `x.y = value`

song1

length_in_seconds	213	year_recorded	1994

Structs are **not** objects! The . operator is **not** a method call! (more later)

Consts, Enums, #define in C

[for reference]

- Constant, **`const`**, is assigned a typed value once in the declaration.
  - Value can't change during entire execution of program.

```
const float  golden_ratio = 1.618;
const int    days_in_week = 7;
const double the_law      = 2.99792458e8;
```

  - You can have a constant version of any of the standard C variable types.
- `#define PI (3.14159)` is a CPP (C Preprocessor) Macro.
  - Prior to compilation, preprocess by performing string replacement in the program based on all #define macros.
  - Replace all `PI` with `(3.14159)` → In effect, makes **`PI`** a "constant"
- Enums: a group of related integer constants. E.g.,

```
enum cardsuit {CLUBS,DIAMONDS,HEARTS,SPADES};
enum color {RED, GREEN, BLUE};
```

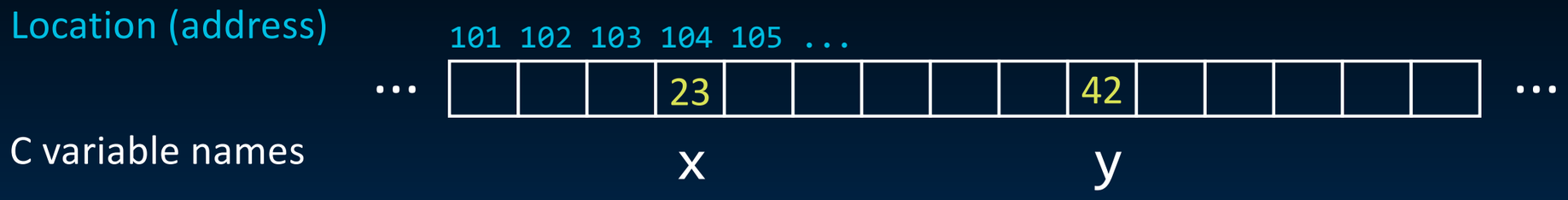
Agenda

Pointers

- More C Features
- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings
- Endianness, Word Alignment

Memory Is a Single Huge Array

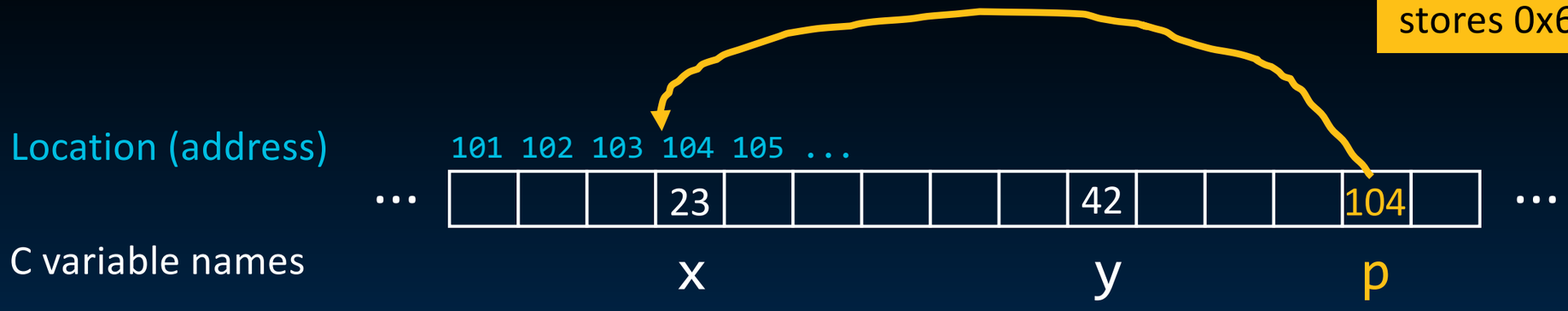
- Consider memory to be a byte-addressed array.
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
- Don't confuse the **address** referring to a memory location with the **value** stored in that location.
- For now, the abstraction lets us think we have access to ∞ memory, numbered from 0…



Pointers store addresses

- An **address** refers to a particular memory location.
- In other words, it "points" to a memory location.
- **Pointer**: A variable that contains the address of another variable.

Addresses are commonly written in hexadecimal format, e.g., p stores 0x68 (104 in hex).

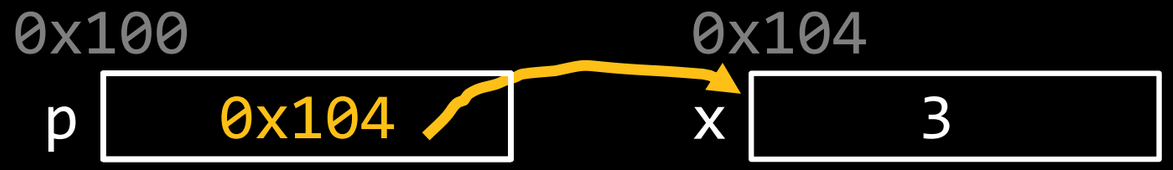


Pointer Syntax


```
1  int *p;
2  int x = 3;


3  p = &x;


4  printf("p points to %d\n",
          *p);


5  *p = 5;
```

- Declaration
- Tells compiler that **variable p is address** of an `int`

Pointer Syntax

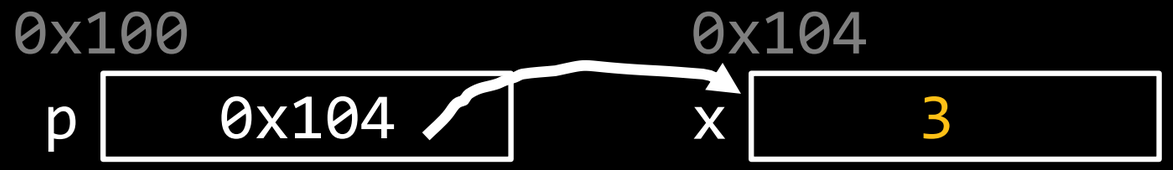



```
1  int *p;
2  int x = 3;


3  p = &x;


4  printf("p points to %d\n",
          *p);

5  *p = 5;
```

- Declaration
- Tells compiler that **variable p is address** of an `int`

- Tells compiler to assign **address of x** to p
- &: "**address operator**" in this context

Pointer Syntax



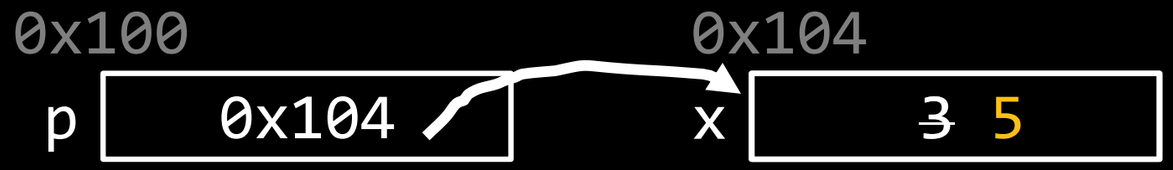

```
1  int *p;
2  int x = 3;


3  p = &x;


4  printf("p points to %d\n",
          *p);

5  *p = 5;
```

