Polymorphism

CIS 15 : Spring 2007

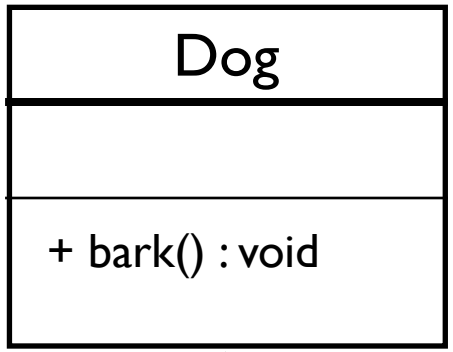
Functionalia

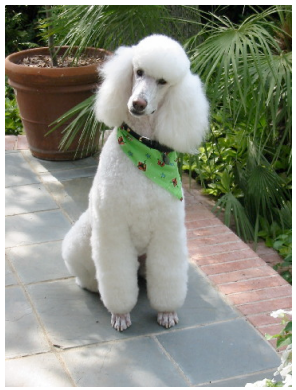
Today:

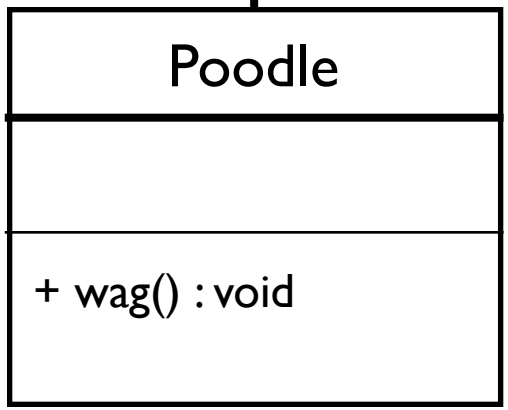
- Polymorphism
- Virtual Functions
- Abstract Classes
- Multiple Inheritance

Dogs

```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void wag() { cout << "wag! wag!" << endl; }
};
```

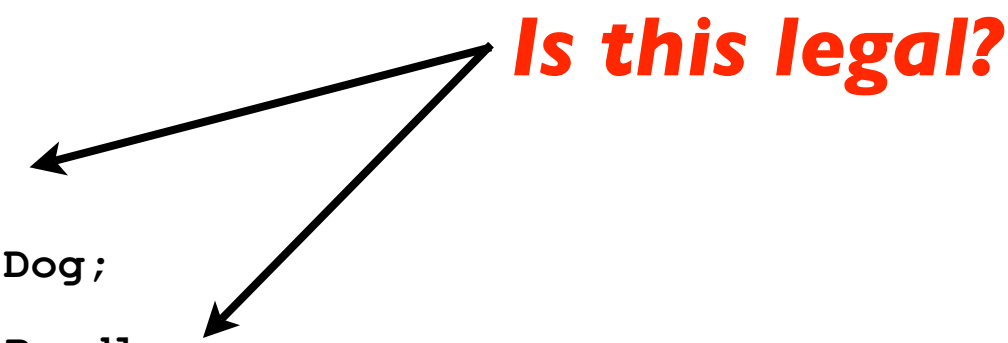






Using a Dog pointer to store a Poodle

```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void wag() { cout << "wag! wag!" << endl; }
};

int main() {

   Dog * pet1;

   Dog * pet2;

   pet1 = new Dog;

   pet2 = new Poodle;

   pet1->bark();

   pet2->bark();

   delete pet1;

   delete pet2;

}
```



Is this legal?

Upcasting

```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void wag() { cout << "wag! wag!" << endl; }
};

int main() {
   Dog * pet1;
   Dog * pet2;
   pet1 = new Dog;
   pet2 = new Poodle;
   pet1->bark();
   pet2->bark();
   delete pet1;
   delete pet2;
}
```

