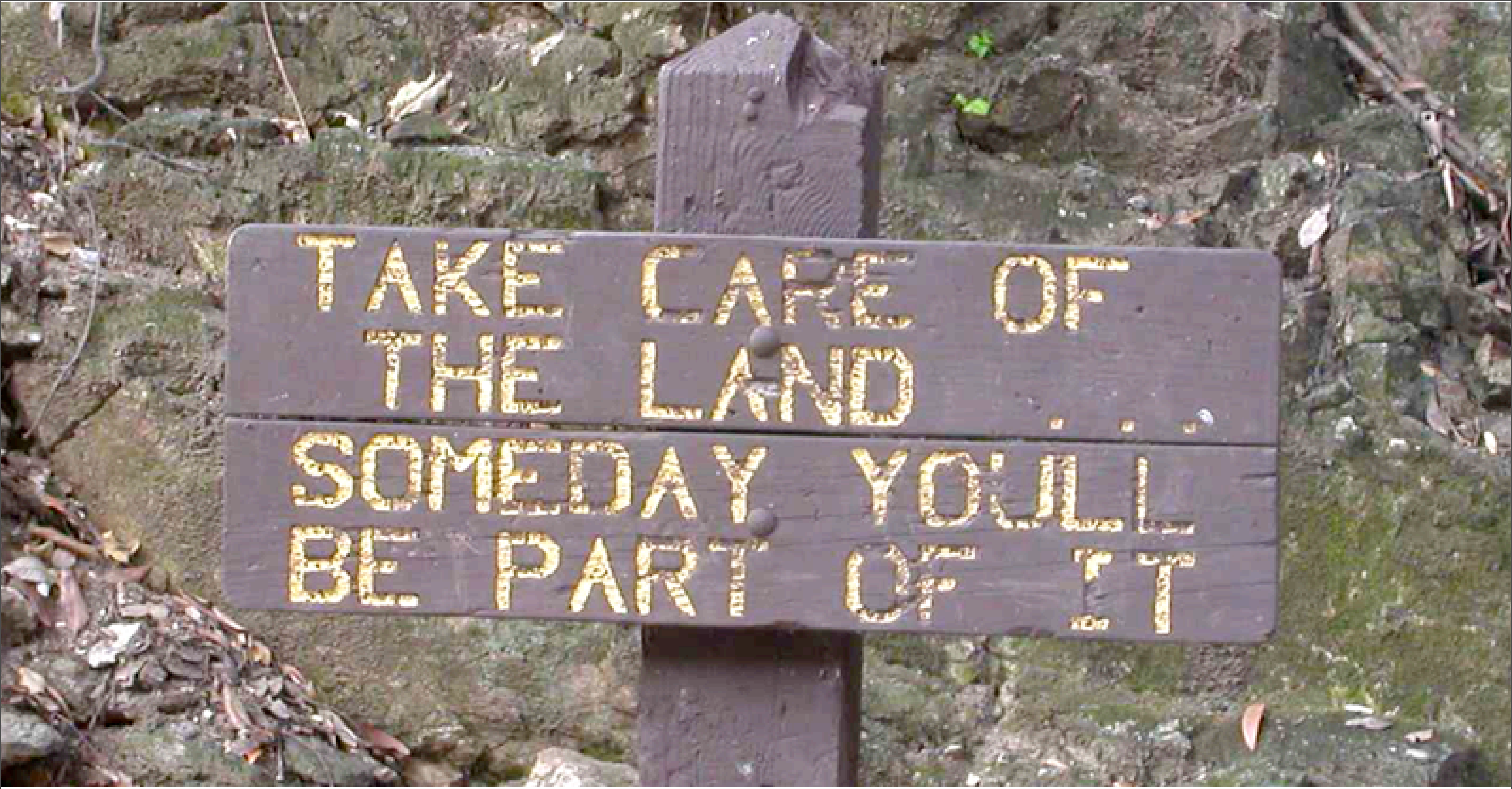


MORE INHERITANCE STUFF

Some Review

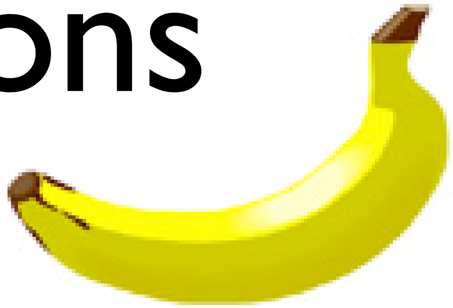
```
int var = 10;

if( AA )
    var = 45;
else
{
    if( BB )
        var = 20;
    else
        var = 16;
}
```

