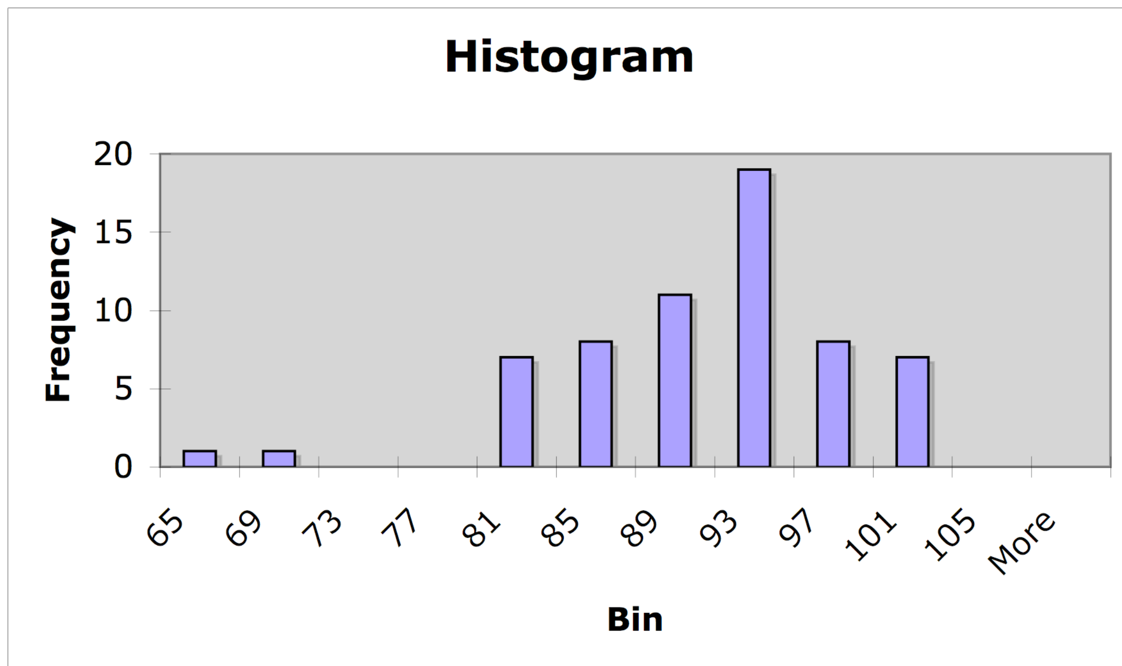
Final Exam Answers

We were very pleased with the results of the final exam. There were quite a few tricky questions on there and the class still pulled off an average grade of **88.8**. There was only one perfect score, but several came close.

Here is the histogram:



There are comments and solutions for each problem below. If you have questions about your exam, take a look at the solution first, then we'll be glad to answer any further questions.

Part 1: True or False (15 points) **Class Average: 12.7/15)**

Indicate whether each statement is **true** or **false**. There is no penalty for guessing, and you do not need to justify your answers.

1) Since it never flushes the output stream buffer, the following program will *never* output anything.

```
int main(int argc, char** argv)
{
    cout << "What happened to Andrae?";
}
```

Answer 1: False. One of the conditions for flushing is that the stream gets destructed, which will happen here when the program exits. *Also*, nobody caught the *Project Runway* reference. I guess I'm the only one whose wife makes them watch that awful show.

2) At a particular time, a file stream can be both "not good" and "not bad." In other words, `stream.good()` could be *false* even if `stream.bad()` is *false*.
Answer 2: True. For example, when an input stream reaches end of file, it is !good() because you can't read any further but it's !bad() because there has not been a fatal error.

3) If a function provides a throw list but doesn't include any exceptions in the list, then it can safely throw *any* exception. In other words, the following function can throw any exception without invoking the unexpected handler:

```
void tomServo(int i) throw()
{
  [etc]
}
```

Answer 3: False. An empty throw list means that you won't throw anything. If you omit the throw list entirely, that means you can throw anything.

4) Destructors should never be `virtual` because you don't really override a destructor when you create a subclass.

Answer 4: False. Destructors should always be virtual!

5) In the following class, `foo` will be constructed before `bar`:

```
#include "Mucus.h"

class Marvin {
     public:
          Marvin() : bar(2), foo(3) {}

          Mucus foo;
          Mucus bar;
};
```

Answer 5: True. Objects are constructed in their list order in the class definition, not their order on the initializer list.

