# CS 106B

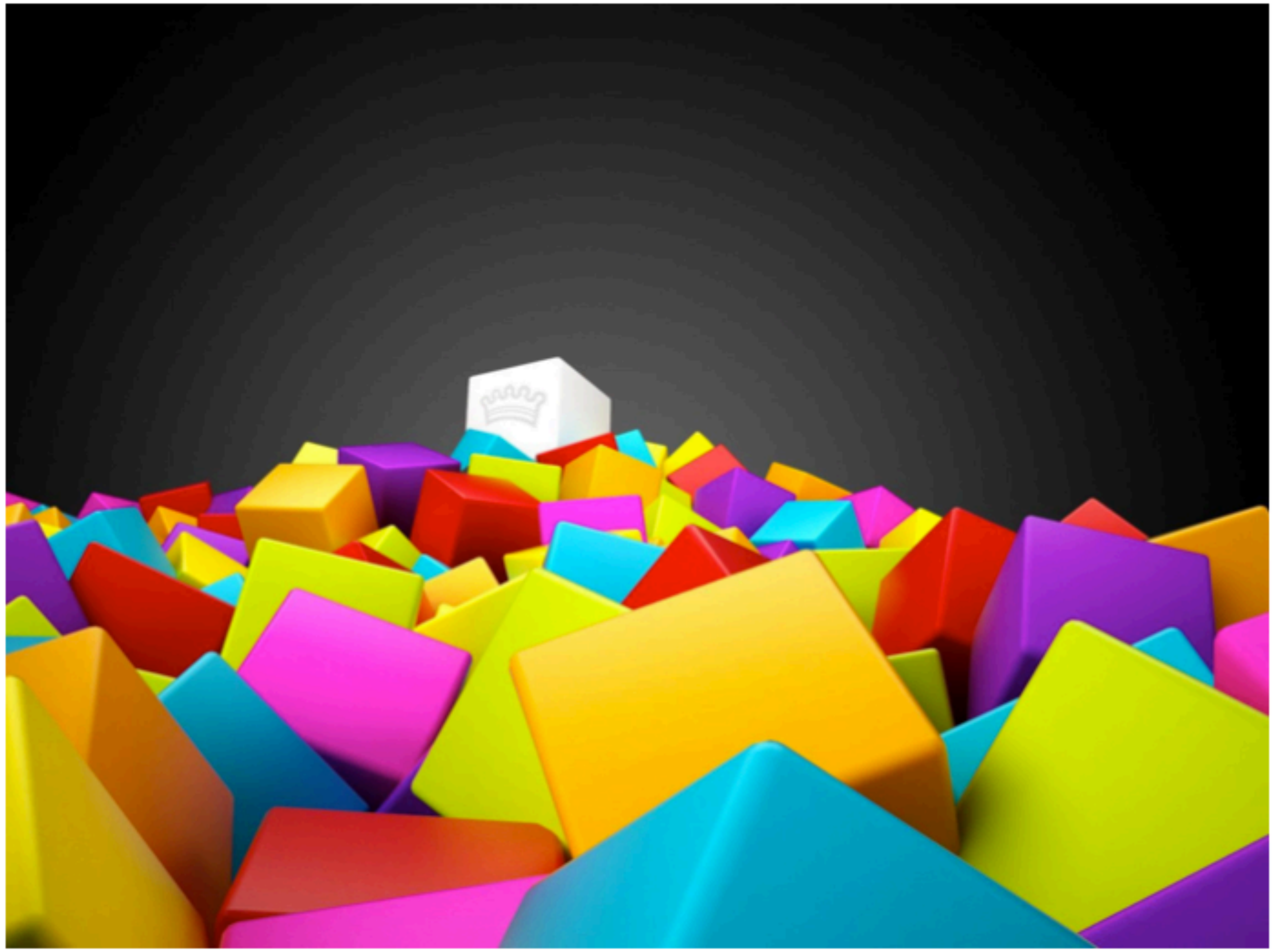
## Lecture 13: Classes

Monday, May 1, 2017

---

Programming Abstractions
Spring 2017
Stanford University
Computer Science Department