- What's the syntax for a switch statement?
- Rewrite this code using the ternary operator.
- How does C++ execute the following command?

```
if( test1 || test2 )
    doStuff();
```

Review Questions

```
class Pet
{
   public:
      Pet();
      ~Pet();
      void play();
      void makeNoise();
   protected:
      string name;
   private:
      string owner;
};

class Dog : public Pet
{
   public:
      Dog();
      void makeNoise();
};

int main()
{
     Dog woofy;
}
```



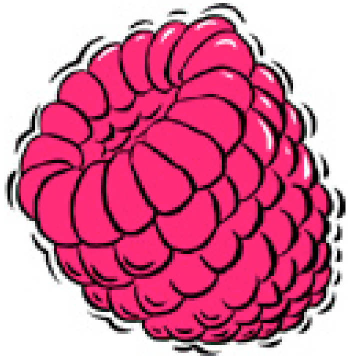
- What is Dog's relationship to Pet?
- What member variables/ functions of Pet are inherited by Dog?
- What kind of class is woofy? Are we dealing with one class or two classes?

```
class Pet
{
    public:
        Pet();
        ~Pet();
        void play();
        void makeNoise();
    protected:
        string name;
    private:
        string owner;
};

class Dog : public Pet
{
    public:
        Dog();
        void makeNoise();
};

int main()
{
      Dog woofy;
}
```

```
class Dog : public Pet
{
  public:
        Pet();
        ~Pet();
        Dog();
        void play();
        void makeNoise();
  private:
        string name;

   (hidden):
        string owner;
};
```

- Dog is a *single class*
- However, Dog has inherited a lot of code from Pet!
- Not all of it is accessible

An alternative...

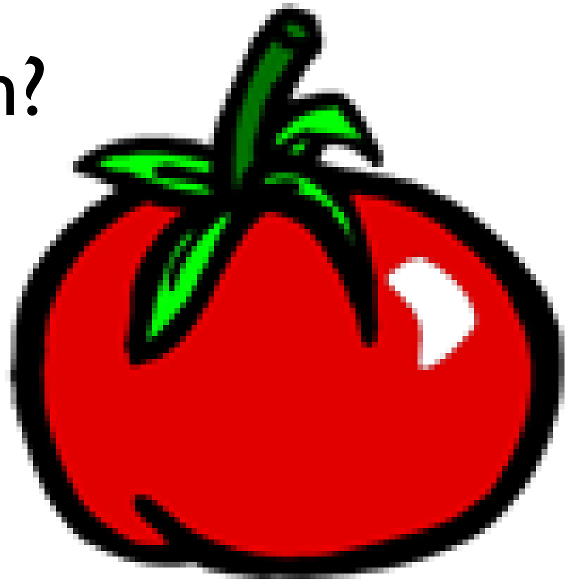


- An alternative to inheritance is called **composition** (or **aggregation**)
- Composition is when one class contains instances of another instead of inheriting from it
  - We use this when inheritance doesn't make sense but we'd still like to have one class be able to use bits of another class

