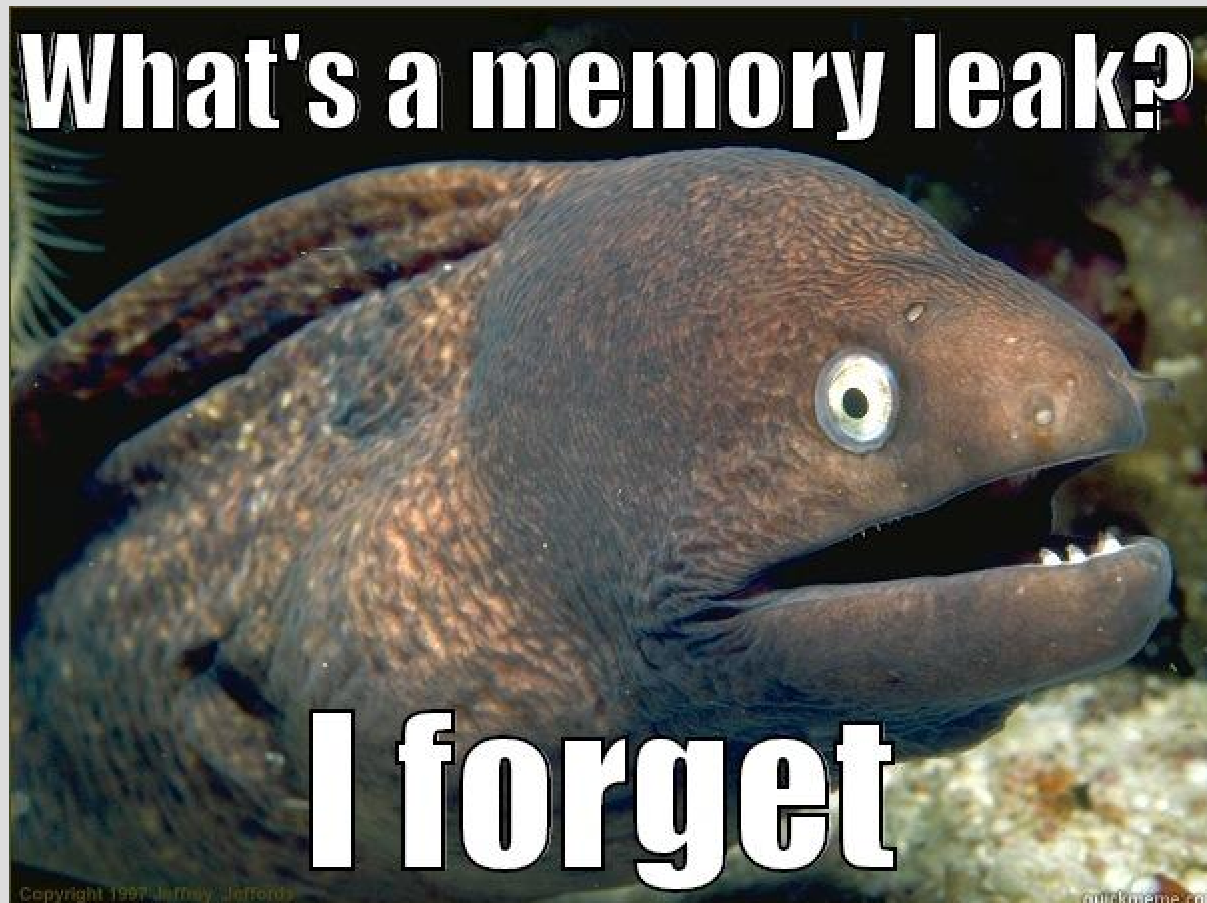
Dynamic memory in class

Ch 9, 11.4, 13.1 & Appendix F



Announcements

Test next week (whole class)

Covers:

-Arrays

-Functions

-Recursion

-Strings

-File I/O

Highlights

- Destructors

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

- Copy constructors

```
class simple
{
    public:
    simple(const simple &o)
};
```

- operator= (basics)

```
classy x;
classy y;
y=x; // equals operator
```

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)

```
// 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?

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

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

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

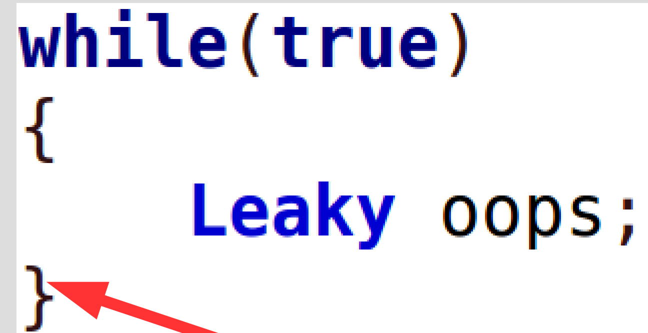
(See: classMemoryLeak.cpp)

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)



oops out of scope = gone

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

Destructors

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

A destructor will always run when a class object/instance/variable is deleted

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

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

constructor

destructor

(See: classMemoryLeakFixed.cpp)

Destructors

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?

Destructors

The benefit of destructors 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: destructor.cpp)

const call-by-reference

What is the difference between these two?