Lecturer: Chris Gregg



reading:
Programming Abstractions in C++, Chapter 6

# When it Compiles but you get a Runtime Error


Source, Reddit, Feb 8, 2017:

https://www.reddit.com/r/ProgrammerHumor/comments/5srs8b/when_it_compiles_but_you_get_a_runtime_error/

# Today's Topics

- Logistics
  - Midterm Thursday, 7-9pm
  - Review Session: Tonight 7-8:30pm, 420-040
- Classes
  - What are they and why are they important?
  - Bouncing Balls
  - Decomposition
  - Encapsulation
  - Elements of a Class, Header files
  - public / private
  - Constructors / Destructors
  - The keyword "this"
  - The Fraction Class
  - operator overloading

# Midterm

- Midterm Thursday, 7-9pm
  - Information: http://web.stanford.edu/class/cs106b/handouts/midterm.html
- Closed Book: You are allowed a 1 page back/front page of notes, plus the reference sheet (also 1 page back/front).
- Will have the option of paper *or* laptop. Make sure your laptop is charged!
- Potential topics:

1. Functions and Pass by Reference
2. Big O
3. Use of Collections (Maps, Sets, Stacks, Queues, Vectors and Grids)
4. Recursion (Definition and Fractals)
5. Recursive Exploration and Backtracking
6. Sorting
7. Memoization

# Bouncing Balls Demo

# Introduction to Classes

Remember how we said that structs are the Lunchables of the C++ world?


Structs gave us the ability to package *data* into one place:

```
struct Lunchable {
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
};
```

# Introduction to Classes

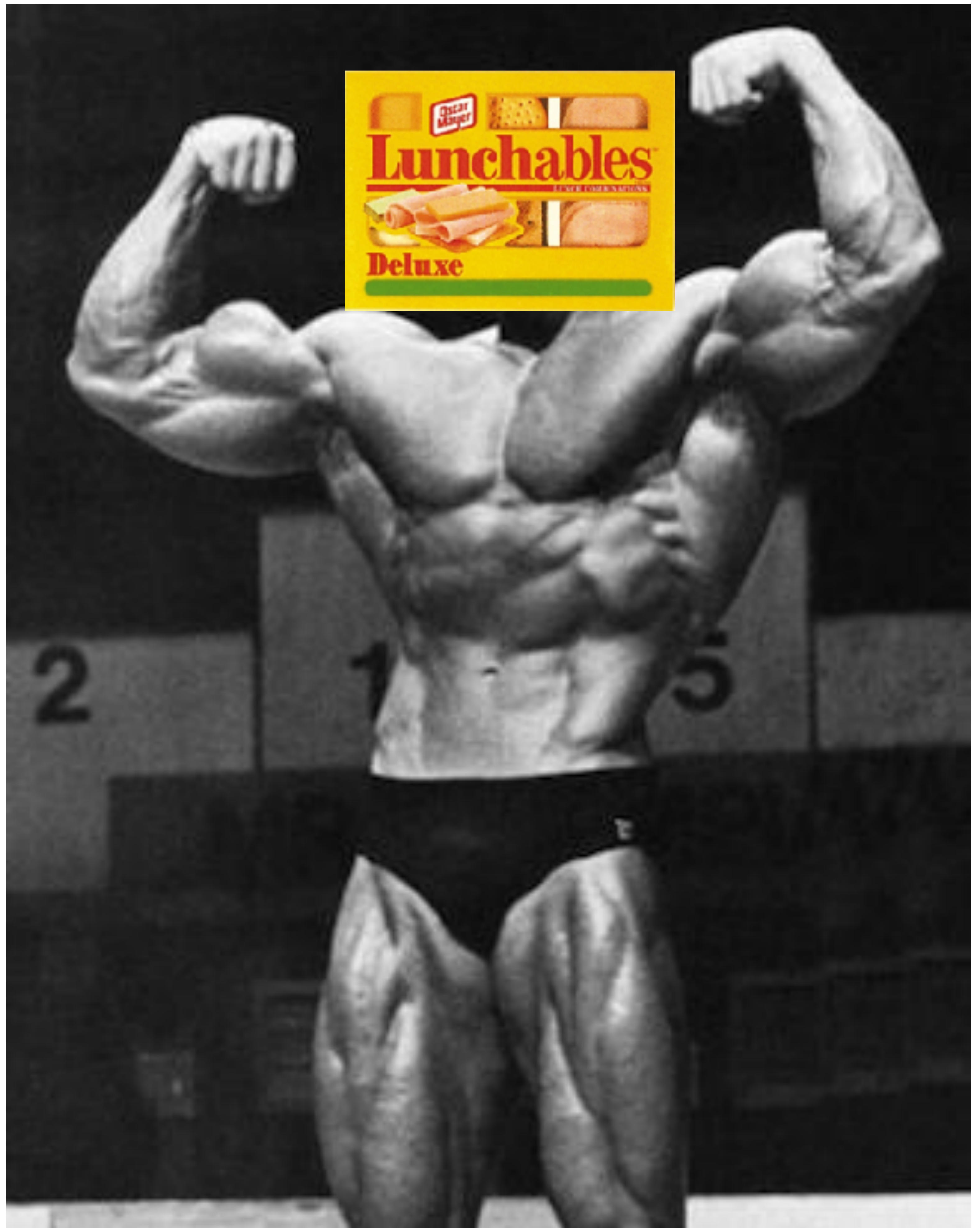
But why stop at data? If we're packaging stuff up, let's *also* package up the functions.