6) Passing a data member as an argument in the initializer list is dangerous because the data member will not have been properly initialized. For example, the following constructor is potentially error-prone:

```
#include "Game.h"
#include "Player.h"
class Taboo : public Game {
     public:
          Taboo() : Game(mPlayer) {}

     protected:
          Player mPlayer;
};
```

Answer 6: True. Remember the order of construction. The superclass is constructed before any data members, so it's a bad idea to pass data members to the superclass.

7) `static` methods cannot be `virtual`. Thus, they are not polymorphic.

Answer 7: True. I was surprised how many people missed this. It's important! static isn't polymorphic – a static method call is bound to the compile time type!

8) Consider the following two classes:

```
class Super {
 public:
   virtual void foo() {
          cout << "Super::foo()" << endl;
   }
   virtual void foo(int i) {
          cout << "Super::foo(" << i << ")" << endl;
   }
};

class Sub : public Super {
 public:
   virtual void foo() {
          cout << "Sub::foo()" << endl;
   }
};
```

Since `Sub` doesn't override the one-argument version of `Super::foo(int)`, the following code will call `Super`'s version, outputting "`Super::foo(3)`":

```
     Sub mySub;
     mySub.foo(3);
```

Answer 8: False. I think a lot of people correctly detected that something bad was happening here, but forgot what. When you override only one of the overloaded versions, the other one gets hidden. You can still call it through a cast, but not from a subclass instance. The code above doesn't compile!

9) The principle of *Refactoring* acknowledges that over time, code gets convoluted and unmaintainable, and needs to be reorganized.

Answer 9: True. I referred to this as *cruft* during lecture. It's all the gunk that accumulates over time that you want to go in and rewrite/reorganize.

10) The software development methodology known as *The Waterfall Model* is an iterative approach that breaks a problem down into a number of smaller individual development cycles.

Answer 10: False. *The Waterfall Method* is **not** iterative. That's its main weakness. Most modern approaches are iterative and break a task into smaller subtasks.

11) *Extreme Programming* says that you should start with designs that are as general as possible, so that they can be reused for different programs.

Answer 11: False. A lot of people answered true, probably because a programmer's natural instinct is that things should be generalized. However, XP claims that you should write the simplest possible solution. Remember the mantra – *no speculative generality!*

12) With a few minor exceptions, the C++ language is a superset of the C language. In other words, most C code will compile and run just fine in C++.

Answer 12: True. This goes way back to lecture 1 and the *all your favorite stuff from C is still present in C++* claim. I think some people were tripped up thinking that there were **no** exceptions, but that's not true. C++ has new reserved words, like "class" which C didn't have. So a C program with a variable named "class" won't compile in C++, but *most* C code will.

13) A function that takes an object of type `Foo` can be called on an object of type `Bar` as long as `Foo` has an implicit `Bar` constructor, as shown below:

```
class Foo {
     public:
          Foo() {}
          Foo(const Bar& inBar) {}
};

void doFoo(Foo inFoo);

int main()
{
     Bar myBar;
     doFoo(myBar);
}
```

Answer 13: True. Remember that a constructor that takes another class (something that looks like a copy constructor, but for another type) is called automatically by the compiler when necessary. Part 3 makes use of this feature.

14) If you overload `operator+`, you *must* return an object of the same type as your parameters. Otherwise, it won't compile. For example, the following code will *not* compile:

```
// I claim that this won't compile
Bar operator+(const Foo& lhs, const Foo& rhs);
```

Answer 14: False. Remember that return types are generally ignored as far as method signatures are concerned. You can return whatever type you want. A MrPibb plus a RedVine can give you a CrazyDelicious.

15) XML defines a syntax for representing data, but the meaning of the data varies from application to application.

Answer 15: True. Recall that XML is just a file format, just a structure. It has no meaning without a specific application to interpret it.

Part 2: Short Answer (20 points) Class Average: 18.76/20

1) One of the problems with C++ exceptions is that they can result in memory leaks as the stack frames are unwound. There are, however, at least three ways that you can protect against exceptions causing memory leaks.