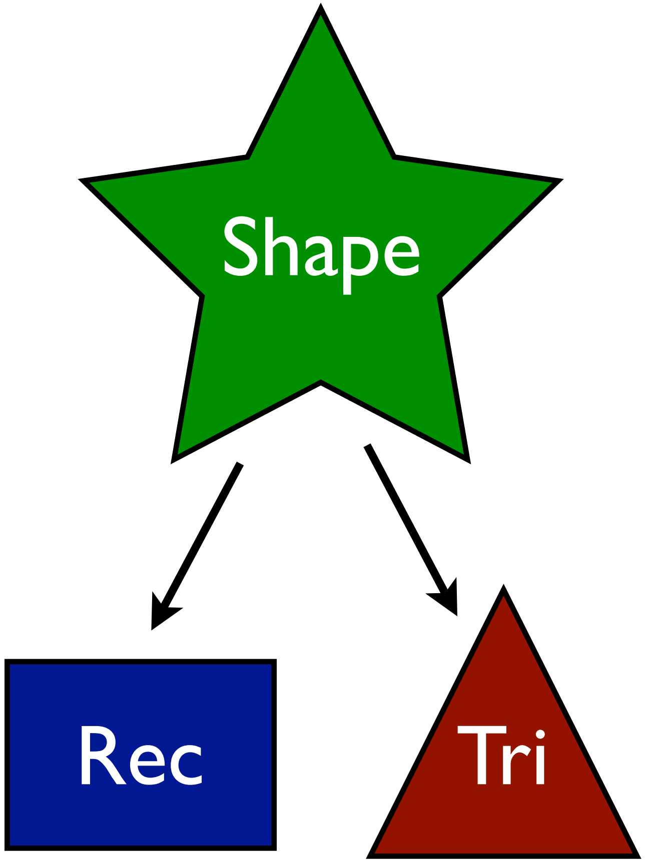
```
class Car
{
   // ...

   private:
      CarEngine e;
}
```

Some Questions

- A common mistake is to try and use inheritance where it doesn't make sense
- When should we use inheritance?
- When should we use composition?
- Is one better than the other?



Object Types




```
Triangle tri;
```

- tri is of type **Triangle**
- We can also say that tri is a **Shape**, too!
- Triangle is derived from Shape, so everything in Shape will also be in every instance of Triangle

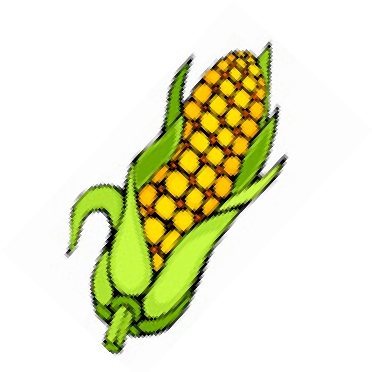
More Object Types

- Since a Triangle is of type Shape, we can refer to it as if it were a Shape.
- This works especially well with pointers:

```
Shape* ptr = new Triangle;
```

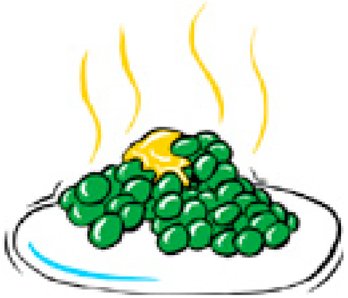
  - What type is **ptr**?
  - What kind of thing is **ptr** pointing to?

Even More Object Types



```
Shape* ptr = new Triangle;
```

- ptr is a *Shape* pointer. Given a pointer, we *can't* tell exactly what kind of thing it's pointing to!
- Only that it's either a Shape, or something derived from Shape
- So it could be Shape, Triangle, Rectangle, Circle, Octrahedron... *any* class derived from shape!



Why this is awesome:

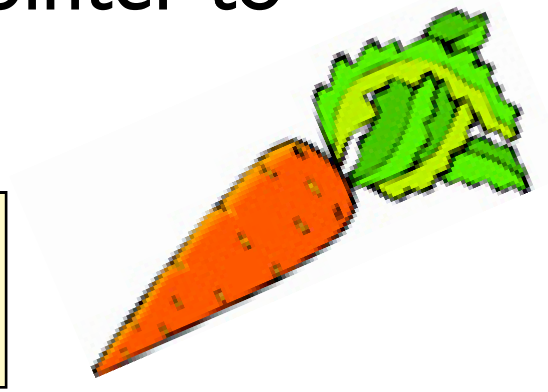
- It lets us treat all kinds of Shapes exactly the same way
- No need to know what type a pointer is actually pointing to - this is called **polymorphism**
- Can only use Shape’s interface

```
void printShapeArea( Shape* s )
{
    cout << “This shape’s area is:”
         << s->area() << endl;
}
```

What type does s point to? Triangle? Rectangle? Circle? Dodecahedron? Polygon? As long as it is derived from Shape, we don’t have to care!

