CS61C

Great Ideas
in
Computer Architecture
(a.k.a. Machine Structures)

Teaching Professor
Dan Garcia

C Pointers, Arrays, and Strings

Agenda

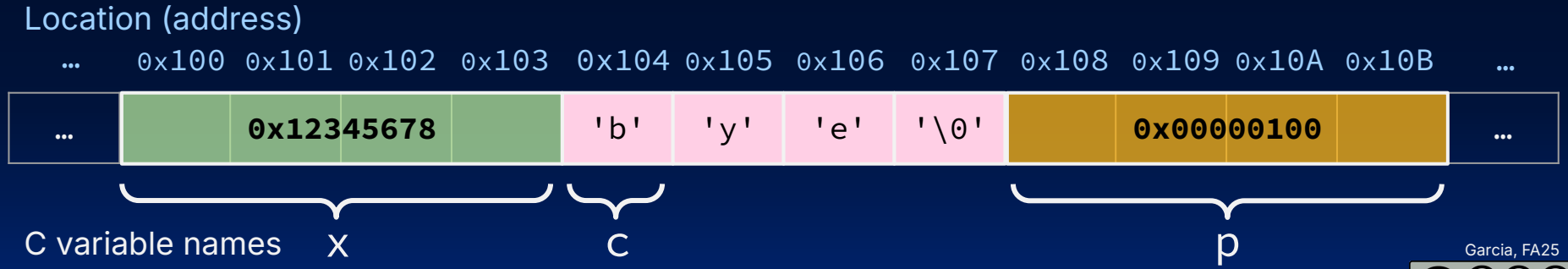
Pointers

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

Memory is a Single Huge Array

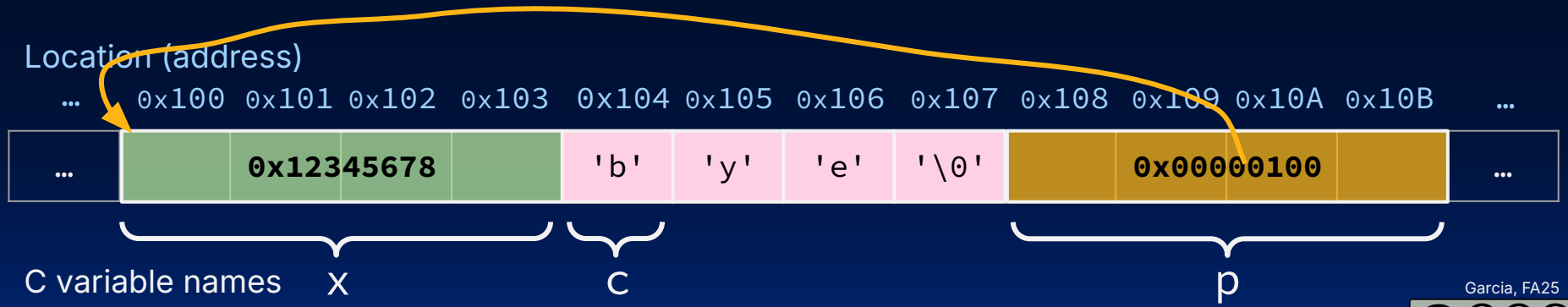
- Consider memory to be a byte-addressed array.
  - Each cell of the array has an **address** associated with it.
  - Each cell also stores some **value**.
  - For now, with this abstraction, lets us think we have access to ∞ memory, numbered from 0…

Addresses are commonly written in hexadecimal format.

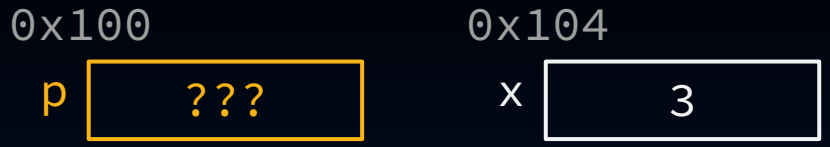


Pointers store addresses

- 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**.
  - For now, with this abstraction, lets us think we have access to ∞ memory, numbered from 0...
- **Pointer**: A variable that contains the address of another variable.
  - In other words, it "**points**" to a memory location.



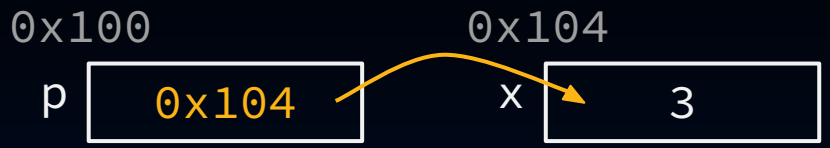
Pointer Syntax




```
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

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

```
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

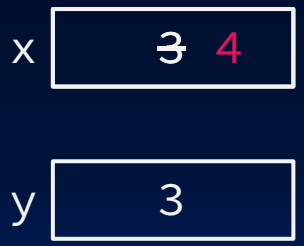
- Changes **value pointed to** by p
- Use deref operator * on left of =

Pointers are Useful When Passing Parameters

C is **pass-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);
```



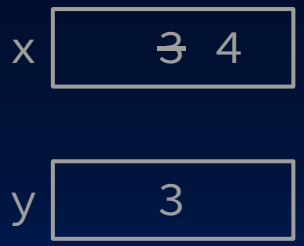
⚠ y is still 3…

Pointers are Useful When Passing Parameters

C is **pass-by-value**: A function parameter gets assigned **a copy of the argument value**.

Changing the function's copy cannot change the original.

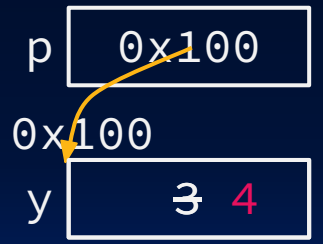
```
void addOne (int x)
{
    x = x + 1;
}

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, the Bad, and the Ugly

- 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.