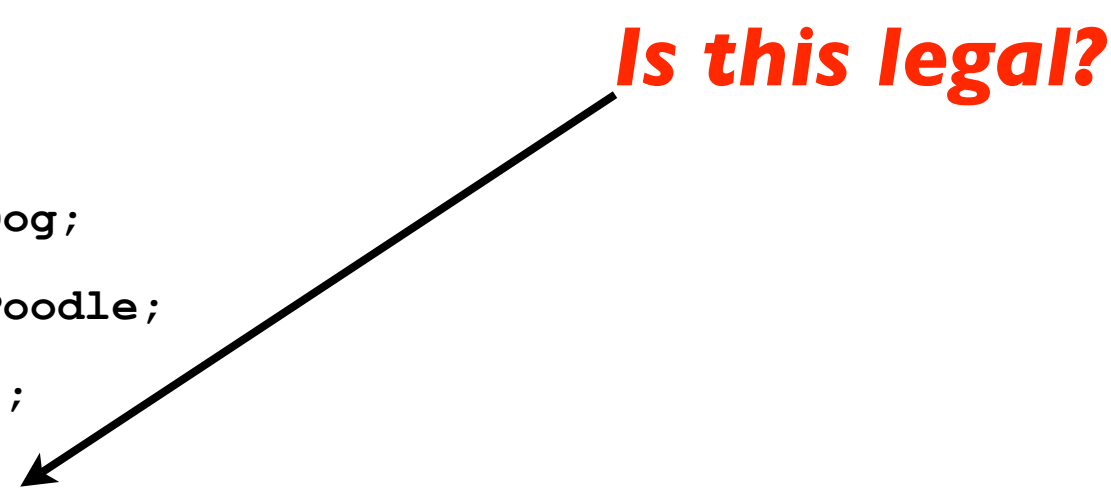
Since Poodle is-a kind of Dog:

A Dog pointer can be used to reference to the Poodle class.
(the Poodle object is upcast to a Dog)

A Poodle always inherits **`bark()`**

Upcasting

```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void wag() { cout << "wag! wag!" << endl; }
};

int main() {

   Dog * pet1;

   Dog * pet2;

   pet1 = new Dog;

   pet2 = new Poodle;

   pet1->bark();

   pet2->wag();

   delete pet1;

   delete pet2;

}
```



Upcasting

```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void wag() { cout << "wag! wag!" << endl; }
};

int main() {

   Dog * pet1;

   Dog * pet2;

   pet1 = new Dog;

   pet2 = new Poodle;

   pet1->bark();

   pet2->wag();

   delete pet1;

   delete pet2;

}
```

No. Compile-time error.

```
error: 'class Dog' has no member named 'wag'
```

The *is-a* relationship does not work in reverse.

Downcasting

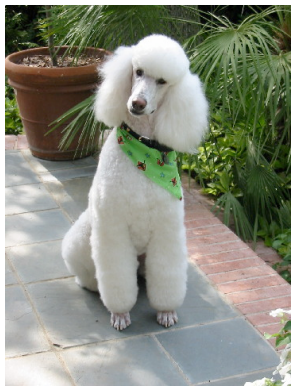
```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void wag() { cout << "wag! wag!" << endl; }
};

int main() {

   Dog * pet;

   pet = new Poodle;

   pet->bark();


   Poodle * pet2 = static_cast<Poodle *>(pet);

   pet2->wag();

   delete pet;

}
```