For example:

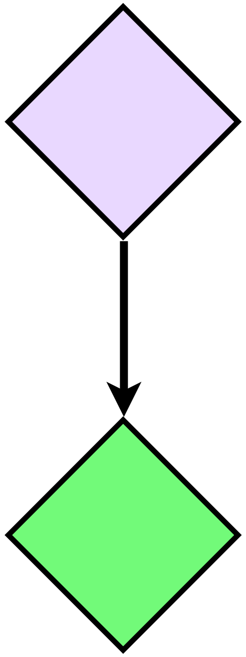
- Here we're defining an array of pointer-to-Shapes:

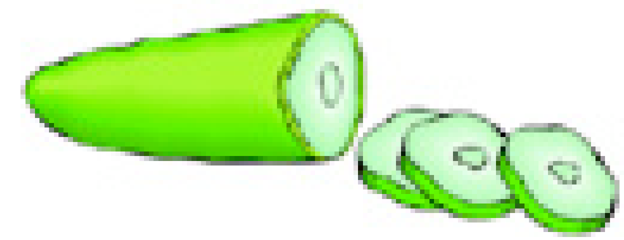


```
Shape* array[10];
```

- Each element in array can be pointing to a different kind of Shape
- They all have a common interface though, so we can treat them all identically

An Issue





FarmAnimal
int weight;

MooCow
void chewCud();
bool hungry;

- We can transparently treat MooCow as a FarmAnimal (this is what polymorphism means!)
- So we can pass MooCow into a function that accepts FarmAnimal.

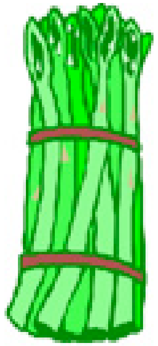
```
void printWeight( FarmAnimal animal )
{
    cout << animal.weight;
}

int main()
{
    MooCow cow;
    printWeight( cow );
}
```

let's talk about this...

- How is cow being passed?
- What type is cow?
- What type does printWeight accept?

Object Slicing

- For this to work, a MooCow must be converted to a FarmAnimal



- The compiler takes all the FarmAnimal bits and leaves behind all the MooCow bits!
- This is called **object slicing**
- It's generally bad.
- To prevent it, use pointers or references instead!

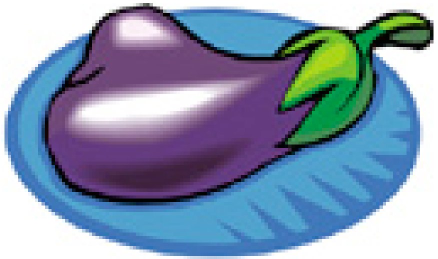
```
void printWeight( FarmAnimal animal )
{
    cout << animal.weight;
}

int main()
{
    MooCow cow;
    printWeight( cow );
}
```

Question

```
class Pet
{
   public:
      void makeNoise()
      {
         cout << "(nothing)";
      }
};

class Cat : public Pet
{
   public:
      void makeNoise()
      {
         cout << "MEOW!";
      }
};
```

- **Pet** has a makeNoise function
- Pet's implementation of makeNoise() isn't good enough for **Cat**, so Cat *overrides* it
- Does this code snippet compile? What's the output?

```
Cat animal;
animal.makeNoise();
```



Question, cont.

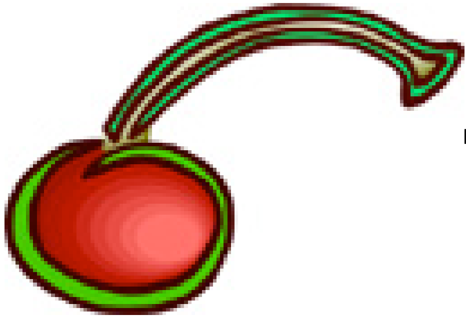
```
class Pet
{
   public:
      void makeNoise()
      {
         cout << "(nothing)";
      }
};

class Cat : public Pet
{
   public:
      void makeNoise()
      {
         cout << "MEOW!";
      }
};
```

- How about this one?

```
Cat* animal = new Cat;
animal->makeNoise();
```

- ... and this one?