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

⚠ So, what are the drawbacks?

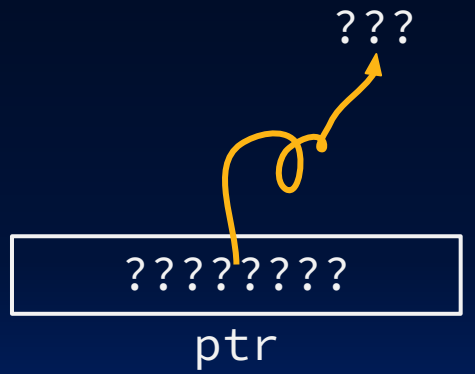
- Pointers are probably the single largest source of bugs in C. Be careful!
  - Most problematic with dynamic memory management
  - Dangling references and memory leaks

} (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

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

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;
```

(more later, but)

- **`void *`** is a type that can point to anything (**generic pointer**).
  - Use sparingly to help avoid program bugs… and security issues…and more…
- 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

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 */ }`

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; }`

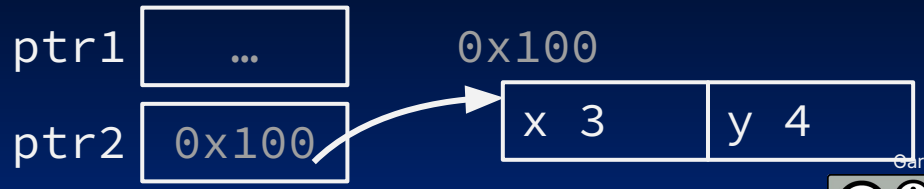
Dot notation: `x.y = value`

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

Struct Pointers

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

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

/* instantiations
   go here… */
…
```

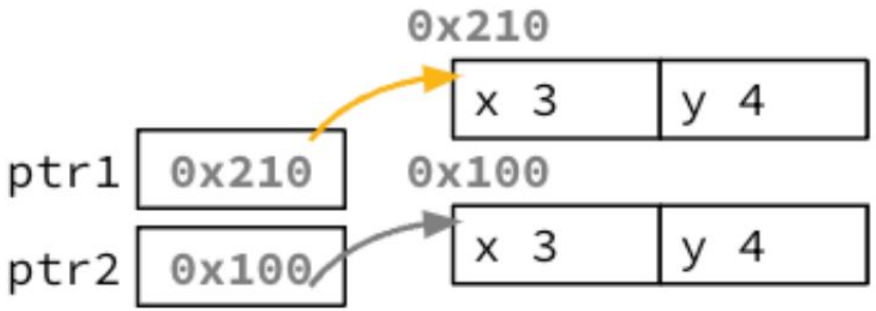
```
/* dot notation */
int h = coord1.x;
coord2.y = coord1.y;

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

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



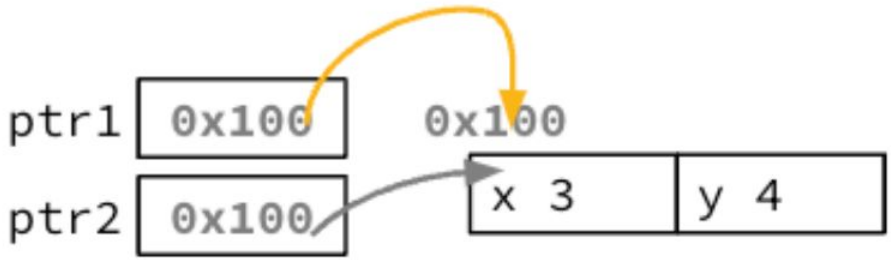
Assume ptr2 points to an initialized Coord struct {3, 4}. What does ptr1 = ptr2; do?

0

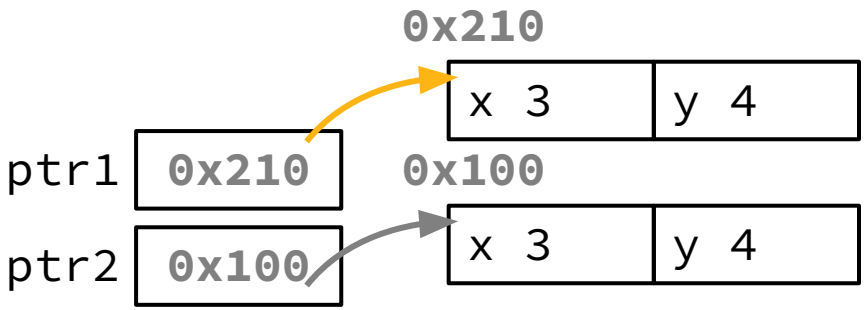
CHOICE A
(click on this side)



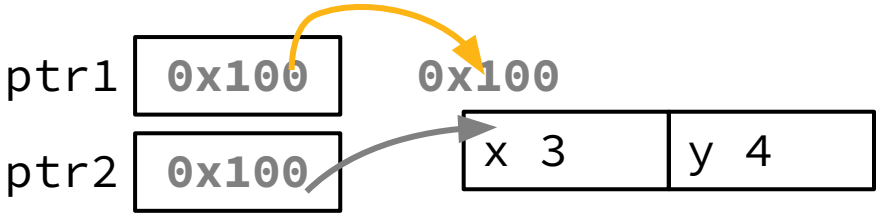
CHOICE B
(click on this side)



CHOICE A
(click on this side)



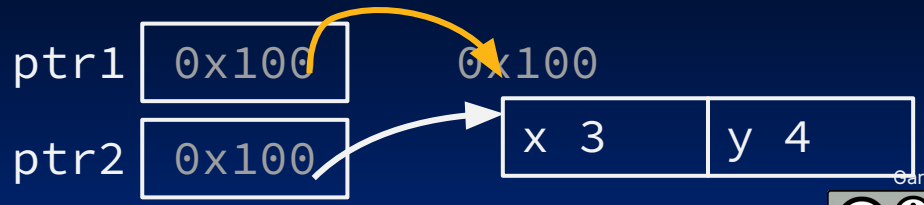
CHOICE B
(click on this side)



Struct Pointers