Name two ways that you can author your functions to protect against exceptions causing memory leaks. Note that they don't have to necessary be *technical* in nature:

Lots of possible answers here. The ones we talked about in class were:

1. Don't use dynamic memory. Keep everything on the stack.
2. Use smart pointers or auto_ptr
3. Catch, clean up, and rethrow all exceptions

There were some other acceptable answers, but these were the most common ones. We also accepted "Don't use exceptions." Some people's answers were way too vague or didn't actually solve the problem. For example, "Use custom exception types" is not a solution, though it may be a good idea for other reasons.

2) Say you've been hired by Microsoft to write a compiler for C*, a new language that's just like C++, except that Microsoft can add whatever new language features it wants without the trouble of going through a standards body.

One of your co-workers proposes a solution to the problem of C++ exceptions causing memory leaks (see previous question). She suggests that when an exception is thrown in C*, the language is smart enough to call delete or delete[] (as appropriate) on every pointer it encounters as it unwinds the stack. In other words, stack-based objects are destructed just like in C++, but pointers are deleted. Describe briefly why this is a bad idea. **Only give one reason**:

To get full credit, you had to only give one answer, as instructed. Acceptable answers pointed out one of the following possibilities:

1. The variable is a pointer, but it's pointing to stack memory and shouldn't be deleted.

2. The variable is a pointer, but the current stack frame doesn't own it, eventually resulting in double deletion.

3. Double deletion occurs for some other reason (stack frame manually cleans up, etc.)

3) The following class, a simple proxy around `SpreadsheetCell`, has a serious bug. `getStringCell()` blindly downcasts mCell even if the underlying cell isn't a `StringSCell`. Rewrite the `getStringCell()` method so that it will return `NULL` if `mCell` isn't a `StringSCell`. **Note:** A full credit answer will *only* change the `getStringCell()` method. However, if you can't find a way to do that, you can change other parts of the code and get partial credit.

```
class CellProxy {
  public:
    CellProxy (bool inIsDoubleCell) {
      if (inIsDoubleCell) {
        mCell = new DoubleSCell();
      } else {
        mCell = new StringSCell();
      }
    }
    StringSCell* getStringCell() {
      return (StringSCell*)mCell;
    }
  protected:
    SCell* mCell;
};
```

Write your answer to #3 here:

```
StringSCell* getStringCell() {
	return dynamic_cast<StringSCell*>(mCell);
}
```

The solution above takes advantage of the fact that dynamic_cast will automatically return NULL if the cast is invalid. This wasn't a test of syntax so as long as you came up with something pretty close to the way dynamic_cast works, you got full credit. If you had the right idea, but your syntax was way off, you got partial credit.

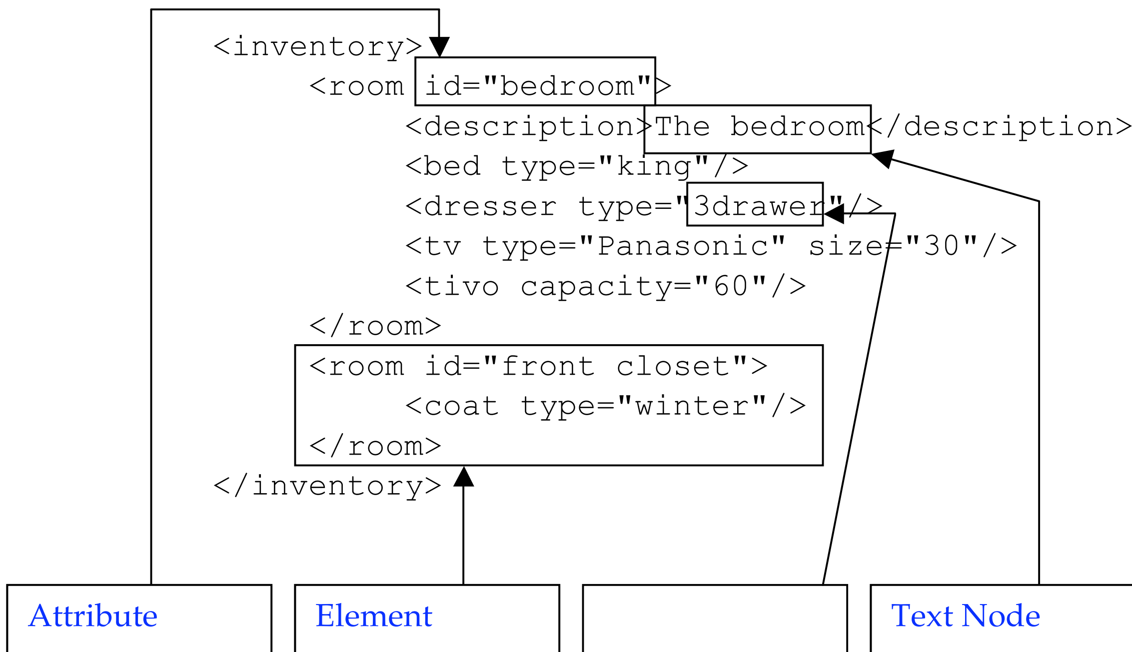
You could also solve this problem using typeid, the other RTTI feature we discussed:

```
	if (typeid(mCell) == typeid(StringSCell*)) return (StringSCell)mCell;
	return null;
```