- Declaration
- Tells compiler that **variable p is address** of an `int`

- Tells compiler to assign **address of x** to p
- &: "**address operator**" in this context

- Gets value pointed to by p
- *: "**dereference operator**" in this context

Pointer Syntax




```
1  int *p;
2  int x = 3;


3  p = &x;


4  printf("p points to %d\n",
          *p);

5  *p = 5;
```

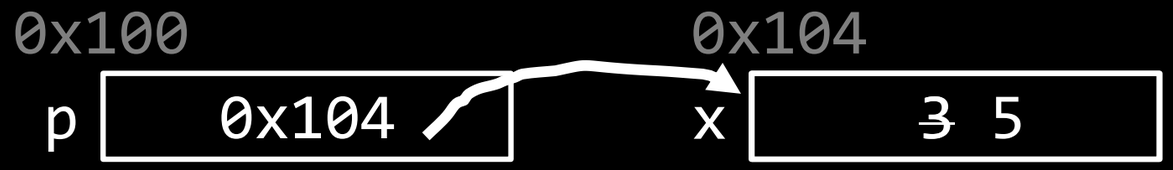
- Declaration
- Tells compiler that **variable p is address** of an `int`

- Tells compiler to assign **address of x** to p
- &: "**address operator**" in this context

- Gets value pointed to by p
- *: "**dereference operator**" in this context

- Changes value pointed to by p
- Use deref operator * on left of =

Pointer Syntax




```
1  int *p;
2  int x = 3;


3  p = &x;


4  printf("p points to %d\n",
          *p);

5  *p = 5;
```

The "*" is used in two ways:
Declaration (L1): Indicate p is a pointer
Dereference (L4,L5): Value pointed to by p

- Declaration
- Tells compiler that **variable p is address** of an `int`
- Tells compiler to assign **address of x** to p
- &: "**address operator**" in this context
- Gets value pointed to by p
- *: "**dereference operator**" in this context
- Changes value pointed to by p
- Use deref operator * on left of =

Pointers are Useful When Passing Parameters

- C passes parameters "**by value**":
  - A function parameter gets assigned a copy of the argument value.

Changing the function's copy cannot change the original.

```
void addOne (int x)
{
    x = x + 1;
}

int y = 3;
addOne(y);
```

⚠ **y is still 3…**

Pointers are Useful When Passing Parameters

- C passes parameters "by value":
  - A function parameter gets assigned a copy of the argument value.

Changing the function's copy cannot change the original.

```
void addOne (int x)
{
    x = x + 1;
}

int y = 3;
addOne(y);
```

⚠ y is still 3…

To get a function to change a value, pass in a pointer.

```
void addOne (int *p)
{
    *p = *p + 1;
}

int y = 3;
addOne(&y);
```

✅ y is now 4!

Pointers in C … The Good, Bad, and the Ugly

Why use pointers?

- To pass a large struct or array to a function, it's easier/faster/etc. to pass a pointer.
  - Otherwise, we'd need to copy a huge amount of data!
- At the time C was invented (early 1970s), compilers didn't produce efficient code, so C was designed to give human programmer more flexibility.
  - Nowadays, computers are 100,000x faster; compilers are also way, way, way better.
- Still used for low-level system code, as well as implementation of "pass-by-reference" object paradigms in other languages.
- In general, pointers allow cleaner, more compact code.

⚠ So, what are the drawbacks?

- Pointers are probably the single largest source of bugs in C. **Be careful!**
  - Most problematic with dynamic memory management
  - Dangling references and memory leaks

(more later)

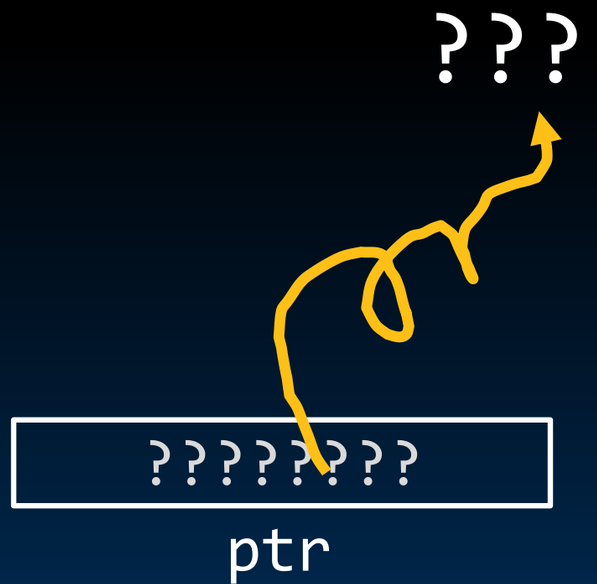
Common C Bug: Garbage Addresses

- Declaring a pointer just allocates space to hold the pointer.
  - It does not allocate something to be pointed to!

- Recall: **Local variables in C are not initialized**.
  They may contain *anything*.

- What does the following code do?

```
void f()
{
        int *ptr;
        *ptr = 5;
}
```



Agenda

Using Pointers Effectively

- More C Features
- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings
- Endianness, Word Alignment

Pointers to Different Data Types

- Pointers are used to point to a variable of a particular data type.
  - Normally a pointer can only point to one type.

```
int *xptr;
char *str;
struct llist *foo_ptr;
```

- `void *` is a type that can point to anything (generic pointer).
  - Use sparingly to help avoid program bugs… and security issues… and a lot of other bad things!
- You can even have pointers to functions…
  - `int (*fn) (void *, void *) = &foo;`
    - `fn` is a function that accepts two `void *` pointers and returns an `int` and is initially pointing to the function `foo`.
  - `(*fn)(x, y);` will then call the function

(more later)

NULL pointers…

- The pointer of all 0s is special.
  - The "NULL" pointer, like in Java, Python, etc…

```
char *p = NULL;
```

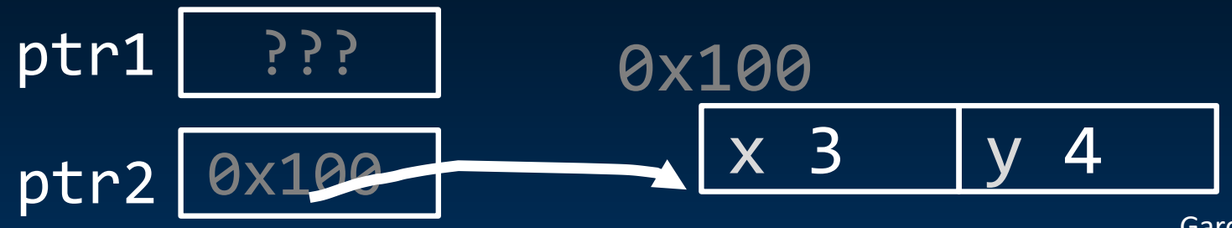
- If you write to or read from a null pointer, your program should crash.

