INHERITANCE

Revue

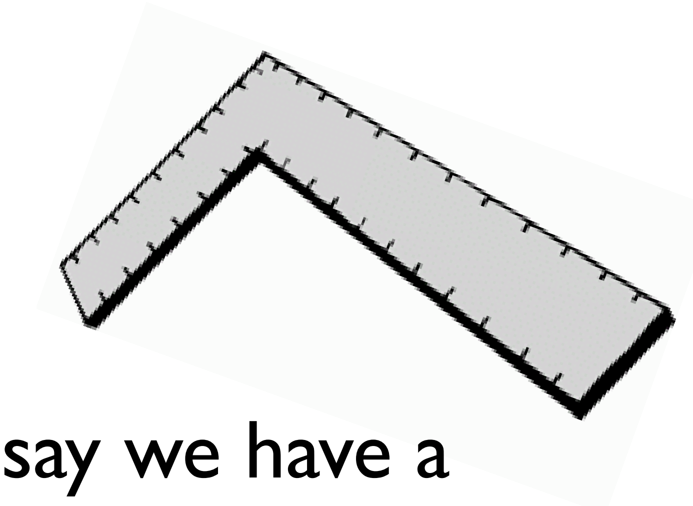
- How do you declare a class or a function to be a **friend**? What's the point?
- What would an **operator==** function look like for Number?
- How about an **operator!=** function?
- How about an **operator==** function that would let you compare a **Number** to an **int**?
- What's a **static** member variable?
- What sorts of data can a static member function access?

```
class Number
{
  public:

  private:
    int n;
};
```

Inheritance

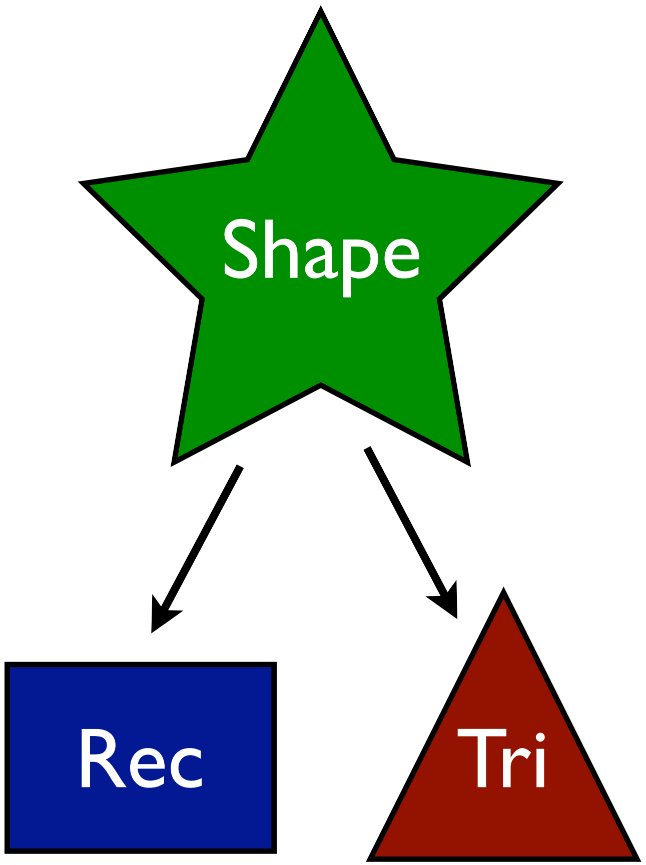
- **Inheritance** is a C++ feature in which one class can "inherit" the member functions and variables from another class
- The new class (the one doing the inheriting) is called the **derived class**
- The class we're inheriting from is called the **base class**

```
class Rectangle
{
  public:
    Rectangle();

    // skipping stuff...

    int area();
    void draw();

  private:
    Color innerColor;
    Color lineColor;
    int lineWidth;
    int x, y;
    int width, length;
    int id;
};
```



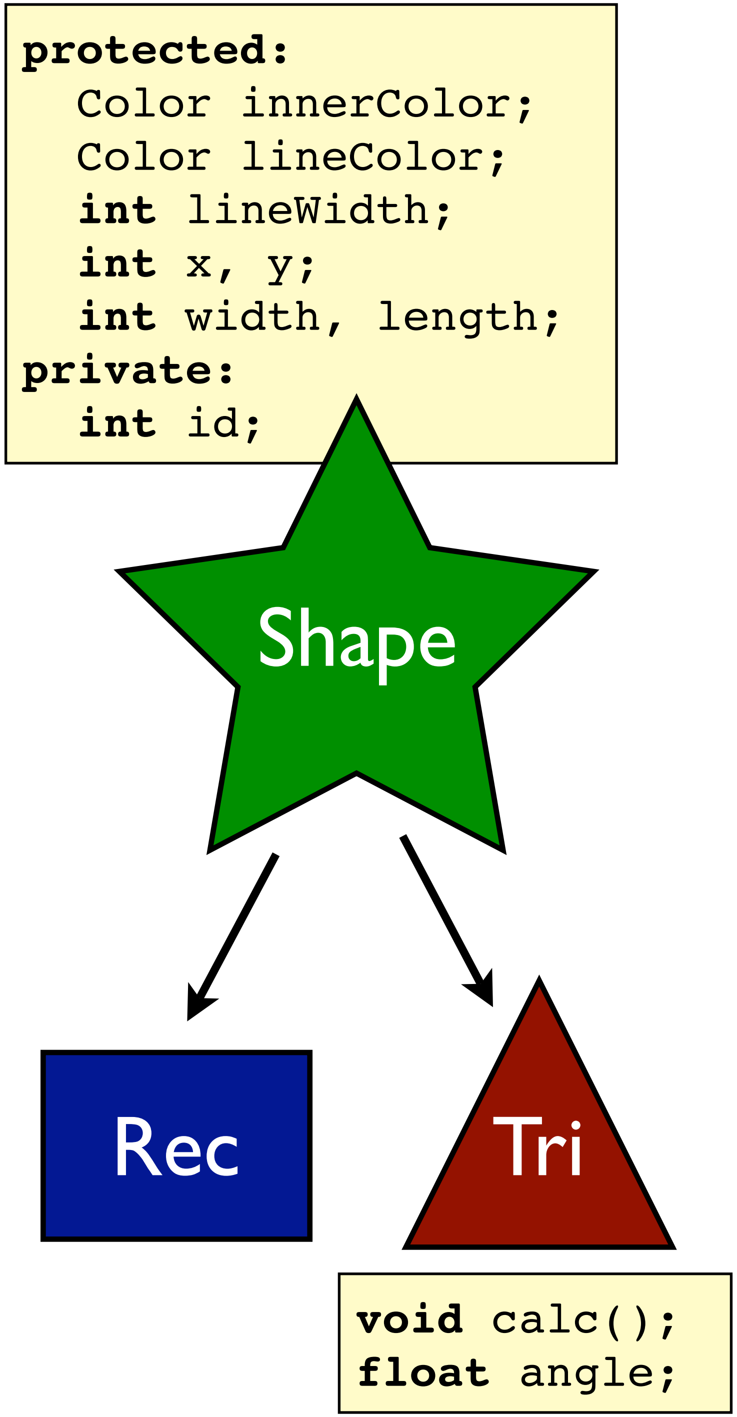
- Let's say we have a **Rectangle** class, with a fair amount of stuff in it
- We'd like to build a simple **Triangle** class
- Most of the code would be the same between these two classes!
- area(), draw() would change

Inheritance



- We could "inherit" most of **Triangle**'s code from **Rectangle**
- A better way: move most of Rectangle's code into a new base class - **Shape** - and derive both Triangle and Rectangle from Shape
- Triangle and Rectangle now only need to implement specific features: the general stuff can be stuck in the Shape class