Again, you got full credit if your syntax was pretty close.

If you forgot about RTTI, you could do this by setting a flag in the constructor for partial credit.

4) Place the following XML node types in the appropriate boxes: **Element, Attribute, Text Node.** Since there are more boxes than terms, one box will remain empty.



5) In class we discussed *double dispatch*, a technique that allows you to use polymorphism on two objects. We can actually extend double dispatch to three objects using the same technique.

Here's the situation: due to soaring real estate costs, the San Francisco Zoo has to consolidate its animals so they now have three animals in each cage. In this question, you'll implement the `canLiveWith()` method on `Animal`. **`canLiveWith(a, b)` returns `true` if the `Animal` in question is not eaten by either of its potential roommates**.

For example, a `Lion` is not eaten by a `Chicken` or a `Horse`, so the following calls will return **true:**

```
myLion.canLiveWith(myChicken, myHorse); // true
myLion.canLiveWith(myHorse, myChicken); // true
```

On the other hand, a `Chicken` is eaten by a `Lion`, so chickens will refuse to live with lions. Both of the following lines will return **false:**

```
myChicken.canLiveWith(myLion, myOtherLion); // false
myChicken.canLiveWith(myMouse, myLion); // false
```

Using triple dispatch, implement the `canLiveWith()` method for `Chicken` by filling in the class definition below. The implementations of `eatenBy()` have been provided for you – you'll want to use them in your implementations.

```
class Animal {
  public:
    virtual bool canLiveWith(const Animal& a,
                             const Animal& b) = 0;
    virtual bool eatenBy(const Lion& inLion) const = 0;
    virtual bool eatenBy(const Chicken& inChick) const = 0;
    virtual bool eatenBy(const Mouse& inMouse) const = 0;
    virtual bool eatenBy(const Horse& inHorse) const = 0;
};
```

```
class Chicken : public Animal {
  public:
    virtual bool canLiveWith(const Animal& a,
                             const Animal& b) {


          return (!a.eats(*this) &&
                  !b.eats(*this));



    }

    // Here's a reminder of how double dispatch works:
    virtual bool eats(const Animal& a) {
        return a.eatenBy(*this);
    }

    bool eatenBy(const Lion& l) const { return true; }
    bool eatenBy(const Chicken& c) const {return false; }
    bool eatenBy(const Mouse& m) const { return false; }
    bool eatenBy(const Horse& h) const { return false; }
};

// Assume that Lion, Horse, and Mouse have been implemented
// already.
```

This was a long setup for a really short question. I was pleased that most of the class was able to extend double dispatch into a third dimension without any problem.

Part 3: Object Tracing (15 points) Class Average: 13.44/15
What will this program print? Your answer goes on the next page.

```
class Enemy {
  public:
    Enemy() { cout << "E ctor" << endl; }
    Enemy(int i) { cout << "E ctor " << i << endl; }
    Enemy(const Enemy& src) {cout << "E copy ctor"<< endl;}
    Enemy& operator=(const Enemy& rhs) {cout<<"E="<<endl;}
    virtual ~Enemy() { cout << "E dtor" << endl; }

    void hornet(int i=7) const {  // Not virtual!
             cout << "E::hornet " << i << endl;
    }
};

class Scott : public Enemy {
  public:
    Scott() : Enemy(1) { cout << "S ctor" << endl; }
    Scott& operator=(const Scott& rhs) {cout<<"S="<<endl;}
    virtual ~Scott() { cout << "S dtor" << endl; }

    void hornet(int i=7) const {
             cout<<"S::hornet " << i << endl;
    }
};

void foo(const Enemy& inKlep) {
  Enemy theEnemy;
  inKlep.hornet(2);
}

int main(int argc, char** argv) {
  Scott kleper;
  Enemy blah = kleper;
  foo(kleper);
  foo(3);
  cout << "Done!" << endl; // Don't forget me!
}
```