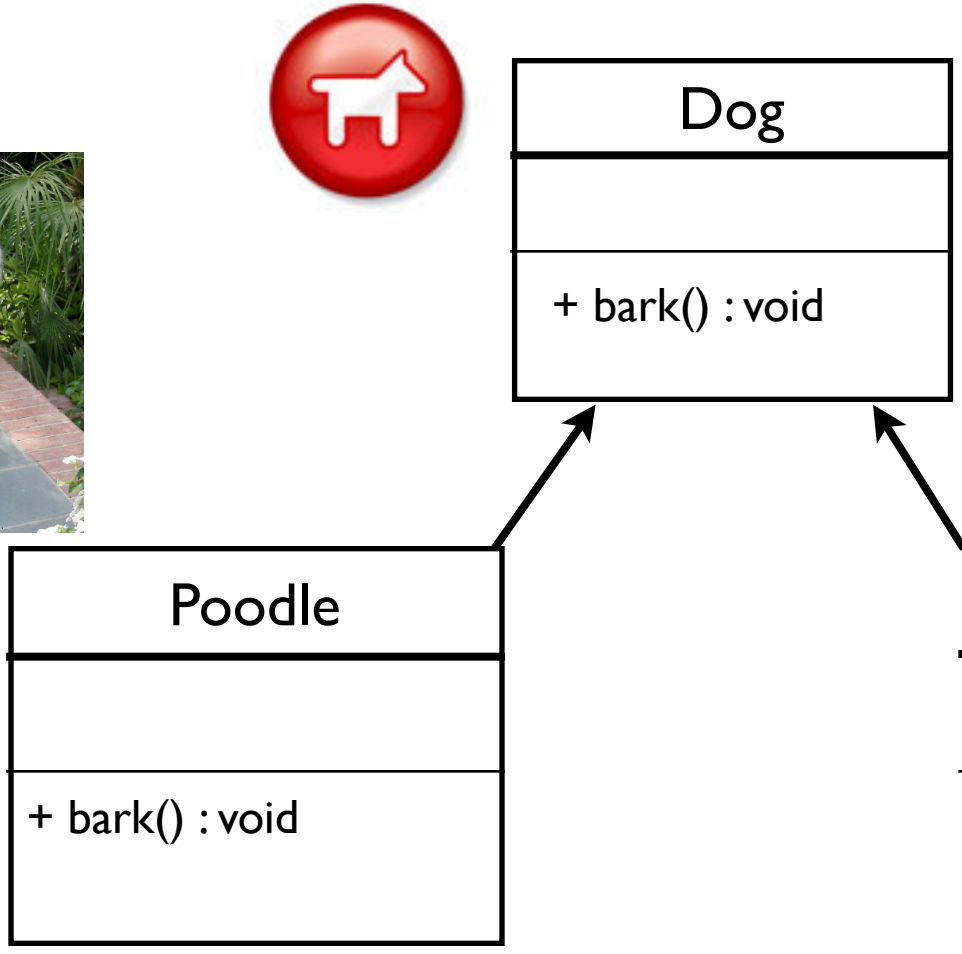
Work-Around

Recast the Dog pointer to a Poodle Pointer

messy

Polymorphism





Polymorphism allows for a reference variable or an object pointer reference objects of *different* types (i.e. Poodle vs. Retriever), and to call the correct member functions(i.e. bark()), depending on the type of object being referenced.

Dogs and their `bark()`

```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
   public:
      void bark() { cout << "arf! arf!" << endl; }
};

int main() {

   Dog * pet;

   pet = new Dog;

   pet->bark();

   delete pet;

}
```

What kind of bark?

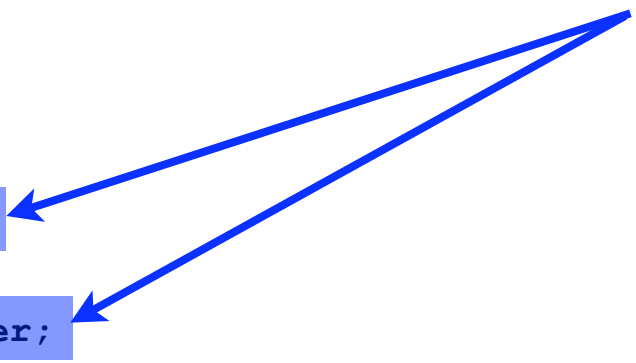
Dogs and their `bark()`

```
class Dog {
   public:
      void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
   public:
      void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
   public:
      void bark() { cout << "arf! arf!" << endl; }
};

int main() {

   Dog * pet;

   pet = new Dog;

   pet->bark();

   delete pet;

}
```

bark bark!

Dogs and their `bark()`

```
class Dog {
    public:
        void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    srand(time(NULL));
    Dog * pet;
    if((rand()%2) == 0)
        pet = new Poodle;
    else
        pet = new Retriever;
    pet->bark();
    delete pet;
}
```



Suppose we do not know the type of Dog at run-time.

Dogs and their `bark()`

```
class Dog {
    public:
        void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

Simplified.

What kind of bark?

Dogs and their `bark()`

```
class Dog {
    public:
        void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

bark bark!