Inheritance 2



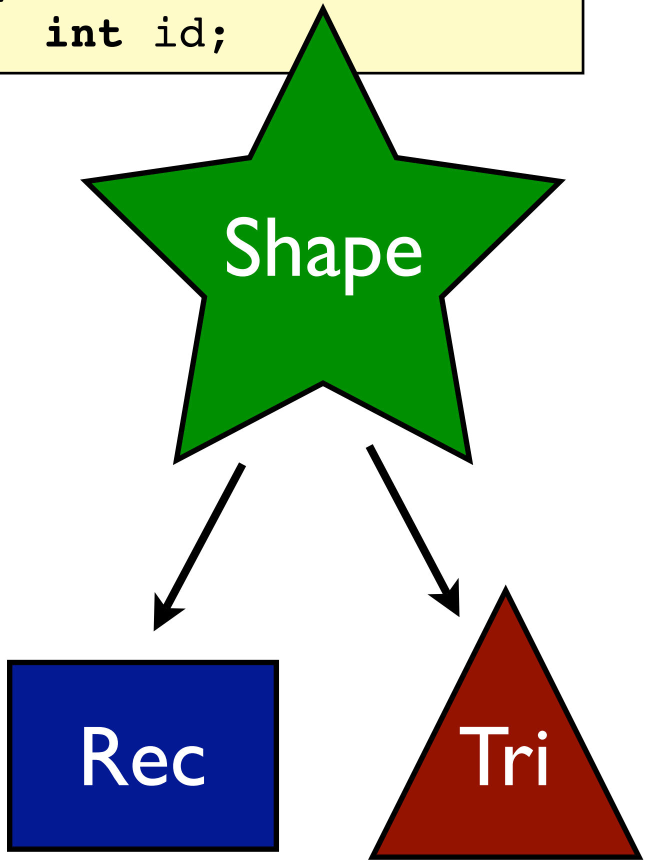
- Derived classes inherit everything in the base class(es)
- Each instance of Triangle has:
  - All the member variables and functions from the Shape class
  - And all the member variables and functions from Triangle
- Triangles has copies of x, y, id, etc. But can it *access* them?

Access Specifiers

- **public** means the same thing it always did
- **private** too: private members can only be accessed from within the class - not any others (including any derived classes!)
- **New! protected** variables can be accessed by the class *and* any derived classes - but not any other class!

Access

```
protected:
  Color innerColor;
  Color lineColor;
  int lineWidth;
  int x, y;
  int width, length;
private:
  int id;
```




```
void calc();
float angle;
```

- So, in this set of classes:
  - innerColor, lineColor, lineWidth, x, y, width, height are all accessible by **Shape**, **Triangle**, **Rectangle**, and no other classes
- id is *only* accessible by **Shape**
- Same access rules apply for member functions

...syntax

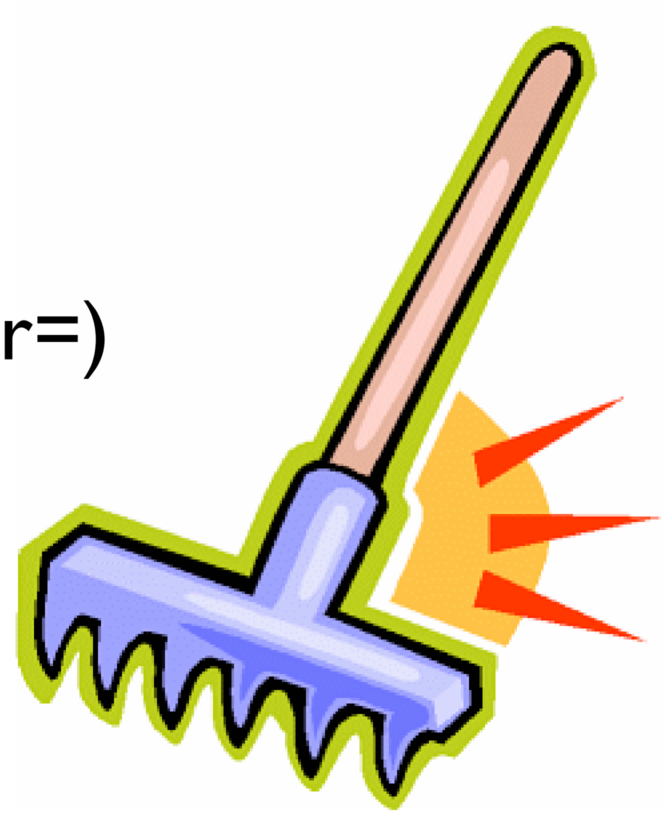
class name → colon → **public**, followed by base class name

```
class Triangle : public Shape
{
  public:
    Triangle();
    int area();

  private:
    void calc();
    // etc...
};
```

- Base class must already be declared here
- Triangle can have all its own stuff - methods, vars, whatever

Inheritance

- What gets inherited?
  - All member variables, (nearly) all functions
- What does **not** get inherited?
  - constructors and destructors
  - Assignment operators (operator=)
  - Friends



Constructors

- Remember, a constructor gets called for *every* class that gets instantiated
  - Sometimes it's a behind-the-scenes constructor, but there always is one!
- With inheritance, there are (at least) two classes involved: the base class and the derived class
- So, at least two constructors are getting called!

```
class Base
{
  public:
    Base()
    { cout << "base\n"; }
};

class Derived : public Base
{
  public:
    Derived()
    { cout << "derived\n"; }
};


int main()
{
   Derived d;
   return 0;
}
```

Snippet

- What is the output of this program?

Construction Order

```
class Base
{
  public:
    Base()
    { cout << "base\n"; }

   Base( int x )
    { cout << "base 2\n"; }
};

class Derived : public Base
{
  public:
    Derived()
    { cout << "derived\n"; }
};
```

- Base classes will always be constructed *before* any derived classes. (Why?)
- The base class constructor is getting called, even though it's not being called explicitly
- If Base has multiple constructors, which one gets called?

Constructor Init List

- C++ will call the default constructor for any base classes automatically
- If there *is* no default constructor (when would that be?) then we have to explicitly call one
- This requires special syntax called the **constructor init list**.

Constructor Init Lists

```
class Base
{
  public:
    Base()
    { cout << "base\n"; }

   Base( int x )
    { cout << "base 2\n"; }
};

class Derived : public Base
{
  public:
    Derived();
};
```

```
Derived::Derived()
  : Base(5)
{
}
```

Constructor Init List

- The constructor init list lets you pass parameters to the base class constructor
- This is like a function call: it will call the correct overloaded constructor

More CIL

```
class Derived : public Base
{
  public:
    Derived();
  private:
    int x, y;
};
```

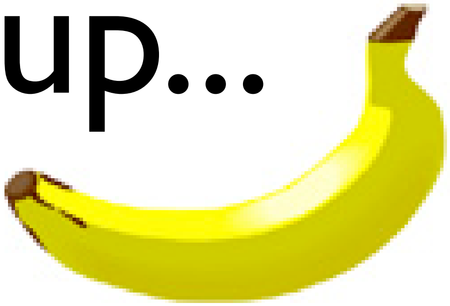
```
Derived::Derived()
  : Base(5), x(5), y(18)
{
}
```

- The CIL can be used for regular member variables, too
- Here, x and y are integers being initialized in the Constructor Init List
- This happens before the constructor body executes!

Coding

- Let’s play with inheritance!

Backing up...



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

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

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?