It would be really nice if we could do this:

```
struct Lunchable {
    string meat;
    string dessert;
    int numCrackers;
    bool hasCheese;
    int countCalories();
};
```

# Introduction to Classes

Guess what? **We can do this!** Even in structs!
(but not in C, only in C++)




Once we have the ability to package up data *and* functions into one structure, we have a super-powerful tool, called an "**object**" that *knows how to perform functions on itself, and carries around its own data.*

So-called "Object Oriented Programming" has led to the creation of most of the large programs we use today.

# The Need for New Types

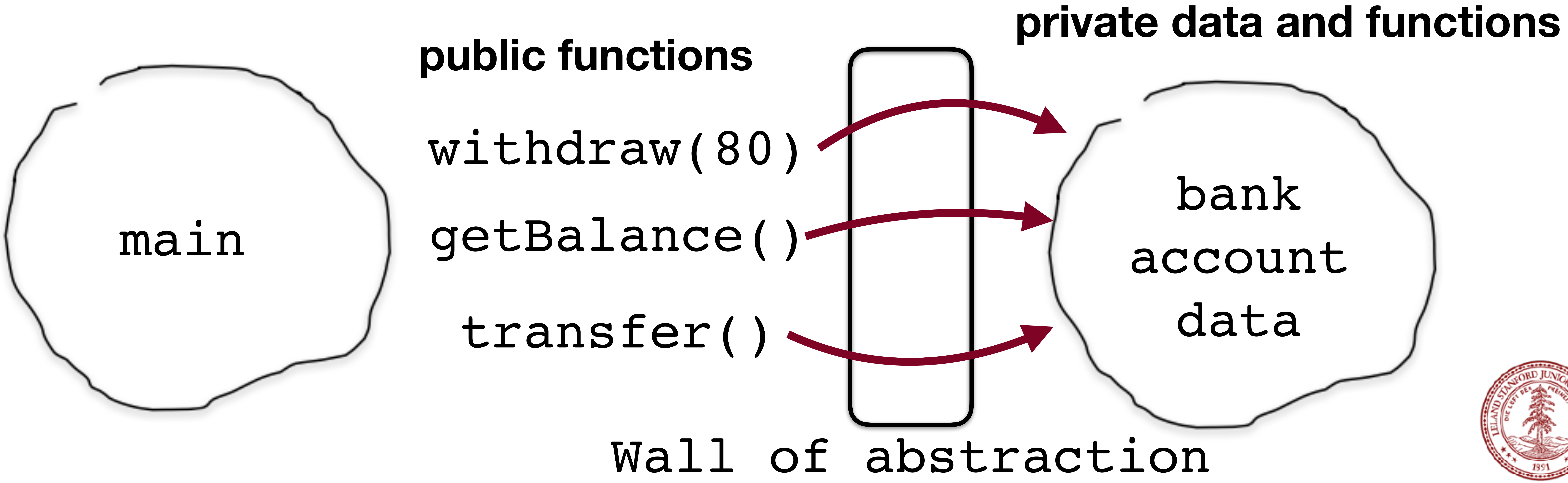
- A C++ "class" is simply a very-slightly modified struct (details to follow).
- As with structs, we sometimes want new types:
  - A calendar program might want to store information about dates, but C++ does not have a Date type.