- Since "0 is false", its very easy to do tests for null:
  - `if(!p) { /* p is a null pointer */ }`
  - `if(q) { /* q is not a null pointer */ }`

Structs, Revisited

```
typedef struct {
    int x;
    int y;
} Coord;

/* declarations */
Coord coord1, coord2;
Coord *ptr1, *ptr2;

/* instantiations
   go here… */
```

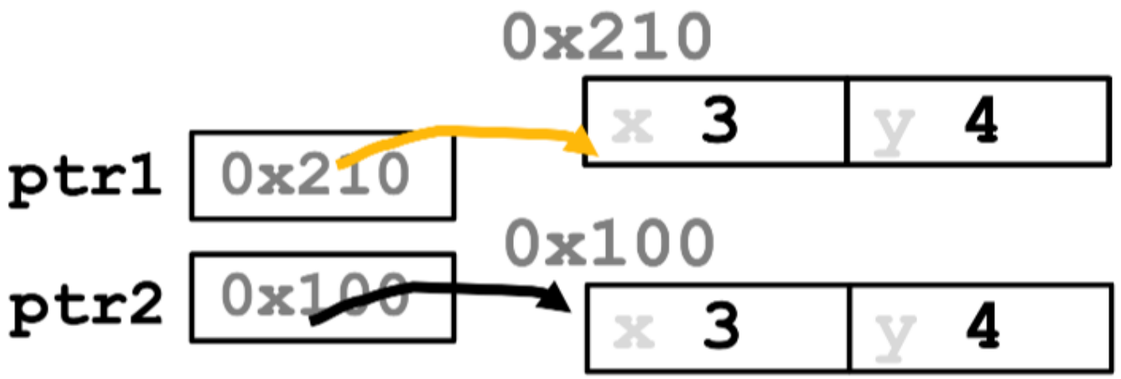
```
/* dot notation */
int h = coord1.x;
coord2.y = coord1.y;

/* arrow notation = deref + struct access*/
int k;
k = (*ptr1).x;
k = ptr1->x;  // equivalent

/* This compiles, but what does it do? */
ptr1 = ptr2;
```



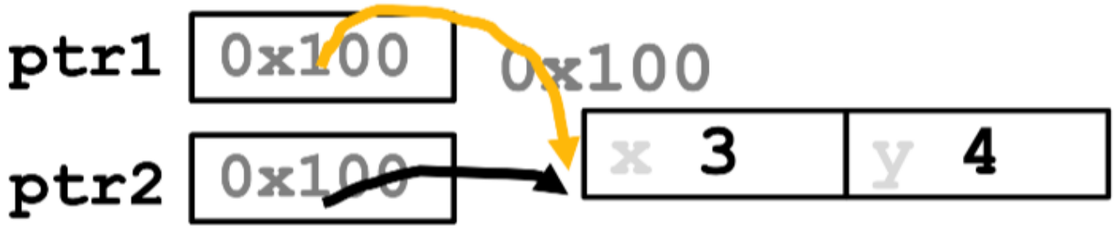
Assume ptr2 points to an initialized Coord struct {3, 4}.
What does ptr1 = ptr2; do?

0

CHOICE A
(click on this side)



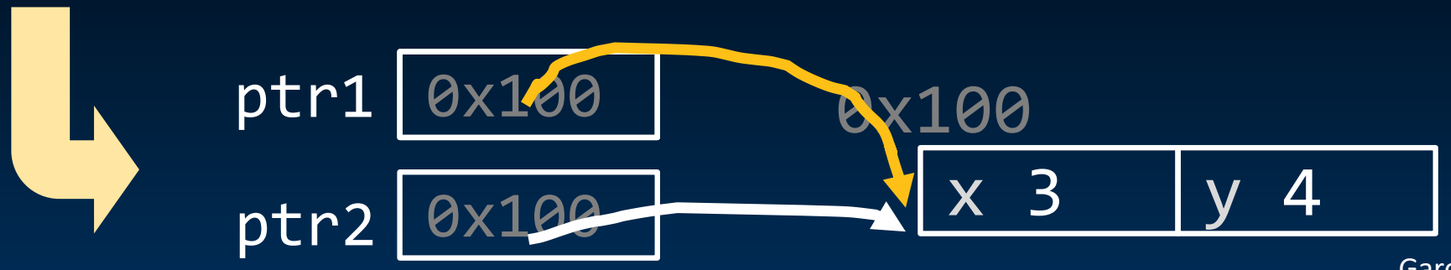
CHOICE B
(click on this side)



Pointers and Structures

```
typedef struct {
    int x;
    int y;
} Coord;

/* declarations */
Coord coord1, coord2;
Coord *ptr1, *ptr2;

/* instantiations
   go here… */
```

```
/* dot notation */
int h = coord1.x;
coord2.y = coord1.y;

/* arrow notation */
int k;
k = ptr1->x;
k = (*ptr1).x; // equivalent

/* This compiles, but what does it do? */
ptr1 = ptr2;
```



Agenda

Arrays, Pointer Arithmetic

- More C Features
- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings
- Endianness, Word Alignment

A C array is really just a big block of memory

- Declaration:
  - `int arr[2];`
  - …declares a 2-element integer array

arr

- Declaration and initialization
  - `int arr[] = {795, 635};`
  - declares and fills a 2-elt integer array

arr

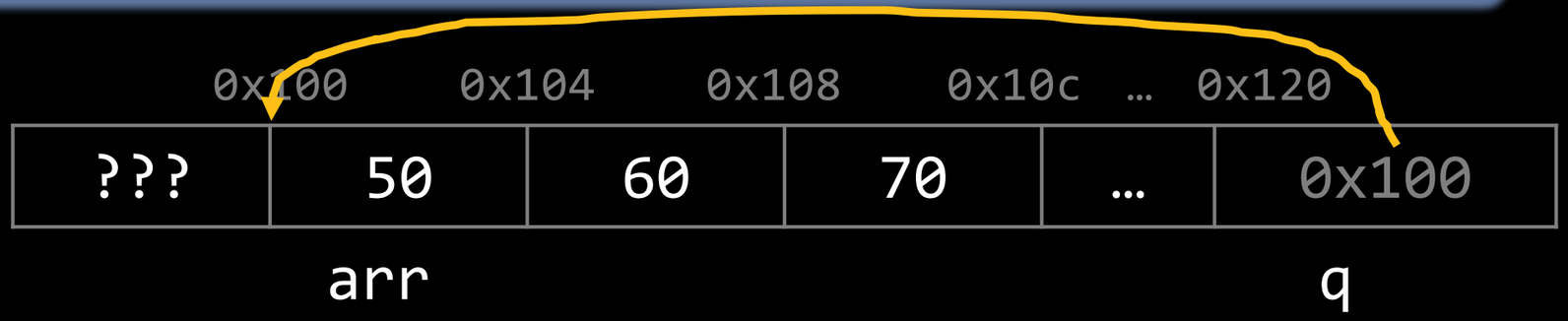
- Accessing elements:
  - `arr[num]`
  - returns the numth element.
  - This is shorthand for **pointer arithmetic**.

```
arr[0]; // 795
```

