CSCI 200: Foundational Programming Concepts & Design Lecture 29

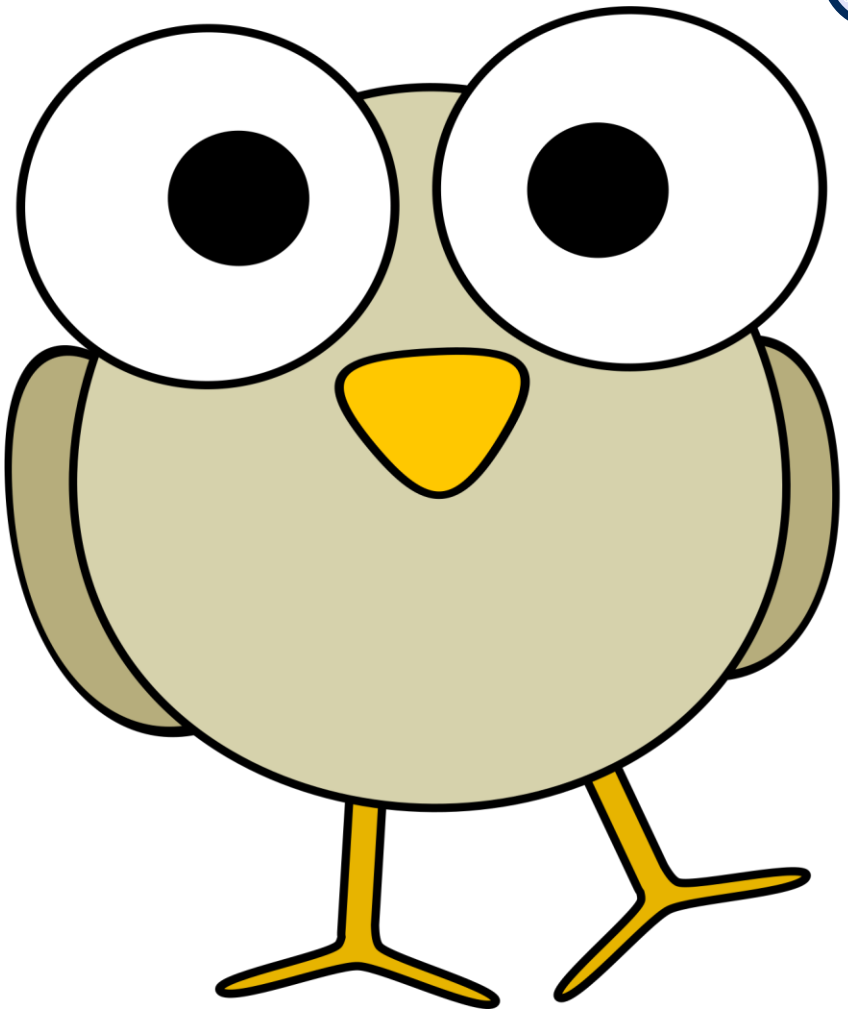
Object-Oriented Programming & Inheritance: Runtime Polymorphism

Download Lecture 29 Starter Code

Previously in CSCI 200

- Subtype Polymorphism
  - Object can behave as both derived class type or base class type
  - Type of object determined at compile time
- Children can override parent method

Questions?



Learning Outcomes For Today

- Give examples of polymorphism at run-time through subtype polymorphism with virtual functions.

On Tap For Today

- Polymorphism
  - Compile Time
- Virtual Functions
  - Run Time Polymorphism
- Practice

On Tap For Today

- Polymorphism
  - Compile Time
- Virtual Functions
  - Run Time Polymorphism
- Practice