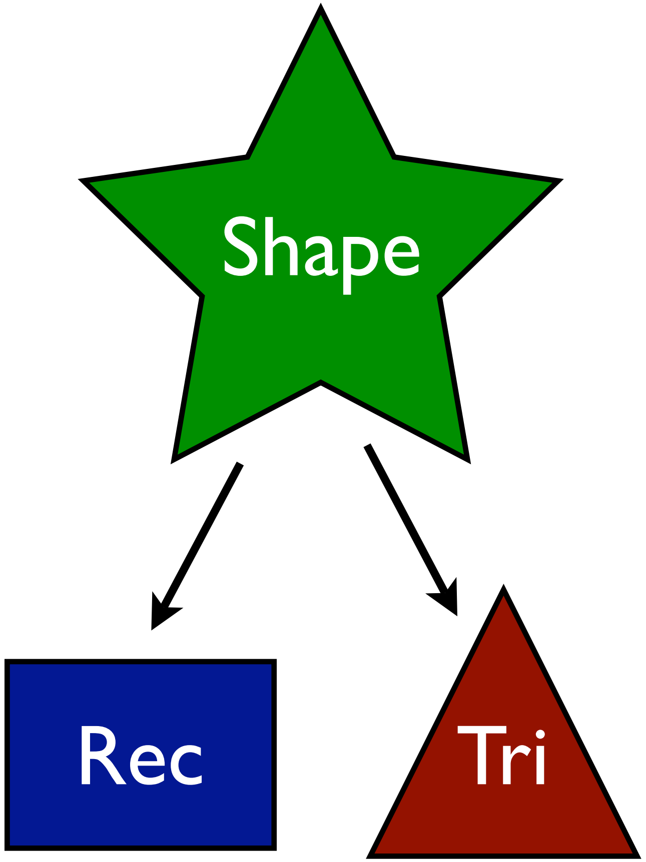
```
class Pet
{
   public:
      Pet();
      ~Pet();
      void play();
      void makeNoise();
   protected:
      string name;
   private:
      string owner;
};

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

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

looks like:

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

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

- Dog is a *single class*
- However, Dog has also inherited everything from Pet!

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

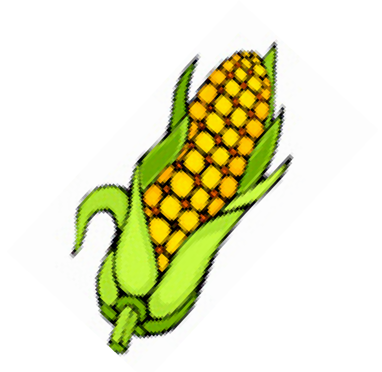
- Since a Triangle is of type Shape, we can refer to it as if it were a Shape.
- This works especially well with pointers:

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

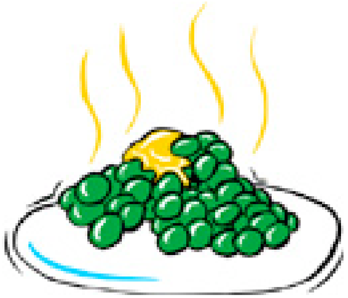
  - What type is **ptr**?
  - What kind of thing is **ptr** pointing to?

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!
- It can only point to 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:

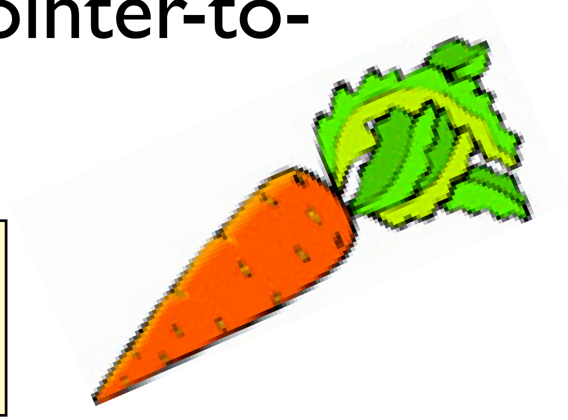
- It lets us treat all kinds of Shapes exactly the same way
- No need to know what type a pointer is actually pointing to - this is called **polymorphism**
- Can only use Shape’s interface

```
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:

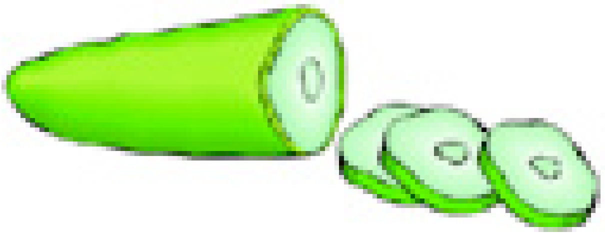
- Here we're defining an array of pointer-to-Shapes:

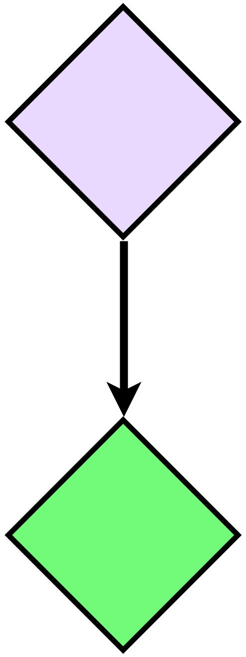


```
Shape* array[10];
```

- 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;

let's talk about this...

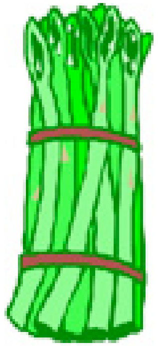
- How is cow being passed?
- What type is cow?
- What type does printWeight accept?

- 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 );
}
```

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!

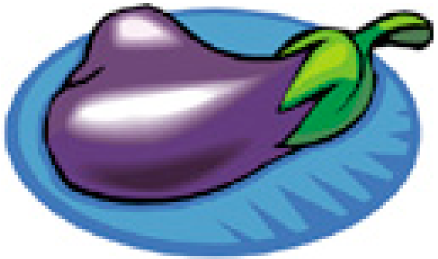
```
void printWeight( FarmAnimal animal )
{
    cout << animal.weight;
}

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.

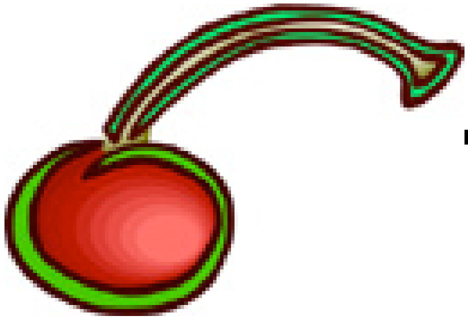
```
class Pet
{
    public:
        void makeNoise()
        {
            cout << "(silence)";
        }
};

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

- How about this one?

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

- ... and this one?

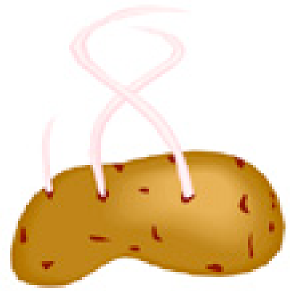
```
Pet* animal = new Cat;
animal->makeNoise();
```



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();
```



