CS 61C

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Dan Garcia

UC Berkeley
Teaching Professor
Lisa Yan

C Pointers, Arrays, and Strings

指针 数组 字符串

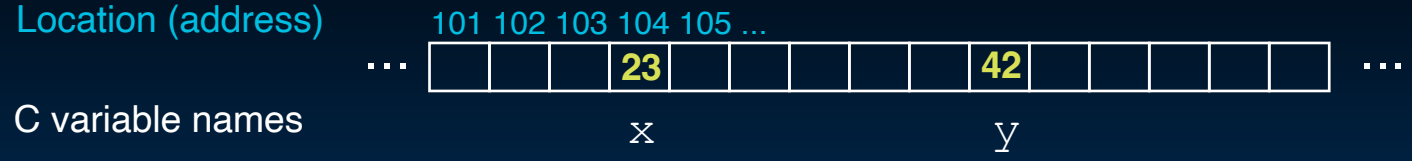
Pointers

- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Word Alignment
- Function Pointer Example (recorded)

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…**

address
value

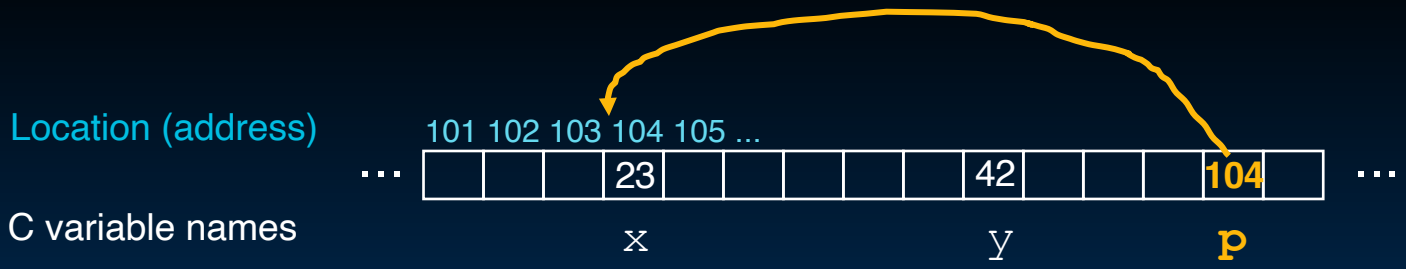


address help us locate the value

Memory Is a Single Huge Array

- 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.

pointer stores address of a variable

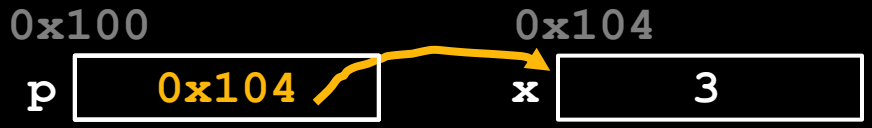


Pointer Syntax


```
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;
```

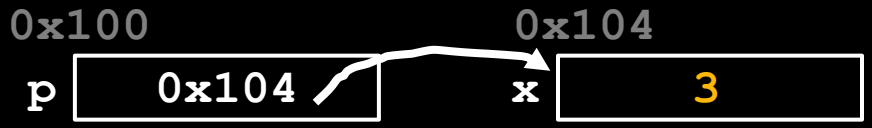
- Declaration
- Tells compiler that variable `p` is address of an `int`

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

assign the address of x to p

& means to acquire the address of a variable

Pointer Syntax

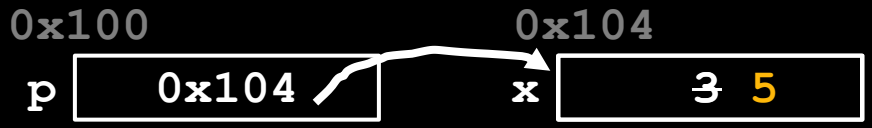



```
1  int *p;
2  int x = 3;


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

* mean acquire the value according to its address

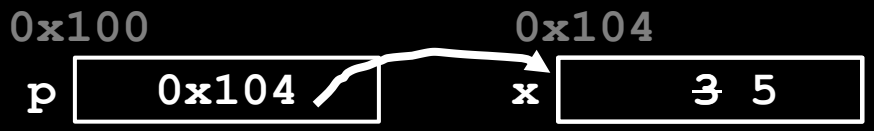
Pointer Syntax