```
int sum(int x, int y);
int sum(const int &x, const int &y);
```

const call-by-reference

What is the difference between these two?

```
int sum(int x, int y);
int sum(const int &x, const int &y);
```

First one copies the values into x and y, thus these values exist in multiple places

The second creates a link but does not let you modify the original (see: callByValue.cpp)

const call-by-reference

Classes can be rather big, so in this case using const and '&' can save memory

So a better way to write:

```
bool equals(Point first, Point second)
```

... would be: (function definition the same)

```
bool equals(const Point & first, const Point & second)
```

In fact, without & creates a copy, which is a new object and thus runs a constructor

Copy constructor

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

This built-in copier makes the boxes hold identical values... but is this good enough?

Issues with copying? (Hint: recent material)

(See: copyIssues.cpp)

Copy constructor

Destructors are nice because they can automatically clean up memory

However, you have to be careful that you do not cause things to delete twice

This primarily happens when a copy is made poorly (a good copy is a “deep copy”) i.e. all pointers should not be shared between copies, you recursively remake the pointers

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):

```
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)

Copy constructor

You will use a copy when:

1. You use an '=' sign when declaring a class
2. You call-by-value a class as an input to a function (i.e. do not use &)
3. You return an inputted class to function

(Third the compiler sometimes skips)

(See: placesCopyConstructorRuns.cpp)

Copy constructor

The most common class we have used is the “string” class

Lines like this were running copy constuctor:

```
string sent = "This is a sentence";
string firstWord = sent.substr(0, 4);
```

It actually converts lines like this:

```
string firstWord = string(sent.substr(0,4));
```

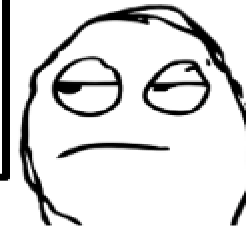
Operator =

Ch 11.4 & Appendix F

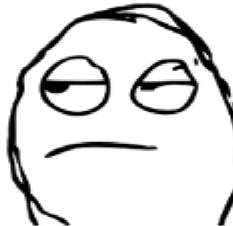
*me, a C/C++ developer learning java for the first time

*the laptop

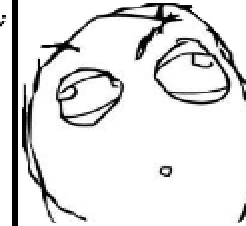
```
int x;
```



```
int foo[] = new int[100];
```

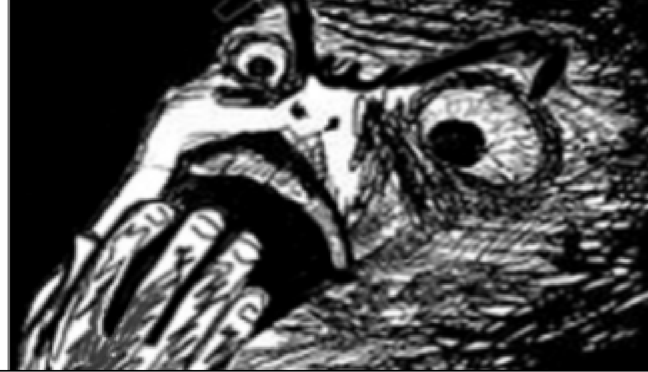


```
int foo[] = new int[100];
foo =
```



```
int foo[] = new int[100];

foo = new int [50];
```



Copy constructor: arrays

How would you copy a dynamically created array inside a class?

```
class rng{
private:
    double* array;
public:
    rng();
    rng(const rng &original) //write me!
};
```

What if this was a normal array?

```
rng::rng()
{
    array = new double[10];
    for(int i=0; i < 10; i++)
    {
        array[i] = rand()%100; /0-99
    }
}
```

(see: copyArray.cpp)

Copy constructor vs. '='

There is actually two ways in which you can use the '=' sign...

1. The copy constructor, if you ask for a box on that same line

```
classy x;
classy y = x; // copy constructor
```

2. Operator overload, if you already have a box when using '=';

```
classy x;
classy y; //y gets box
y=x; // equals operator
```

(See: copyVsEquals.cpp)

Overload =

What is the difference between copy and '='?

Overload =

What is the difference between copy and '='?

“copy” is a constructor, so it creates new boxes

'=' is changing the value of an existing box
(but same idea: not sharing the same address)

The “proper” way to implement '=' is
nuanced... see code comments if you care
(See: overloadEquals.cpp)

TLDR

When using “new” in a constructor, you also should make:

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

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

this

Consider the following code:

```
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()?

this

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

this points to itself

```
BadPublic* getMe()
{
    return this;
}
```

(See: thisSelfPointer.cpp)

typedef

Side note: If you want to rename types, you can do that with typedef command:

original name new synonymous name

```
typedef int DefinatelyNotAnInt;
DefinatelyNotAnInt x;
x=3;
int y = x;
cout << y;
```

(See: redefiningTypes.cpp)

If you have always been bothered that we use "double" instead of "real", go ahead and fix it!