Why?

bark() is ***statically bound***

```
class Dog {
    public:
        void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

The compiler binds the function call to the only class it knows about at the time, the Dog class.

Also called **early binding**.

`virtual` functions allow for ***virtual binding***

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

Now, the function that will be called depends on the *type* of the object and is decided at **runtime**.

What kind of bark?

`virtual` functions allow for ***virtual binding***

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

woof! woof!

virtual binding also called ***late binding*** because the function to call is decided at a *later* time during runtime.

All `bark()` functions are virtual in hierarchy

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        virtual void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        virtual void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

Although not necessary to explicitly define them as virtual:

A function defined as *virtual* in the base class is *virtual* in the derived classes.

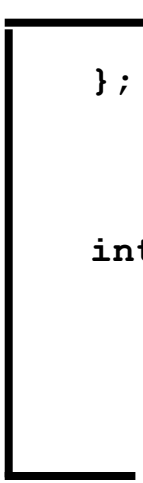
Virtual Functions extend through ***Multiple Levels*** of inheritance

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class FrenchPoodle : public Poodle {
    public:
        void bark() { cout << "ouah! ouah!" << endl; }

};

int main() {
    Dog * pet;
    pet = new FrenchPoodle;

    pet->bark();
    delete pet;
}
```

(**source** http://www.georgetown.edu/faculty/ballc/animals/dog.html)

Virtual Functions extend through ***Multiple Levels*** of inheritance

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class FrenchPoodle : public Poodle {
    public:
        void bark() { cout << "ouah! ouah!" << endl; }
};


int main() {
    Dog * pet;
    pet = new FrenchPoodle;

    pet->bark();
    delete pet;
}
```



ouah! ouah!

(**source** http://www.georgetown.edu/faculty/ballc/animals/dog.html)

Redefining versus Overriding

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class FrenchPoodle : public Poodle {
    public:
        void bark() { cout << "ouah! ouah!" << endl; }
};
```

overrides

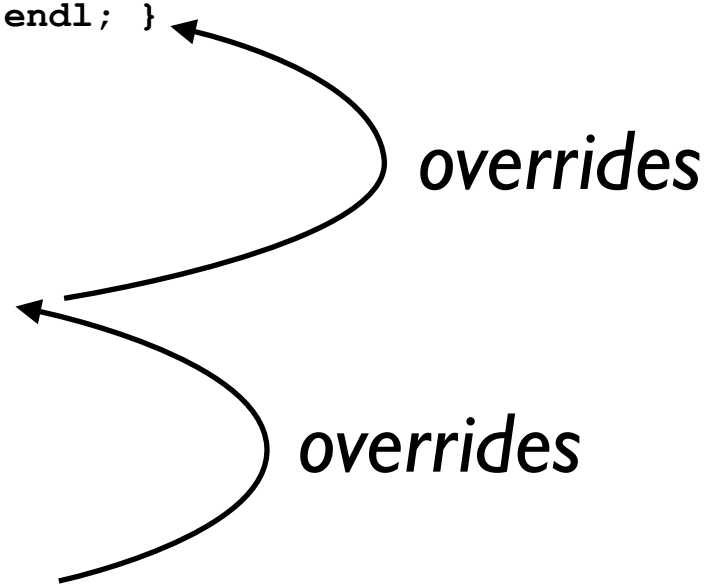
overrides

```
int main() {
    Dog * pet;
    pet = new FrenchPoodle;

    pet->bark();
    delete pet;
}
```

Virtual functions ***override*** a Base Class's function

Redefining versus Overriding

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class FrenchPoodle : public Poodle {
    public:
        void bark() { cout << "ouah! ouah!" << endl; }
};
```

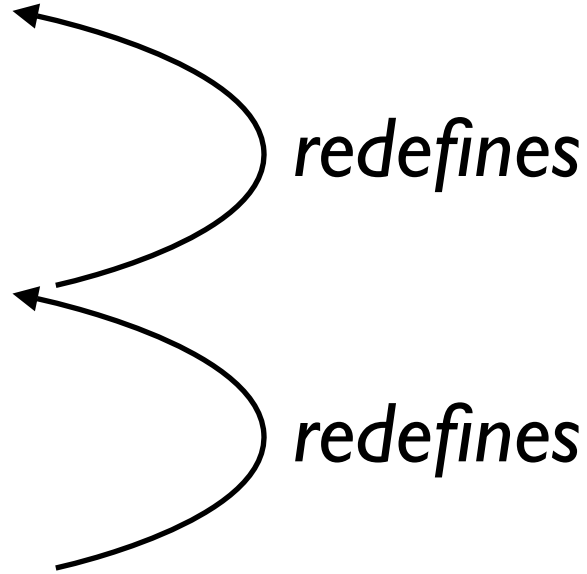


Virtual functions ***override*** a Base Class's function

dynamic and run-time

Redefining versus Overriding