Pointer Arithmetic

Equivalent:

`a[i]` ⇔ `*(a+i)`




```
// 32-bit unsigned int array
uint32_t arr[] = {50, 60, 70};

uint32_t *q = arr;
```

pointer + n

Adds `n*sizeof(`"whatever pointer is pointing to"`)` to memory address.

pointer – n

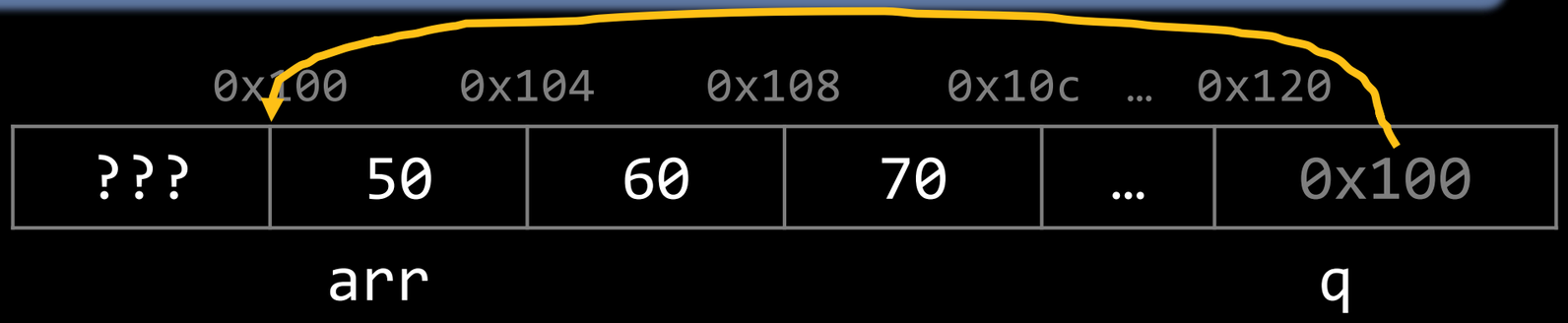
Subtracts `n*sizeof(` "whatever pointer is pointing to" `)` from memory address.

`*sizeof`: compile-time op for # of bytes in object.

Pointer Arithmetic

Equivalent:

`a[i]` ⟺ `*(a+i)`



pointer + n

Adds n*`sizeof`("whatever pointer is pointing to") to memory address.

pointer – n

Subtracts n*`sizeof`("whatever pointer is pointing to") from memory address.

*`sizeof`: compile-time op for # of bytes in object.

```
// 32-bit unsigned int array
uint32_t arr[] = {50, 60, 70};

uint32_t *q = arr;

printf("    *q: %d is %d\n", *q, q[0]);
printf("*(q+1): %d is %d\n", *(q+1), q[1]);
printf("*(q-1): %d is %d\n", *(q-1), q[-1]);
```

```
    *q: 50 is 50
*(q+1): 60 is 60
*(q-1): ??? is ???
```

Handling Pointers (1/2)

How to get a function to change a pointer?

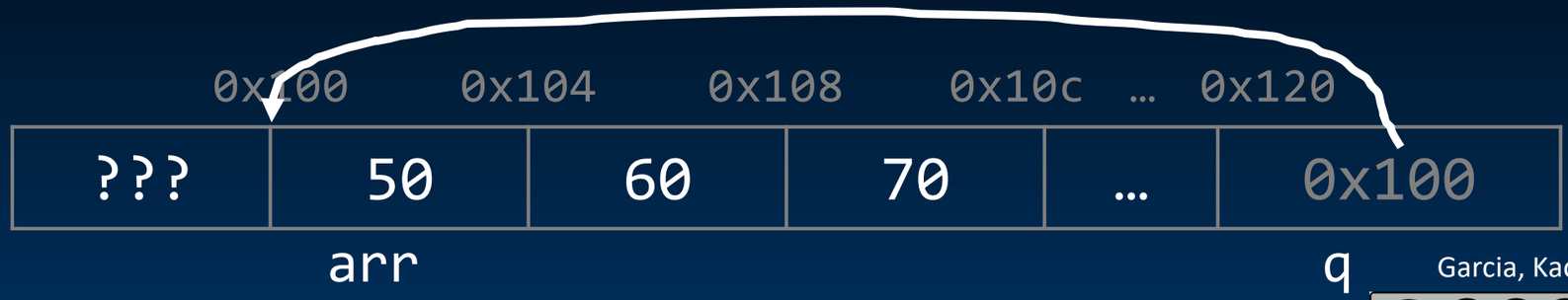
- Suppose we want `increment_ptr` to change where q points to.

Remember: C is pass-by-value!

```
1 void increment_ptr(int32_t *p)
2 {  p = p + 1;  }

3 int32_t arr[3] = {50, 60, 70};
4 int32_t *q = arr;
5 increment_ptr(q);
6 printf("*q is %d\n", *q);
```

What gets printed?

```
*q is 50
```



Handling Pointers (2/2)

How to get a function to change a pointer?

- Suppose we want `increment_ptr` to change where q points to.