```
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 variable pointed to by `p`
- Use dereference operator `*` on left of `=`

Pointer Syntax



The "*" is used two ways here:
Declaration (L1): Indicate p is a pointer
Dereference (L4-5): 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 variable pointed to by `p`
- Use dereference operator `*` on left of =

Pointers are Useful When Passing Parameters

- **Java and C pass parameters "by value":** 值传递
  - A procedure/function/method gets a copy of the parameter.

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...

can't alter the origin functions will create a new local variable and the value is copied from parameter.

Pointers are Useful When Passing Parameters

- **Java and C pass parameters "by value":**
  - A procedure/function/method gets a copy of the parameter.

Changing the function's copy cannot change the original.

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

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

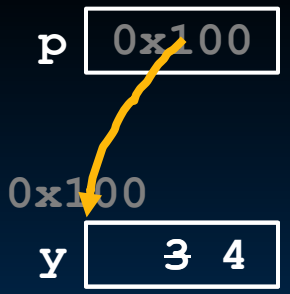
x: 3 4

y: 3

⚠ 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? pass a large struct or array to function

- 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 next time)

Common C Bug: Garbage Addresses

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

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

- What does the following code do?

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

Using Pointers Effectively

- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Word Alignment
- Function Pointer Example (recorded)

Pointers to Different Data Types

- Pointers are used to point to any data type.
- Normally a pointer can only point to one type.
  - `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!

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

void *

- 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

Check out Dan's recorded function pointer example:
https://www.youtube.com/watch?v=PlIYbp0fgY4

NULL pointers...

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

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

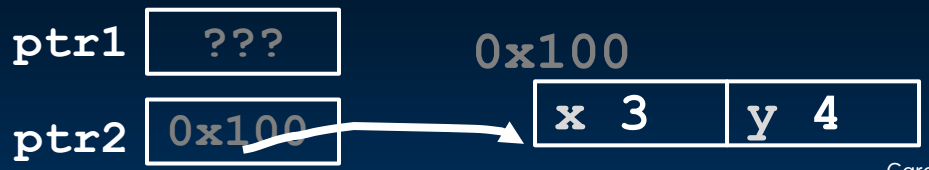
- **Since "0 is false", its very easy to do tests for null:**
  - `if(!p) { /* p is a null pointer */ }`
  - `if(q) { /* q is not a null pointer */ }`

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

Structs, Revisited

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

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

/* instantiations
   go here… */
```

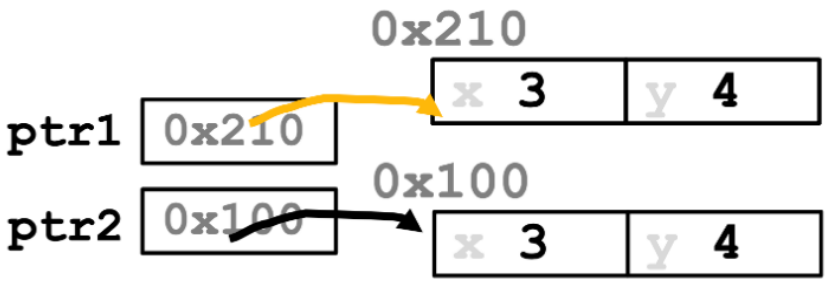
```
/* dot notation */
int h = coord1.x;
coord2.y = coord1.y;

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

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



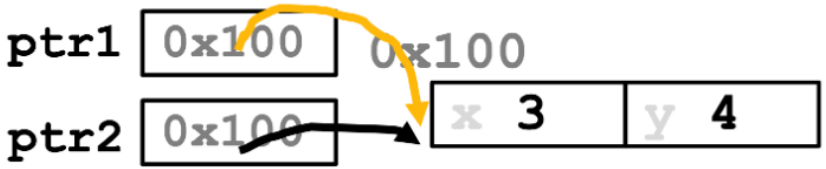
Assume ptr2 points to an initialized Coord struct {3, 4}.