  - A student registration system needs to store info about students, but C++ has no Student type.


  - A music synthesizer app might want to store information about users' accounts, but C++ has no Instrument type.

# Classes: Encapsulation

- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation.* A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:



# Classes: Encapsulation

- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation.* A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

**private data and functions**

```
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
}
```


Wall of abstraction

# Classes: Encapsulation

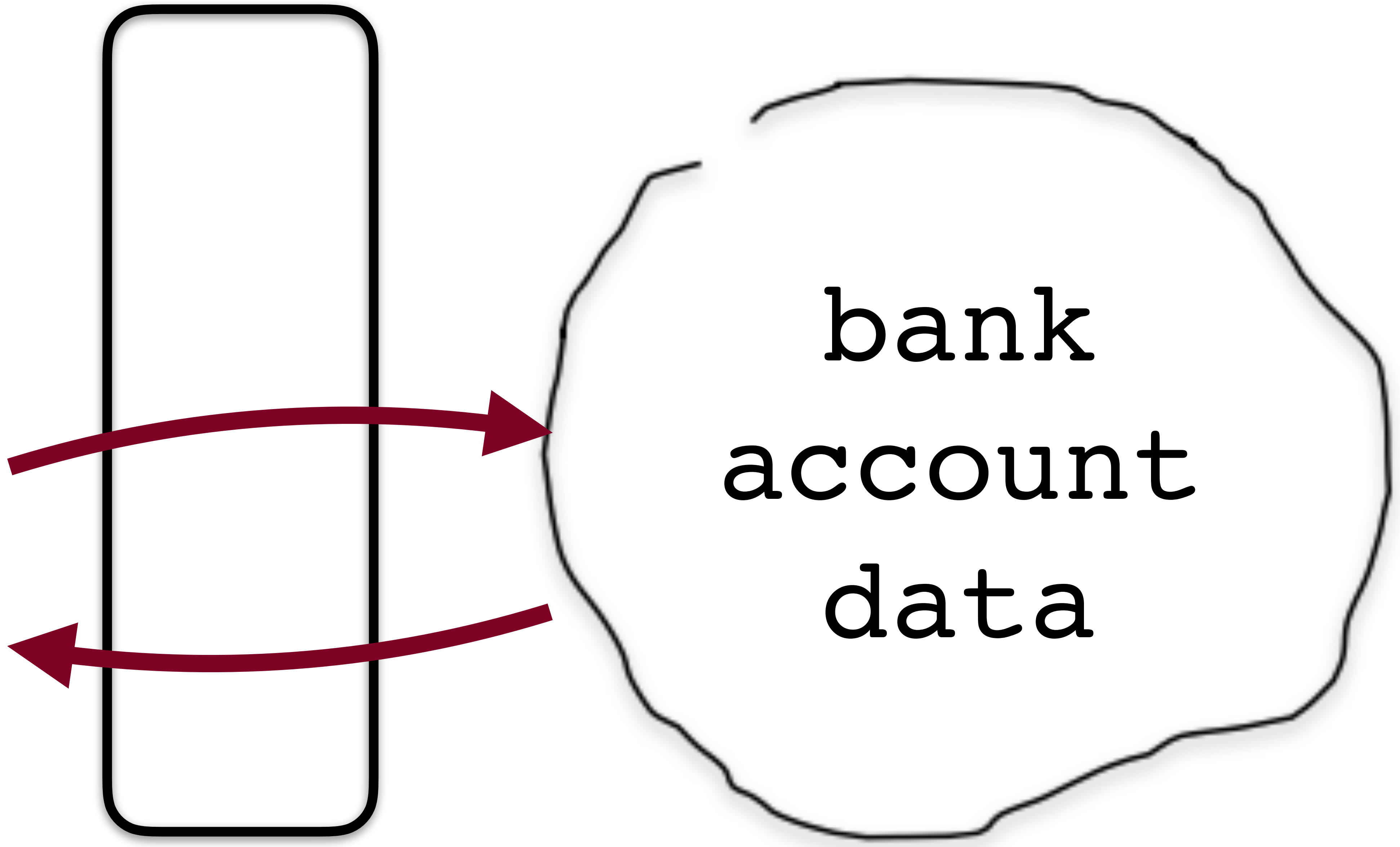
- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation.* A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
}
```

# Classes: Encapsulation

- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation.* A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
}
```