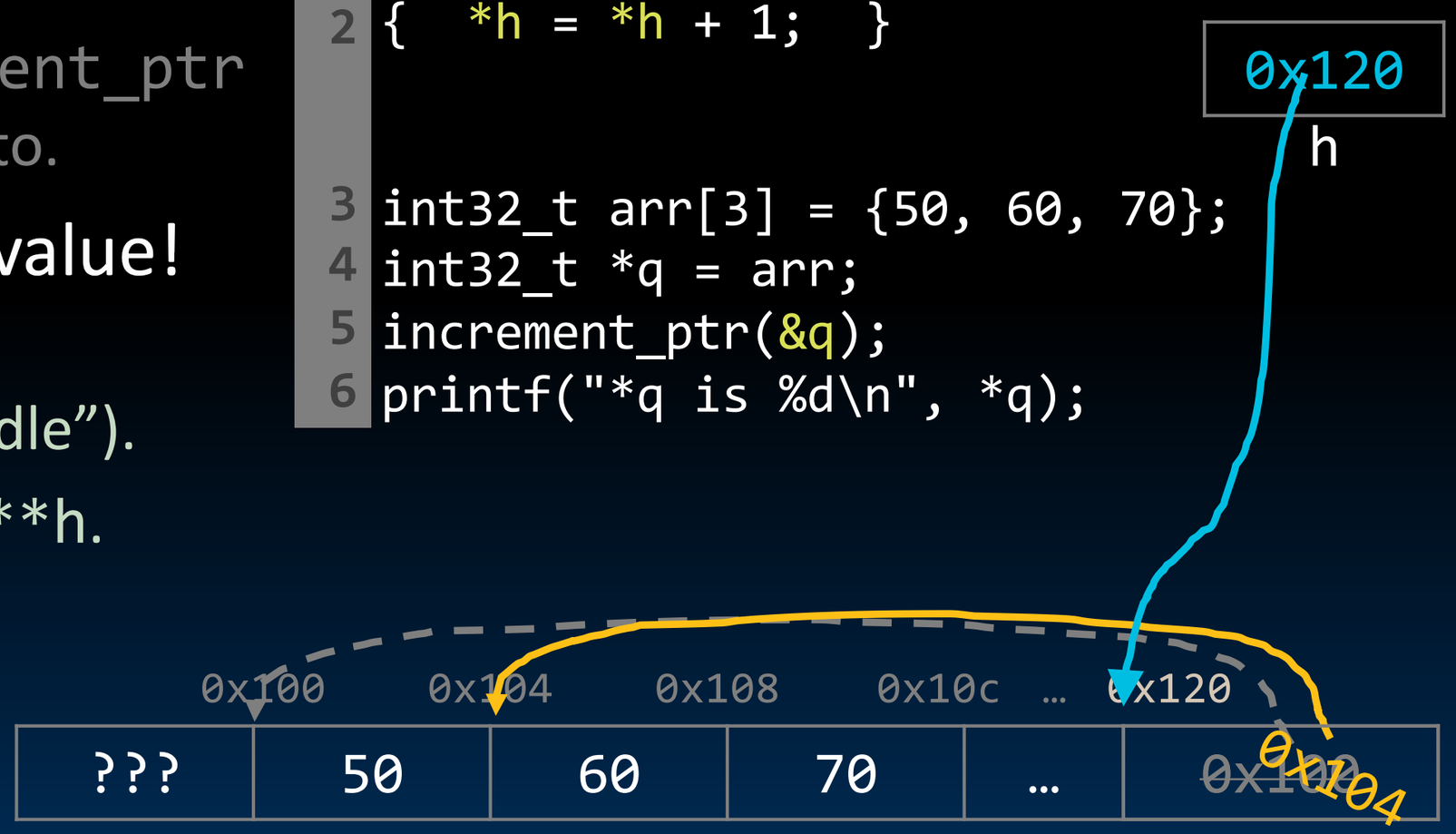
Remember: C is pass-by-value!

- Instead, pass **a pointer to a pointer** ("handle").
- Declared as `data_type **h`.

Now, what gets printed?

```
*q is 60
```

```
1 void increment_ptr(int32_t **h)
2 {  *h = *h + 1;  }

3 int32_t arr[3] = {50, 60, 70};
4 int32_t *q = arr;
5 increment_ptr(&q);
6 printf("*q is %d\n", *q);
```



Agenda

Array Pitfalls

- More C Features
- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings
- Endianness, Word Alignment

Array Pitfall #1 of 123234983

- Declare array and initialize all elements of an array of known size **n**:
  - ⚠ Wrong

```
int i, ar[10];
for(i = 0; i < 10; i++){ ... }
```

  - Strongly encouraged

```
int ARRAY_SIZE = 10;
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Why? SINGLE SOURCE OF TRUTH!
  - Utilize indirection and avoid maintaining two copies of the number 10!

Arrays vs Pointers

- Arrays are (almost) identical to pointers.
  - `char *string` and `char string[]` are nearly identical declarations
  - They differ in **very subtle** ways: incrementing, declaration of filled arrays…(more in a bit)

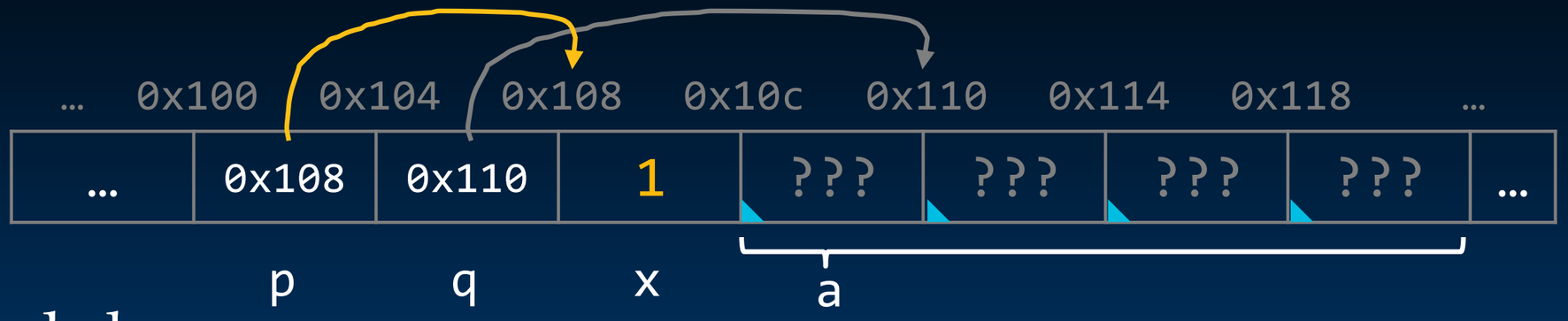
- Accessing Array Elements
  - `arr` is an array variable, but it looks like a pointer in many respects (though not all).
  - `arr[0]` is the same as `*arr`
  - `arr[2]` is the same as `*(arr+2)`

An array variable is a "pointer" to the first (0-th) element.

Arrays are not implemented as you'd think…

```
int *p, *q, x;
int a[4];
p = &x;
q = a + 1;

*p = 1;
printf("*p:%d, p:%x, &p:%x\n", *p, p, &p); // %d: signed decimal, %x: hex

*q = 2;
printf("*q:%d, q:%x, &q:%x\n", *q, q, &q);

*a = 3;
printf("*a:%d, a:%x, &a:%x\n", *a, a, &a);
```

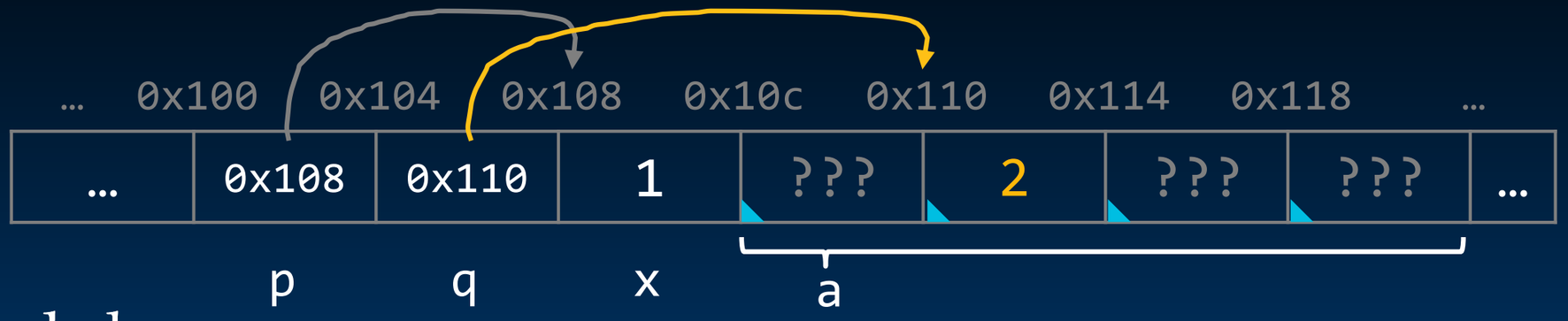
```
*p:1, p:108, &p:100
```



Arrays are not implemented as you'd think…