```
class Dog {
    public:
        void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class FrenchPoodle : public Poodle {
    public:
        void bark() { cout << "ouah! ouah!" << endl; }
};
```



Non Virtual functions ***redefine***
a Base Class's function

static and compile time

Destructors

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
        Poodle() { chewy = new Toy; }
        ~Poodle() { delete chewy; }
    private:
        Toy * chewy;
};
```


Poodle contains a dynamically allocated object (a chew `Toy`)

```
int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

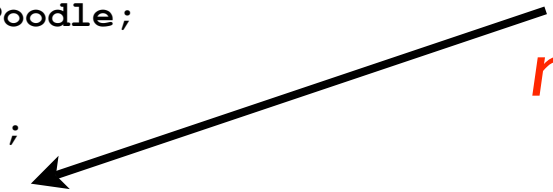
Destructors

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
        Poodle() { chewy = new Toy; }
        ~Poodle() { delete chewy; }
    private:
        Toy * chewy;
};



int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

Poodle object is referenced through a Base Class pointer.

Destructors

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
        Poodle() { chewy = new Toy; }
        ~Poodle() { delete chewy; }
    private:
        Toy * chewy;
};



int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

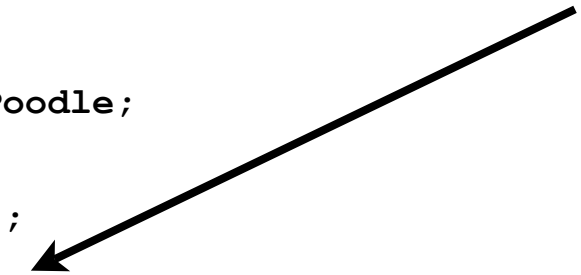


Will all of the dynamic memory be de-allocated?

Destructors

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
        Poodle() { chewy = new Toy; }
        ~Poodle() { delete chewy; }
    private:
        Toy * chewy;
};
```

```
int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

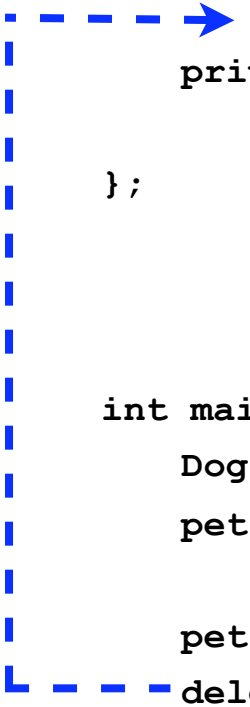


No.
Default Destructor is called (**~Dog()** *)*

Solution?

Virtual Destructors

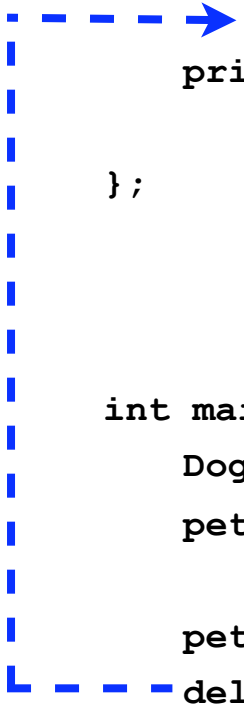
```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
        virtual ~Dog();
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
        Poodle() { chewy = new Toy; }
        ~Poodle() { delete chewy; }
    private:
        Toy * chewy;
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```



Declaring the Base Class Destructor Virtual guarantees that the *appropriate* Object's Destructor is called.

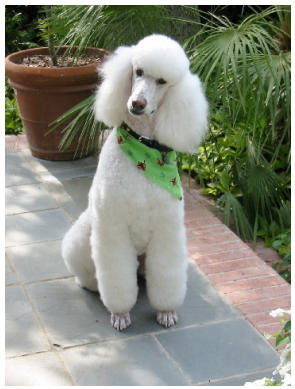
Virtual Destructors

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
        virtual ~Dog();
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
        Poodle() { chewy = new Toy; }
        ~Poodle() { delete chewy; }
    private:
        Toy * chewy;
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```



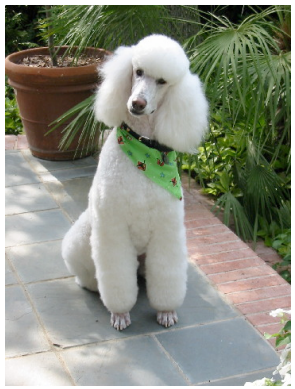
If the Base Class contains virtual functions, it is always good programming practice to declare the destructors **`virtual`**.