What does ptr1 = ptr2; do?

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;
```

Pointer Arithmetic

- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Word Alignment
- Function Pointer Example (recorded)

A C array is really just a big block of memory

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

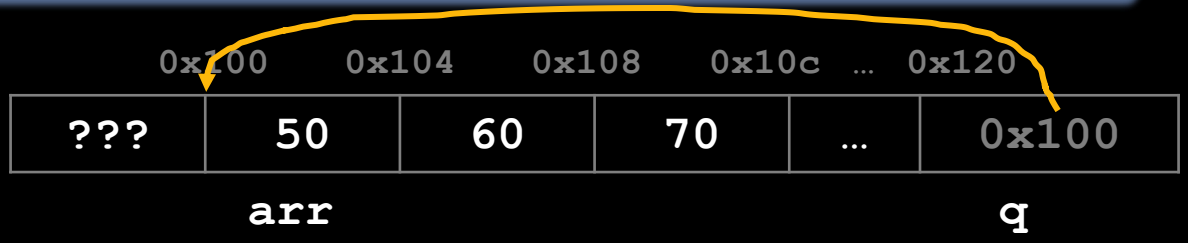
- **Declaration and initialization**
  - `int arr[] = {795, 635};`
  - declares and fills a 2-elt integer array

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

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

Pointer Arithmetic

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




```
// 32-bit signed int array
int32_t arr[] = {50, 60, 70};

int32_t *q = arr;
```

pointer ± integer

pointer + n

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

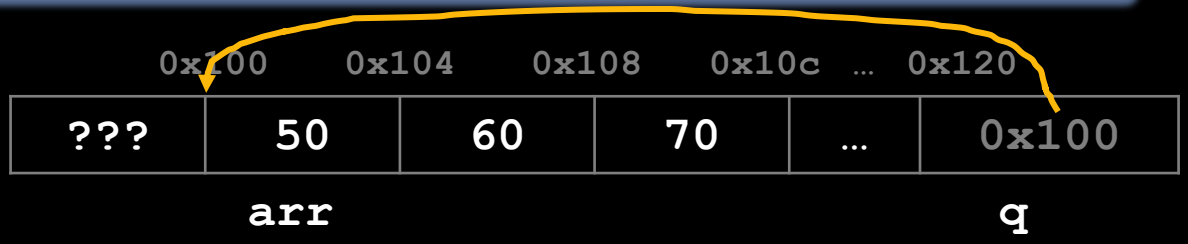
pointer – n

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

`a[i] ≡ *(a+i)`

Pointer Arithmetic

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



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

`a[i] ≡ *(a+i)`

```
// 32-bit signed int array
int32_t arr[] = {50, 60, 70};

int32_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 ???
```

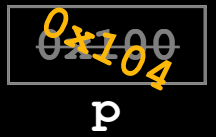
Handling Pointers (1/2)

How to get a function to change a pointer?

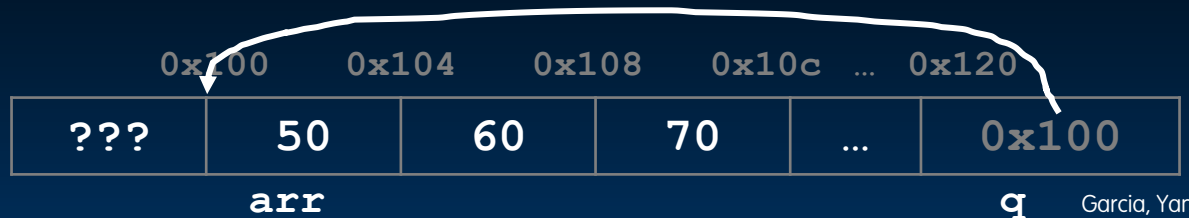
- Suppose we want `increment_ptr` to change where `q` points to.
- What gets printed?

