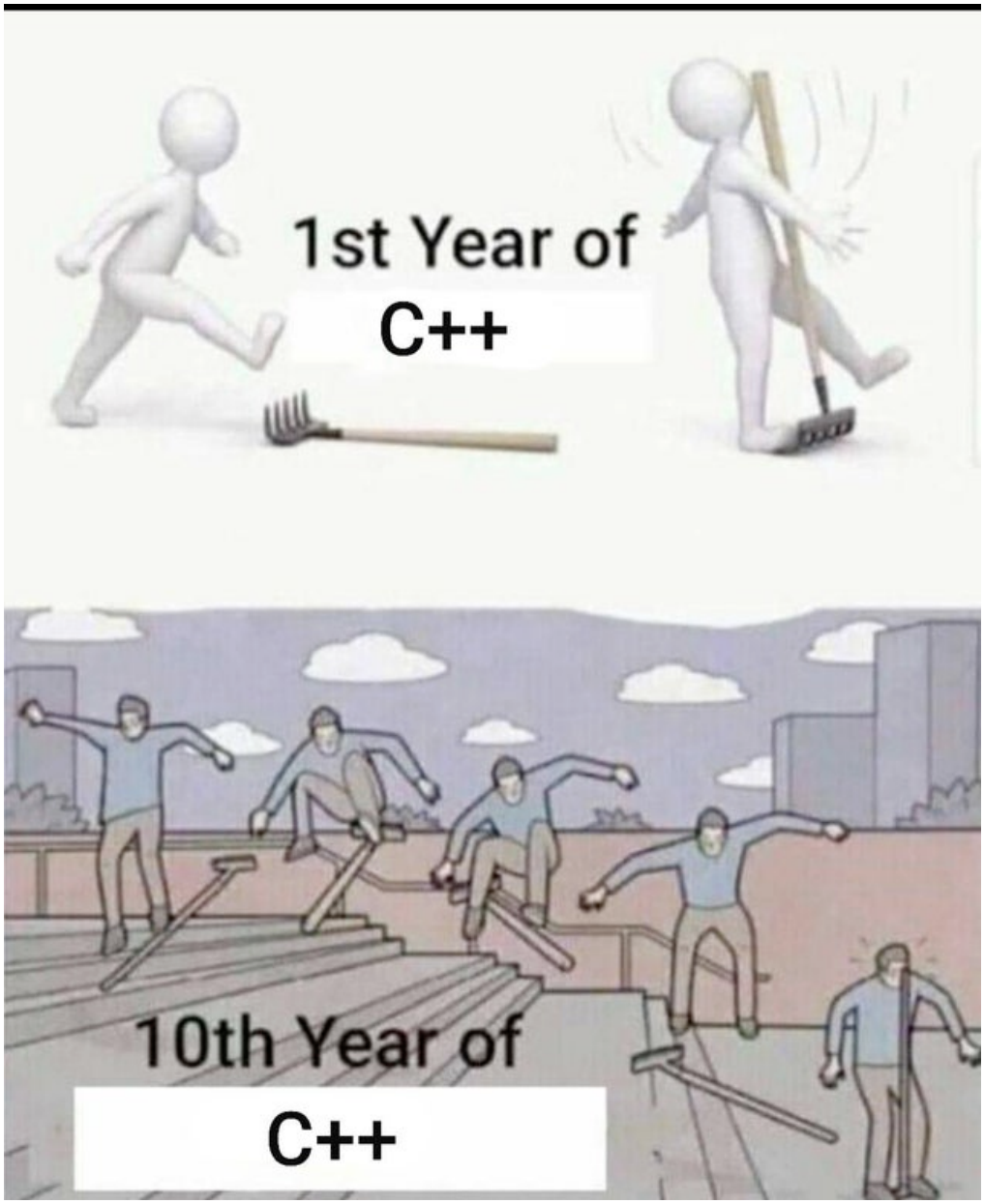
Week 2



Questions?

- anything?

Pointers and Addresses

- Carey's slides (review pointers and go over dynamic memory allocation)
  - https://drive.google.com/file/d/1U3tJyYhOZRMpZQSwh4AkLTVjXroHcYu5/view

Initializer Lists

```
class House {
  public:
    House(double price, int squareFt);
    // ...
  private:
    double mPrice;
    int    mSquareFt;
    // ...
}

// old constructor
House::House(double price, int squareFt) {
  mPrice    = price;
  mSquareFt = squareFt;
}

// TODO: how can I apply an initializer list to my constructor?
House::House(double price, int squareFt)
  :mPrice(price), mSquareFt(squareFt)
{

}

// create an instance of Class House
House exampleHouse(5, 5);
```

Order of Construction/ Destruction

```
// CONSTRUCTION ***************************************************

1. ------------------------------

2. Initialize member variables, first looking at the initializer list
     if dataMember not in initializer list:
       if dataMember is a built-in type -> leave it uninitialized
       if dataMember is a Class         -> call the default constructor
         // anything to be careful of here? (make sure your class has a default
         // constructor defined)

3. Run the constructor
```

```
// DESTRUCTION ***************************************************

1. Run body of the destructor

2. Destroy each member variable
     if dataMember is a built-in type -> do nothing
     if dataMember is a Class        -> call the Class' destructor

3. ------------------------------
```

