Review: Static arrays

Rule: a call to `f` always takes the same amount of space on the stack

```
void f()
{
  const int ADDER = 10;
  const size_t SIZE = 3;
  int data[SIZE];
  for (size_t i = 0; i < SIZE; ++i) {
    data[i] = i + ADDER;
  }
}
```

a pointer to an integer

int*

cin >> size;

data = new int[size];

returns location in the heap w/ room for our array

- pointer math
- dereference

*(data + i) = i + ADDER;

data[i] = i + ADDER;

delete[] data;

Dynamically-allocated arrays

What if we wanted to create an array whose size is dynamic (i.e., known only at runtime)?

```
int* p = new int(5);
// h70 (location of ONE int
```

~~delete[size] data~~

```
delete[] data;
```

Pointer Math

Consequence:

Dereference a pointer

*data ⇒ the int that data points to

Cow* cp = new Cow[3];

*cp → Cow *data → int

data + 1 ⇒ data + (size of 1 int)

cp + 1 ⇒ cp + (size of 1 Cow)

Object Lifetime for Pointers

Pointers are primitive (regardless of the type they point to).

`data`

- Allocation: at opening { of f'n
- Initialization: at declaring line
- Use: (scope)
- Destruction: at } of declaring block
- Deallocation: at closing } of f'n

dynamically allocated array (`*data`)

- Allocation:
- Initialization:
- Use:
- Destruction:
- Deallocation:

Class Exercise

```
int** p = new int*[5];
p[0] = new int[3];
for(size_t i=0; i<5; ++i) {
    delete[] p[i];
}
delete[] p;
delete[] p;
```

double delete