Polymorphism

Partial Listing of the MFC Class Hierarchy

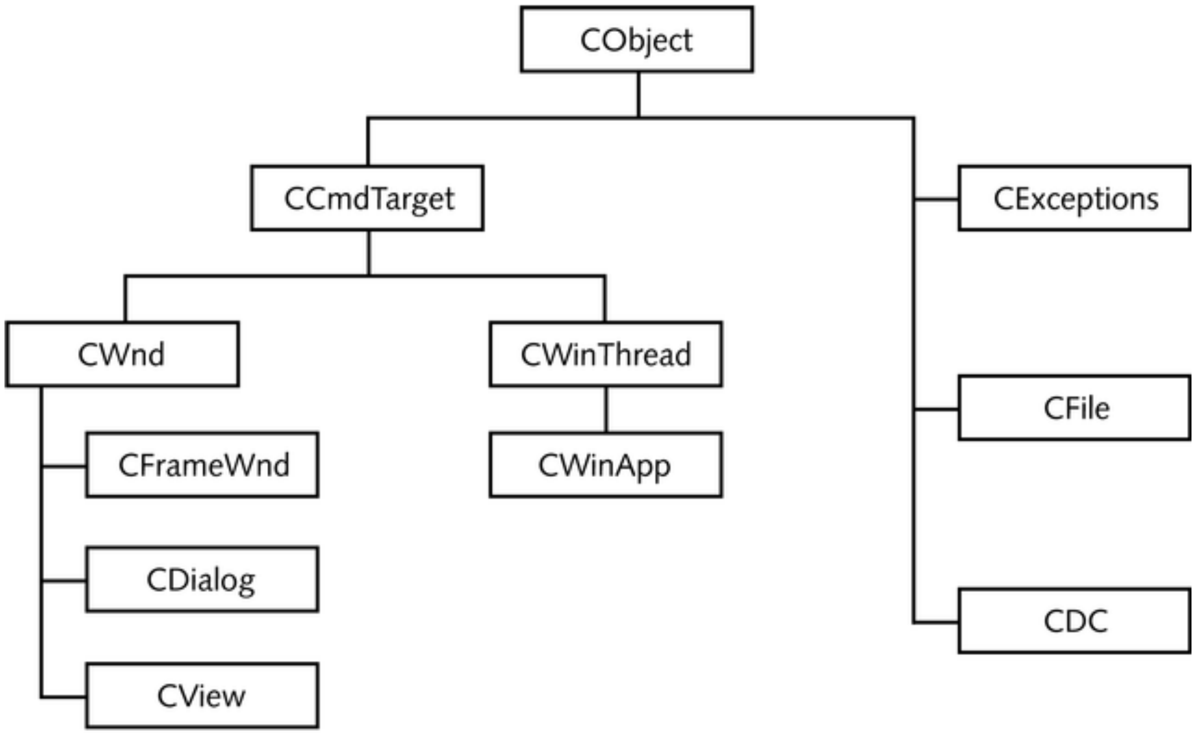


Figure 10-5 Partial listing of the MFC class hierarchy

http://msdn.microsoft.com/ko-kr/library/ws8s10w4(v=vs.100).aspx

Review — Accessing Members of Base and Derived Classes

```
class B {
public:
   void m();
   void n();
   ...
} // class B

class D: public B {
public
   void m();
   void p();
   ...
} // class D
```

- The following are legal:–

```
B_obj.m() //B's m()
B_obj.n()

D_obj.m() //D's m()
D_obj.n() //B's n()
D_obj.p()

B_ptr->m() //B's m()
B_ptr->n()

D_ptr->m() //D's m()
D_ptr->n() //B's n()
D_ptr->p()
```

Review — Accessing Members of Base and Derived Classes (continued)

```
class B {
public:
   void m();
   void n();
   ...
} // class B

class D: public B {
public
   void m();
   void p();
   ...
} // class D
```

