Designing Abstractions

Apply to Section Lead!

Application Online
Due Thursday, February 14th, 11:59PM

Black in CS Presents...

Hack 101 is an event that aims to prepare students for Treehacks2019/hackathons!
All are welcome!

HACK101: HOW TO HACKATHON

Wednesday, February 13th
6 - 7:20pm
GATES 415

A WHOLE DINNER WILL BE SERVED
RSVP HERE (HTTP://BIT.LY/RSVPHACK101)
QUESTIONS/LOST? EMAIL MAMADOU@STANFORD.EDU

GTGTC Presents…

Want to inspire high school girls to code?

Apply to be a mentor for **Girls Teaching Girls To Code**'s annual Code Camp on Saturday, April 6!

As a mentor, you will help teach 200+ high school girls programming basics and then lead (in small groups) an exploratory workshop that you design. In the past, our students have overwhelmingly cited the mentors as their favorite part of the day.

Fill out a short application **HERE** by Saturday, February 9th!

Feel free to check out our website at http://girlsteachinggirlstocode.org/, or email gtgtc.stanford@gmail.com with any questions.

Alternate Exams

- As a reminder, our midterm is Tuesday, February 19th from 7:00PM – 10:00PM.
  - We'll talk about that more next time.
- If you have OAE accommodations or otherwise can't make that exam, you should have heard back from Kate with information.
- If not, we don't know that you need an alternate exam, and you should contact us ASAP.

Onward and Forward!

Designing Abstractions

ab·strac·tion

[...]

freedom from
representational
qualities in art

Source: Google

ab·strac·tion

[...]

the process of considering something independently of its associations, attributes, or concrete accompaniments.

Source: Google

Vector Map

Set Queue

***In Plato's Cave, No. 4* by Robert Motherwell**

Building a rich vocabulary of abstractions makes it possible to ***model and solve*** a wider class of problems.

Question One:

How do we create new abstractions to model ideas not precisely captured by the standard container types?

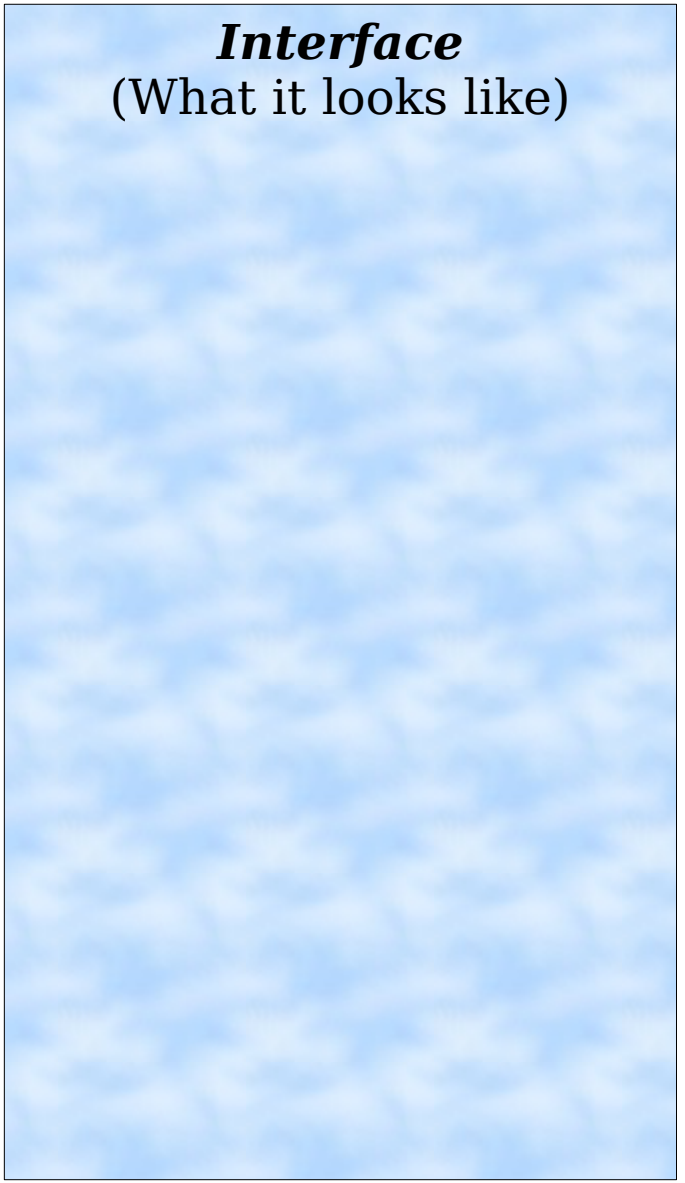
Question Two:

How do the abstractions we've been using so far work, and how can we use that knowledge to build richer abstractions?

Classes in C++

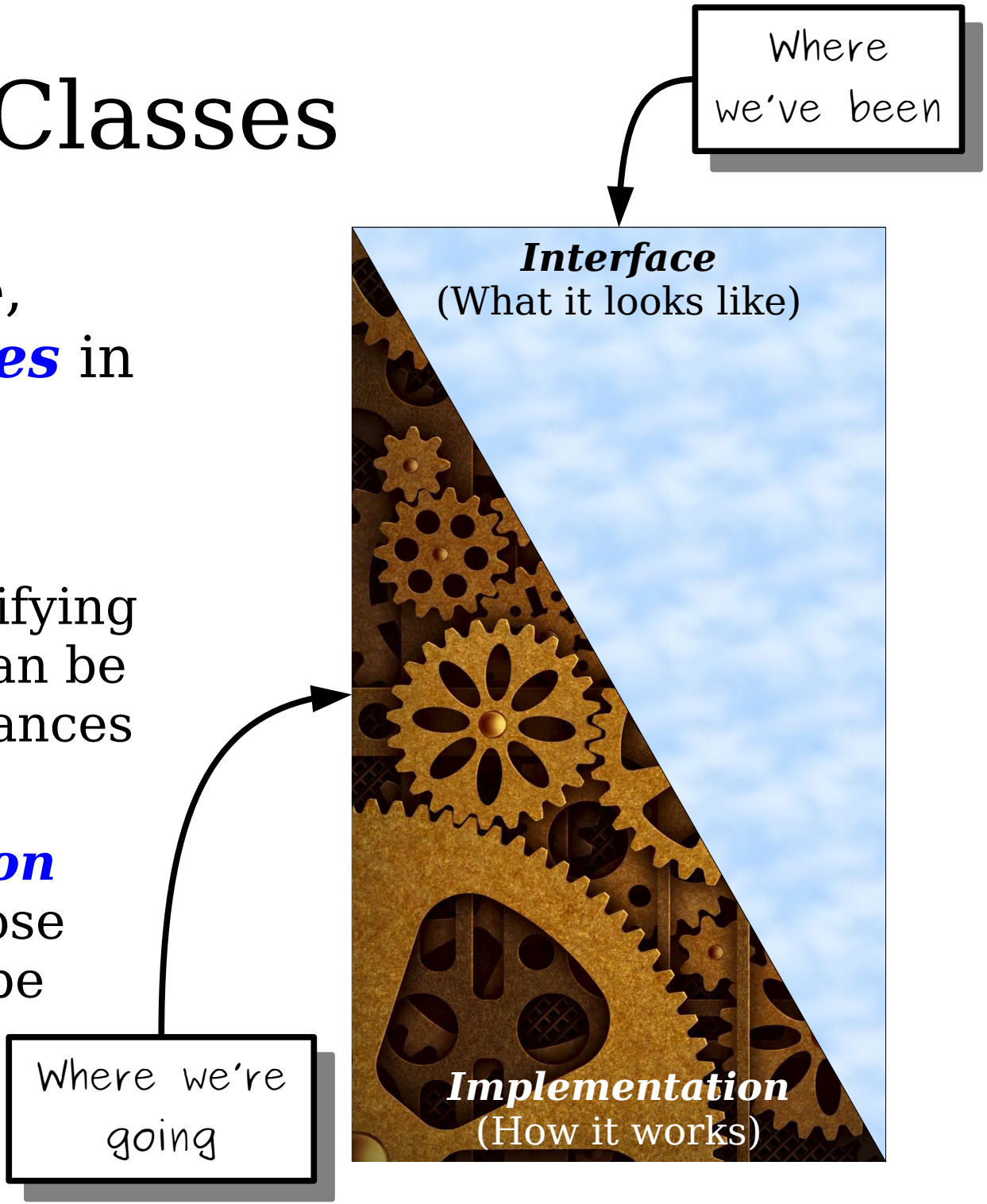
Classes

- `Vector`, `Stack`, `Queue`, `Map`, etc. are ***classes*** in C++.
- Classes contain
  - an ***interface*** specifying what operations can be performed on instances of the class.



Classes

- `Vector`, `Stack`, `Queue`, `Map`, etc. are ***classes*** in C++.
- Classes contain
  - an ***interface*** specifying what operations can be performed on instances of the class, and
  - an ***implementation*** specifying how those operations are to be performed.