```
*q is 50
```

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




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



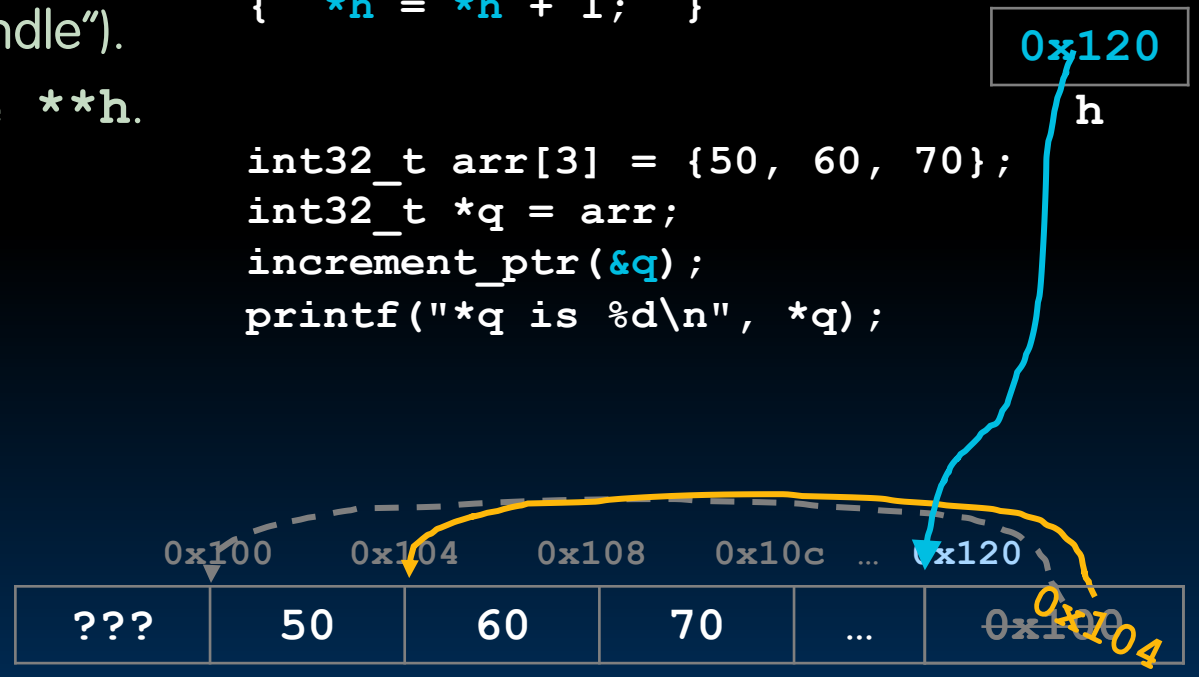
Handling Pointers (2/2)

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
```

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

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



Array Pitfalls

- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Word Alignment
- Function Pointer Example (recorded)

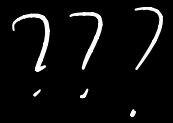
Array Pitfall #1 of 123234983

- 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)

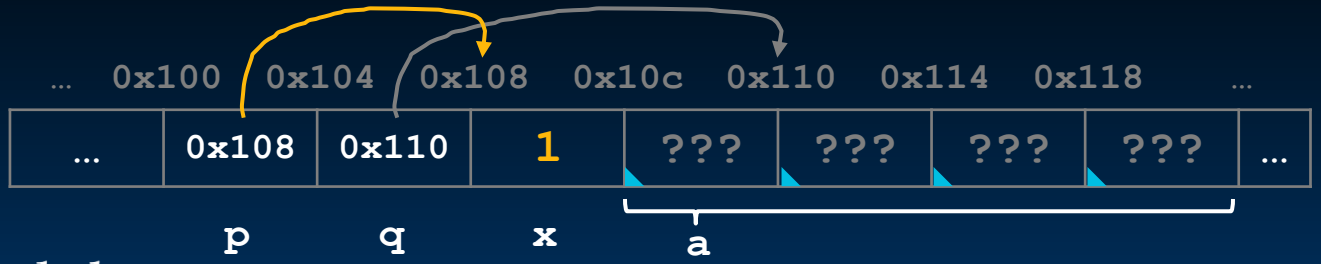
- **Accessing Array Elements**
  - `arr` is an array variable, but it looks like a pointer in many respects (though not all).
  - `arr[0]` is the same as `*arr`
  - `arr[2]` is the same as `*(arr+2)`