Class **D** *redefines* method **m()**

- The following are legal:–

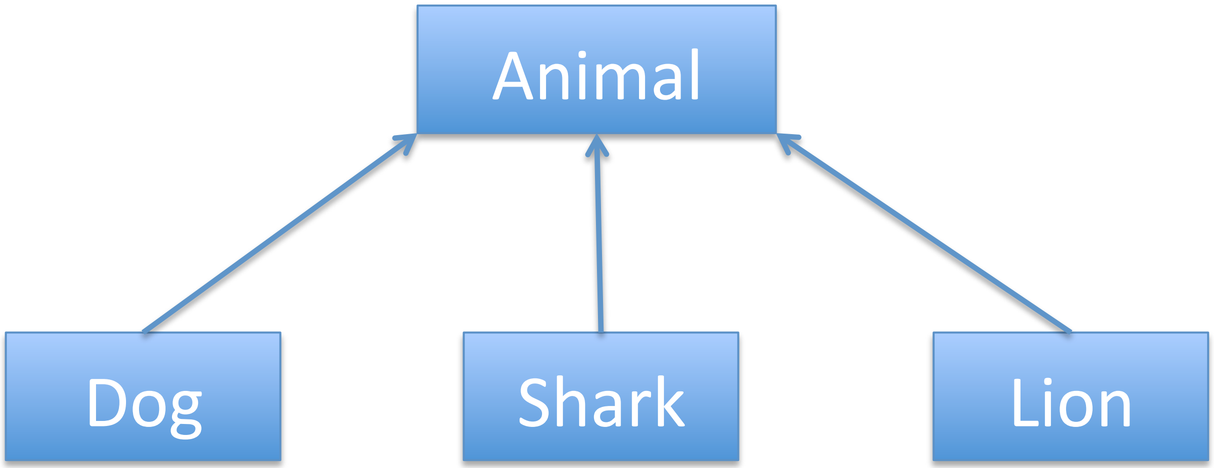
  B_ptr = D_ptr;

- The following are *not* legal:–

  D_ptr = B_ptr;
  B_ptr->p();

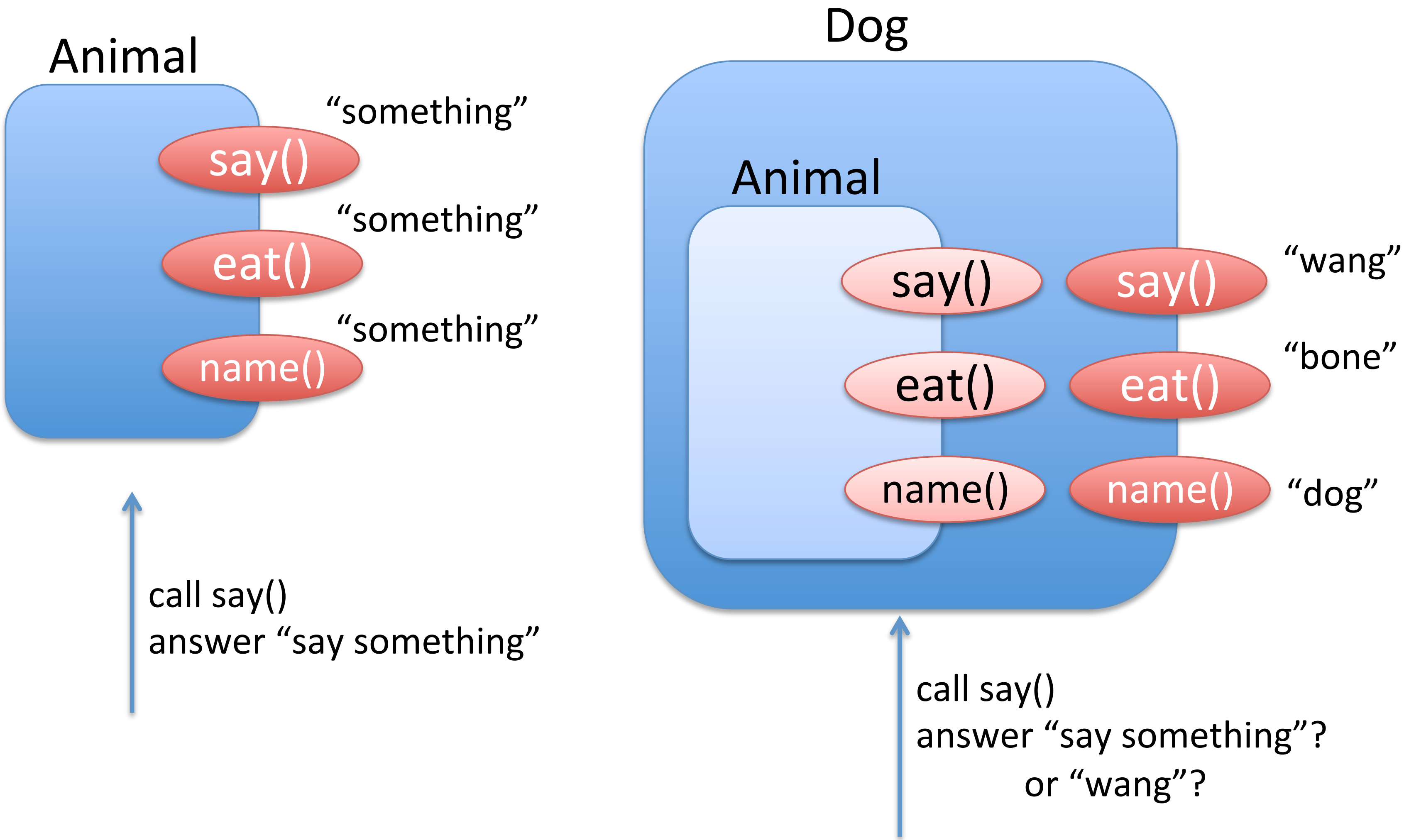
  Even if **B_ptr** *is known to point to an object of class* **D**

