Arrays And Pointers

Queens College, CUNY
CS211 Lab
Instructor: Kent Chin
Your Computer’s Memory...

...is just a giant 1D array!
2D Arrays in Memory

```c
int b[3][3];
```

What you visualize:

```
b[0][0] b[0][1] b[0][2] 
b[1][0] b[1][1] b[1][2] 
b[2][0] b[2][1] b[2][2] 
```

In memory:

```
b[0][0] b[0][1] b[0][2] b[1][0] b[1][1] b[1][2] b[2][0] b[2][1] b[2][2] 
```

**2D arrays are 1D arrays of 1D arrays!**
You’re used to coding like this...

```c
int a[4] = {1, 5, 0, 7};
a[0] = 4;
cout << a[2] << endl;
```

...here’s another way.

```c
*(a+0) = 4;
cout << *(a+2) << endl;
```
Slight Detour

```c
int a[4] = {1, 5, 0, 7};
```

The array name itself refers to the contiguous block of memory where the array sits.

Array names are not pointers, though they can be treated like pointers in many cases. Recall that we dereference a pointer to access whatever is being pointed to.

```c
cout << *a << endl;  // 1
cout << *(a+0) << endl; // 1
cout << *(a+2) << endl; // 0
cout << *(a+1) << endl; // 5
```
Fun Fact!

How we would usually print array a:

```cpp
for (int i = 0; i < 4; ++i)
    cout << a[i] << ' ';  
```

Note the following: \(a[i] = *(a+i) = *(i+a) = i[a]\)

So the following works too...

```cpp
for (int i = 0; i < 4; ++i)
    cout << i[a] << ' ';  
```

...never do this though!
Back On Track

```cpp
int b[3][3];
```

Assume integers and memory addresses are 4 bytes.

```
// sizeof prints the "actual" size of the array if called within
// same scope where array was made
cout << sizeof(b); // 36 (bytes)
// b+0 is seen as a pointer type by compiler
cout << sizeof(b+0); // 4 (bytes)
// *(b+0) gives you the 1D array at b[0]
cout << sizeof(*(b+0)); // 12 (bytes)
```
**Printouts**

```
cout << b; //0x22fe30
```

Here, `b` is seen as a *pointer to the first element* of the array.

```
cout << b+1; //0x22fe3c
```

Here, `b+1` points to the **next array element**, which is 12 bytes *past* array base (start) address.
Here, `&b` holds the *address of the array*, and is therefore a **pointer to the entire array**.

cout << &b; //0x22fe30

&b + 0
0x22fe30

b[0][0] b[0][1] b[0][2] b[1][0] b[1][1] b[1][2] b[2][0] b[2][1] b[2][2]

b+0

b+1

b+2

cout << &b+1; //0x22fe54

&b + i brings us *(i * sizeof(b)) bytes past the array’s base (starting) address*

&b+i

b+0

0x22fe30

b+1 36 bytes b+2

b+3

0x22fe54