UML and Virtual Functions



Virtual Functions in the Base Class are commonly shown in *italics* in UML.

Abstract Base Class



Sometimes the Object Hierarchy should start with a common Base Class that is ***Generic*** and ***Abstract*** and would not ever be implemented itself.

Here, **`Poodles`** and **`Retrievers`** would be instantiated, but never the **`Dog`** class on its own. (Of course **`Dog`** pointers would still be used.)

Make `Dog` an Abstract Class

```
class Dog {
    public:
        virtual void bark() { cout << "bark bark!" << endl; }
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

Virtual functions in Abstract Base classes would not have an code associated to them.

Make `Dog` an Abstract Class

```
class Dog {
    public:
        virtual void bark() = 0;
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Poodle;

    pet->bark();
    delete pet;
}
```

They are *pure virtual* functions.

Indicated by the "= 0"

Have no body or definition in base class.

Make `Dog` an Abstract Class

```
class Dog {
    public:
        virtual void bark() = 0;
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Dog;

    pet->bark();
    delete pet;
}
```

An Abstract Class cannot be instantiated.

```
error: cannot allocate an object of abstract type 'Dog'
```

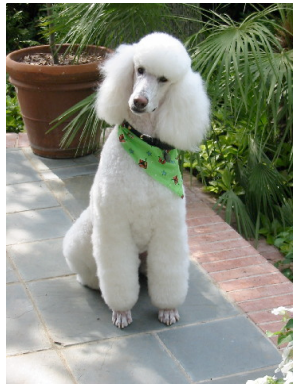
Multiple Inheritance

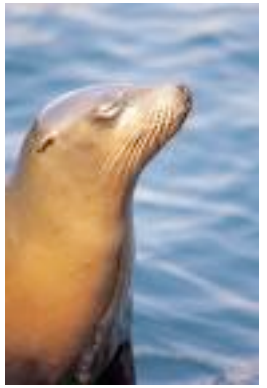
```
class Dog {
    public:
        virtual void bark() = 0;
};

class Poodle : public Dog {
    public:
        void bark() { cout << "woof! woof!" << endl; }
};

class Retriever : public Dog {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};

int main() {
    Dog * pet;
    pet = new Dog;

    pet->bark();
    delete pet;
}
```

An Abstract Class cannot be instantiated.

```
error: cannot allocate an object of abstract type 'Dog'
```

Sea Lions Bark too! Multiple Inheritance





C++ is an Object Oriented Language that supports Multiple Inheritance in classes, which allow for a more extensible hierarchies of Classes.

A Class can Inherit from more that One Class

```
class MarineMammal {
    ...
};

class Barkable {
    public:
        virtual void bark() = 0;
};

class SeaLion : public MarineMammal, public Barkable {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};
```

A `SeaLion` is a type of `MarineMammal` and is a `Barkable` object.

Inherits all members of both Classes (i.e. and their hierarchies).

Constructors are called in Order that they are Listed

1. 2.

```
class SeaLion : public MarineMammal, public Barkable {
    public:
        void bark() { cout << "arf! arf!" << endl;
        SeaLion() {}
};
```

3.

1. `MarineMammal` constructor is called.
2. `Barkable` constructor is called.
3. `SeaLion` constructor is called.

Date and Time Classes : Multiple Inheritance

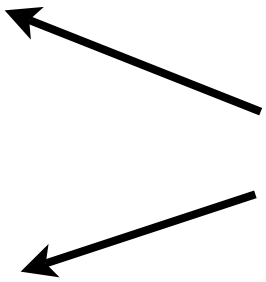
```
class Date {
   protected:
      int day;
      int month;
      int year;
   public:
      Date(int d, int m, int y) {
         day = d; month = m; year = y;
      }
};

class Time {
   protected:
      int hour;
      int min;
      int sec;
   public:
      Time(int h, int m, int s) {
         hour = h; min = m; sec = s;
      }
};
```

DateTime Combined Class with Constructor

```
class DateTime : public Date, public Time
{
   public:
      DateTime(int dy, int mon, int yr, int hr, int mn, int sc);
      ...
};