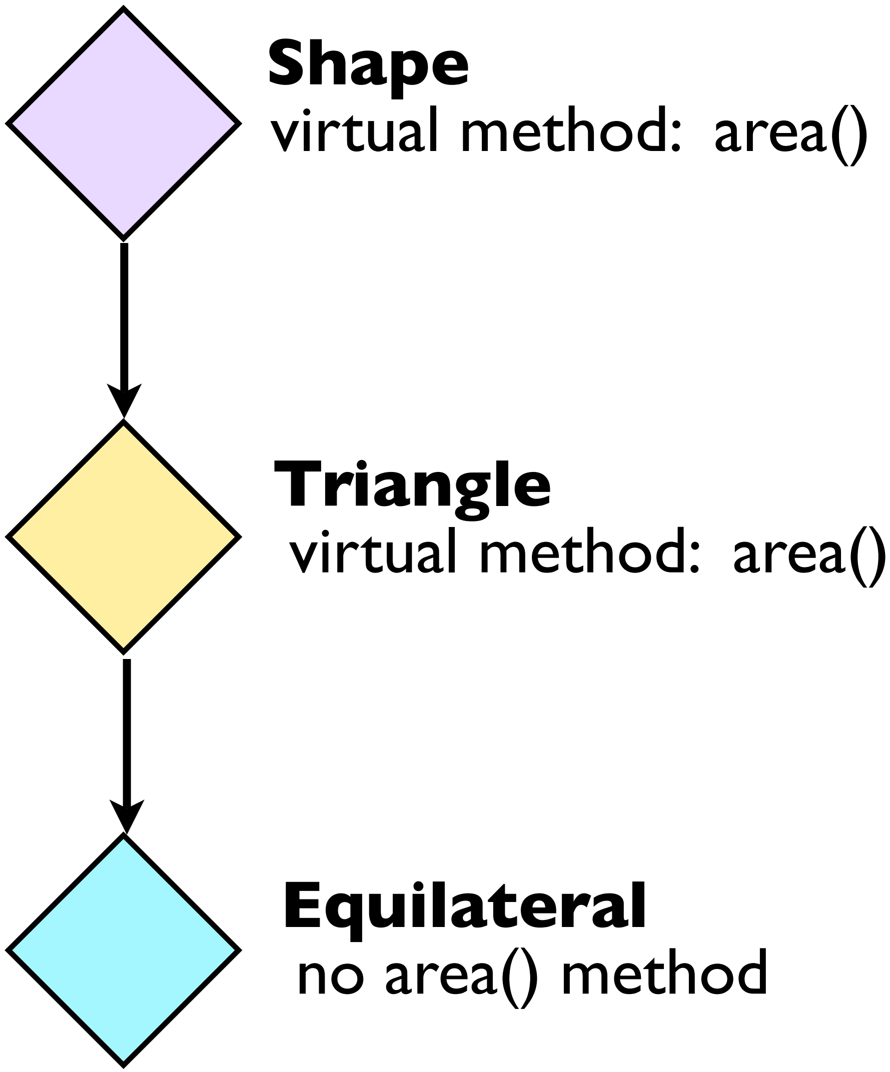
So then:

```
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?

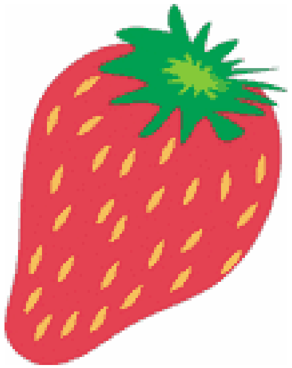
```
Shape* s = new Equilateral;
s->area();
```

Virtual: How-to

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

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

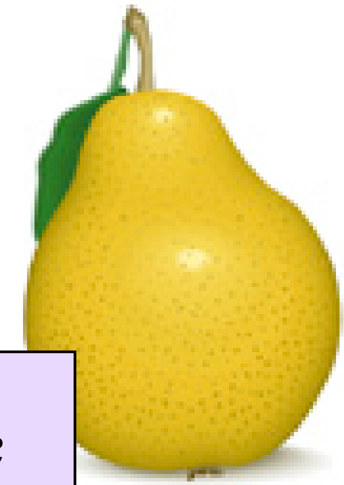
- To declare a virtual method, stick the keyword **virtual** before its return type
- This automatically makes every overridden version of the method virtual too
- Only works in one direction: marking Cat::makeNoise as virtual doesn't make Pet::makeNoise virtual!

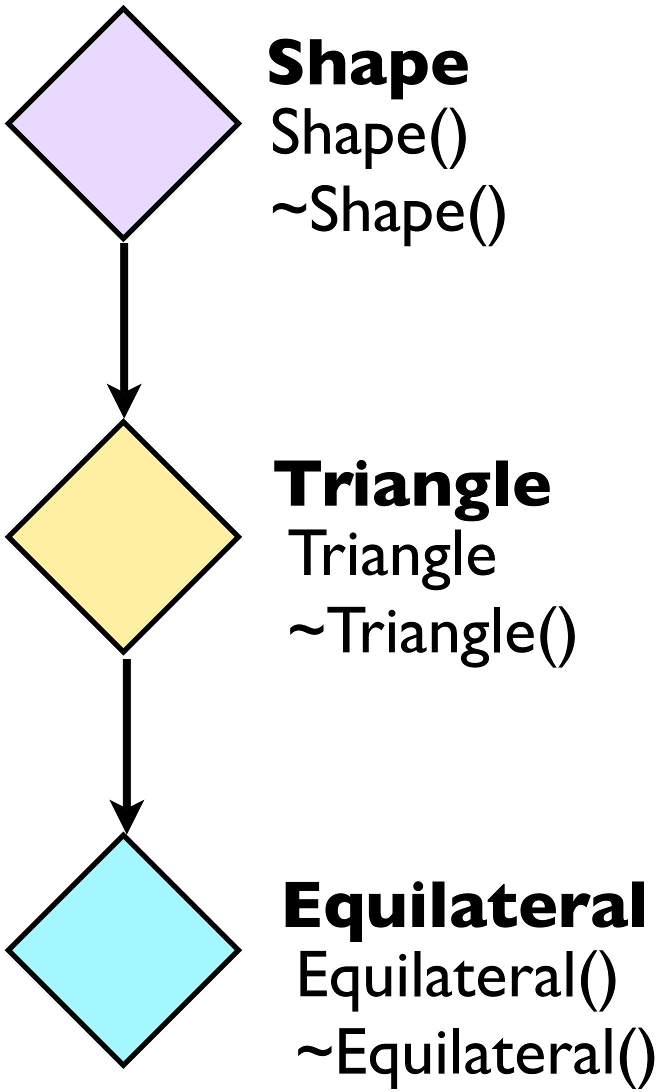
Virtual Rules



- 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

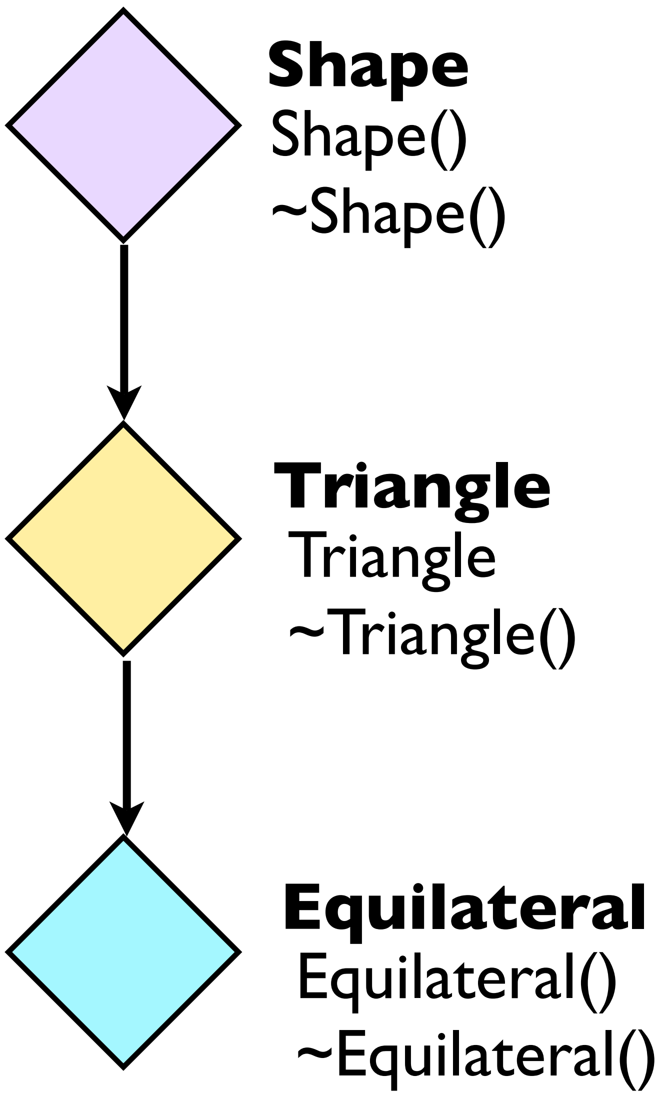





```
Equilateral e;
```

- 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();
};
```

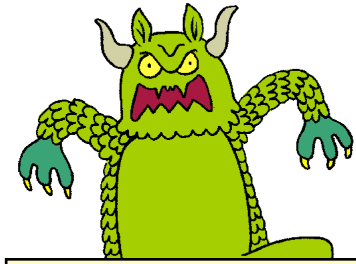
- When using inheritance, always make your destructors virtual!
- Again, making a virtual base class constructor makes all inherited destructors also be virtual

Overrided Functions

```
class Car
{
   public:
      void vroom()
      {
          cout << "Car::vroom\n";
      }
};

class Geo : public Car
{
  public:
     void vroom()
      {
          cout << "Geo::vroom\n";
      }
};
```

- So far we've been saying that overrided functions "hide" their base class versions
- What would this code fragment output?

```
Geo prizm;
prizm.vroom();
```