```
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;
```



Agenda

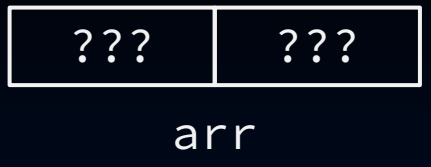
Arrays, Pointer Arithmetic

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

A C array is really just a big block of memory

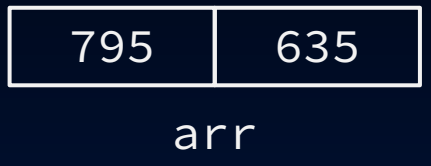
Declaration:

- `int arr[2];`
- ...declares a 2-element integer array



Declaration and initialization

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



Accessing elements:

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

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

Pointer Arithmetic

Equivalent:

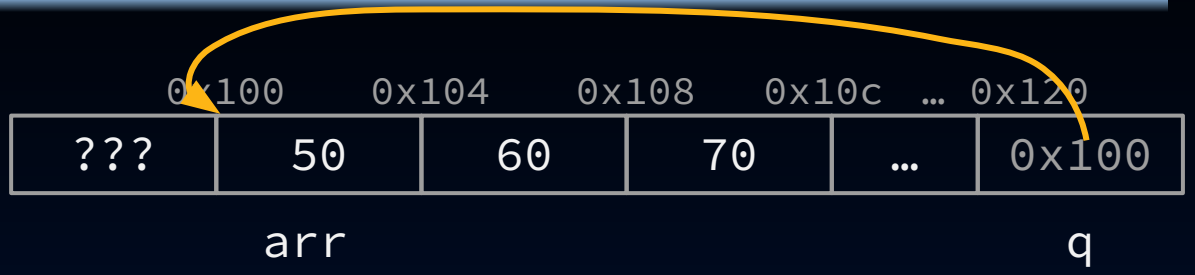
`a[i]` ⟺ `*(a+i)`

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.




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

uint32_t *q = arr;
```

Pointer Arithmetic

Equivalent:

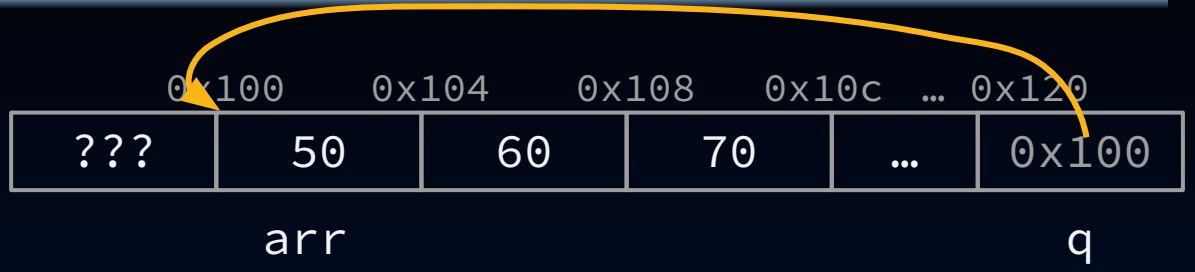
`a[i]` ⟺ `*(a+i)`

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.




```
// 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.