```
int *p, *q, x;
int a[4];
p = &x;
q = a + 1;

*p = 1;
printf("*p:%d, p:%x, &p:%x\n", *p, p, &p); // %d:signed decimal,%x:hex

*q = 2;
printf("*q:%d, q:%x, &q:%x\n", *q, q, &q);

*a = 3;
printf("*a:%d, a:%x, &a:%x\n", *a, a, &a);
```

```
*p:1, p:108, &p:100
*q:2, q:110, &q:104
```

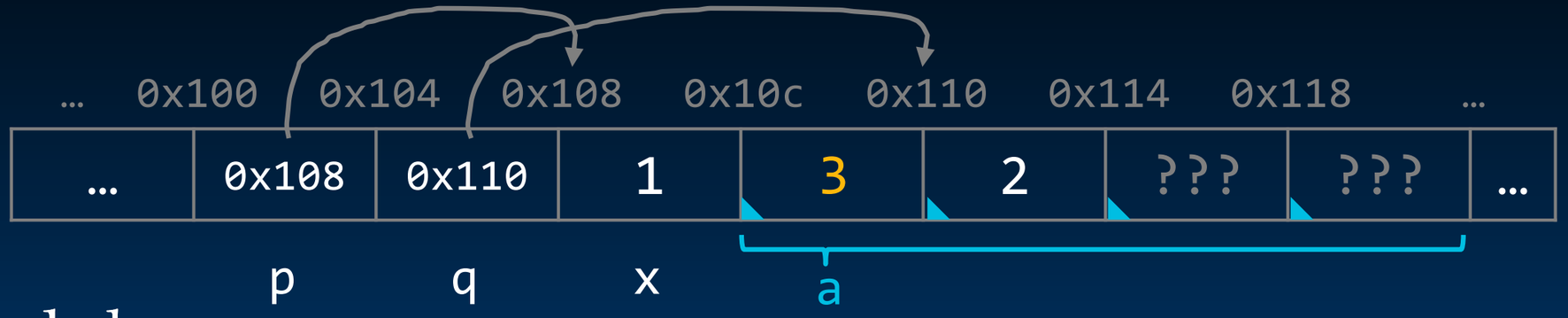


Arrays are not implemented as you'd think…

```
int *p, *q, x;
int a[4];
p = &x;
q = a + 1;

*p = 1;
printf("*p:%d, p:%x, &p:%x\n", *p, p, &p); // %d:signed decimal,%x:hex

*q = 2;
printf("*q:%d, q:%x, &q:%x\n", *q, q, &q);

*a = 3;
printf("*a:%d, a:%x, &a:%x\n", *a, a, &a);
```

```
*p:1, p:108, &p:100
*q:2, q:110, &q:104
*a:3, a:10c, &a:10c   ⚠
```

K&R: "An array name is not a variable"



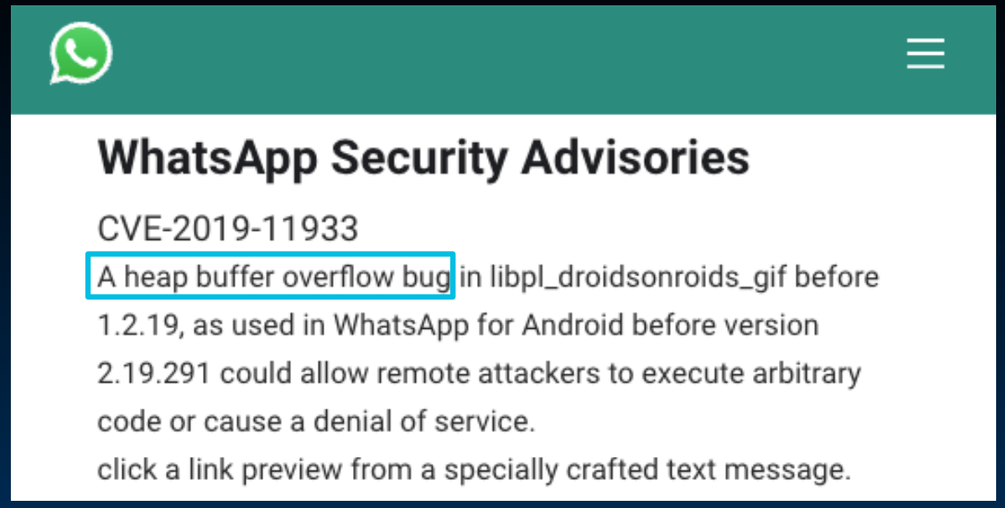
Dan: "A C array is really just a big block of memory"

Array Are Very Primitive (1/3)

1. Array bounds are not checked during element access.
   - Consequence: We can accidentally access off the end of an array!

```
int ARRAY_SIZE = 100;
int foo[ARRAY_SIZE];
int i;
....
for(i = 0; i <= ARRAY_SIZE; ++i) {
    foo[i] = 0;
}
```

   - Corrupts other parts of the program...
     - Including internal C data
   - May cause crashes later.



WhatsApp Security Advisories

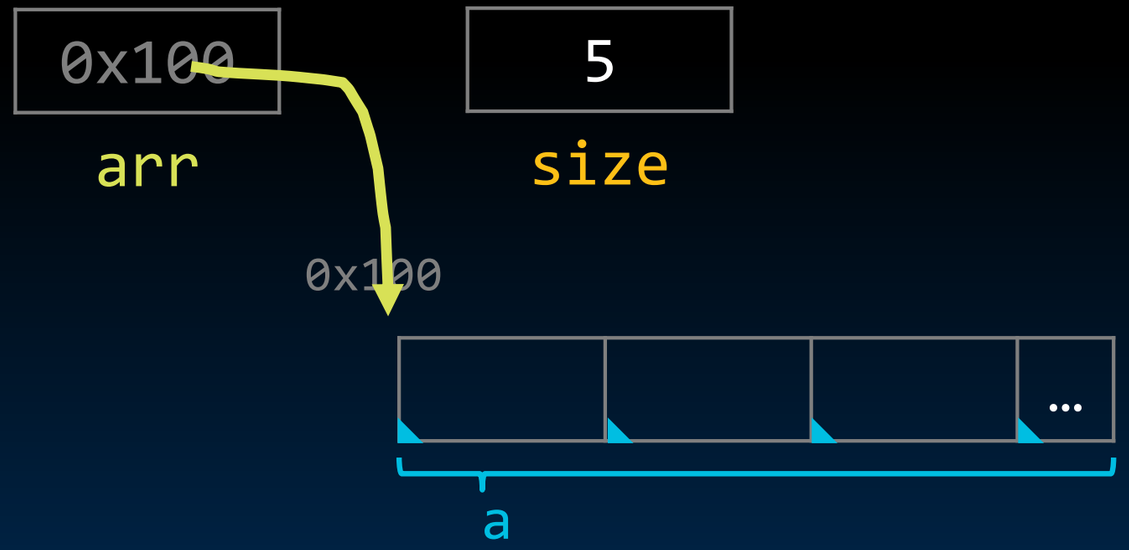
CVE-2019-11933

A heap buffer overflow bug in libpl_droidsonroids_gif before 1.2.19, as used in WhatsApp for Android before version 2.19.291 could allow remote attackers to execute arbitrary code or cause a denial of service.
click a link preview from a specially crafted text message.

Array Are Very Primitive (2/3)

1. Array bounds are not checked during element access.
   - Consequence: We can accidentally access off the end of an array!
2. An array is passed to a function as a pointer.
   - Consequence: The array size is lost! Be careful with `sizeof()`!