# Classes: Encapsulation

- The reason we want encapsulation is so that the end user of our class does not have direct access to the data -- we, as the class designer, control the data completely:

```
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
    checking.balance = 81.2345;
}
```

If we allowed this, our internal class data might not be in a state we can handle (e.g., too many decimal places for a monetary value)

**private data and functions**


Wall of abstraction

# Classes: Encapsulation

- So, we block the ability for someone using our class to directly touch the data, and we force them to go through our own functions:

```
int main() {
    BankAccount checking("Bob", 42);
    checking.withdraw(80);
    cout << checking.getBalance() << endl;
    checking.setBalance(81.2345);
}
```

Because we control the function, we can do a check on this, and enforce the "only two decimals" limit.

**private data and functions**


Wall of abstraction

# Elements of a Class

- **member variables**: State inside each object.
  - Also called "instance variables" or "fields"
  - Declared as private
  - Each object created has a copy of each field.
- **member functions**: Behavior that executes inside each object.
  - Also called "methods"
  - Each object created has a copy of each method.
  - The method can interact with the data inside that object.
- **constructor**: Initializes new objects as they are created.
  - Sets the initial state of each new object.
  - Often accepts parameters for the initial state of the fields.

# Class Interface Divide

| Interface | Source |
| --- | --- |
| *name*.h | *name*.cpp |
| Client reads | Implementer writes |
| Shows methods and states instance variables | Implements methods |

# Structure of a .h file

```
// classname.h
#pragma once

class ClassName {
    // class definition
};
```

This basically says, "if you see this file more than once while compiling, ignore it after the first time"
(so the compiler doesn't think you're trying to define things more than once)

# Structure of a .h file

```
// classname.h
#pragma once

class ClassName {
    // class definition
};
```

This basically says, "if you see this file more than once while compiling, ignore it after the first time"
(so the compiler doesn't think you're trying to define things more than once)

Older format, not as nice:

```
// classname.h
#ifndef _CLASSNAME_H
#define _CLASSNAME_H

class ClassName {
    // class definition
};

#endif
```

# Structure of a .h file

```
// in ClassName.h
class ClassName {
public:
    ClassName(parameters);            // constructor
    returnType func1(parameters);     // member functions
    returnType func2(parameters);     // (behavior inside
    returnType func3(parameters);     //  each object)

private:
    type var1;     // member variables
    type var2;     // (data inside each object)
    type func4(); // (private function)
};
```

# Encapsulation defined in .h

```
// in MyClass.h
class MyClass {
public:
    MyClass(parameters);                // constructor
    returnType func1(parameters);    // member functions
    returnType func2(parameters);    // (behavior inside
    returnType func3(parameters);    //  each object)

private:
    type var1;    // member variables
    type var2;    // (data inside each object)
    type func4(); // (private function)
};
```

Any class *instance* can directly use anything defined as public (but you **never** directly call a constructor):

```
MyClass a;
a.func1(arguments)
```

# Encapsulation defined in .h

```
// in MyClass.h
class MyClass {
public:
    MyClass(parameters);              // constructor
    returnType func1(parameters);    // member functions
    returnType func2(parameters);    // (behavior inside
    returnType func3(parameters);    //  each object)

private:
    type var1;    // member variables
    type var2;    // (data inside each object)
    type func4(); // (private function)
};
```

Class *instances* can **not** directly use anything defined as private:

```
MyClass a;
a.var1 = 2; // error!
```

# Constructors and (eventually) Destructors

```
// in MyClass.h
class MyClass {
public:
    MyClass(); // default constructor
    MyClass(parameters); // constructor
  ...
};
```

When a class instance is created, we say that it is "constructed":

```
string s1; // uses default constructor

string s2("I'm a string"); // uses a constructor
                           // that takes 1 string parameter

string s3 = "I'm a string"; // different! (we'll get to that)
```

# The Implicit Parameter

**implicit parameter**:
The object on which a member function is called.

- During the call `chris.withdraw(...)`, the object named `chris` is the implicit parameter.