```
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);
```

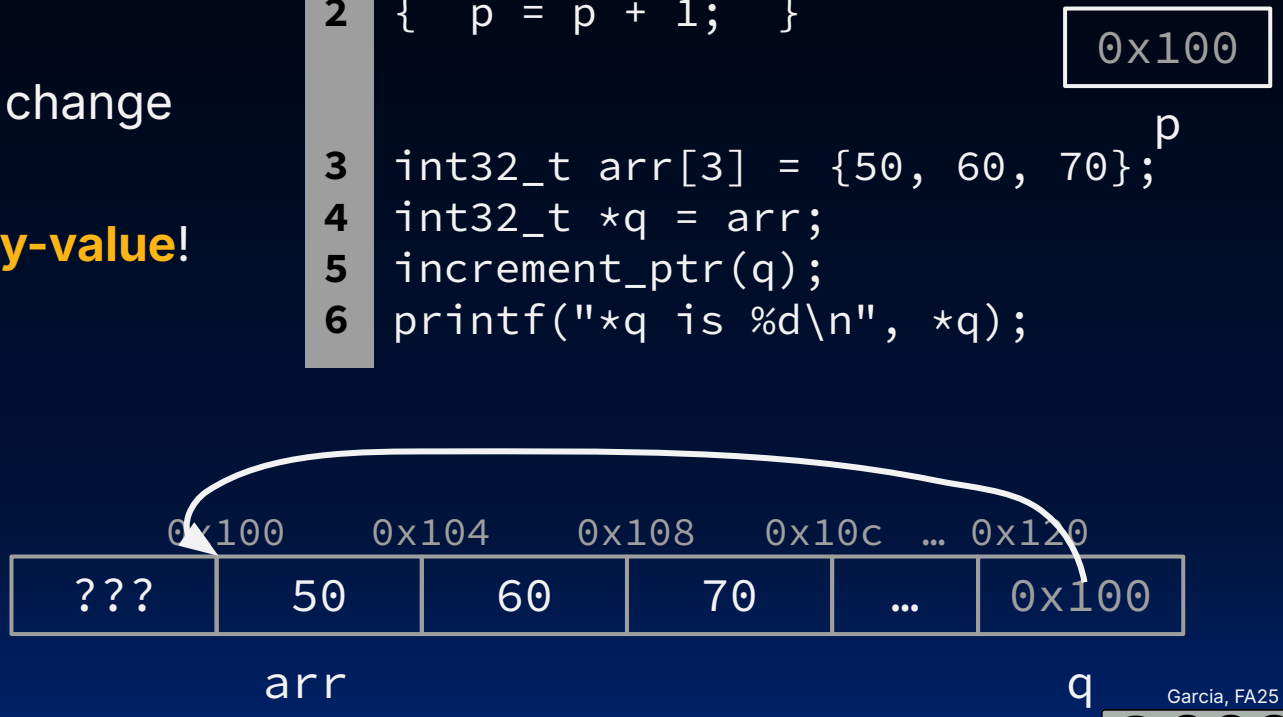
Handling Pointers (1/2)

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);
```



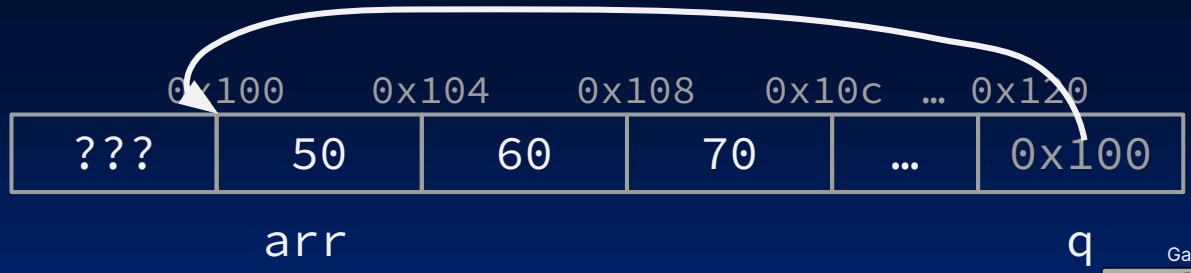
Handling Pointers (1/2)

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);
```



Print output:

```
*q is 50
```

Handling Pointers (2/2)

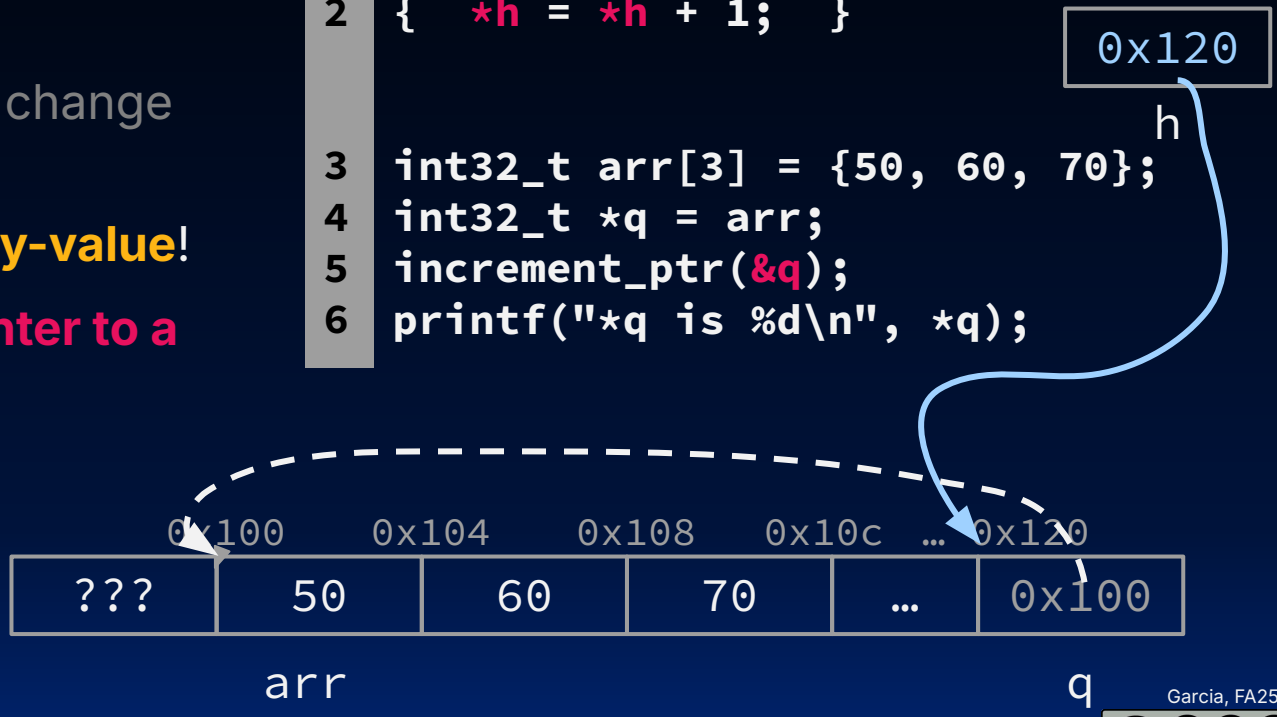
How to get a function to change a pointer?

- 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`.

```
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);
```



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`.

```
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);
```

Print output:

```
*q is 60
```

Agenda

Array Pitfalls

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

Array Pitfall #1 of 123234983

Declare array and initialize all elements of an array of known size n:

- ⚠

```
int i, arr[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.

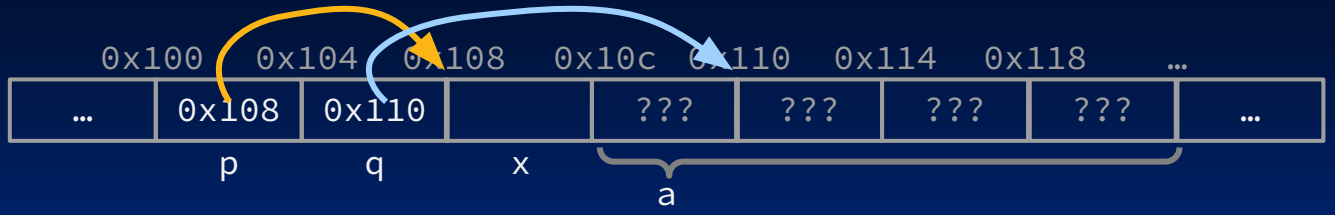
- `char *string` and `char string[]` are nearly identical declarations
- They differ in **very subtle** ways: incrementing, declaration of filled arrays…(more in a bit)

Accessing Array Elements

- `arr` is an array, 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)`

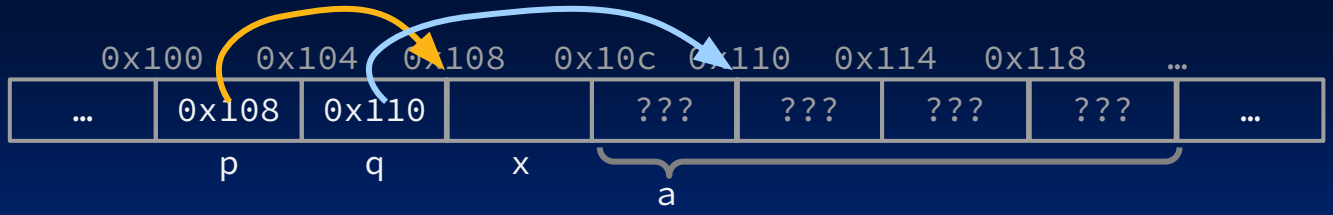
Arrays are not implemented as you'd think...

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



Arrays are not implemented as you'd think…

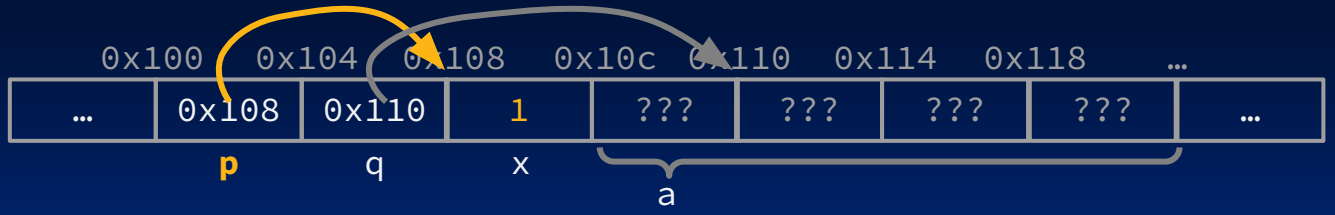
```
int *p, *q, x;
int a[4];
p = &x;
q = a + 1;

*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);
```