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

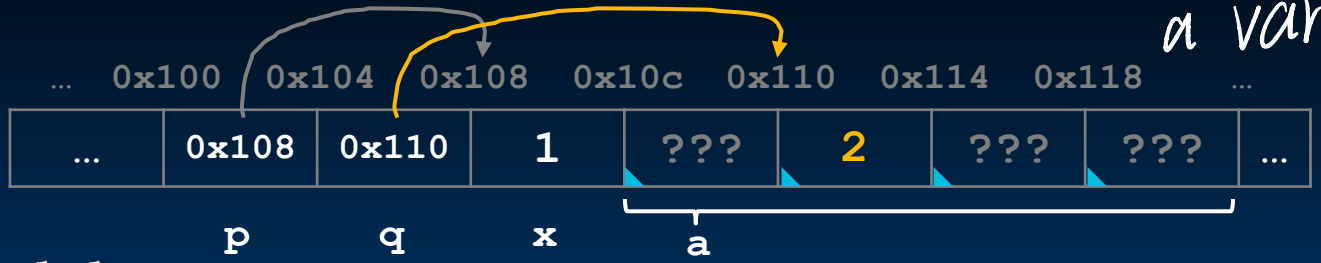


Arrays are not implemented as you'd think…

```
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
```

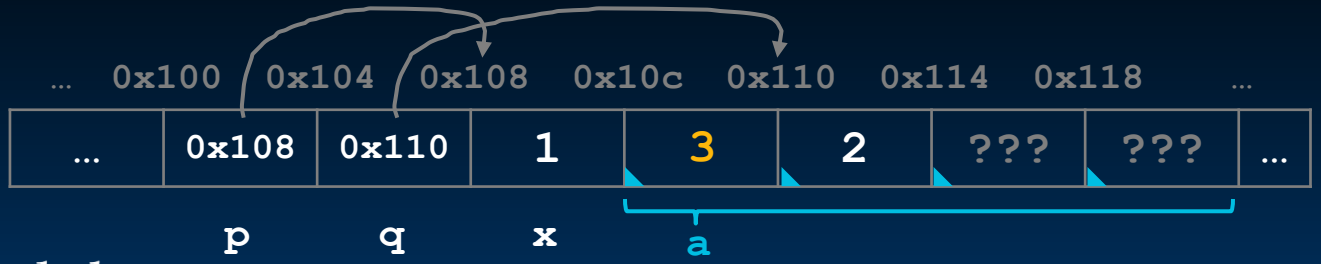
An array name is not a variable.



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"

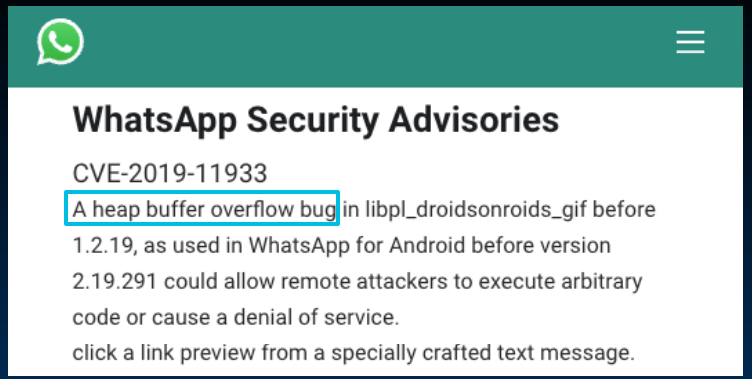
Lisa: "A C array is really just a big block of memory"