Write your answer to Part 3 here:

Most of the class heeded my comment and noticed that since hornet() wasn't virtual, the call in foo() would go to Enemy's version, not Scott's.

```
E ctor 1          // Construct Enemy part of kleper
S ctor            // Construct Scott part of kleper
E copy ctor       // Construct blah as copy of kleper
                  // call foo(kleper)
E ctor            // Construct theEnemy;
E::hornet 2       // call non-virtual hornet() on E
E dtor            // Destruct theEnemy
E ctor 3          // create temporary E using implicit ctor
                  // call foo(temporary E)
E ctor            // Construct theEnemy
E::hornet 2       // call non-virtual hornet() on E
E dtor            // Destruct theEnemy
E dtor            // Destruct temporary E
Done!             // Output Done message
E dtor            // Destruct blah
S dtor            // Destruct Scott part of kleper
E dtor            // Destruct Enemy part of kleper
```

Part 4: STL Tracing (15 points) **Class Average: 13.34/15**

1) The following lines of code use STL algorithms and methods to modify an STL container. In each box, write what the previous line would print out. If the line would cause an error, indicate so.

```
void printVector(const vector<char>& inVec)
{
  cout << "[";
  for (size_t i = 0; i < inVec.size(); i++) {
    cout << inVec[i] << ",";
  }
  cout << "]" << endl;
}

void nmakr(char& ref) {
    ref = 'n';
}

class nfindr {
  public:
    bool operator()(char ch) { return (ch == 'n'); }
};

int main() {
  vector<char> v(5, 's');
  printVector(v);
```

```
[s,s,s,s,s,]
```

```
  v.push_back('g');
  printVector(v);
```

```
[s,s,s,s,s,g,]
```

```
for_each(v.begin()+1, v.end()-1, nmakr);
printVector(v);
```

```
[s,n,n,n,n,g]
```

```
v.push_back('g');
printVector(v);
```

```
[s,n,n,n,n,g,g,]
```

```
// Reminder: accumulate() adds all elements in the range
// together, starting with the third arg ('a')
char ch = accumulate(v.begin(), v.begin(), 'a');

cout << ch << endl;
```

```
a    // of course we wouldn't expect you to memorize ASCII!
```

```
printVector(v);
```

```
[s,n,n,n,n,g,g,]
```

```
// Reminder: count_if() returns the number of elements in
// the range that satisfy the condition in the third arg.
int i = count_if(v.begin(), v.end(),
              not1(bind2nd(greater_equal<char>(), 'n')));

cout << i << endl;
```

```
2
```

```
    vector<char>::iterator loc;
    // Reminder: find() returns the position of the first hit
    loc = find(v.begin(), v.end(), 'p');
    bool b;
    if (loc == v.end()) b = true; else b = false;
    cout << boolalpha << b << endl;
```

```
true
```

```
    nfindr finder;
    loc = find_if(v.begin(), v.end(), finder);
    if (loc == v.begin() + 2) b = true; else b = false;

    cout << noboolalpha << b << endl;
```

```
0
```

```
    vector<char> findme;
    findme.push_back('n');
    findme.push_back('g');

    // Reminder: search() finds the range specified in args 3
    // and 4 within the range specified in args 1 and 2
    loc = search(v.begin(), v.end(), findme.begin(),
                                     findme.end());

    cout << *loc << endl;
```

```
n
```

```
}
```

The most common problem was forgetting about the boolalpha and noboolalpha manipulators.

Part 5: Implementation (30 points)
One of the most difficult parts of starting a company is finding the right group of investors to fund your venture. So many people out there with so much money to throw away – what's an entrepreneur to do?