Class Hierarchy



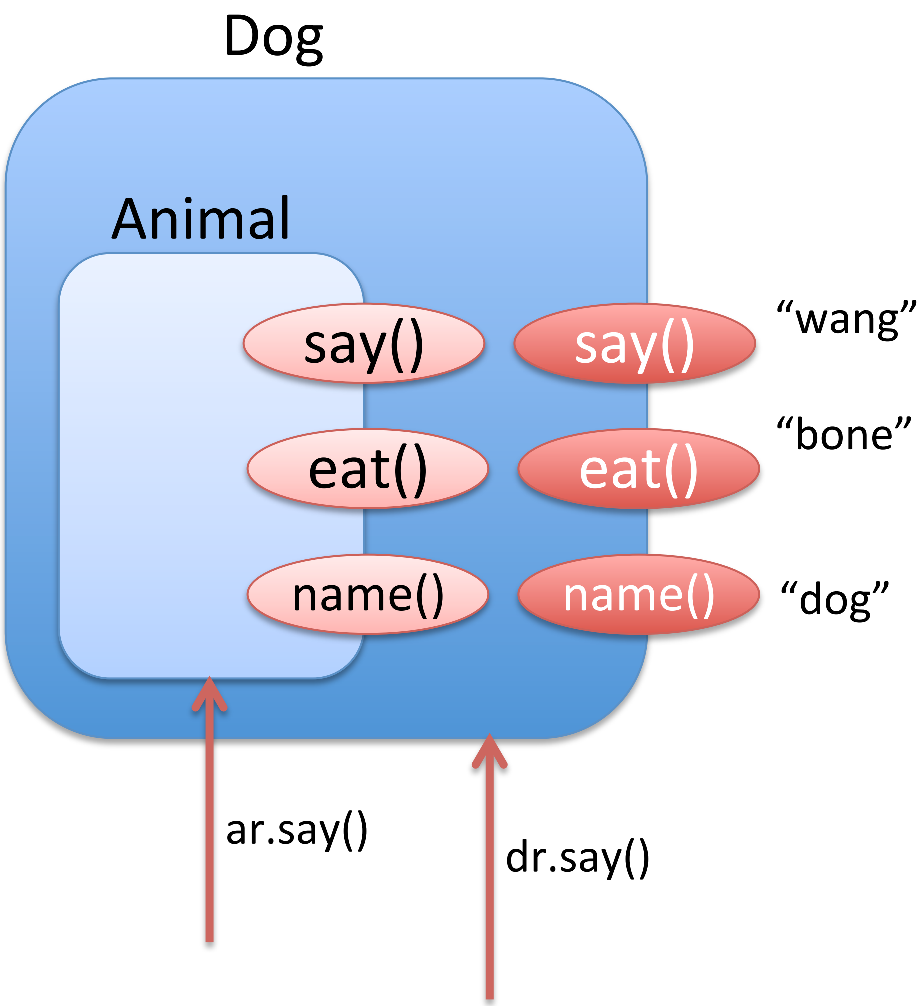
- Define how animals make sound and eat
  - say(), eat()

Call ambiguity



Call ambiguity

- Dog zong;
- zong.say();
- Dog &dr = zong;
- dr.say();
- Dog *dp = &zong;
- dp->say();
- Animal &ar = zong;
- ar.say();
- Animal *ap = &zong;
- ap->say();



Why Polymorphism

- Cat is an Animal (Cat is a type of Animal)
- *Upcasting* is possible
- A function:

```
explain( Animal &an ) {
    cout << an.name() << " eats " << an.eat() << endl;
}
```

- explain(zong)

