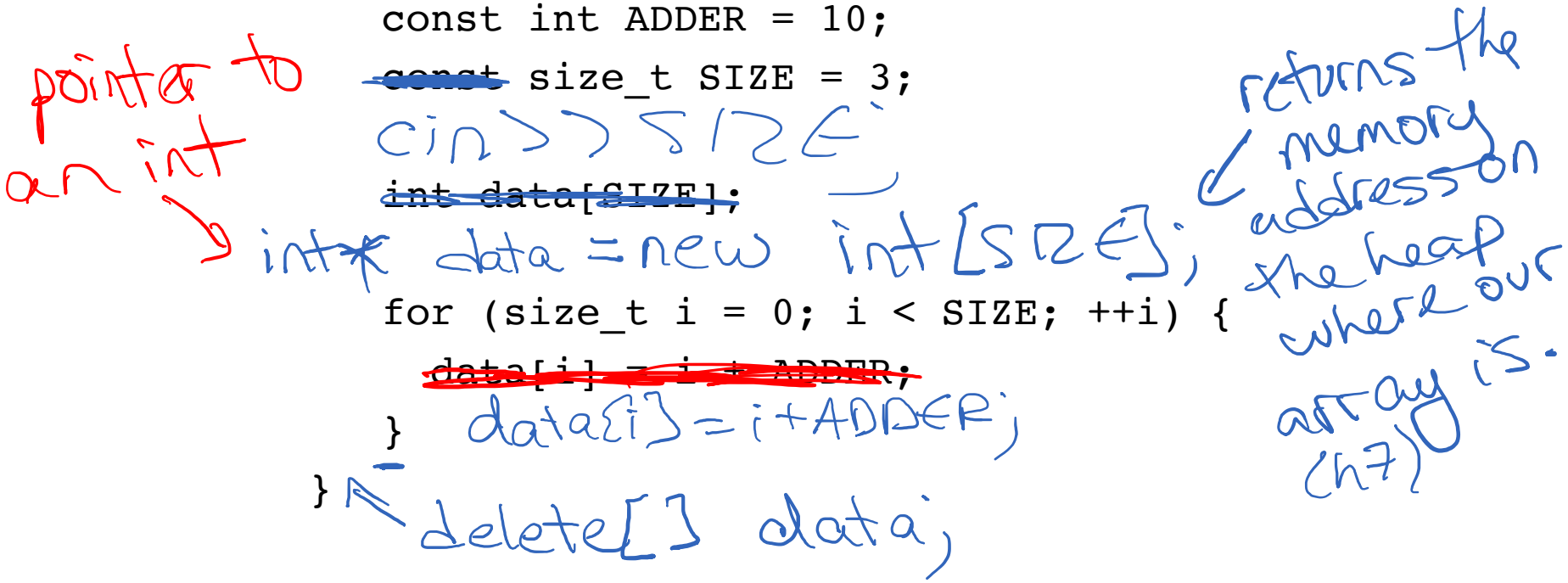
Review: Static arrays

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;
  }
}
```



Dynamically-allocated arrays

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

Pointer Math

Consequence:

int* data = h7;

data + 1 = (h7 + size of 1 integer)

Dereferencing

*data ⟹ the value at data.

*data = 11;

```
for(size_t i=0; i<size; ++i) {
    *(data+i) = ADDER+i;
}
```

Object Lifetime for Pointers

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

data

- Allocation: at function's {
- Initialization: at declaring line: do nothing
- Use: (during scope)
- Destruction: at closing } of declaring block; do nothing
- Deallocation: at function's }

dynamically allocated array (`*data`)

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

```
delete[size] data;
delete[] data;
delete
delete[]
```

```
Cow* p = new Cow;
...
delete p;
```

Class Exercise

```
Cow** p = new Cow*[3];
p[0] = new Cow;
```

Ex 1:

```
delete myInt;
```

Ex 2:

```
delete[] dynArray;
```

Ex 3:

```
delete[] myCows;
for (size_t i=0; i<3; ++i){
    delete myCows[i];
}
delete[] myCows;
```