Overrided Functions

```
class Car
{
    public:
        void vroom()
        {
            cout << “Car::vroom\n”;
        }
};

class Geo : public Car
{
   public:
       void vroom()
        {
            cout << “Geo::vroom\n”;
            base::stuff();
        }
};
```

- “Hidden” doesn’t mean “gone”, though!
- Sometimes you might want to call the base class version of a function...
- You can do that using the scope resolution operator (::)

What does this print now?

```
Geo prizm;
prizm.vroom();
```

Some Weird Syntax...

```
class Car
{
   public:
      void vroom()
      {
          cout << "Car::vroom\n";
      }
};

class Geo : public Car
{
  public:
     void vroom()
      {
          cout << "Geo::vroom\n";
      }
};
```

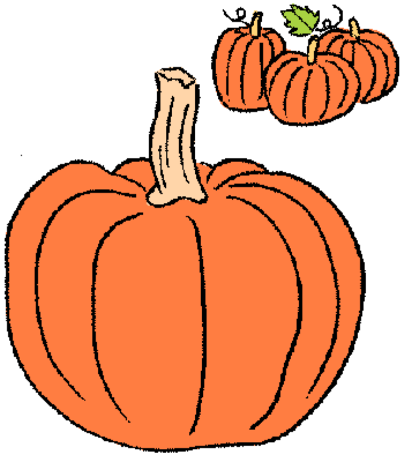
- You can even do this from *outside* a class
- Say you want to call the base class version of **vroom**() from the main function:

```
int main()
{
  Geo prizm;
  prizm.base::vroom();
}
```

Question

```
void vroom()
{
   cout << “Global Vroom!!\n”;
}

class Car
{
   public:
      void vroom()
      {
         cout << “Car::vroom\n”;
      }
};

class Geo : public Car
{
  public:
     void vroom()
      {
         cout << “Geo::vroom\n”;
         Global vroom()?
      }
};
```

- What if we add *another* vroom() function - a global one?
- Could we call that from Geo::vroom()?



Question

```
void vroom()
{
   cout << “Global Vroom!!\n”;
}

class Car
{
   public:
      void vroom()
      {
          cout << “Car::vroom\n”;
      }
};

class Geo : public Car
{
  public:
     void vroom()
      {
          cout << “Geo::vroom\n”;
          ::vroom();
      }
};
```

- When used on its own, :: means “access the global scope, not the local scope”
- So, to call the global vroom() function, we use the :: operator to call the containing scope

A Useless Function

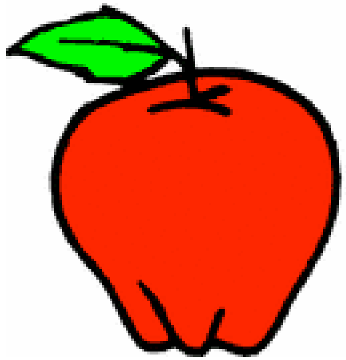
```
class Pet
{
    public:
        void makeNoise()
        {
            cout << "(silence)";
        }
};
```

- 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

Rules

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

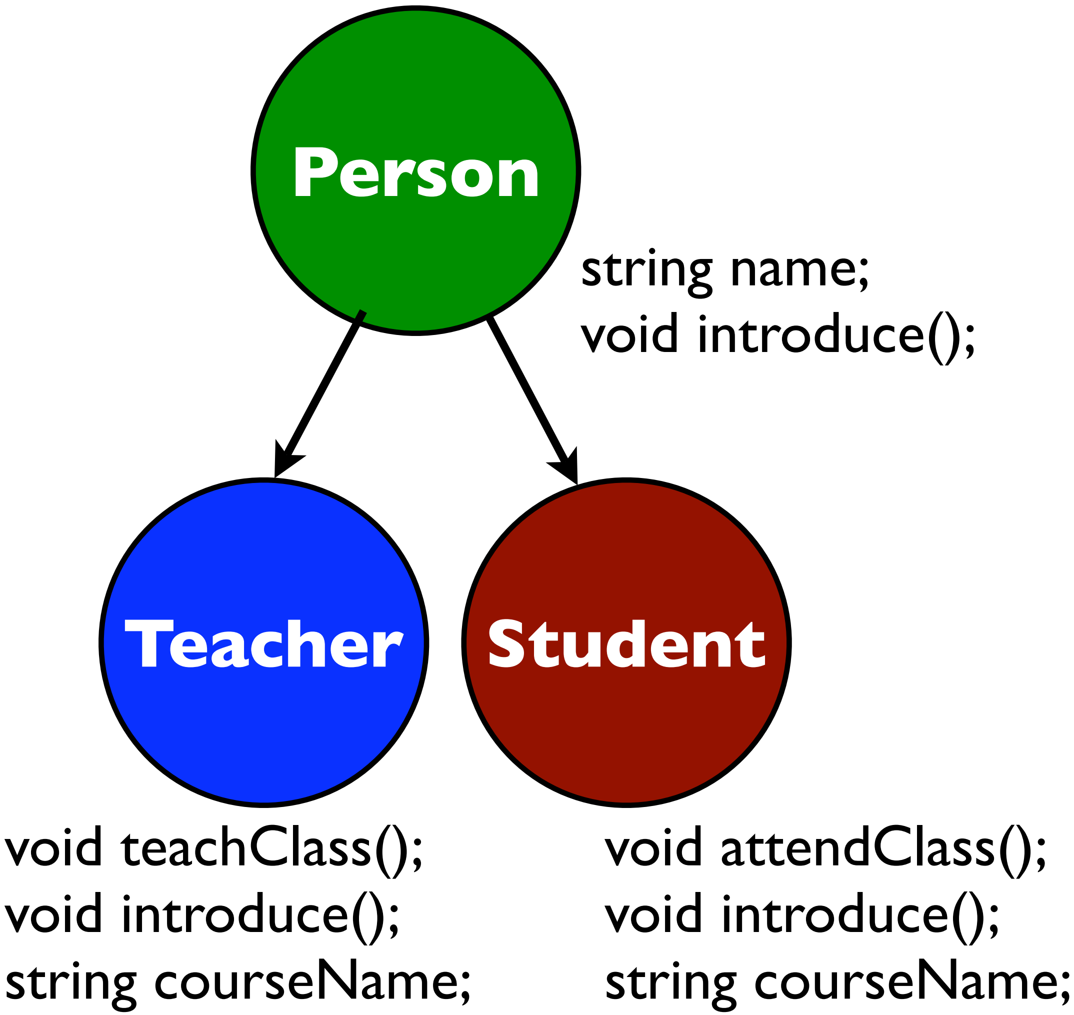
we declare a method to be abstract by tacking "= 0" onto the declaration

- 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!