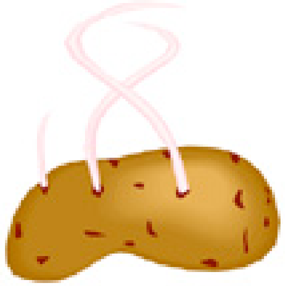
```
Pet* animal = new Cat;
animal->makeNoise();
```



The Problem

- C++ uses ***static type checking*** (early binding) - types are checked at compile time, not run-time (late binding)!
- A major design goal of C++: produce code that runs as quickly as possible
- What's happening here:
  - We have a pointer of type Pet
  - Pet has a method called makeNoise
  - Therefore, Pet::makeNoise is called

```
Pet* animal = new Cat;
animal->makeNoise();
```



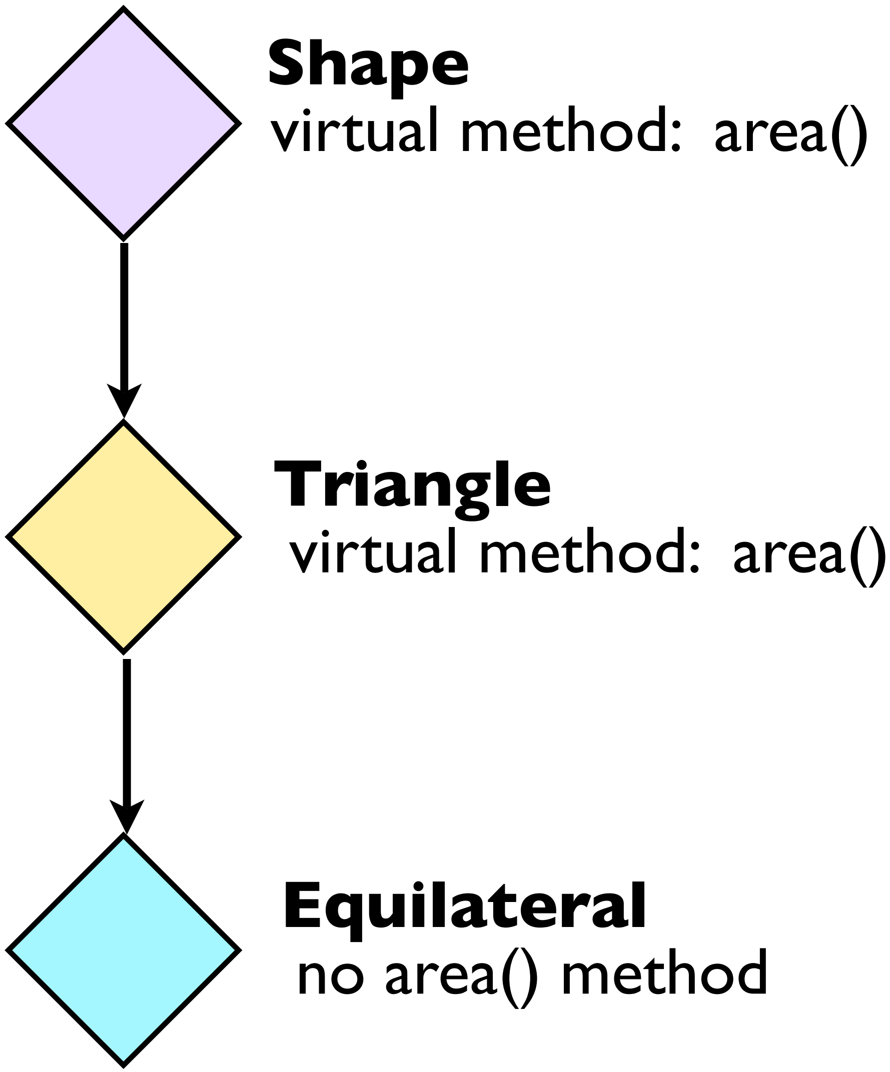
So then:

```
class Pet
{
    public:
        void makeNoise()
        {
            cout << "(nothing)";
        }
};

class Cat : public Pet
{
    public:
        void makeNoise()
        {
            cout << "MEOW!";
        }
};
```

```
Pet* animal = new Cat;
animal->makeNoise();
```

- The compiler sees animal as a **Pet**, instead of a **Cat**
- Therefore Pet::makeNoise() is getting called instead of Cat::makeNoise()
- How do we tell the compiler to figure out the correct version of makeNoise to call?

Virtual Methods



- To do this, we can mark a method as **virtual**.
- The compiler will use run-time type identification to call the *most specific* version of the method that it can!

what version of area() gets called?

