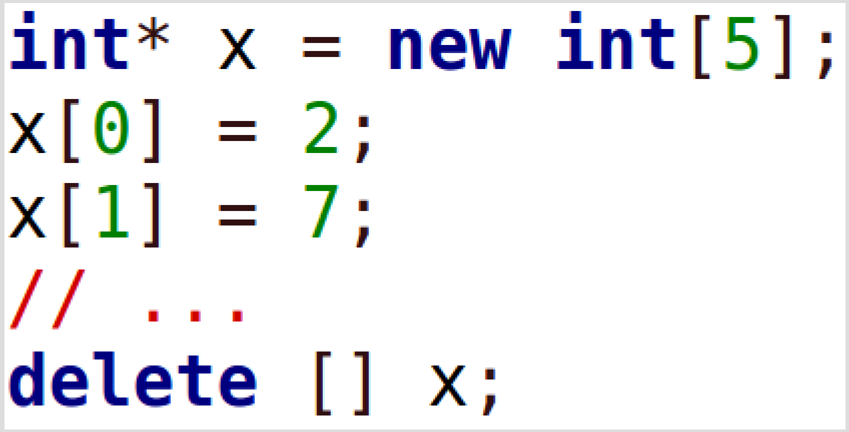
Pointers and memory

Ch 9, 11.4, 13.1 & Appendix F

MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Highlights

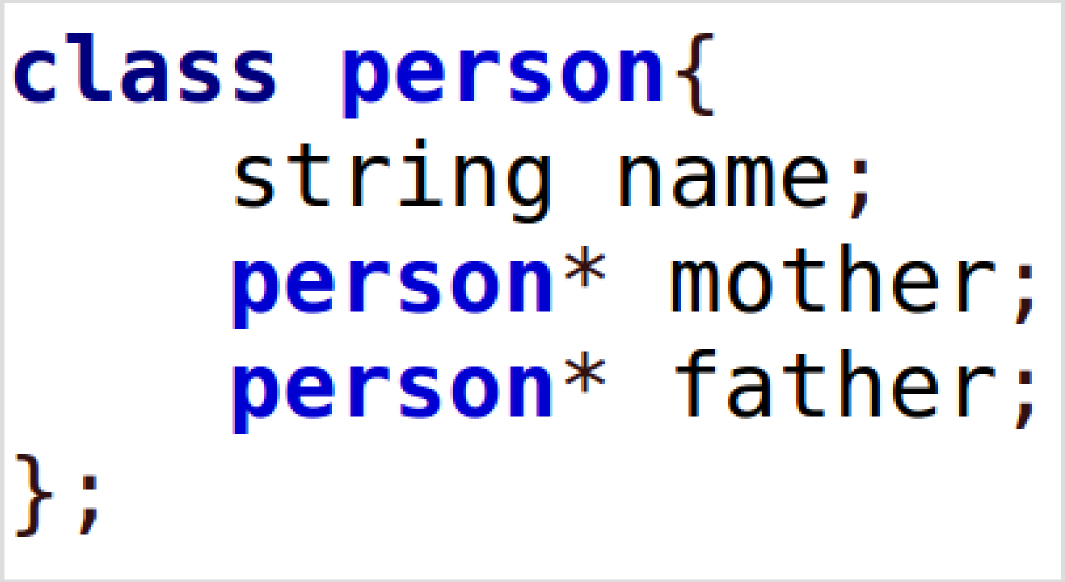
- dynamic arrays



```
int* x = new int[5];
x[0] = 2;
x[1] = 7;
// ...
delete [] x;
```

Person class

The ability to have non-named boxes allows you to more easily initialize pointers



```
class person{
    string name;
    person* mother;
    person* father;
};
```