Creating our own Classes

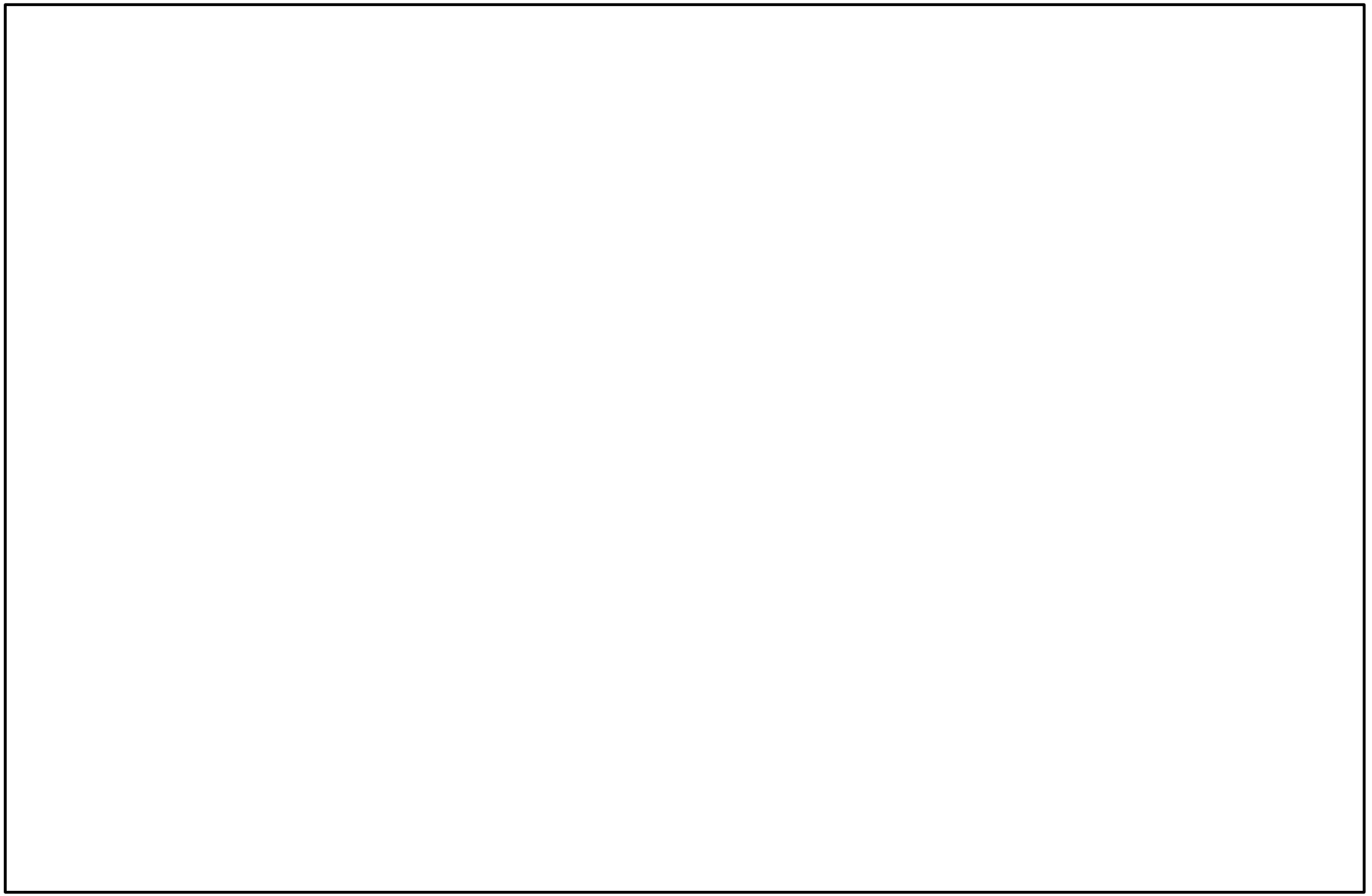
Random Bags

- A ***random bag*** is a data structure similar to a stack or queue. It supports two operations:
  - ***add***, which puts an element into the random bag, and
  - ***remove random***, which returns and removes a random element from the bag.
- Random bags have a number of applications:
  - Simpler: Shuffling a deck of cards.
  - More advanced: generating artwork, designing mazes, and training self-driving cars to park and change lanes. *(Curious how? Come talk to me after class!)*
- Let's go create our own custom `RandomBag` type!

Classes in C++

- Defining a class in C++ (typically) requires two steps:
  - Create a ***header file*** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.
  - Create an ***implementation file*** (typically suffixed with `.cpp`) that contains the implementation of the class.
- Clients of the class can then include the header file to use the class.

What’s in a Header?



What's in a Header?

```
#ifndef RandomBag_Included
#define RandomBag_Included



#endif
```

This boilerplate code is called an ***include guard***. It's used to make sure weird things don't happen if you include the same header twice.

Curious how it works? Come talk to me after class!

What's in a Header?

```
#ifndef RandomBag_Included
#define RandomBag_Included



class RandomBag {




};

#endif
```

This is a ***class definition***. We're creating a new class called `RandomBag`. Like a `struct`, this defines the name of a new type that we can use in our programs.

What's in a Header?

```
#ifndef RandomBag_Included
#define RandomBag_Included



class RandomBag {









};