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);
```

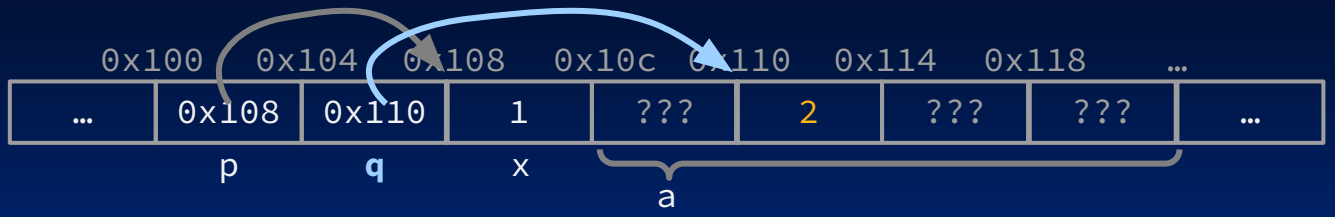
```
*p:1, p:108, &p:100
```



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
```

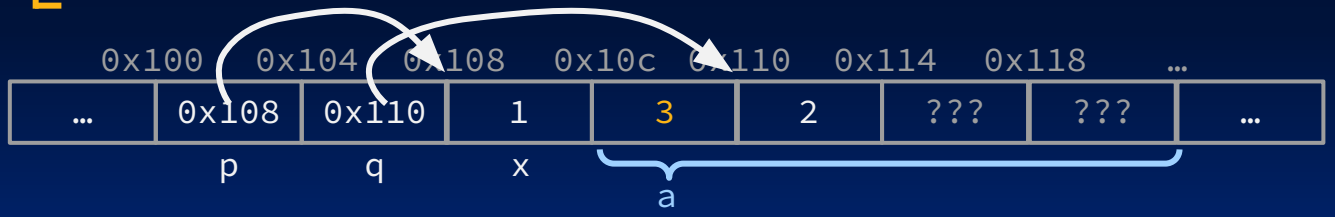


Arrays are not implemented as you'd think…

```
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"



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

Agenda

Strings

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

C Strings

[READ ON YOUR OWN]

A C string is just an array of characters, followed by **a null terminator**.

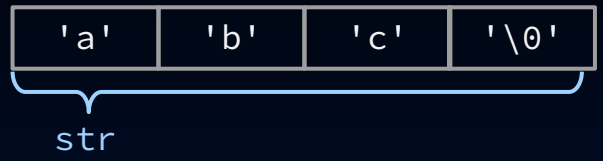
- Null terminator: the byte 0 (number) aka `'\0'` character)

The standard C library `string.h` assumes **null-terminated strings.**

- String operations often abstract away null terminators when returning values!

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

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




```
… strlen(str) … // 3 ⚠️

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

*for actual strlen() implementation, see glibc

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
- Strings are arrays of characters with a null terminator
  - The length is the # of characters, but memory needs 1 more for \0
- (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!"

Agenda

[THESE WERE MOVED TO THE END <BEGIN>]

More C Features

Full slides for reference in L03.

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

From Last Time

[from last time]

```
1  #include <stdio.h>
2  int main(int argc, char *argv[]) {
3      int x = 0;
4      int y;
5
6      printf("before: x=%d, y=%d\n",
               x, y);
7      x++;
8      y += x; // y = y + x
9      printf(" after: x=%d, y=%d\n",
               x, y);
10     return 0;
11 }
```

%d Placeholder for argument, where
d: format value as decimal numeral

Print output:

```
before: x=0, y=22621
 after: x=1, y=22622
```

Declaring a C Variable Does Not Also Initialize It

- Variables are **not automatically initialized to default values**!

- If a variable is not initialized in its declaration, **it holds garbage**!
  - The contents are undefined…
  - …but C still lets you use uninitialized variables!
- ⚠ **Danger** ⚠: Bugs sometimes may only manifest after you've built other parts of your program.

```
int x; // declaration
…
x = 42; // initialization
```


```
// OK
int y = 38;
```

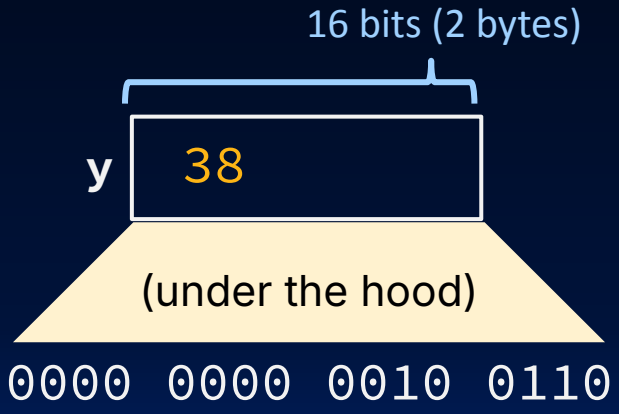
Why are variables typed in C? [from last time]

- C variables are typed.
  - Types of ***variables*** can't change, e.g., y cannot store a float.
  - However, you can **typecast *values*** (more later)

```
uint16_t y = 38;
```

(unsigned 16-bit integer, see `stdint.h`)

- A variable's type helps the compiler determine **how to translate the program to machine code** designed for the computer's architecture:
  - **How many bytes** the variable takes up in memory, and
  - **What operators** the variable supports, etc.