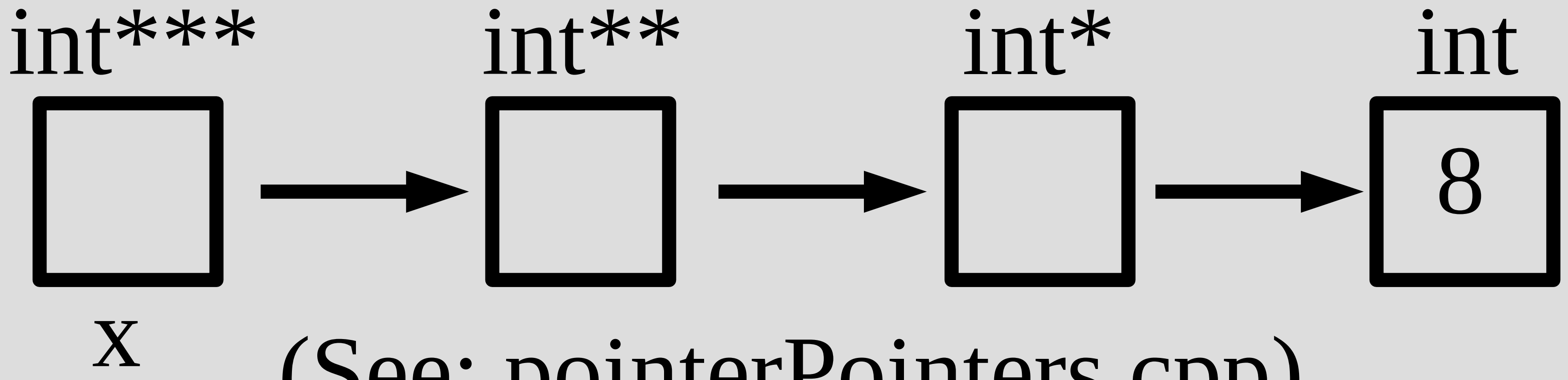
(See: personV3.cpp)

Pointer to pointer

You can have multiple stars next to types:

```
int*** x;
```

Each star indicates **how many arrows** you need to follow before you find the variable



(See: pointerPointers.cpp)

What pointers can/cannot do

Pointers CAN do

```
int *ptr;
int x = 2;
ptr = &x;
```

```
// pointer to...
int **ptr2;
// .. a pointer!
int *ptr;
int x = 10;
ptr2 = &ptr;
ptr = &x;
```

Pointers CANNOT do

```
int *ptr;
ptr = new int;
*ptr=3;
int x;
&x = ptr;
//cannot relabel/move box
```

```
int *ptr;
double x = 2.5;
ptr = &x;
// may seem weird...
```