#endif
```

Don't forget to add this semicolon! You'll get some Hairy Scary Compiler Errors if you leave it out.

What's in a Header?

```
#ifndef RandomBag_Included
#define RandomBag_Included



class RandomBag {
public:









private:

};

#endif
```

What's in a Header?

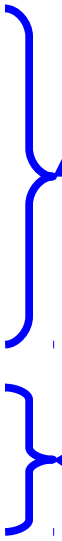
```
#ifndef RandomBag_Included
#define RandomBag_Included


class RandomBag {
public:


private:

};

#endif
```



The ***public interface*** specifics what functions you can call on objects of this type.

Think things like the `Vector`'s `.add()` function or the `TokenScanner`'s `.nextToken()`.

The ***private implementation*** contains information that objects of the class type will need in order to do their job properly. This is invisible to people using the class.

What's in a Header?

```
#ifndef RandomBag_Included
#define RandomBag_Included



class RandomBag {
public:
   void add(int value);
   int  removeRandom();



private:

};

#endif
```

These are ***member functions*** of the `RandomBag` class. They're functions you can call on objects of the type `RandomBag`.

All member functions need to be declared in the class definition. We'll implement them in our .cpp file.

What's in a Header?

```
#ifndef RandomBag_Included
#define RandomBag_Included

#include "vector.h"

class RandomBag {
public:
   void add(int value);
   int  removeRandom();



private:
   Vector<int> elems;
};

#endif
```

This is a ***data member*** of the class. This tells us how the class is implemented. Internally, we're going to store a `Vector<int>` holding all the elements. The only code that can access or touch this `Vector` is the `RandomBag` implementation.

