Dynamic memory in class
April 6, Ch 11.4 & Appendix F

What's a memory leak?
I forget
- Deconstructors

class simple{
public:
    int x;
    simple(); // constructor (default)
    ~simple(); // deconstructor (cannot overload)
};
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
Review: constructors

Constructors are special functions that have the same name as the class.

Use a constructor to create an instance of the class (i.e. an object of the blueprint)

```cpp
// all three the same
string a = string("one way");
string b("another way");
string c = "overloaded operator way";
```
Person class

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

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

(See: personV4.cpp)
Constructors + dynamic

What if we have a variable inside a class that uses dynamic memory?

```cpp
class simple{
public:
    int* xArray;
    simple();
};
```

simple::simple()
{
    xArray = new int[3];
}

When do we stop using this class?
What do we do if the int* was private?

(See: classMemoryLeak.cpp)
Often, we might want a class to retain its information until the instance is deleted.

This means either:
1. Variable's scope ends (automatically deleted)

2. You manually delete a dynamically created class with the delete command.

```cpp
while(true)
{
    Leaky oops;
}
```
Constructors + dynamic

A good analogy is file I/O, as there are 3 steps:

1. Open the file (read or write)
2. Use the file
3. Close the file

The constructor is basically requiring step 1 to happen

Do you want 3 to be automatic or explicit?
Deconstructors

Just as a constructor **must** run when a class is created...

A **deconstructor** will always run when a class object-instance-variable is deleted.

Deconstructors (like constructors) must have the same name as the class, but with a `~`:

```cpp
public:
    Unleaky();
    ~Unleaky();
```

(See: classMemoryLeakFixed.cpp)
Deconstructors

The benefit of deconstructors is the computer will run them for you when a variable ends. This means you do not need to explicitly tell it when to delete the dynamic memory, simply how it should be done.

This fits better with classes as a blueprint that is used in other parts of the program. (see: deconstructor.cpp)