DateTime::DateTime(int dy, int mon, int yr, int hr, int mn, int sc) :
Date(dy, mon, yr), Time(hr, mt, sc)
{
   ...
}
```

DateTime Combined Class with Constructor

```
class DateTime : public Date, public Time
{
   public:
      DateTime(int dy, int mon, int yr, int hr, int mn, int sc);
      ...
};
```



Constructor Definition

```
DateTime::DateTime(int dy, int mon, int yr, int hr, int mn, int sc) :
Date(dy, mon, yr), Time(hr, mt, sc)
{
   ...
}
```

DateTime Combined Class with Constructor

```
class DateTime : public Date, public Time
{
   public:
      DateTime(int dy, int mon, int yr, int hr, int mn, int sc);
      ...
};


DateTime::DateTime(int dy, int mon, int yr, int hr, int mn, int sc) :
Date(dy, mon, yr), Time(hr, mt, sc)
{
   ...
}
```

Inherited Constructors

Interfaces

```
class MarineMammal {
    ...
};

class Barkable {
    public:
        virtual void bark() = 0;
};

class SeaLion : public MarineMammal, public Barkable {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};
```

An Abstract Base class can be thought of as an ***interface*** (through its pure virtual functions) for a class that derives from it.

Multiple Inheritance allows for multiple interfaces.

Interfaces

```
class MarineMammal {
    ...
};
```

```
class Barkable {
    public:
        virtual void bark() = 0;
};
```

Barkable Interface allows other classes to pass the bark() message to SeaLion.

```
class SeaLion : public MarineMammal, public Barkable {
    public:
        void bark() { cout << "arf! arf!" << endl; }
};
```

An Abstract Base class can be thought of as an ***interface*** (through its pure virtual functions) for a class that derives from it.

Multiple Inheritance allows for multiple interfaces.

Interfaces

```
class Printable {
   public:
      virtual void print(string printer) = 0;
};

class Emailable {
   public:
      virtual void email(string sender, string rcpt) = 0;
};

class WordDoc : public Document, public Printable, public Emailable {
   public:
      void WordDoc();
      ...
};
```

Java has a type of class that is called an Interface that serves this purpose.

More Stuff About Classes

Friends

A class can declare a function of another class or another class itself a ***friend***. A ***friend*** can have access to the private members of a class.

Friends

A class can declare a function of another class or another class itself a ***friend***. A ***friend*** can have access to the private members of a class.

This is specified as a list of functions and classes in the class description.

```
class Club {
   private:
      vector<string> VIPs;
   public:
      bool gainAccess(string name);

   friend void Promoter::addToVIPList(Club & c, string name);
};
```

Friend keyword

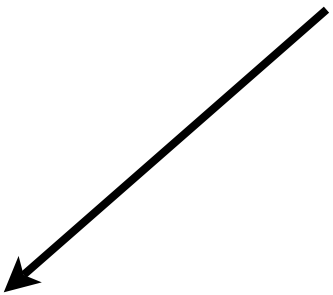
Friends

A class can declare a function of another class or another class itself a ***friend***. A ***friend*** can have access to the private members of a class.

This is specified as a list of functions and classes in the class description.

```
class Club {
   private:
      vector<string> VIPs;
   public:
      bool gainAccess(string name);

   friend void Promoter::addToVIPList(Club & c, string name);
};
```

Function that has private access



Friends

```
class Club {
   private:
      vector<string> VIPs;
   public:
      bool gainAccess(string name);

   friend void Promoter::addToVIPList(Club & c, string name);
};

class Promoter {
   public:
      void addToVIPList(Club & c, string name) {
         c.VIPs.push_back(name);
      }
};
```

Access to private member through dot-notation

Forward declaration of Promoter

```
class Promoter;

class Club {
    private:
        vector<string> VIPs;
    public:
        bool gainAccess(string name);

    friend void Promoter::addToVIPList(Club & c, string name);
};

class Promoter {
    public:
        void addToVIPList(Club & c, string name) {
            c.VIPs.push_back(name);
        }
};
```

Need to let the Club Class know that the Promoter class exists.

Can also friend Classes as a whole

```
class Promoter;

class Club {

    private:

        vector<string> VIPs;

    public:

        bool gainAccess(string name);


    friend class Promoter;

};


class Promoter {

    public:

        void addToVIPList(Club & c, string name) {

            c.VIPs.push_back(name);

        }

};
```

Static Members of Classes

Classes can have static member variables, single instances of a variable which do not belong to any specific object instance of that class.

They act like global variables.