Using Initializer Lists w/ Objects

```
class City {
  public:
    City(double mayorPrice, int mayorSquareFt);
    // ...
  private:
    House mayorHouse;
    // House* mayorHouse;  // TODO: is there a different way I could've done it?
                           // making it a pointer doesn't call the default
                           // constructor, but we now need to do dynamic
                           // allocation
    // ...
}

// TODO: how do I initialize mayorHouse?
City::City(double mayorPrice, int mayorSquareFt)
  :mayorHouse(mayorPrice, mayorSquareFt) // use the initializer list!
{
  // would not compile
  // mayorHouse(mayorPrice, mayorSquareFt)

  // if I used the pointer approach, this code is what I'd need to do
  // and I'd need to make sure to delete it in my destructor
  mayorHouse = new House(mayorPrice, mayorSquareFt)

}

// does initializer list order matter?
// if an object depends on other member variables to be constructed first,
// then yes. It's good practice to match the order you declare your variables
// and the order used in the constructor list.
```

Resource Management

Dynamically Allocated Arrays

```
// Arrays in CS31
// size needed to be known at compile time
int nums[5] = {1, 2, 3, 4, 5}
cout << *nums << endl;
cout << *(nums+1) << endl;

// Now we can create arrays with sizes that aren't known at compile time

// wait for the user to input a size at run time
int size;
cin >> size;

// TODO: create an array of length size
// int nums[size];  // won't compile

int* nums = new int[size]

// ... do stuff with array
nums[0] = 5;                // type requires a default constructor
                            // here we're using a built in type, which has one

// TODO: delete the array
delete[] nums;
num = nullptr;

*nums;  // is there a problem with this?
        // if we didn't set num to nullptr we'd have a dangling pointer
        // and de-referencing it would have undefined behavior

// similar to creating/destroying objects
```

Classes that Hold a Resource

```
class Disc {
  public:
    // TODO: dynamically create an int array for m_students to hold n students
    Disc(int n)
      :m_nStudents(n)
    {
      // make sure m_nStudents is initialized before the array (otherwise it'll
      // be a junk value
      m_students = new int[m_nStudents];
    }

    // TODO: do we need a destructor?
    // yes, because we are handling a resource
    ~Disc() {
      delete[] m_students;
    }

    // this function uses our resource
```

```
    void favoriteAnimal() {
      // blobfish and alpaca were candidates
      cout << "Giraffe" << endl;
      cout << m_students[0]; // assumes we always have at least 1 student
    }

  private:
    int m_nStudents;
    int *m_students;
}
```

Copy Constructor

```
// what happens if you try to copy an object that doesn't have a
// copy constructor defined?

// you get a shallow copy, which can result in some funny behavior
// we want a deep copy

// Our class manages a resource, the array m_students
// we need to define a destructor, copy constructor, and assignment operator
  // to prevent mishandling of the resource

class Disc {
  public:
    Disc(int n){
      m_nStudents = n;
      m_students = new int[n];
    }

    // TODO: write the copy constructor
    Disc(const Disc &src) {    // src needs to be a reference otherwise you
                               // need a copy constructor to pass-by-value,
                               // for which you need a copy constructor, etc.
      m_nStudents = src.m_nStudents;

      m_students = new int[m_nStudents];
      for (int i = 0; i < m_nStudents; i++) {
        m_students[i] = src.m_students[i]
      }
    }

    // TODO: do we need a destructor?
    // yes, we're allocating memory earlier and need to call delete
    ~Disc() {
      delete [] m_students;
      m_students = nullptr;
    }

    void favoriteAnimal() {
      // cat, turtle, alpaca were other contenders in the chat
```

```
      cout << "Blue whale" << endl;
      cout << m_students[0]; // assumes we always have at least 1 student
    }

  private:
    int m_nStudents;
    int *m_students;
}

// What is a copy constructor? (example with ints)
int x = 5;
int b = x;

int main() {
  Disc a2(60);

  if (true) {
    // Before writing our copy constructor, b2 was a shallow copy
    Disc b2 = a2;

    // when b2 went out of scope here, it deleted m_students, which was actually
    // the same array that a2 pointed to
    // a2.favoriteAnimal() wouldn't work because it access the array, which would
    // have already been deleted!
  }

  a2.favoriteAnimal();
}

// TODO: call copy constructor in another way
Disc a2(60);

Disc a3(&a2);

// how else can we use it?
// now we can pass our object by value, which may be slow as it has to copy
```

Assignment Operator

```
// what happens if we don't define an assignment operator?
// again, shallow copies but also memory leaks when trying to assign

class Disc {
  public:
    Disc(int n){
      m_nStudents = n;
      int* m_students = new int[n];
    }

    // copy constructor
    Disc(const Disc &src) {
```

```
      this->m_nStudents = src.m_nStudents;
      m_students = new int[m_nStudents];

      for (int i = 0; i < m_nStudents; i++) {
        m_students[i] = src.m_students[i];
      }
    }

    // TODO: add assignment operator
    Disc& operator= (const Disc &src) {

      // check for aliasing
      if (&src == this) {
        return *this;
      }

      // delete the old resource and allocate a new one of the correct size
      delete[] m_students;

      m_nStudents = src.m_nStudents;
      m_students = new int[m_nStudents];

      for (int i = 0; i < m_nStudents; i++) {
        m_students[i] = src.m_students[i];
      }

      // returning a reference allows us to chain the operator
      // e.g. x1 = x2 = x3
      return *this;
    }

    ~Disc() {
      delete [] m_students;

      m_students = nullptr;
    }

    void favoriteAnimal() {
      cout << "Blue whale" << endl;
    }


  private:
    int m_nStudents;
    int *m_students;
}

// example with ints (built in type)
int x1 = 5;
int x2 = 10;

x2 = x1;

// example with resource management
Disc steakSauce(10);  // haha..
Disc a2(60);
```

```
Disc a3(20);

steakSauce = a3 = a2;

a2 = a2;  // test if aliasing works

// swap function?
// The copy constructor and assignment operator have some code in common.
// While you shouldn't call one from the other, you can put some code in a
// seperate swap function that both functions call. This is considered more
// modern style.

// explained around 1:24:00 in the April 7th: Resource Management lecture
```