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

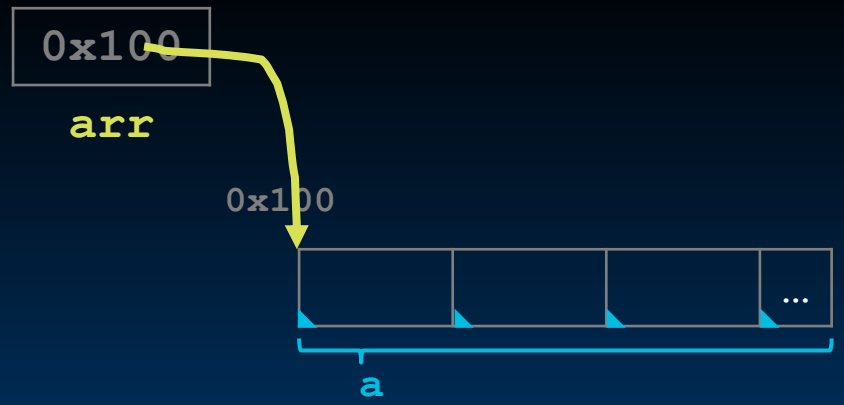
A heap buffer overflow bug in libpl_droidsonroids_gif before 1.2.19, as used in WhatsApp for Android before version 2.19.291 could allow remote attackers to execute arbitrary code or cause a denial of service.
click a link preview from a specially crafted text message.

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!

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);
    bar(b, 10);
    …
}
```



Must always explicitly pass array size/length!

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

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

the scope is not valid

Fix: Dynamic memory allocation! (next time)

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!
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 with `gdb`…

Strings

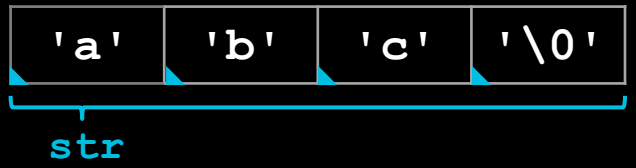
- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Word Alignment
- Function Pointer Example (recorded)

C Strings

string.h

- **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.**
  - In particular, string length operation does **not** include the null terminator when you ask for length of a string!

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




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

... strlen(str) ... // 3
```

Word Alignment

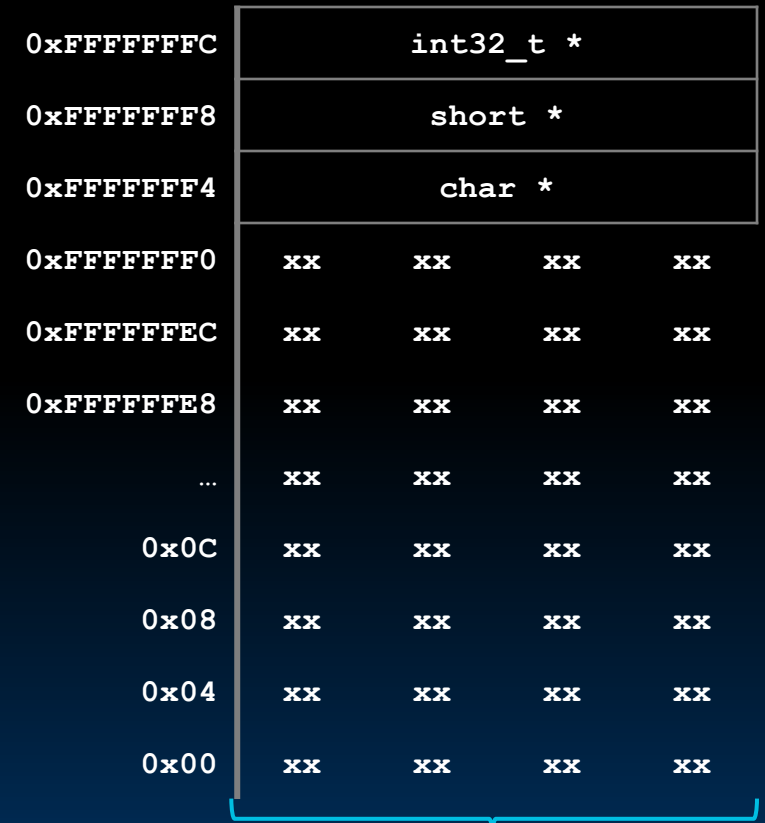
- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Word Alignment
- Function Pointer Example (recorded)

Memory and Addresses

1 byte = 8 bit

- **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.