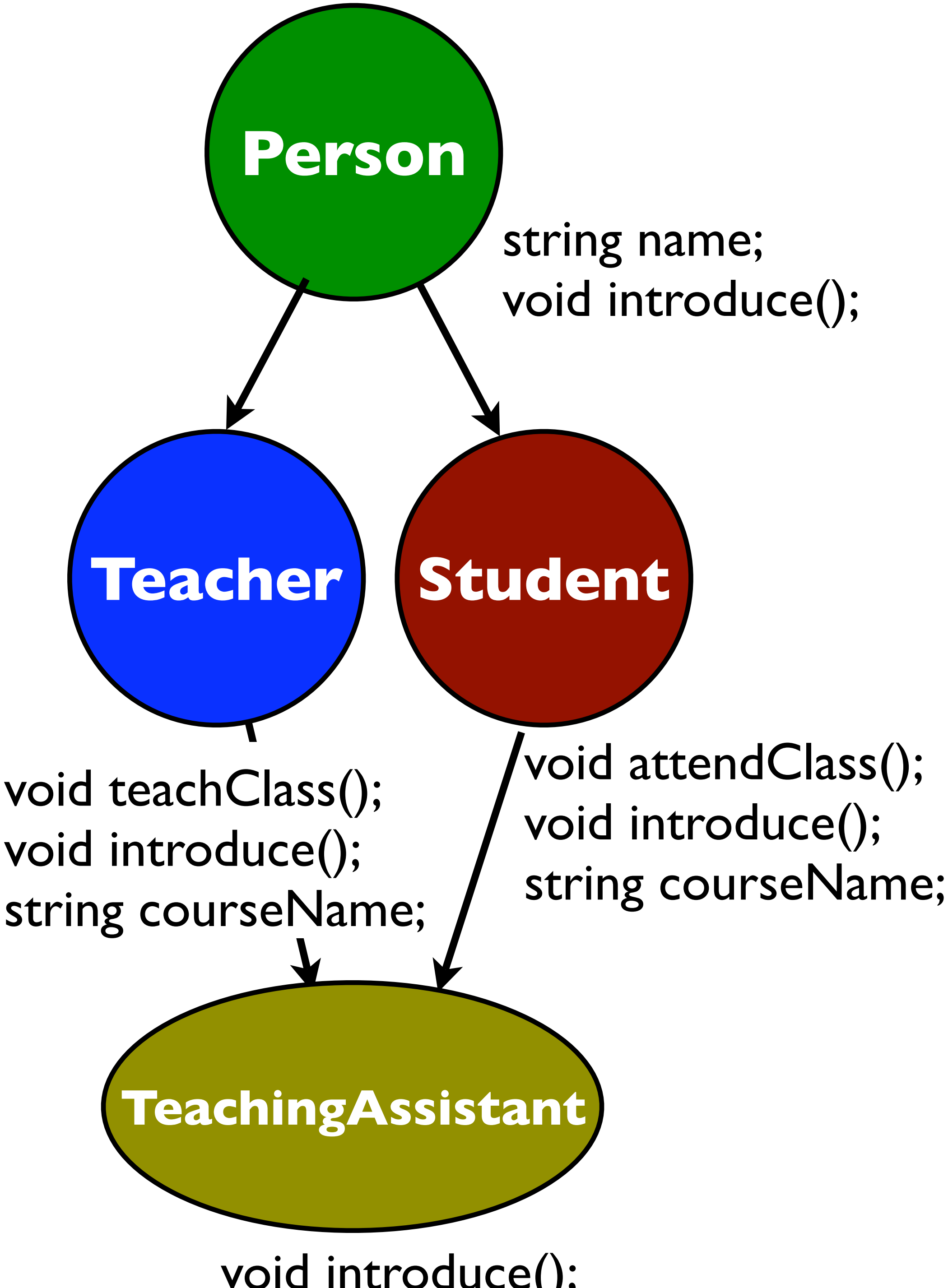
More Coding

- Let's play with inheritance!
- Again!

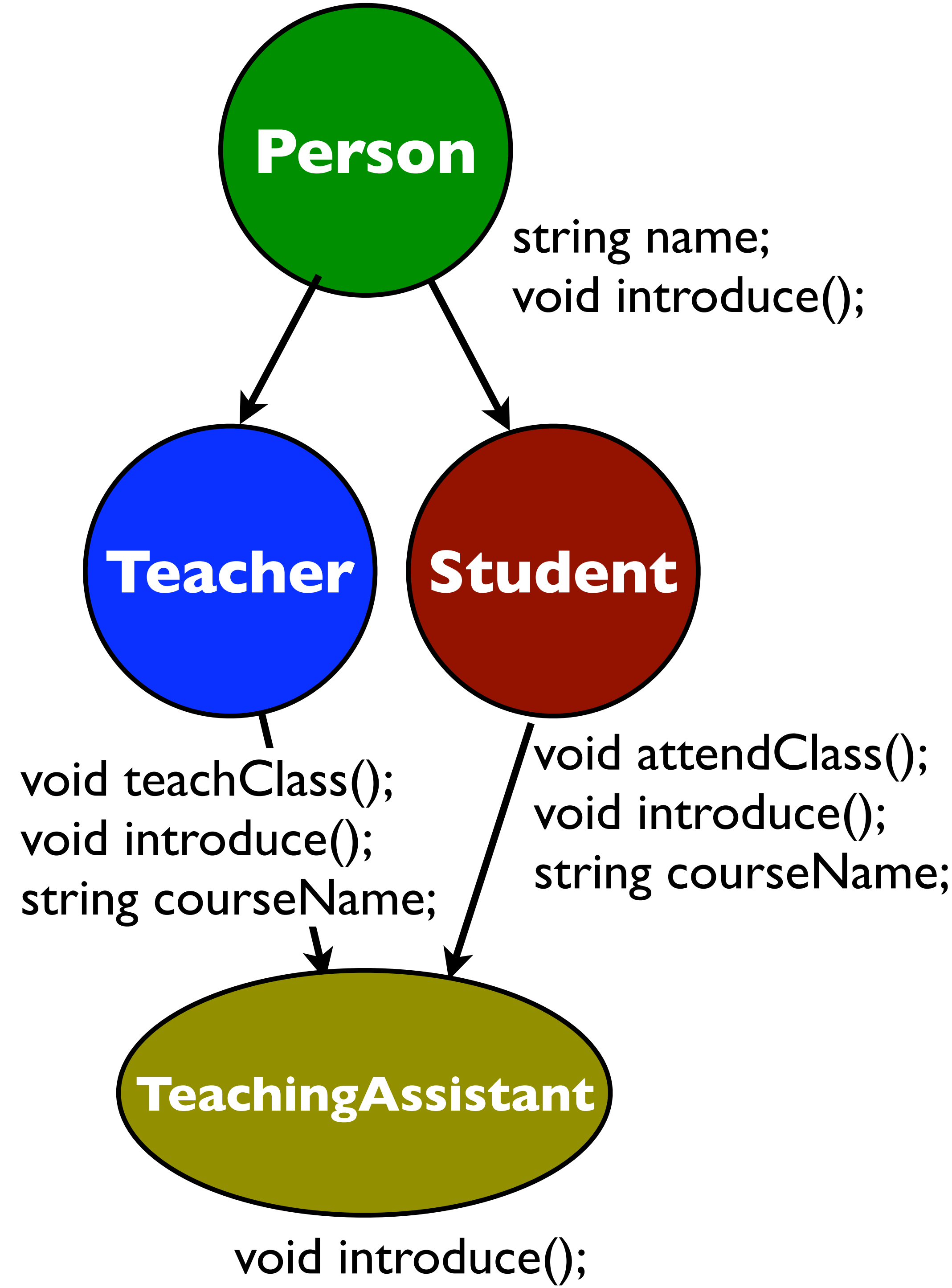
Multiple Inheritance

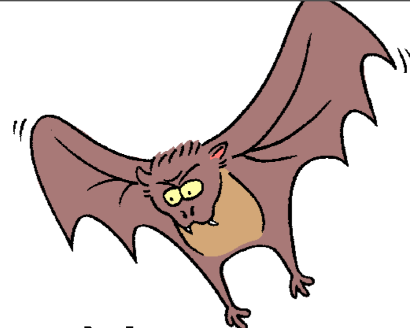


- Sometimes inheriting from a single class isn't enough!
- Say we've got the simple class hierarchy to the left:
- What do we do when we want to define a **TeachingAssistant** class?
  - A TeachingAssistant both teaches *and* attends classes
  - No one base class is enough!



- We have to make **TeachingAssistant** inherit from *both* Teacher and Student!
- So: our new TA class will inherit *all* the stuff from both base classes!
- How would we write an introduce method that explains what course the TA teaches, *and* what course he/she studies?