The C `bool`: `true` or `false`? [from last time]

- Originally, the boolean was not a built-in 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)

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

- Nowadays:
  - `true` and `false` provided by `stdbool.h`.
  - Built-in type as of C23 (but we are using C17)

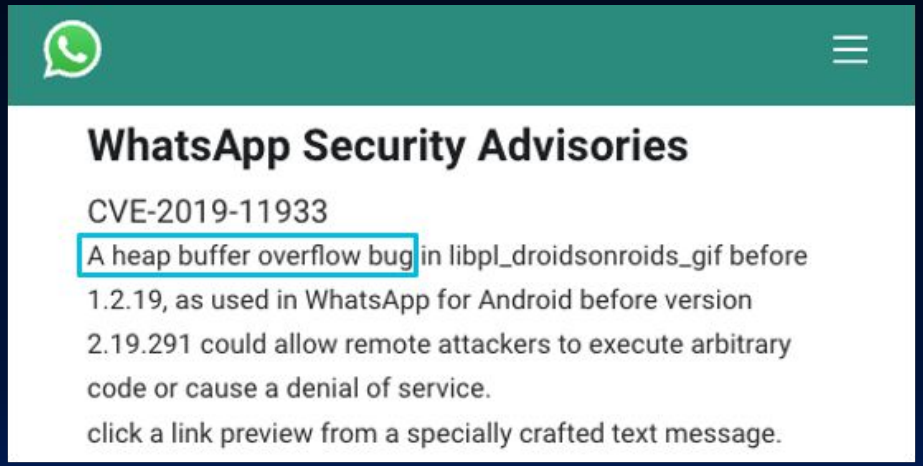
[THESE WERE MOVED TO THE END <END>]

Arrays Are Very Primitive (1/3)

[GOT TO HERE]

1. Array bounds are not checked during element access.
   - Consequence: We can accidentally access off the end of an array
   - Corrupts other parts of program…
     - Including internal C data
   - May cause crashes later.

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



"Buffer overflow"

Arrays Are Very Primitive (2/3)

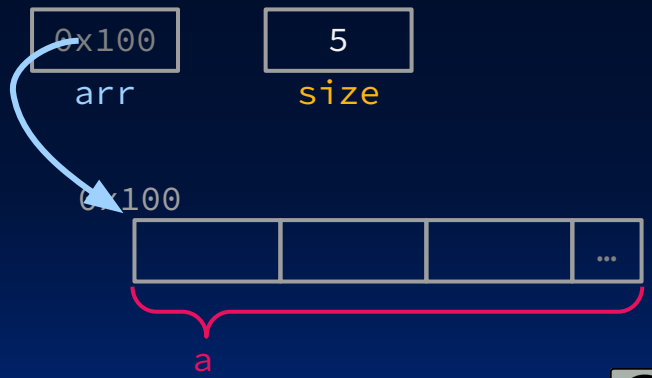
[for next time]

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.



Arrays Are Very Primitive (3/3)

[for next time]

3. Declared arrays are only allocated while the scope is valid.

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

Solution:
Dynamic memory allocation!

(more later)

Arrays Are Very Primitive, Summary [for next time]

1. Array bounds are not checked during element access.
2. An array is passed to a function as a pointer.
3. Declared arrays are only allocated while the scope is valid.

Consequences? Segmentation faults, bus errors, …

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

Agenda

[READ ON YOUR OWN]

Strings

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

Agenda

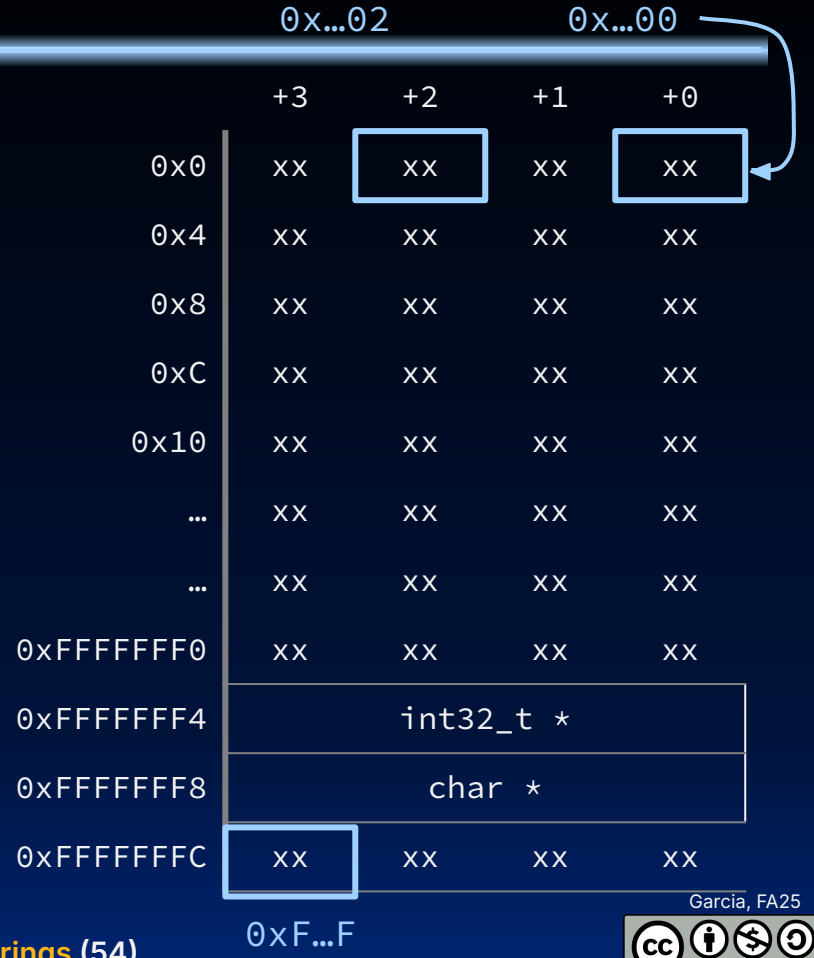
[FOR F 1/31]