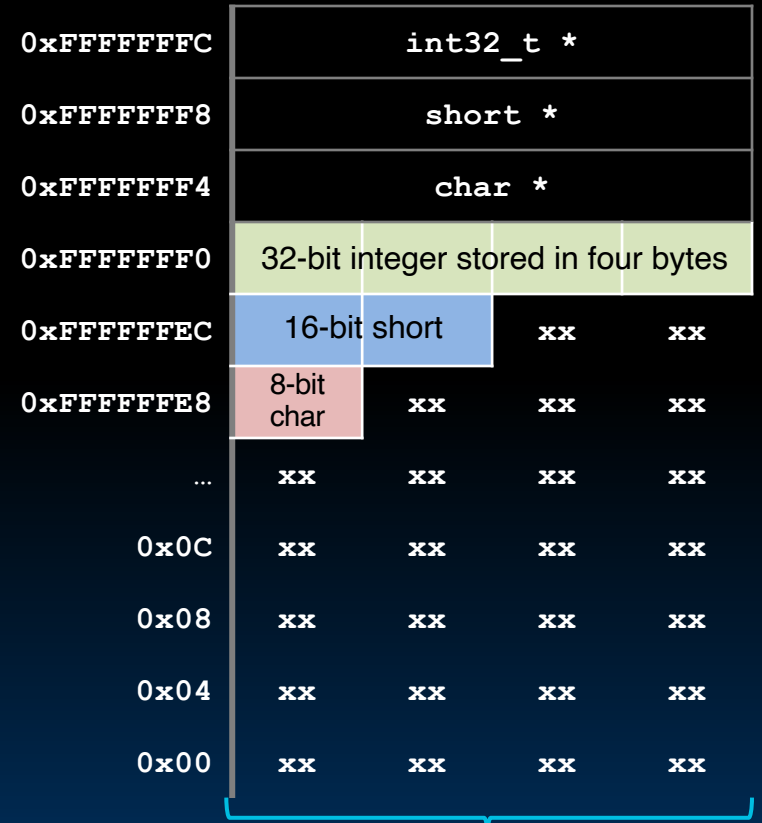
```
sizeof(int *) ==
    sizeof(char *) == 4
```



Word Alignment and Endianness

4 byte

- **A C pointer is just an abstracted memory address.**
  - Pointer type declaration tells the compiler how many bytes to fetch on each dereference.
- **But 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.