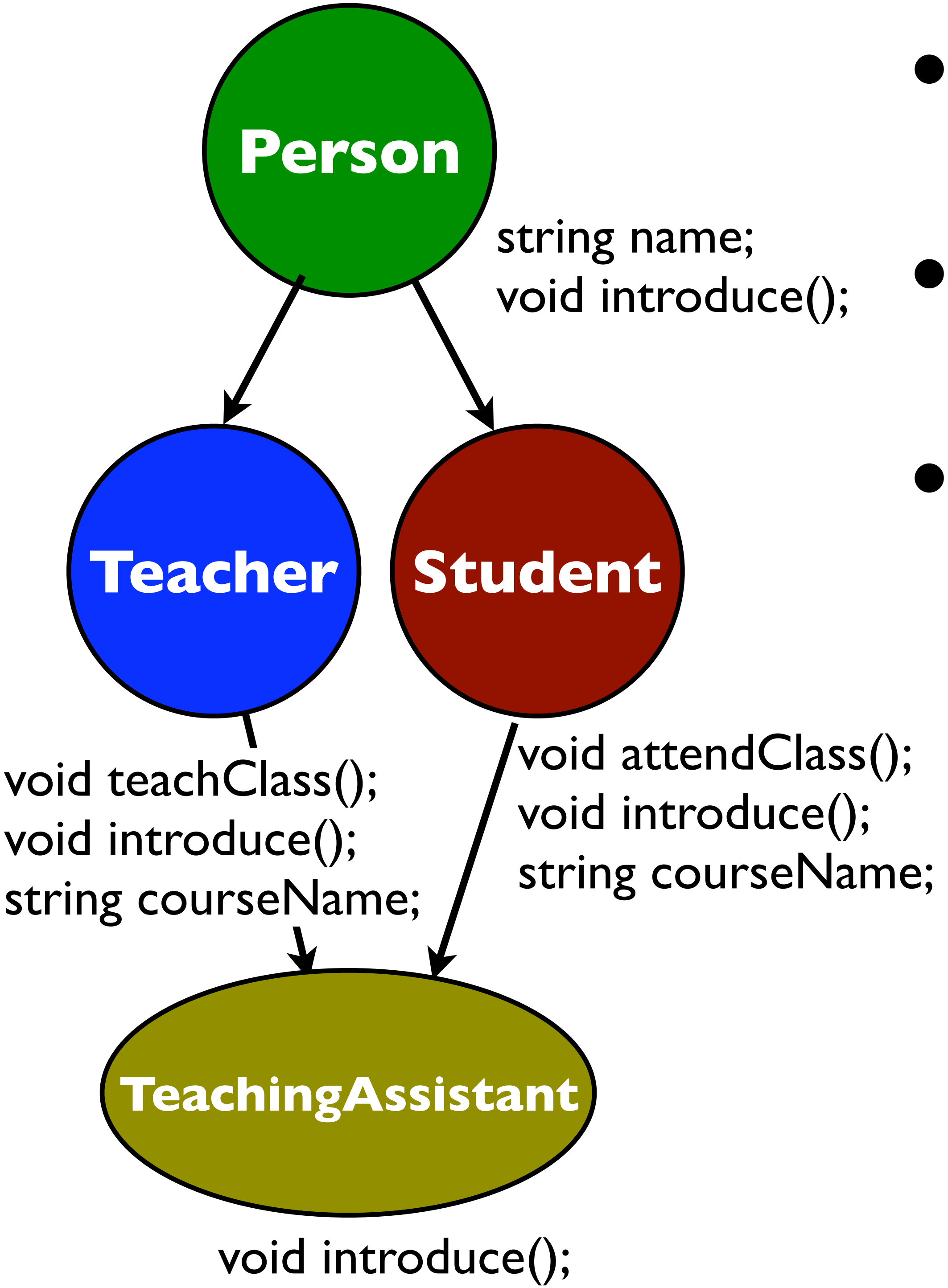
- How many courseName variables are there in TeachingAssistant?
- How do we print out the right version at the right time?

```
void TA::introduce()
{
    cout << “I teach: ”;
    cout <<  (?)
    cout << “I study: “;
    cout <<  (?)
}
```

Multiple Inheritance

```
class Teacher : public Person
{   // declaration mostly omitted
  public:
    Teacher( string name );
};

class Student : public Person
{   // declaration mostly omitted
 public:
    Student( string name );
};

class TA :
      public Teacher, public Student
{
  public:
    TA() :
      Student(name), Teacher(name)
    {}
};
```

- Doing this is pretty simple:
- Just add to the list of classes your class inherits from
- You may need to add to the constructor init list too!



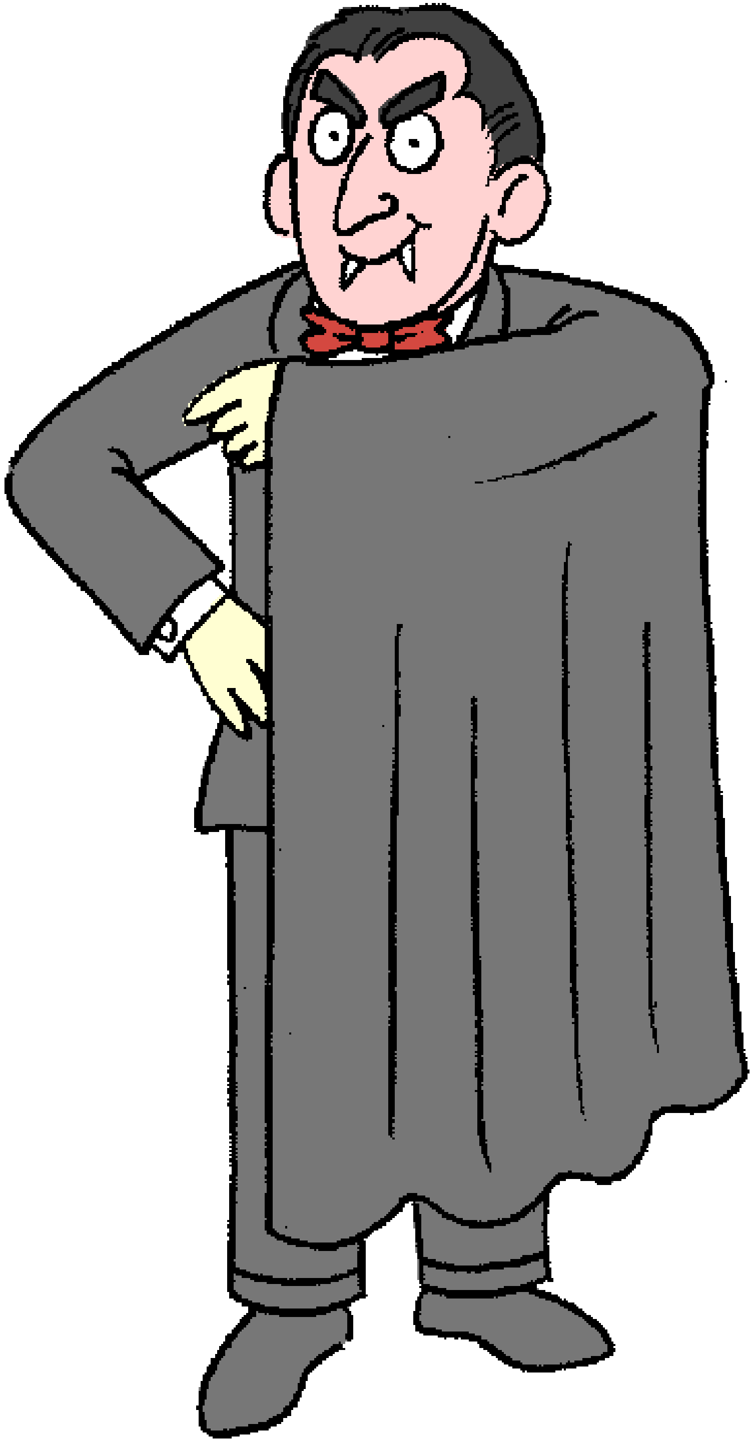
- One problem you may have noticed:
- How many copies of **name** does TeachingAssistant have?
- Which one do we use? Does it matter?

