Name: _____

Today's Date: _____

Today's Goals:

- Explain the problem that is solved by *dynamic allocation*
- Safely use dynamically-allocated memory
- Explain what a *pointer* is

Today's Question(s)

Why can't the arrays we've seen so far change size?

Lingering Questions

What is still unclear after today's class?

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:

Object Lifetime for Pointers

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

`data`

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

dynamically allocated array (`*data`)

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

Class Exercise

Exercise 1

```
int main() {
    int* myInt = new int{0};
    const size_t myConstant = 4;
    Cow myCows[myConstant];

    // TODO: AVOID MEMORY LEAKS

}
```

Exercise 2

```
void triple(int* intArray, const size_t SIZE) {
    for (size_t i=0; i<SIZE; ++i) {
        intArray[i] *= 3;
    }
}

int main() {
    size_t size = 3;
    int* dynArray = new int[size];

    for (size_t i=0; i<size; ++i) {
        dynArray[i] = i;
    }

    triple(dynArray);

    // TODO: AVOID MEMORY LEAKS

}
```

Exercise 3

```
Cow** makeCowPtrArray(const size_t SIZE) {
    Cow** cowPtrArray = new Cow*[SIZE];

    for (size_t i = 0; i<SIZE; ++i) {
        cowPtrArray[i] = new Cow;
    }
    return cowPtrArray;
}


int main() {
    Cow** myCows = makeCowPtrArray(4);
    // TODO: AVOID MEMORY LEAKS

}
```