```
          same as: int *arr
int bar(int arr[], unsigned int size)
{
    … arr[size - 1] …
}
int main(void)
{
    int a[5], b[10];
    …
    bar(a, 5);
    …
}
```



   - You should always explicitly **include array length as a parameter**.

Array Are Very Primitive (3/3)

1. Array bounds are not checked during element access.
   - Consequence: We can accidentally access off the end of an array!
2. An array is passed to a function as a pointer.
   - Consequence: The array size is lost! Be careful with `sizeof()`!
3. Declared arrays are only allocated while the scope is valid.

```
char *foo() {
    char string[32]; ...;
    return string;
}
```
is incorrect

Solution:
Dynamic memory allocation!
(more later)

Array Are Very Primitive, Summary

1. Array bounds are not checked during element access.
   - Consequence: We can accidentally access off the end of an array!
2. An array is passed to a function as a pointer.
   - Consequence: The array size is lost! Be careful with `sizeof()`!
3. Declared arrays are only allocated while the scope is valid.

Segmentation faults and bus errors:

- These are VERY difficult to find; be careful!
- You'll learn how to debug these in Lab 02 with **gdb**…

Agenda

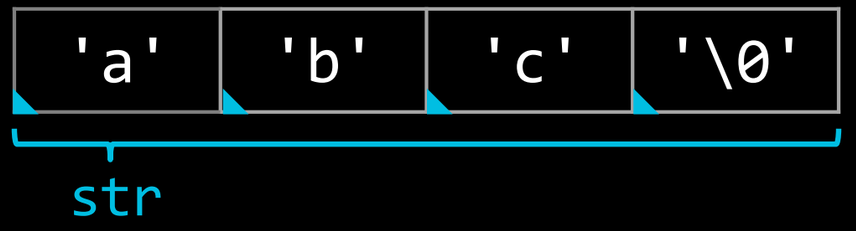
Strings

- More C Features
- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings
- Endianness, Word Alignment

C Strings

- A C string is just an array of characters, followed by **a null terminator**.
  - Null terminator: the byte 0 (number) aka `'\0'` character)

```
char str[] = "abc";
```



- The standard C library `string.h` assumes null-terminated strings.
  - String operations do **not** include the null terminator, e.g., when you ask for length of a string!