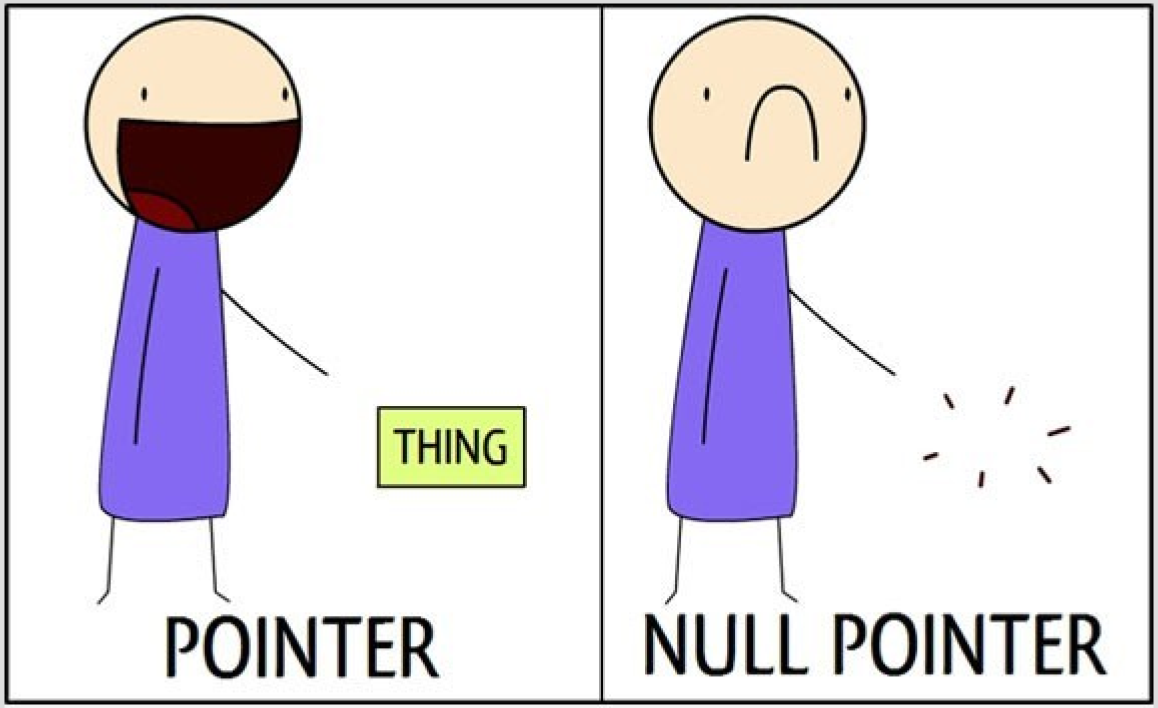
nullptr

When you type this, what is ptr pointing at?

```
int *ptr;
```

Answer: nullptr (or NULL)

```
int *ptr = nullptr;
```



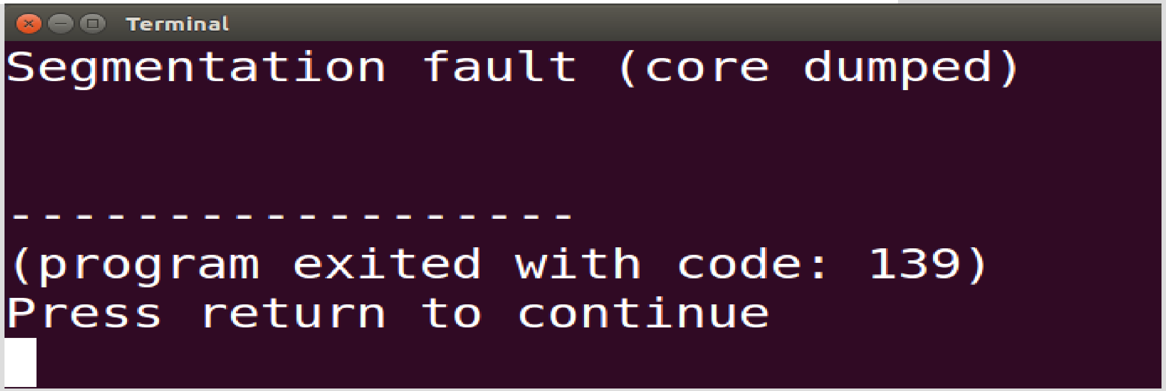
nullptr

The null pointer is useful to indicate that you are not yet pointing at anything

However, if you try to de-reference it (use *), you will seg fault

```
int *ptr = nullptr;
cout << *ptr << endl;
```

Do not try to ask the computer to go here



```
Segmentation fault (core dumped)


--------------------
(program exited with code: 139)
Press return to continue
```

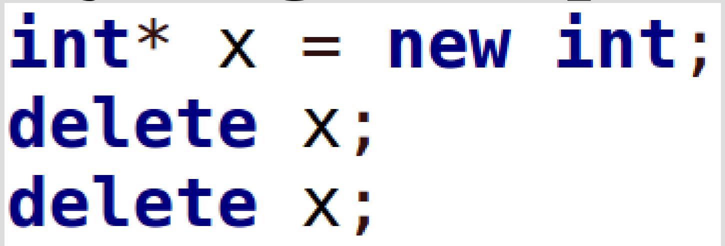
(see: nullptr.cpp)

Multiple deletes

Every new should have one corresponding delete command (one for one always)

The delete command gives the memory where a variable is pointing back to the computer

However, the computer will get angry if you try to give it places you do not own (i.e. twice)



```
int* x = new int;
delete x;
delete x;
```

Dynamic arrays

One of the downsides of arrays, is that we needed to have a fixed size

To get around this we have been making them huge and only using a part of it:

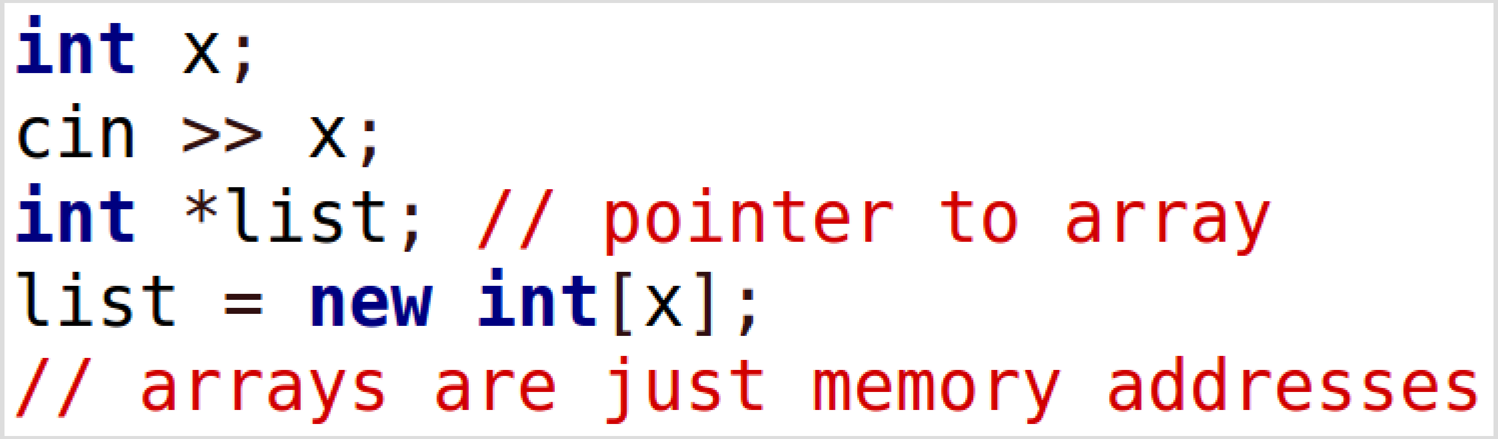
```
const int SIZE = 400;
int list[SIZE]; // SIZE must be const
```

Then we need to keep track of how much of the array we are currently using

Dynamic arrays

Arrays are memory addresses (if you pass them into function you can modify original)

So we can actually make a dynamic array in a very similar fashion



```
int x;
cin >> x;
int *list; // pointer to array
list = new int[x];
// arrays are just memory addresses
```

(this memory spot better to store large stuff)

Dynamic arrays

One important difference to normal pointers

When you delete an array you must do:

```
int *list; // pointer to array
list = new int[x];
delete [] list;
```

need empty square brackets

If you do the normal one, you will only delete a single index (list[0]) and not the whole thing

```
int *list; // pointer to array
list = new int[x];
delete list; // BAD BAD BAD BAD BAD
```

(See: dynamicArrays.cpp)

Functions & pointers

Another issues with arrays is that we could not return them from functions

Since arrays are memory addresses, we would only return a pointer to a local array

However, before this local array would just fall out of scope, but no more as dynamic memory stays until you manually delete it (See: returnArrays.cpp)

Dynamic 2D arrays

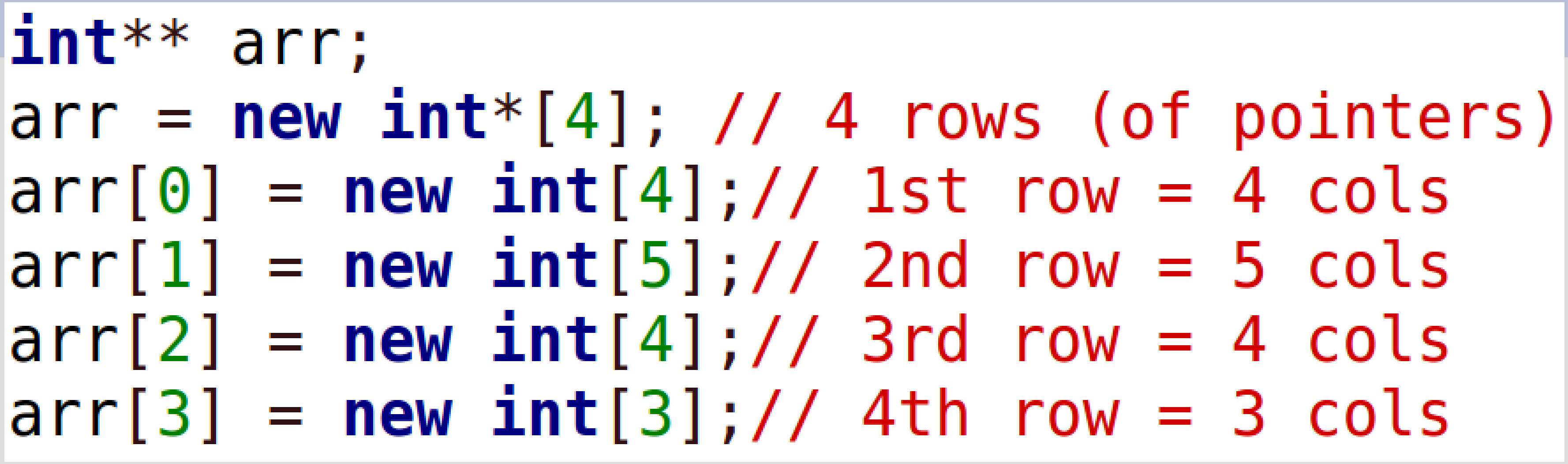
Since pointers can act like arrays...
(i.e. int* acts like int [])

