Dynamic memory in class

Nov 24, Ch 11.4 & Appendix F

What's a memory leak?

I forget
Announcements

HW 6B - too hard, will turn into an extra credit problem

No labs this week (Thursday is a holiday)

Homework 8 will be posted tomorrow, due next Wednesday
- Deconstructors

```cpp
class simple{
    public:
        int x;
        simple(); // constructor (default)
        ~simple(); // destructor (cannot overload)
};
```

- this

```cpp
void simple::foo()
{
    this->x=2;
}
```
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";
```
Constructors + dynamic

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

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

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

(See: classMemoryLeak.cpp)
Constructors + dynamic

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.

```c
while(true) {
    Leaky oops;
}
```

`oops out of scope = gone`
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 happen hidden 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.
Consider the following code:

```cpp
BadPublic test;
test.x=3;

int* intPtr = &(test.x);
intPtr = test.getX();

BadPublic* bpPtr = &test;
bpPtr = test.getMe();
```

class BadPublic {
    public:
        int x;
        int* getX();
        BadPublic* getMe();
};

How do we write `getX()` and `getMe()`?
Q: It seems you should have information about yourself, but how do you access that?

A: Inside every class, there is a this pointer, that points to yourself.

(See: thisSelfPointer.cpp)
Copy constructor

Review:
Difference between these two?

```c
void doSomething(int x)
void doSomething(int &x)
```
Copy constructor

Review:
Difference between these two?

```c
void doSomething(int x)
void doSomething(int &x)
```

First one makes a copy of the input for a local variable inside the function

Second one links directly to the original variable (as it uses the memory address)
Copy constructor

There is actually a built-in copier (much like there is a built-in default constructor)

You will use a copy when:
1. You use an '==' sign with classes
2. You call-by-value a class as an input to a function (i.e. do not use &)

Issues with copying? (Hint: recent material) (See: copyIssues.cpp)
Copy constructor

To avoid double deleting (crashes program) or multiple pointers looking at the same spot...

We have to redefine the copy constructor if we use dynamic memory

The **copy constructor** is another special constructor (same name as class):

```cpp
Dynamic();
~Dynamic();
Dynamic(const Dynamic &d);
```
Copy constructor

In a copy constructor the “const” is optional, but the call-by-reference is necessary (the '&')

Why?
Copy constructor

In a copy constructor the “const” is optional, but the call-by-reference is necessary (the ' &) 

Why? 
If you did not use a &, you would make a copy which would call a copy constructor... 
which would make a copy... 
which would call a copy constructor... 
which crashes your computer! 
(See: copyConstructor.cpp)
When using dynamic memory with classes, there is one more problematic use: the '==' sign. The “proper” way to implement this is nuanced (and it is late)... see code comments.

We should ensure that no memory addresses are getting copied (otherwise they overlap)

(See: overloadEquals.cpp)
When using pointers in a class or dynamic memory, you should create:

1. Deconstructor
2. Copy constructor
3. Overload '==' operator

Typically the built-in functions are not sufficient if you use a “new” or '*size