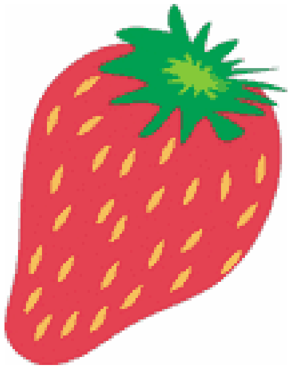
```
Shape* s = new Equilateral;
s->area();
```

Virtual: How-to

```
class Pet
{
  public:
    virtual void makeNoise()
    {
       cout << "(nothing)";
    }
};

class Cat : public Pet
{
  public:
    void makeNoise()
    {
       cout << "MEOW!";
    }
};
```

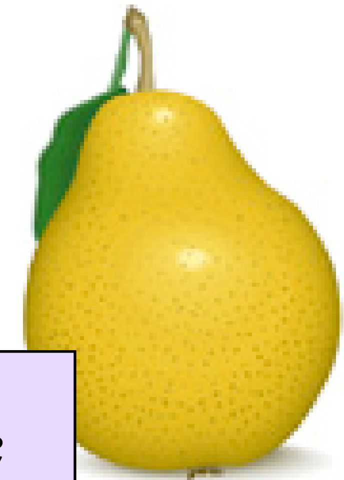
- To declare a virtual method, stick the keyword **virtual** before its return type
- This automatically makes every overridden version of the method virtual too
- Only works in one direction: marking Cat::makeNoise as virtual doesn't make Pet::makeNoise virtual!

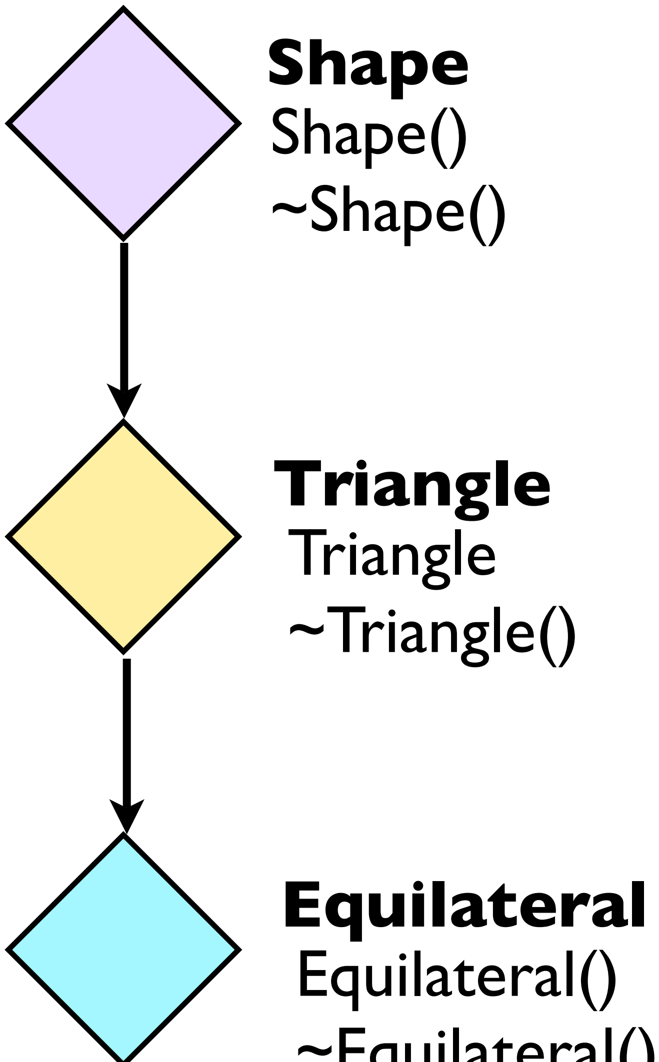
Virtual Rules



- Virtual methods are slightly slower than non-virtual methods (why?)
- Static methods can't be virtual, and virtual methods can't be static
- One way to make this a non-issue: make every base-class method virtual. (why does this work?)
- If in doubt: make your methods virtual