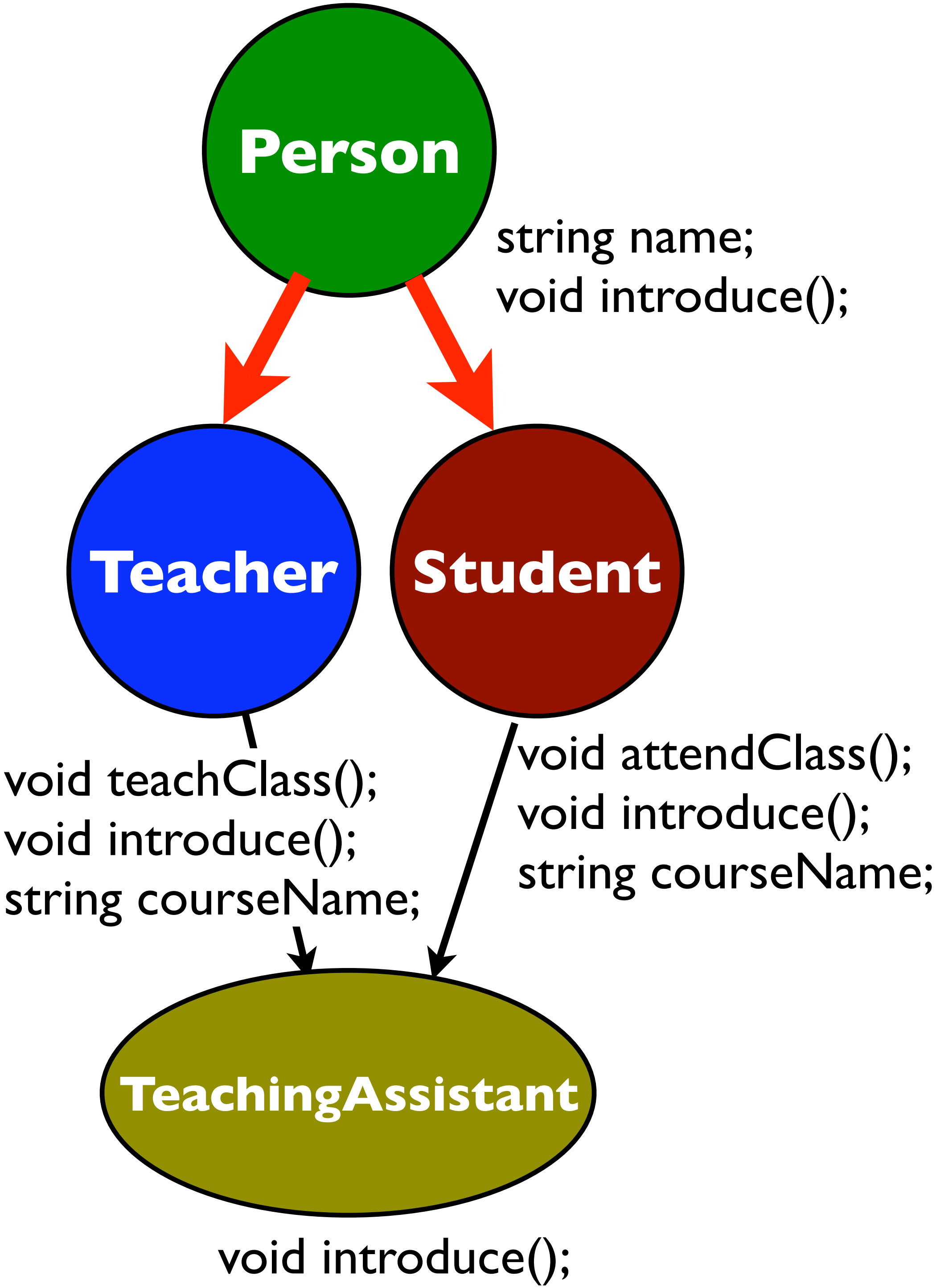
```
void TA::introduce()
{
    cout << "My name is:";
    cout <<  (?)
    cout << "I teach: ";
    cout <<  (?)
    cout << "I study: ";
    cout <<  (?)
}
```

- **TeachingAssistant** is derived from both **Student** and **Teacher**
- Both Student and Teacher inherited a **name** attribute from **Person**
- Therefore, TeachingAssistant has **two** copies of **name**!
- This might be OK but it might not: could each copy of name have a different value?

Virtual Inheritance

- The way to solve this: **virtual inheritance**
- If you inherit "virtually" from a base class, you tell the compiler:
  - there must be one instance of that base class if someone inherits from the current class
- This is weird, and ugly, but it solves the problem neatly





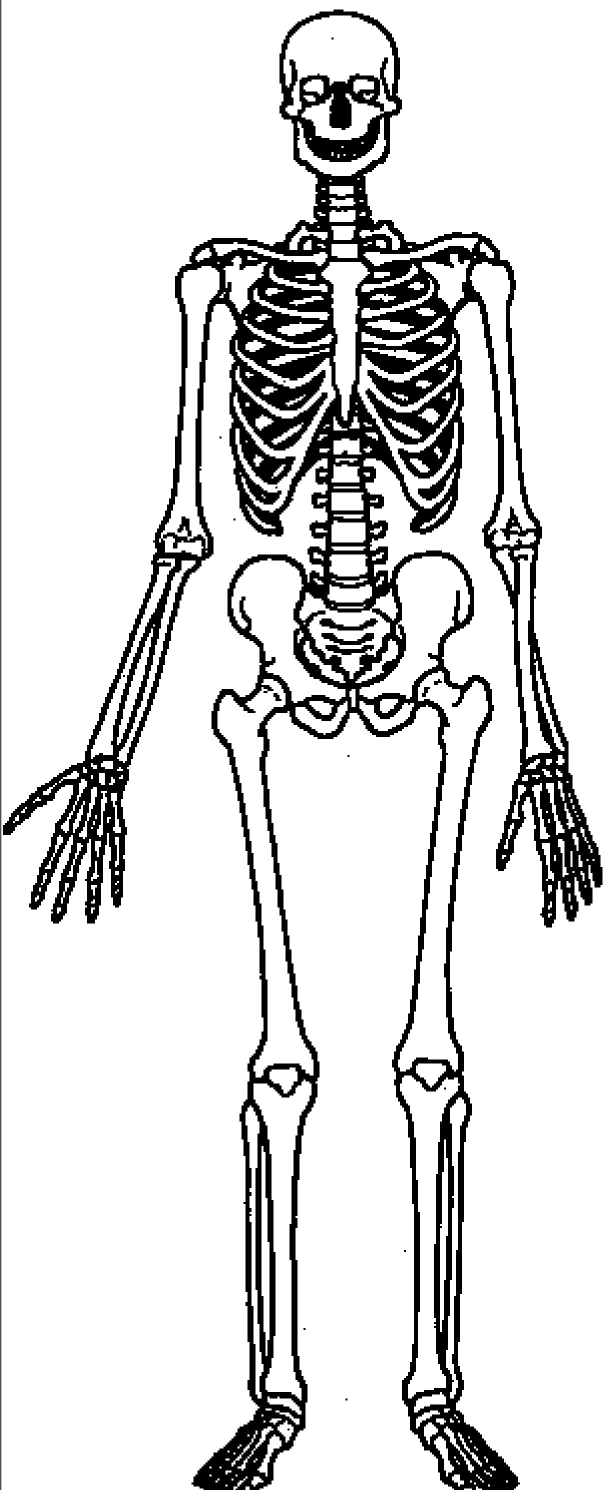
how this works:

- Before we had **two** copies of name in TeachingAssistant
- Now, Teacher and Student are inheriting *virtually* from Person (red arrows)
- So there will be only *one* copy of Person in any class inherited from Teacher and Student
- ... aka TeachingAssistant, only has a single copy of Person - (therefore, name)

Virtual Inheritance

```
// declarations mostly omitted...
class Person
{
    string name;
};

class Teacher : virtual public Person
{
  public:
    Teacher( string name );
};

class Student : virtual public Person
{
 public:
    Student( string name );
};

class TA :
      public Teacher, public Student
{
  public:
    TA() :
      Student(name), Teacher(name)
    {}
};
```

- To inherit virtually, just stick the keyword **virtual** right before the **public**
- This has nothing to do with virtual functions!
- Why do *both* Student and Teacher use virtual inheritance? Is this necessary?



Multiple Inheritance

- Many people disagree on the usefulness of Multiple Inheritance
  - Most newer languages don't support MI at all, or only a small subset of it
- If you find yourself needing to use MI a lot, consider redesigning your classes so you don't!
- Not used nearly as widely as regular inheritance