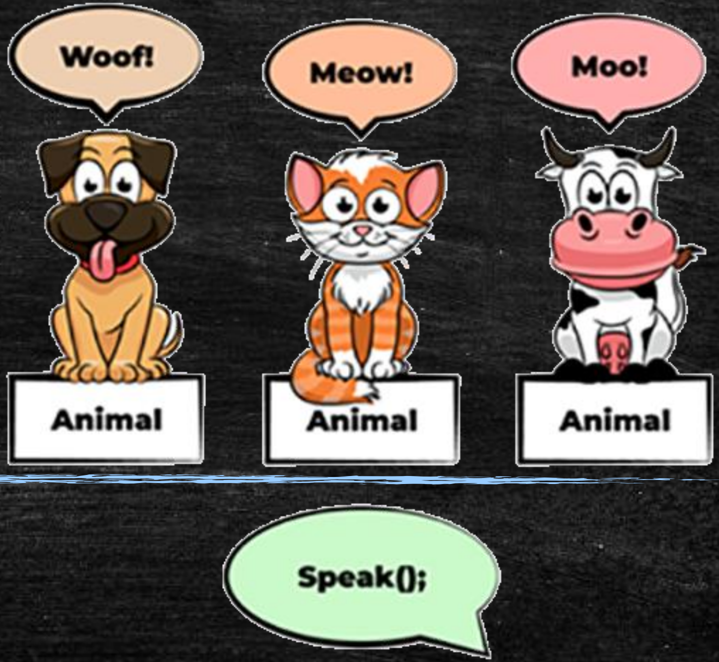
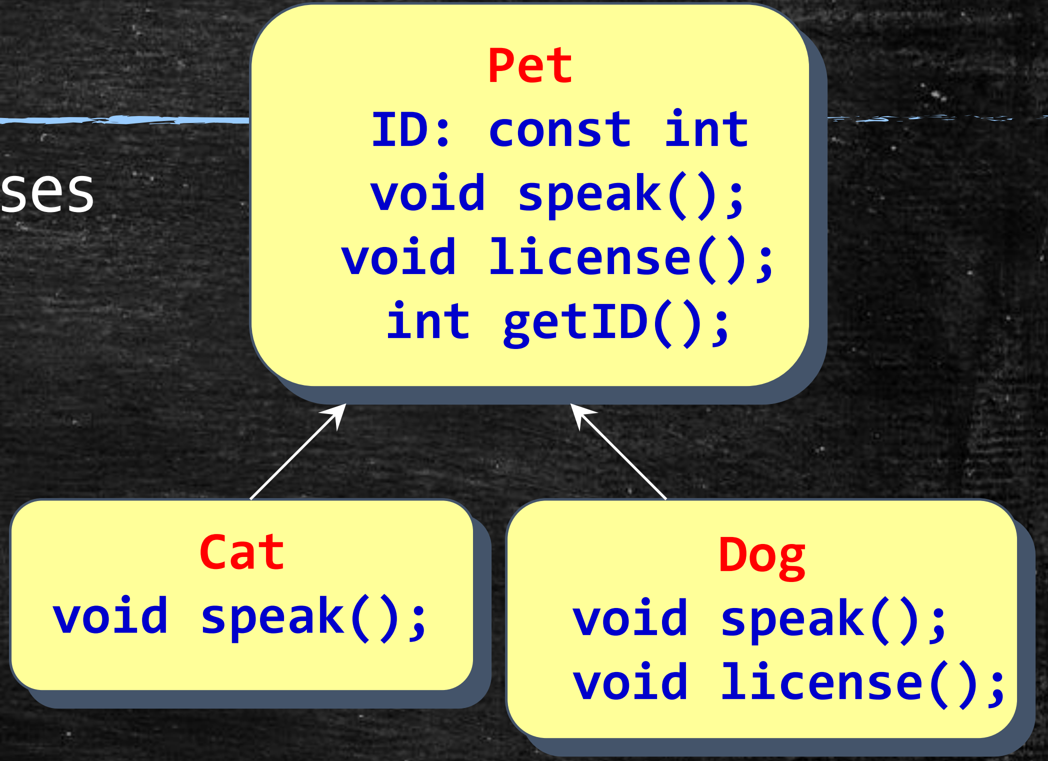
Polymorphism