What's in a Header?

```
#ifndef RandomBag_Included
#define RandomBag_Included

#include "vector.h"

class RandomBag {
public:
   void add(int value);
   int  removeRandom();




private:
   Vector<int> elems;
};

#endif
```

```
#include "RandomBag.h"
```

If we're going to implement the `RandomBag` type, the .cpp file needs to have the class definition available. All implementation files need to include the relevant headers.

```
class RandomBag {
public:
   void add(int value);
   int  removeRandom();



private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"


void RandomBag::add(int value) {

}
```

The syntax

`RandomBag::add`

means "the `add` function defined inside of `RandomBag`." The `::` operator is called the ***scope resolution operation*** in C++ and is used to say where to look for things.

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();



private:
   Vector<int> elems;
};
```

```
#include "RandomBag.h"


void RandomBag::add(int value) {

}
```

If we had written something like this instead, then the compiler would think we were just making a free function named add that has nothing to do with `RandomBag`'s version of `add`. That's an easy mistake to make!

```
class RandomBag {
public:
   void add(int value);
   int  removeRandom();



private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"


void RandomBag::add(int value) {
   elems += value;
}
```

We don't need to say what `elems` is. The compiler knows we're inside `RandomBag`, and so it knows that this means "the current `RandomBag`'s collection of elements."

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();



private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (elems.isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, elems.size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}
```

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();



private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (elems.isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, elems.size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}
```

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();

   int  size();
   bool isEmpty();

private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (elems.isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, elems.size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}

int RandomBag::size() {
   return elems.size();
}
```

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();

   int  size();
   bool isEmpty();

private:
   Vector<int> elems;
};
```

```
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (elems.isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, elems.size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}

int RandomBag::size() {
   return elems.size();
}

bool RandomBag::isEmpty() {
   return size() == 0;
}
```

This code calls our own `size()` function. The class implementation can use the public interface.

```
mBag {

(int value);
oveRandom();

   int  size();
   bool isEmpty();

private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}

int RandomBag::size() {
   return elems.size();
}

bool RandomBag::isEmpty() {
   return size() == 0;
}
```

That's such a good idea, let's do this up here as well.

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();

   int  size();
   bool isEmpty();

private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}

int RandomBag::size() {
   return elems.size();
}

bool RandomBag::isEmpty() {
   return size() == 0;
}
```

```cpp
class Random
public:
   void add(int value);
   int  removeRandom();

   int  size() const;
   bool isEmpty() const;

private:
   Vector<int> elems;
};
```

This use of the `const` keyword means "I promise that this function doesn't change the object."

```
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}

int RandomBag::size() const {
   return elems.size();
}

bool RandomBag::isEmpty() const {
   return size() == 0;
}
```

We have to remember to put it here too as well!

```
class RandomBag {
public:
   void add(int value);
   int  removeRandom();

   int  size() const;
   bool isEmpty() const;

private:
   Vector<int> elems;
};
```

```cpp
#include "RandomBag.h"
#include "random.h"

void RandomBag::add(int value) {
   elems += value;
}

int RandomBag::removeRandom() {
   if (isEmpty()) {
      error("Aaaaahhh!");
   }

   int index = randomInteger(0, size() - 1);
   int result = elems[index];
   elems.remove(index);

   return result;
}

int RandomBag::size() const {
   return elems.size();
}

bool RandomBag::isEmpty() const {
   return size() == 0;
}
```

```cpp
class RandomBag {
public:
   void add(int value);
   int  removeRandom();

   int  size() const;
   bool isEmpty() const;

private:
   Vector<int> elems;
};
```

Your Action Items

- ***Read Chapter 6 of the textbook.***
  - There's a ton of goodies in there about class design that we'll talk about later on.
- ***Keep working on Assignment 4.***
  - If you're following our suggested timetable, you should be done with Doctors Without Orders by the end of the evening.
  - Start working on Disaster Planning over the weekend.

Next Time

- ***Dynamic Allocation***
  - Where does memory come from?
- ***Constructors and Destructors***
  - Taking things out and putting them away.
- ***Implementing the Stack***
  - Peering into our tools!