Inheritance

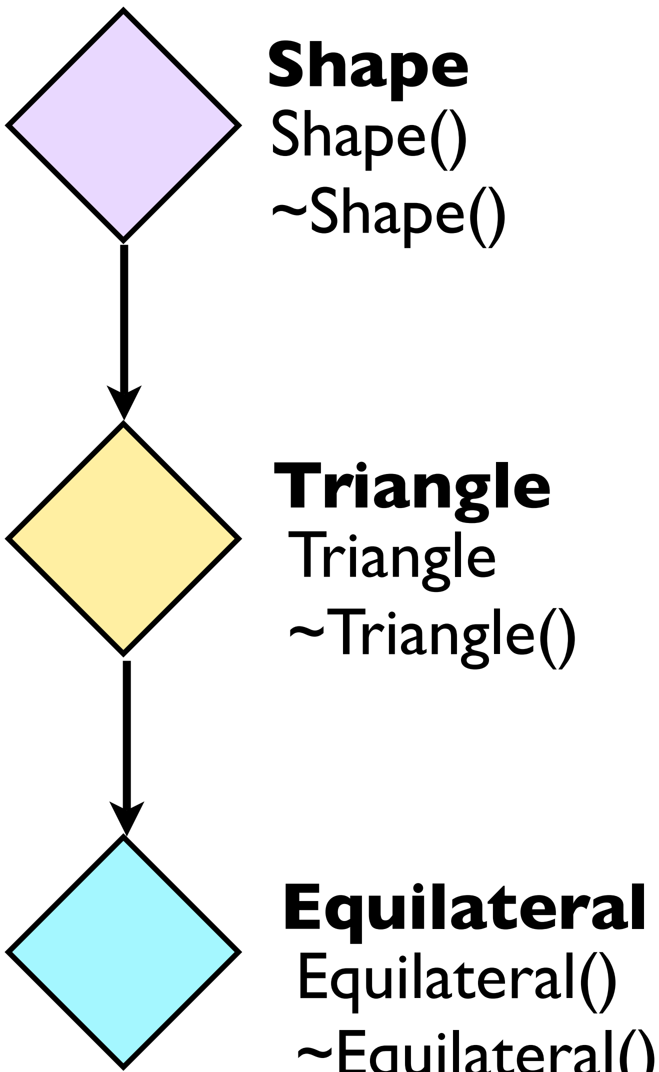





```
Equilateral e;
```

- Small review: in which order are the constructors executed?
- How about the destructors? What would make sense here?

Virtual Destructors




```
Shape* s = new Equilateral();
...
delete s;
```

- A destructor is a method like any other, and the same rules apply
- Destructors need to be marked virtual!
- What *should* happen here?
- What *does* happen, if the destructor is not virtual?

The Fix

```
class Pet
{
   public:
      virtual ~Pet();
};

class Cat : public Pet
{
  public:

    // doesn't need to be
    // marked virtual!
    ~Cat();
};
```

- When using inheritance, always make your destructors virtual!
- Again, making a virtual base class constructor makes all inherited destructors also be virtual

A Useless Function

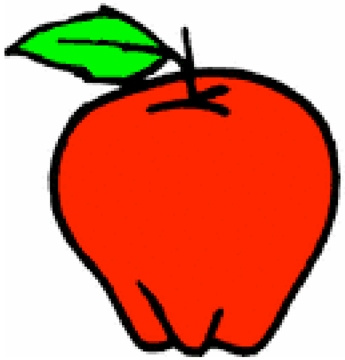
```
class Pet
{
    public:
        void makeNoise()
        {
            cout << "(nothing)";
        }
};
```

- Earlier, we saw this implementation of the makeNoise() function:

- It's kinda useless.
- Its only purpose is to help define an interface: to provide a function for derived classes to override
- So it's not important what Pet::makeNoise *itself* does!

Abstract Methods

- An **abstract method** is a declaration of a method, without a definition
- We're telling the compiler:
  - This method won't be defined in this class, but
  - Any usable derived class *must* implement this method!
- These are also known as **pure virtual** methods



Abstract Methods

- A class with an abstract method is known as an **abstract class**
- An abstract class can't be instantiated!
- To be usable, all methods have to be defined. Since abstract classes have undefined methods (the abstract ones!) they can't be instantiated
- To be usable, a derived class *must* override all abstract methods

The Last One

```
class Pet
{
  public:
    virtual void makeNoise() = 0;
    virtual string getName();
};
```

- This turns the class into an abstract class
- Weird C++ rule: every class needs to have at least one "regular" virtual method when also using abstract methods!

we declare a method to be abstract by tacking "= 0" onto the declaration