... int** can act like a two dimensional array

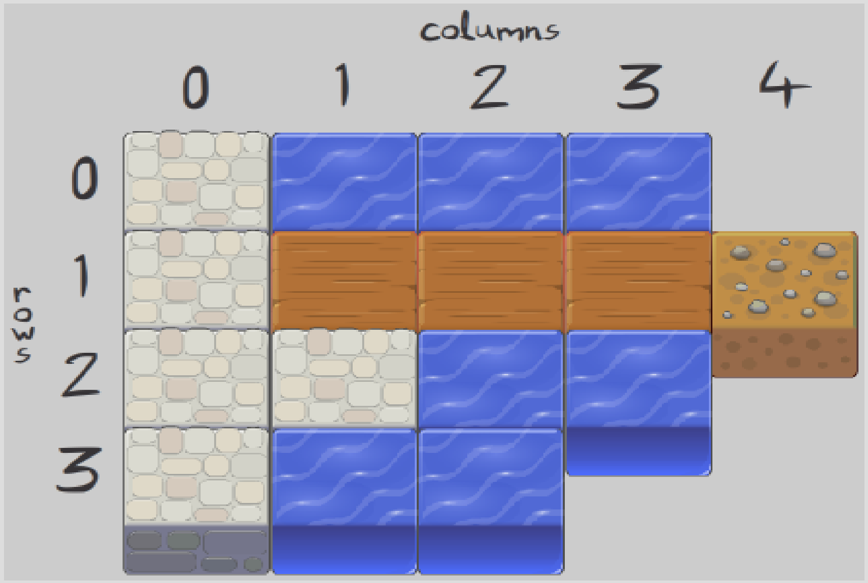
But need to use new to create each column individually (but can change the size of them)

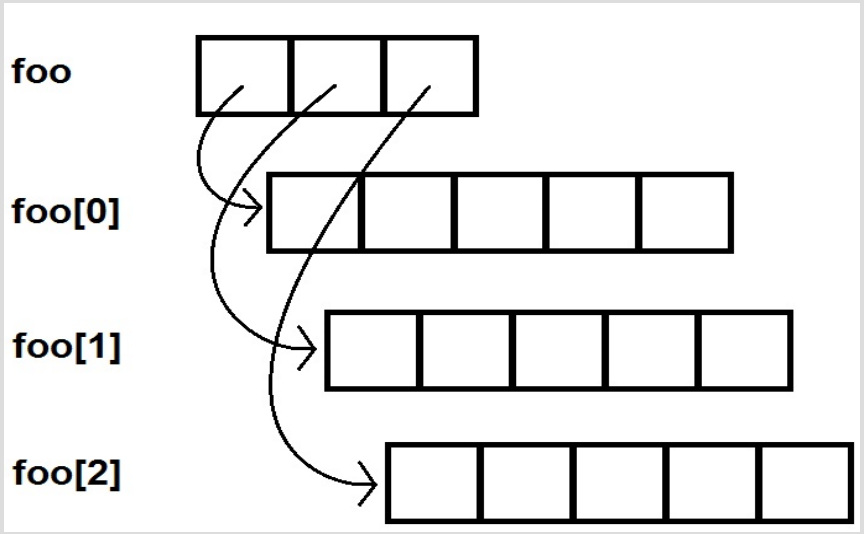
When deleting, same structure but backwards (delete each column, then rows)