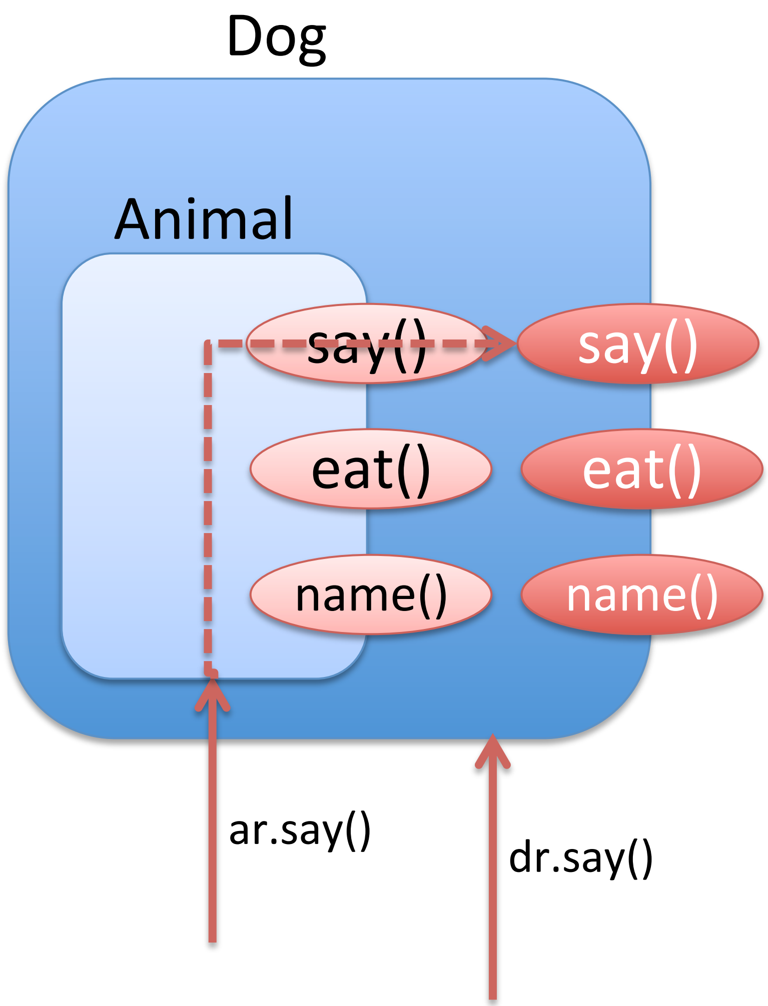
Polymorphism

- *Polymorphism:*– from the Greek "having multiple forms"
    - In programming languages, the ability to assign a different meaning or usage to something in different contexts
- The ability to declare functions/methods as **`virtual`** is one of the central elements of polymorphism in *C++*

Polymorphism

- make a function *virtual*

```
class Animal {
    virtual void say();
}
class Dog : public Animal
{
    void say();
}
```



Abstract and Concrete Classes

- *Abstract Classes*
  - Classes from which it is never intended to instantiate any objects
    - Incomplete—derived classes must define the “missing pieces”.
    - Too generic to define real objects.
  - Normally used as base classes and called *abstract base classes*
    - Provide appropriate base class frameworks from which other classes can inherit.
- *Concrete Classes*
  - Classes used to instantiate objects
  - Must provide implementation for *every* member function they define

Abstract Class

- A class only for polymorphism
- Has interfaces, but no implementation
  - pure virtual function

```
class Animal {
    virtual void say() = 0;
    virtual void eat() = 0;
    virtual void name() = 0;
}
```

Pure `virtual` Functions

- A class is made *abstract* by declaring one or more of its virtual functions to be "pure"
  - I.e., by placing `"= 0"` in its declaration

- Example

```
virtual void draw() const = 0;
```

  - `"= 0"` is known as a *pure specifier*.
  - Tells compiler that there *is no* implementation.

Pure `virtual` Functions (continued)

- Every *concrete* derived class must override all base-class pure **`virtual`** functions
  - with concrete implementations

- If even one pure virtual function is not overridden, the derived-class will also be *abstract*
  - Compiler will refuse to create any objects of the class
  - Cannot call a constructor

Purpose

- When it does not make sense for base class to have an implementation of a function

- Software design requires *all* concrete derived classes to implement the function
  - Themselves

Why Do we Want to do This?

- To define a *common public interface* for the various classes in a class hierarchy
  - Create framework for abstractions defined in our software system

- The heart of *object-oriented programming*

- Simplifies a lot of big software systems
  - Enables code re-use in a major way
  - Readable, maintainable, adaptable code

Abstract Classes and Pure `virtual` Functions

- *Abstract* base class can be used to declare pointers and references referring to objects of any derived concrete class

- Pointers and references used to manipulate derived-class objects polymorphically

- Polymorphism is particularly effective for implementing layered software systems – e.g.,
    1. Reading or writing data from and to devices.
    2. *Iterator* classes to traverse all the objects in a container.

Example – Graphical User Interfaces

- All objects on the screen are represented by derived classes from an abstract base class

- Common windowing functions
  - Redraw or refresh
  - Highlight
  - Respond to mouse entry, mouse clicks, user actions, etc.

- Every object has its own implementation of these functions
  - Invoked polymorphically by windowing system