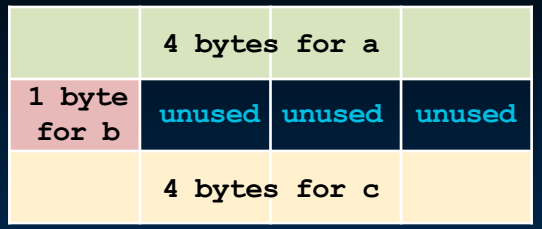


Structures, Revisited, Again

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

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

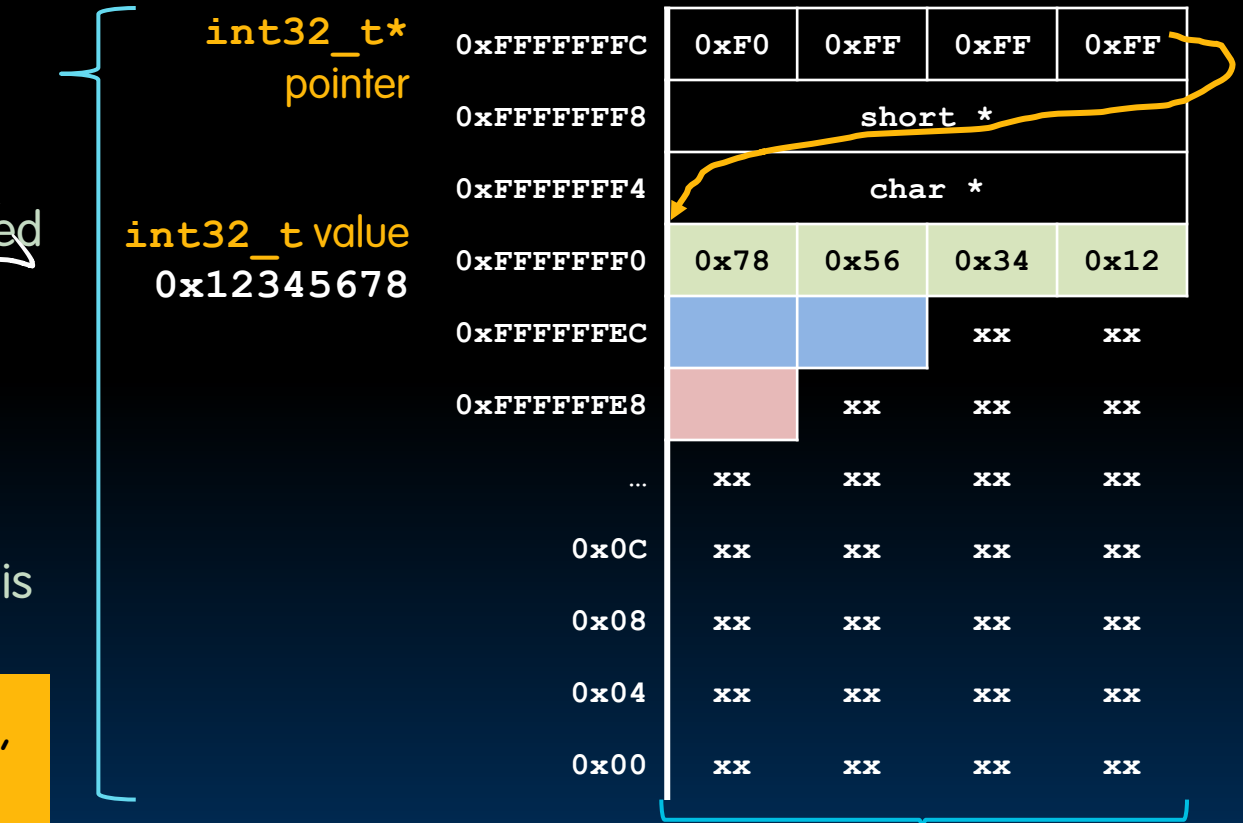
  - For this struct, the actual layout on a 32b architecture would be as follows.
    - Note the 3 bytes of padding
    - `sizeof(struct foo) == 12`



When debugging, watch out for 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)

Check out HW02 guide video, to be released soon



And in Conclusion…

- Pointers and arrays are pretty much the same
- C knows how to increment pointers
- C is an efficient language, 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!"

Function Pointer Example (Recorded)

- Pointers
- Using Pointers Effectively
- Pointer Arithmetic
- Array Pitfalls
- Strings
- Word Alignment
- Function Pointer Example (recorded)

Recording:
https://www.youtube.com/watch?v=PlIYbp0fgY4

map (actually mutate_map easier)

```
#include <stdio.h>

int x10(int), x2(int);
void mutate_map(int [], int n, int(*)(int));
void print_array(int [], int n);

int x2 (int n) {    return 2*n;    }
int x10(int n) {    return 10*n;   }

void mutate_map(int A[], int n, int(*fp)(int)) {
    for (int i = 0; i < n; i++)
        A[i] = (*fp)(A[i]);
}

void print_array(int A[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ",A[i]);
    printf("\n");
}
```

```
% ./map
3 1 4
6 2 8
60 20 80
```

```
int main(void)
{
    int A[] = {3,1,4}, n = 3;
    print_array(A, n);
    mutate_map (A, n, &x2);
    print_array(A, n);
    mutate_map (A, n, &x10);
    print_array(A, n);
}
```