CS 150 – C++ Programming I
Lecture 28

Polymorphism

- In *Pet01* you have the following classes
  - Pet, Cat, Dog
  - Every Cat and every Dog *IS-A* Pet
- Cat inherits 2 member functions
  - `license()` and `getID()`
  - It redefines `speak()`
- Dog inherits getID()
  - It edefines `speak()` and `license()`



```
Pet
ID: const int
void speak();
void license();
int getID();
```

```
Cat
void speak();
```

```
Dog
void speak();
void license();
```

Static Polymorphism

```
// 1. Create objects of each class
Pet p; Cat c; Dog d;

// 2. Ask each of them to speak!
cout << "Asking three types of pet to speak" << endl;
p.speak();
c.speak();
d.speak();
```

- Each object responds appropriately to the request
  - Compiler looks at the declared type of p, d & c
  - Calls the correct function based upon that type
  - The type must be known at compile-time
- Called static polymorphism or early binding

Polymorphic Functions

- Instead, what we really want are polymorphic functions
  - Instead of writing several overloaded functions like this:

```
void show(Cat c){ c.speak(); }
void show(Dog d){ d.speak(); }
```

  - Why can't we do this instead?

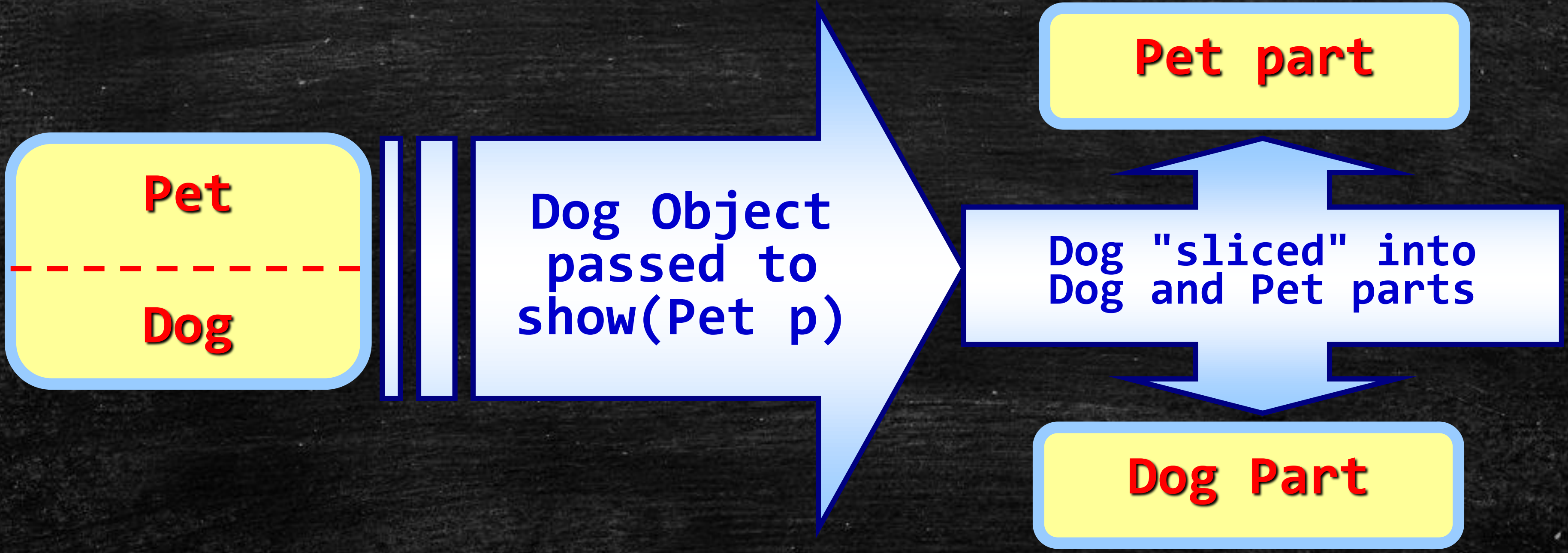
```
void show(Pet p){ p.speak() }
void show(Pet* p) { p->speak(); }
```

  - And then call `show(myDog);`

- After all, every Dog ***IS-A*** Pet, right? So, that should work!