You're about to solve the problem by writing some C++ classes to simulate the different types of investors and their various idiosyncrasies. As you may already know, there are really only four types of investors in the world:

Investor Type: VC
A VC is a traditional venture capitalist. They will invest in *anything* except:

- They will not invest in a deal where the company value is < $5,000,000.
- They will not invest in a deal where their investment is < $1,000,000.
- They will not invest in a deal that has more than 3 other investors.

Investor Type: Top-Tier VC
A Top-Tier VC has an established name and slightly more rigorous standards:

- They will not invest in a deal where the company value is < $8,000,000.
- They will not invest in a deal that **any** other VC's are investing in.
- Other than that, they have the same rules as a regular VC.

Investor Type: Angel
An angel investor is a wealthy individual with some money to throw around:

- They will never invest more than $500,000.
- Other than that, they'll invest in anything.

Investor Type: Rich Uncle
Your rich uncle has told you since you were four that he'd invest in you:

- He will invest up to $100,000.
- He will only invest if there is at least one other investor involved.

1) Draw a simple class hierarchy diagram (boxes and arrows) showing the relationship between Investor (the base class), VC, TopTierVC, Angel, and RichUncle. Don't introduce any classes other than those five.

The only requirement here is that you showed TopTierVC as a subclass of VC. There was a clear relationship stated between these two classes. Even if you were of the opinion that they were significantly different in practice, you're ignoring the fact that a TTVC is **defined** as a VC with some extra rules. Imagine that the VC rules change – the TTVC should change automatically. If you didn't make it a subclass but you justified your decision, it was a minor deduction.

2) For each subclass (omitting Investor), implement the following method:

```
/**
  * likesDeal()
  *
  * Determines if an investor will invest given a
  * particular scenario.
  *
  * @return true if the given investor will invest.
  *
  * @param totalValue the total company value (in $)
  * @param investment this investor's contribution (in $)
  * @param inOthers the other potential investors
  */
virtual bool likesDeal(int totalValue, int myPortion,
                       vector<Investor*> inOthers);
```

Use this page and the next page (if necessary) for your answer.

It turns out that the entire class was able to turn logic rules in English into code. The only complications came with the TopTierVC implementation:

```
bool TopTierVC::likesDeal(int totalValue, int myPortion,
                         vector<Investor*> inOthers)
{
     if (totalValue < 8000000) return false;

     for (size_t i = 0; i < inOthers.size(); i++) {
          if (typeid(inOthers[i] == typeid(VC*)) {
               return false;
          }
     }

     return VC::likesDeal(totalValue, myPortion, inOthers);
}
```

There were two main mistakes that we saw. First, many people failed to realize that TTVC's won't invest if there is another **VC** in the investment. They just checked inOthers.size(), which could be any investor type. Second, many people reimplemented the VC rules inside of TopTierVC instead of "chaining up" to the superclass. If you had TTVC as a subclass of VC in your diagram, there was no reason not to do this – think of the case where the VC definition needs to change.

Continue your answer to Part 5, Number 2 here:

Again, if you didn't remember the RTTI syntax, but you were close enough, you got full credit. Worth mentioning – some people reverted to Java on this problem and used instanceof for RTTI and super.likesDeal() instead of VC::likesDeal(). These are errors that a compiler would catch so we didn't take off points.

If you didn't remember RTTI, there was another solution. You could add an "isVC()" method that the two VC classes override to return true.

}

3) Fill in the implementation of willDealWork(). This function returns true if the proposed deal is acceptable to all of the given investors. The function takes the total size of the deal, a vector of investors, and a vector of ints representing the corresponding investor's proposed contribution.

You may find the `insert()` method of `vector` useful in your solution:

```
     void vector<T>::insert(Iterator insertPoint,
                            Iterator start, Iterator end);
```

```
bool willDealWork(int totalValue,
                  vector<Investor*> inVestors,
                  vector<int> inAmounts)
{
     for (size_t i = 0; i < inVestors.size(); i++) {
        vector<Investor*> others;
        others.insert(others.end(), inVestors.begin(),
                      inVestors.begin() + i);
        others.insert(others.end(), inVestors.begin() + i +
                      1, inVestors.end());
        if (!inVestors[i]->likesDeal(total,
                 amountsInDollars[i], others)) return false;
     }
     return true;
}
```

The challenge here is that the likesDeal() method on an individual investor takes an array of the *other* investors, not including themselves. So you need to somehow remove the current investor from the vector that you pass in.

}