Memory, Words, and Endianness

- Pointers
- Using Pointers Effectively
- Arrays, Pointer Arithmetic
- Array Pitfalls
- Strings

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:
  - 4B words
  - word-sized pointers = 4B pointers
  - `sizeof(int *) == … == sizeof(char *) == 4`



Example program memory, on hive

see: endianness.c

```
#include <stdio.h>
#include <stdint.h>

int main(int argc, char *argv[]) {
  int32_t value = 0x12345678;
  char str[] = "bye";
  char str2[] = "hello";
  int16_t short_val = 0xaabb;
  …

  return 0;
}
```

	+3	+2	+1	+0
0x0	xx	xx	xx	xx
0x4	xx	xx	xx	xx
…	xx	xx	xx	xx
0x7F…FE164	0xaa	0xbb	xx	xx
0x7F…FE168	0x12	0x34	0x56	0x78
0x7F…FE16C	'y'	'b'	xx	xx
0x7F…FE170	'e'	'h'	'\0'	'e'
0x7F…FE174	'\0'	'o'	'l'	'l'
0x7F…FE178	xx	xx	xx	xx
…	xx	xx	xx	xx
0xFFFFFFFC	xx	xx	xx	xx

Word Alignment

- We often want **word alignment**:
  - Some 32-bit 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

- Based on this diagram, does the hive machine processor do word-alignment?

	+3	+2	+1	+0
0x0	xx	xx	xx	xx
0x4	xx	xx	xx	xx
…	xx	xx	xx	xx
0x7F…FE164	0xaa	0xbb	xx	xx
0x7F…FE168	0x12	0x34	0x56	0x78
0x7F…FE16C	'y'	'b'	xx	xx
0x7F…FE170	'e'	'h'	'\0'	'e'
0x7F…FE174	'\0'	'o'	'l'	'l'
0x7F…FE178	xx	xx	xx	xx
…	xx	xx	xx	xx
0xFFFFFFFC	xx	xx	xx	xx

Endianness

- The hive machines are **little endian**.
  - Words are stored **least significant byte (LSB) first**.
  - (contrast: big endian, most significant byte (MSB) stored first)
  - LSB of integers have lowest byte address

```
int32_t value = 0x12345678;
```

  - Diagram makes it easy to read integers, harder to read strings:

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

'b'	'y'	'e'	'\0'

	+3	+2	+1	+0
0x0	xx	xx	xx	xx
0x4	xx	xx	xx	xx
…	xx	xx	xx	xx
0x7F…FE164	0xaa	0xbb	xx	xx
0x7F…FE168	0x12	0x34	0x56	0x78
0x7F…FE16C	'y'	'b'	xx	xx
0x7F…FE170	'e'	'h'	'\0'	'e'
0x7F…FE174	'\0'	'o'	'l'	'l'
0x7F…FE178	xx	xx	xx	xx
…	xx	xx	xx	xx
0xFFFFFFFC	xx	xx	xx	xx

For you to try at home with gdb

[at home]
endianness.c

```
$ gdb endianness
(gdb) b 12 # set breakpoint at line 12
(gdb) r      # run, initializing all vars
(gdb) p/x str  # see string bytes
(gdb) p/x str2
(gdb) # print one word, show in hex
(gdb) x/1wx 0x7fffffffe164
(gdb) <enter> # repeat last action
(gdb) … # keep pressing <enter>
(gdb) # LSB in lowest address/word
(gdb) # print 4 bytes, show in hex
(gdb) x/4bx 0x7fffffffe164
(gdb) <enter> # repeat last action
(gdb) … # keep pressing <enter>
(gdb) q
```

	+3	+2	+1	+0
0x0	xx	xx	xx	xx
0x4	xx	xx	xx	xx
…	xx	xx	xx	xx
0x7F…FE164	0xaa	0xbb	xx	xx
0x7F…FE168	0x12	0x34	0x56	0x78
0x7F…FE16C	'y'	'b'	xx	xx
0x7F…FE170	'e'	'h'	'\0'	'e'
0x7F…FE174	'\0'	'o'	'l'	'l'
0x7F…FE178	xx	xx	xx	xx
…	xx	xx	xx	xx
0xFFFFFFFC	xx	xx	xx	xx

Structures, Revisited, Again

Read for HW

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`