The Slicing Problem

- What happens when you pass a Dog object to `show(Pet p)`
  - Moral? You cannot assign a derived object to a base variable



Pet

Dog

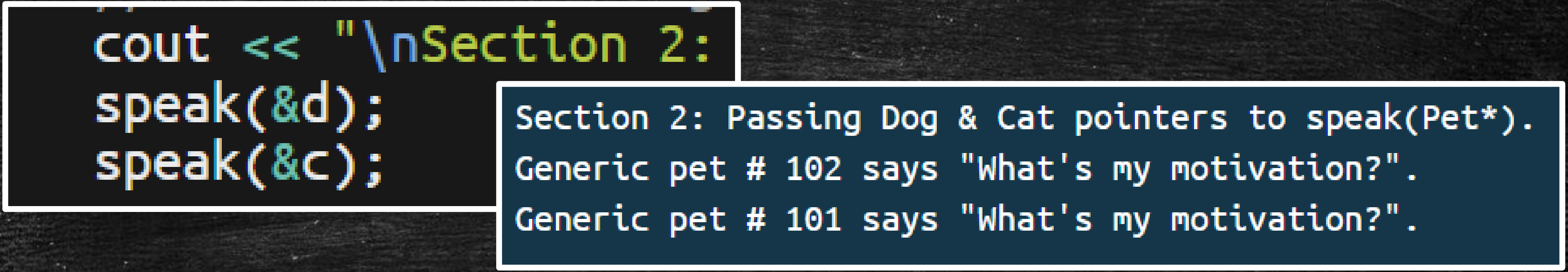
Dog Object passed to show(Pet p)

Dog "sliced" into Dog and Pet parts

Pet part

Dog Part

Pointers and Derived Objects



```
    cout << "\nSection 2:
    speak(&d);
    speak(&c);
```

```
Section 2: Passing Dog & Cat pointers to speak(Pet*).
Generic pet # 102 says "What's my motivation?".
Generic pet # 101 says "What's my motivation?".
```

- But, what about Section 2 where we pass pointers?
  - No slicing occurs when you pass a pointer or reference
- Still doesn't work, unfortunately
  - Function calls are still bound based on declared pointer type
  - Must know which function to call at compile time
- Does this mean that polymorphic functions are impossible?

Late Binding

- We want to tell the function that the `Pet* p` parameter actually points to a `Cat` or `Dog` object
  - We want to wait until runtime to decide which function to call
  - This is called late binding
- The keyword `virtual` in the base class enables late binding
  - This may (but need not) be repeated in the derived class
- Unlike Java, C++ allows both early and late binding
  - That's because early (static) binding is a little more efficient
  - Late binding uses indirection to call the function (a little slower)
  - Each object has a `v-pointer` which points to a `v-table`

Calling Virtual Functions

- Call a `virtual` member function through a base pointer
  - `Cat c; Pet* p = &c; p->speak();`
- Call a polymorphic function only with a pointer or reference
  - `void speak(Pet p) {p.speak();} // not polymorphic`
  - `void speak(Pet *p) {p->speak();} // polymorphic`
  - The polymorphic function must call a virtual function
  - `speak(&c);` the compiler uses the pointer's dynamic type to decide exactly which function to call (`Cat::speak` in this case)
  - This decision is made at run-time