- During the call `aaron.withdraw(...)`, the object named `aaron` is the implicit parameter.

- The member function can refer to that object's member variables.
  - We say that it executes in the context of a particular object.
  - The function can refer to the data of the object it was called on.
  - It behaves as if each object has its own copy of the member functions.

# The Keyword This

- As in Java, C++ has a this keyword to refer to the current object.
  - Syntax: `this->`*`member`*
  - Common usage: In constructor, so parameter names can match the names of the object's member variables:

```
BankAccount::BankAccount(string name, double balance) {
    this->name = name;
    this->balance = balance;
}
```

  - `this` uses `->` not `.` because it is a "pointer"; we'll discuss that later

# Let's Start an Example: The Fraction Class


- As an example of a class, we're going to define a **Fraction** class that can deal with rational numbers directly, without decimals.
- We are going to walk through the class one step at a time, demonstrating the various parts of a class as we go.

# The Fraction Class

- Questions we must answer about the Fraction class:

- What data should the class hold?
- What kinds of functions (public / private) should our class have?
- What constructors could we have?
    - What is a good value for a default fraction?

$$\frac{3}{8}+\frac{6}{4}$$

# The Fraction Class

## Class outline

```
class Fraction {
public:

    Things we want class users to see

private:
    Things we want to keep hidden
    from class users

};
```

# The Fraction Class

## Class outline

```
class Fraction {
public:



private:
    int num;   // the numerator
    int denom; // the denominator




};
```

What data would a Fraction class have?

Why is it private?

# The Fraction Class

```
class Fraction {
public:
     void add(Fraction f);
     void mult(Fraction f);
     double decimal();
     int getNum();
     int getDenom();
     friend ostream& operator<<
        (ostream& out, Fraction &frac);



private:
     int num; // the numerator
     int denom; // the denominator


};
```

What functions should a fraction class be able to do?

Why are they public?

What is this???

# The Fraction Class

```
class Fraction {
public:
     void add(Fraction f);
     void mult(Fraction f);
     double decimal();
     int getNum();
     int getDenom();
     friend ostream& operator<<
        (ostream& out, Fraction &frac);


private:
    int num; // the numerator
    int denom; // the denominator


};
```

What is this???

This defines an operator "overload" to make it possible to use the "<<" operator with cout.

We will write this function in a few minutes.

# The Fraction Class

```
class Fraction {
public:
    Fraction();
    Fraction(int num,int denom);
    void add(Fraction f);
    void mult(Fraction f);
    double decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<
        (ostream& out, Fraction &frac);

private:
    int num; // the numerator
    int denom; // the denominator


};
```

We need to *construct* the class when it is called.

What should a "default" fraction look like?

1 / 1 probably makes the most sense (why not 0/0?)

Should we let the user create an initial fraction, e.g., 3/4?

# The Fraction Class

```
class Fraction {
public:
    Fraction();
    Fraction(int num,int denom);
    void add(Fraction f);
    void mult(Fraction f);
    double decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<
       (ostream& out, Fraction &frac);

private:
    int num; // the numerator
    int denom; // the denominator
    void reduce(); // reduce the fraction
    int gcd(int u, int v);
};
```

Any other functions?

What about reduce?
(necessary for multiplication)

Reduce needs gcd()...

# The Fraction Class

```
#pragma once
#include<ostream>
using namespace std;

class Fraction {
public:
    Fraction();
    Fraction(int num,int denom);
    void add(Fraction f);
    void mult(Fraction f);
    double decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<
        (ostream& out, Fraction &frac);
private:
    int num;   // the numerator
    int denom; // the denominator
    void reduce(); // reduce the fraction
    int gcd(int u, int v);
};
```

Last, but not least...

# The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

Fraction::Fraction()
{




}
```

# The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

Fraction::Fraction()
{
}
```

This tells the compiler what class we are creating. The double-colon is called the "scope resolution operator" because it helps the compiler resolve the scope of the function.