```
… strlen(str) … // 3
```

⚠

```
// possible implementation*
int strlen(char s[])
{
  int n = 0;
  while (*(s++) != 0) { n++; }
  return n;
}
```

*for actual `strlen()` implementation, see glibc

Lab 01: string.h: strcpy() vs strncpy()

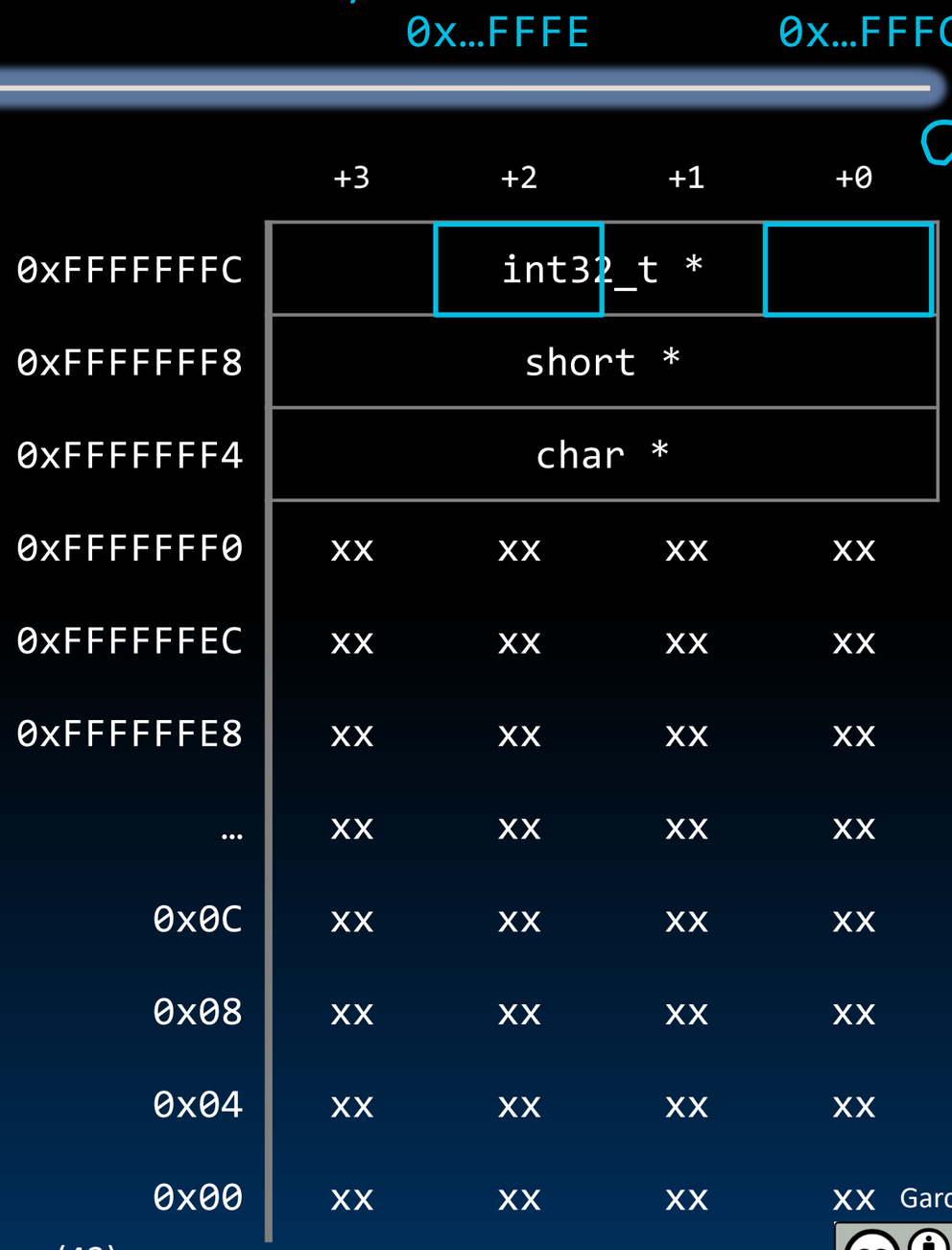
Agenda

Endianness, Word Alignment

- More C Features
- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Endianness, Word Alignment

Memory and Addresses

- Modern machines are "byte-addressable."
  - Hardware's memory composed of 8-bit storage cells; each byte has a unique address
- We commonly think in terms of "word size":
  - aka number of bits in an address
  - A **32b** architecture has **4-byte words**.
  - All pointer sizes on 32b architecture:
    `sizeof(int *) == … == sizeof(char *) == 4`



Endianness

- The hive machines are "little endian".
  - The **least significant byte** of a value is stored first.
- (Contrast with "big endian")
  - (Most significant byte is stored first)

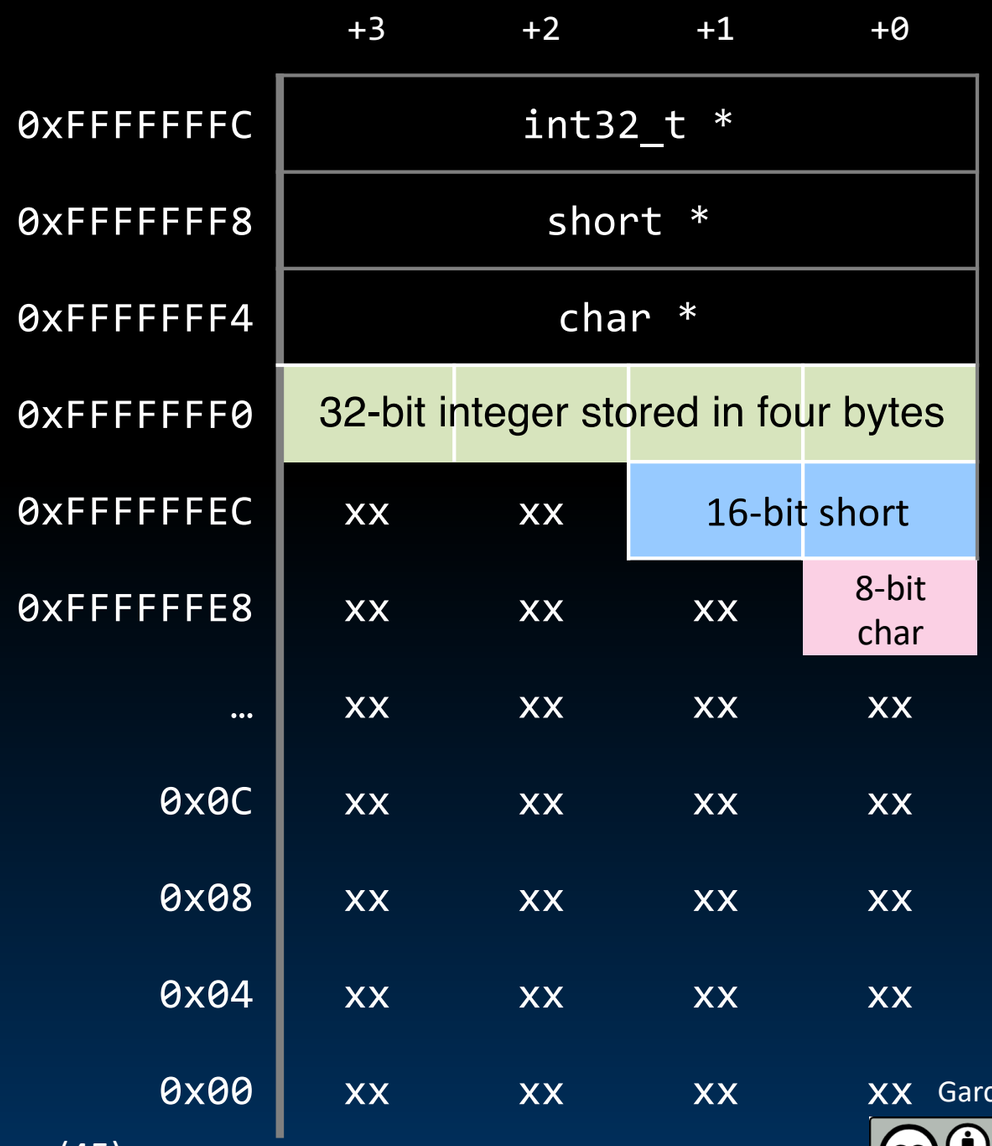
`int32_t* pointer`
`0xFFFFFFF0`

`int32_t value`
`0x12345678`

Word Alignment

Read for HW01

- We often want "word alignment":
  - Some processors will not allow you to address **32b** values without being on **4-byte** boundaries.
  - Others will just be very slow if you try to access "unaligned" memory.



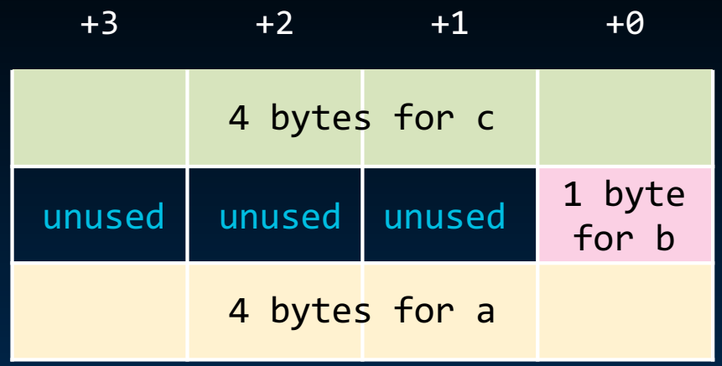
Structures, Revisited, Again

Read for HW01

- A “struct” is really an instruction to C on how to arrange a bunch of bytes in a bucket.
  - Structs provide enough space for the data.
  - C compilers often **align** the data with **padding**.

```
struct foo {
    int32_t a;
    char b;
    struct foo *c;
}
```

  - For this struct, the actual layout on a 32b architecture would be as follows.
    - Note the 3 bytes of padding
    - `sizeof(struct foo) == 12`



More: StackOverflow post

And in Conclusion…

- C pointers and arrays are ***pretty* much the same**, except with function calls.
- C knows how to **increment pointers**.
- C is an efficient language, but with little protection.
  - Array bounds **not checked**
  - Variables **not automatically initialized**
- Use handles to change pointers
- (Beware) The cost of efficiency is more overhead for the programmer.
  - "C gives you a lot of extra rope, don't hang yourself with it!"