Polymorphism at Work

- Exercise: 4 different kinds of cards; should print:

  -
    ```
    Name: John Doe
    Name: John Smith
    ID number: 0800640674
    Name: Star Card
    Card number: 12398437
    Name: John Doe
    Expiration date: 09/30/2009
    ```
- Problem: prints only name and leaks memory
  - Use `make check` or `make grind` to see the errors

Adding Destructors

- *`Billfold`* class stores pointers to objects in a *`vector`*
  - The objects have been created on the heap with `new`
  - The memory is not freed when the *`Billfold`* is destroyed
- A class that manages dynamic memory needs a destructor
  - Each class can have only one destructor (not overloaded)
  - Syntax: `~Billfold()`
  - Should walk through *`vector`* and `delete` each pointer

Virtual Functions & Destructors

- Even though each specific card type has it's own `print()` member function, it isn't being called. Why?
  - Because `print()` isn't declared as `virtual` in `Card`
- Exercise: make `print()` `virtual` and rebuild
- Whoa!!! Why all the new errors?
  - What happens when you delete a `Card*`?
  - Deletes `sizeof(Card)` amount of memory from the heap
  - With `virtual` functions, object on heap may not *be* a `Card`
  - Add `virtual` destructor to classes with `virtual` members

The Stack Classes

- The *Stack* is a classic data structure or ADT
  - Fundamental for many different algorithms
  - Last in-First out sequence (LIFO)
  - Elements added and removed only from one end (top)
  - ADT operations: `push`, `pop`, `top`, `empty`
- Example: a Reverse Polish Notation (RPN) calculator
  - Expressions put operators after the operands: `2 3 +`
  - Operands are popped and answer is pushed on to the stack
  - Starting code uses standard library `stack<double>`
  - Type alias allows us to use the simpler name *Stack*

Stack: public Inheritance

- Exercise 1: public inheritance from *vector*
  - Add inline members push, pop, top, (don't need empty)
  - Call inherited members in definition of inline members
- This works, but there is really a major problem!
  - *Stack* can use *all* of the *vector* methods; that means it can access other elements besides the top and can change the contents of the stack (think deck of cards!)
  - That violates the *Stack* interface or contract!
  - Using public inheritance says that a *Stack* IS-A *vector*!!!
  - That is not true, so public inheritance is inappropriate

Stack: Composition & Layering

- Exercise 2: A better technique is composition & layering
  - Add a `private` *`vector`* data member
  - Add inline member functions `push`, `pop`, `top`, `empty`
  - Forward all requests to the embedded data member
- This kind of class is called an adapter class
  - It adapts the interface of one class, changing it to another
  - This use of composition is known as layering
  - It is slightly different than a has-a, or whole-part relationship

Stack: private Inheritance

- Exercise 3: with private inheritance, the derived class inherits the implementation, but not the interface
  - Private inheritance is an implemented with relationship
- Selectively import the base-class interface like this:
  - public:

```
public:
        using vector::size;
        using vector::push_back;
```

  - Prevents unwanted members being called (unlike IS-A)
  - You can't automatically change the function names
  - You must use inline delegating functions just like layering

What *these* Features MEAN?

- Public Inheritance means IS-A
  - If class D publicly inherits from B, every object of type D is also an object of type B, but not vice-versa.
  - Public inheritance means substitutability
- Private Inheritance: Implemented With
  - If class D privately inherits from B, there is no conceptual relationship between the classes
- Composition: Has-A or Implemented With
  - Can be used as whole-part or for layering

Abstract Functions and Classes

- Design of Pet hierarchy has problem
  - "Generic" pets can't really speak
  - But, we can't remove the speak() member function
  - We would lose the ability to call speak() polymorphically
- Solution: create an abstract function
  - In C++ terminology this is called a pure virtual function
  - `virtual void speak() = 0;`
- Class becomes an ABC or Abstract Base Class
  - Cannot instantiate; only inherit from
  - Concrete derived classes must implement the function

What do *these* features mean?

- Regular virtual functions: children inherit the interface plus a *default* implementation
  - Child *may* (but are not required to) override
- Non-virtual functions: children inherit the interface plus a *mandatory* implementation
  - Child *can't* override (can replace; *shouldn't*)
- Pure virtual function means a mandatory override for the derived class
  - Parent *may* provide partial implementation