Dynamic 2D arrays



```
int** arr;
arr = new int*[4]; // 4 rows (of pointers)
arr[0] = new int[4];// 1st row = 4 cols
arr[1] = new int[5];// 2nd row = 5 cols
arr[2] = new int[4];// 3rd row = 4 cols
arr[3] = new int[3];// 4th row = 3 cols
```

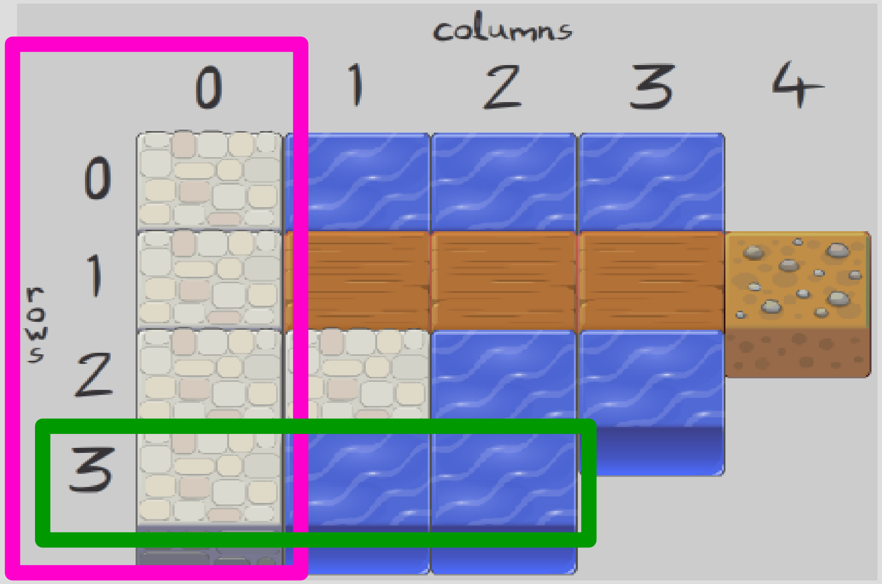


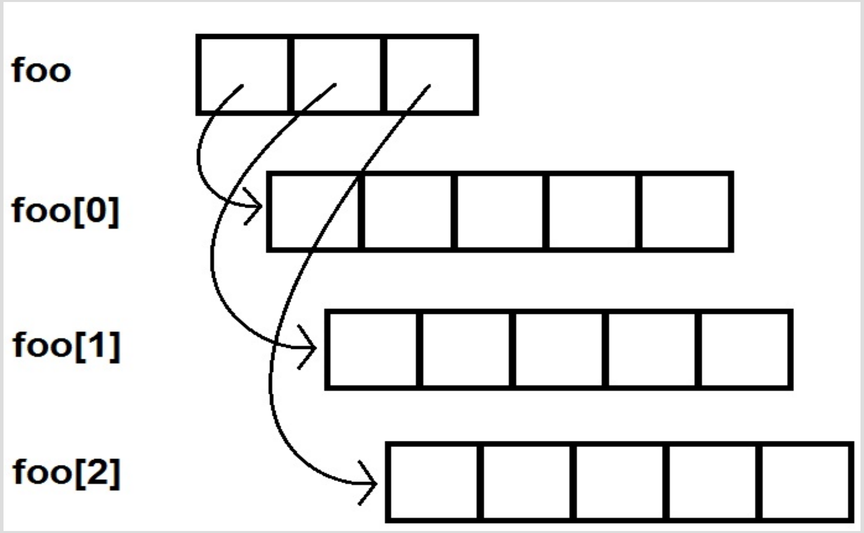


(See: raggedArray.cpp)

Dynamic 2D arrays

```
int** arr;
arr = new int*[4]; // 4 rows (of pointers)
arr[0] = new int[4];// 1st row = 4 cols
arr[1] = new int[5];// 2nd row = 5 cols
arr[2] = new int[4];// 3rd row = 4 cols
arr[3] = new int[3];// 4th row = 3 cols
```





(See: raggedArray.cpp)

Reasons why pointer

Why use pointers?

1. Want to share variables (multiple names for the same box)
2. Dynamic sized arrays
3. Return arrays from functions (or any case of keep variable after scope ends) (DOWN WITH GLOBAL VARIABLES)
4. Store classes within themselves
5. Automatically initialize the number 4 above