# The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

Fraction::Fraction()
{

    num   = 1;
    denom = 1;

}
```

**Pretty simple! We are just setting our two class variables to default values.**

# The Fraction Class

We also have an *overloaded* constructor that takes in two values that the user sets. It is called as follows:

```
// create a
// 1/2 fraction
Fraction fracA(1,2);

// create a
// 4/6 fraction
Fraction fracB(4,6);
```

```
// purpose: an overloaded constructor
//             to create a custom fraction
//            that immediately gets reduced
// arguments: an int numerator
//              and an int denominator
Fraction::Fraction(int num,int denom)
{




}
```

# The Fraction Class

We also have an *overloaded* constructor that takes in two values that the user sets. It is called as follows:

```
// create a
// 1/2 fraction
Fraction fracA(1,2);

// create a
// 4/6 fraction
Fraction fracB(4,6);
```

```
// purpose: an overloaded constructor
//           to create a custom fraction
//          that immediately gets reduced
// arguments: an int numerator
//             and an int denominator
Fraction::Fraction(int num, int denom)
{

    this->num = num;
    this->denom = denom;

   // reduce in case we were given
   // an unreduced fraction
   reduce();


}
```

# The Fraction Class

Let's write some more functions...

```
// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3
```

```
// purpose: to multiply another fraction
// with this one with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(Fraction other)
{



}
```

# The Fraction Class

Let's write some more functions...

```
// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3
```

```
// purpose: to multiply another fraction
// with this one with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(Fraction other)
{
        // multiplies a Fraction
        // with this Fraction
        num *= other.num;
        denom *= other.denom;

        // reduce the fraction
        reduce();
}
```

# The Fraction Class

Let's write some more functions...

```
// get the decimal value
Fraction fracA(1,2);
double f =
fracA.decimal();
cout << f << endl;

output:
0.5
```

```
// purpose: To return a decimal
// value of our fraction
// arguments: None
// return value: the decimal
//                   value of this fraction
double Fraction::decimal()
{



}
```

# The Fraction Class

Let's write some more functions...

```
// get the decimal value
Fraction fracA(1,2);
double f = fracA.decimal();
cout << f << endl;

output:
0.5
```

```
// purpose: To return a decimal
// value of our fraction
// arguments: None
// return value: the decimal
//                value of this fraction
double Fraction::decimal()
{
        // returns the decimal
        // value of our fraction
        return (double)num / denom;
}
```

# The Fraction Class: reduce()

```
void Fraction::reduce() {
    // reduce the fraction to lowest terms
    // find the greatest common divisor
    int frac_gcd = gcd(num,denom);

    // reduce by dividing num and denom
    // by the gcd
    num = num / frac_gcd;
    denom = denom / frac_gcd;
}
```

# The Fraction Class: gcd() — nice recursive function

```
int Fraction::gcd(int u, int v) {
    if (v != 0) {
        return gcd(v,u%v);
    }
    else {
        return u;
    }
}
```

# The Fraction Class: overloading <<

Yes, this syntax is a bit strange.

Basically, we are telling the compiler how to cout our Fraction.
You can do something very similar for Boggle.

```
// purpose: To overload the << operator
// for use with cout
// arguments: a reference to an outstream and the
//            fraction we are using
// return value: a reference to the outstream
ostream& operator<<(ostream& out, Fraction &frac) {
    out << frac.num << "/" << frac.denom;
    return out;
}
```

# References and Advanced Reading

- **Advanced Reading**
  - Overloading the assignment operator:
    - http://www.learncpp.com/cpp-tutorial/9-14-overloading-the-assignment-operator/
  - Constructors and Destructors:
    - http://www.cprogramming.com/tutorial/constructor_destructor_ordering.html

- **References:**
  - https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm
  - http://www.cprogramming.com/tutorial/lesson12.html

# Extra Slides

# The Copy Constructor

## The Copy Constructor

```
Vector<int> a;
a.add(0);
a.add(1);
a.add(1);
a.add(2);
Vector<int> b = a;
     or
Vector<int> b(a); // b gets constructed
                  // with the same elements as b
```

**This doesn't work automatically!**

## The assignment overload:

```
Vector<int> a;
a.add(0);
a.add(1);
Vector<int> b;
b.add(8);
b = a; // a gets copied into b
```