```
class Tree {
    private:
        static int count;
    public:
        Tree() {
            count++;
        }
        ~Tree() {
            count--;
        }
        int getCount() { return count; }
);


int Tree::count = 0;
```

Static Members of Classes

Classes can have static member variables, single instances of a variable which do not belong to any specific object instance of that class.

They act like global variables.

```
class Tree {
    private:
        static int count;
    public:
        Tree() {
            count++;
        }
        ~Tree() {
            count--;
        }
        int getCount() { return count; }
);
```

Static Member Declared

Static Member Defined

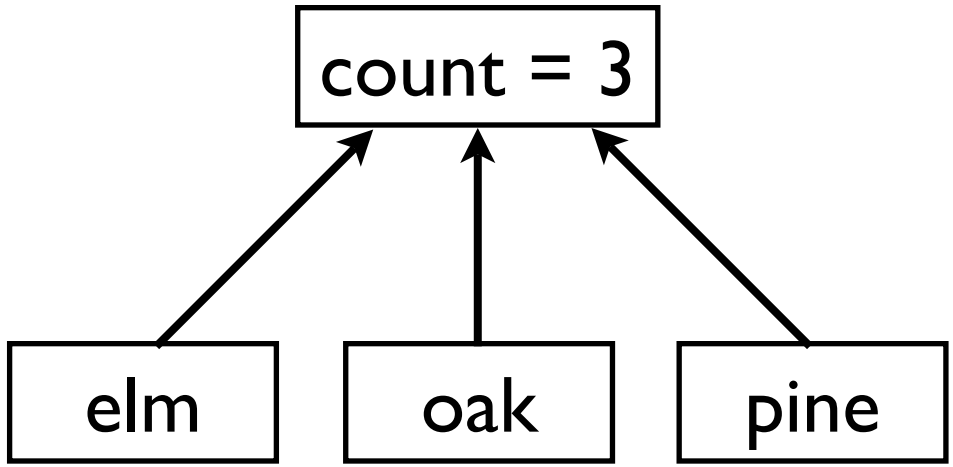
```
int Tree::count = 0;
```

Static Members of Classes

```
class Tree {
    private:
        static int count;
    public:
        Tree() {
            count++;
        }
        ~Tree() {
            count--;
        }
        int getCount() const { return count; }
);
int Tree::count = 0;


int main() {
    Tree elm;
    Tree oak;
    Tree pine;

    cout << "Trees so far: " << elm.getCount() << endl;
}
```

Static Members of Classes

```
class Tree {
  private:
      static int count;
  public:
      Tree() {
         count++;
      }
      ~Tree() {
         count--;
      }
      int getCount() const { return count; }
);
int Tree::count = 0;

int main() {
  Tree elm;
  Tree oak;
  Tree pine;

  cout << "Trees so far: " << elm.getCount() << endl;
}
```



Trees so far: 3

Member Functions can be static as well.

```
class Tree {
  private:
      static int count;
  public:
      Tree() {
         count++;
      }
      ~Tree() {
         count--;
      }
      static int getCount() const { return count; }
);
int Tree::count = 0;


int main() {
  Tree elm;
  Tree oak;
  Tree pine;


  cout << "Trees so far: " << Tree::getCount() << endl;
}
```

Can be accessed without an object instance.

Accessed with the :: scope operator

Static Functions can only access Static Members

```
class Tree {
  private:
       static int count;
  public:
       Tree() {
          count++;
       }
       ~Tree() {
          count--;
       }
       static int getCount() const { return count; }
);
int Tree::count = 0;

int main() {
  cout << "Trees so far: " << Tree::getCount() << endl;
}
```

What is returned in this case?

Static Functions can only access Static Members

```
class Tree {
  private:
       static int count;
  public:
       Tree() {
          count++;
       }
       ~Tree() {
          count--;
       }
       static int getCount() const { return count; }
);
int Tree::count = 0;

int main() {
  cout << "Trees so far: " << Tree::getCount() << endl;
}
```

0

Midterm Review on Thursday.