poly·morph·ism

- *poly* – many
- *morph* – form / behavior
- *ism* – imitation of

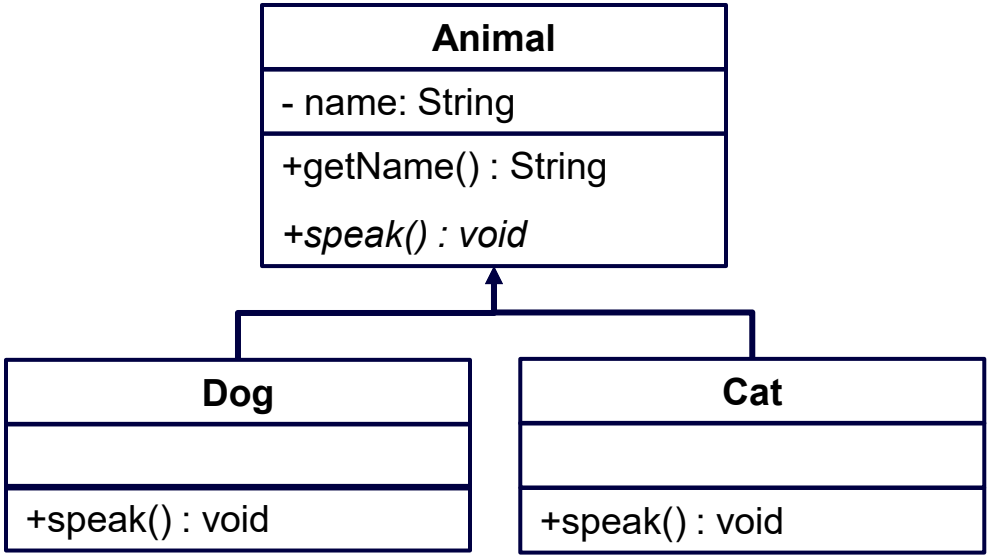
- *polymorphism:*
  - having many forms
  - having many behaviors

Polymorphism

```
Dog odie;
Cat garfield;

cout << odie.getName() << " ";
odie.speak();

cout << garfield.getName() << " ";
garfield.speak();
```



- **odie** is a **Dog** and an **Animal**
- **garfield** is a **Cat** and an **Animal**
  - Can exhibit behaviors of different types

Subtype Polymorphism

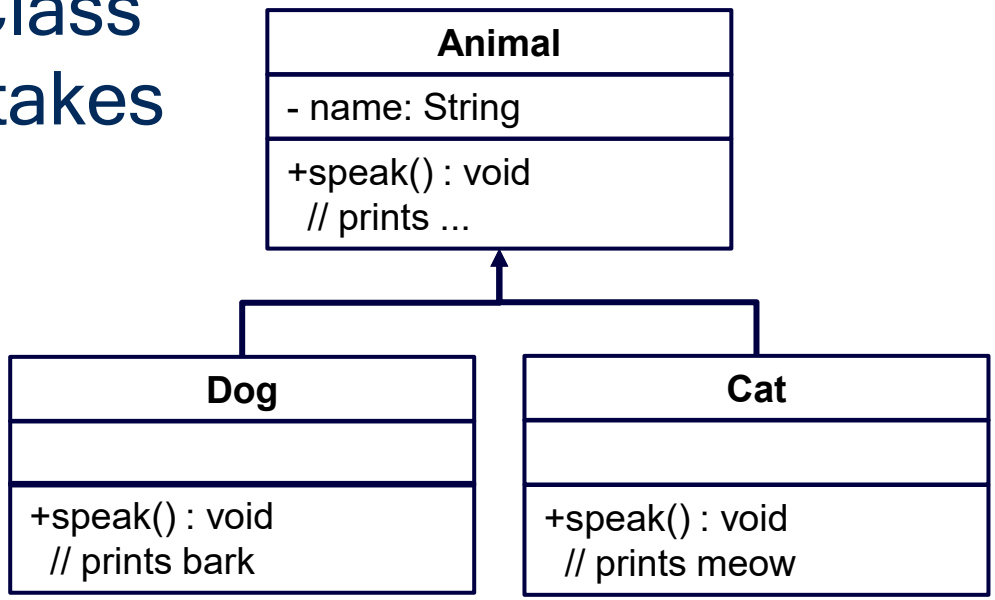
```
Dog odie;
cout << odie.getName() << " ";  // treat odie as an Animal
odie.speak();                   // treat odie as a Dog
```

- **`odie`** is a **`Dog`** and an **`Animal`**
  - Can exhibit behaviors of different types

- At compile-time, form & behavior is known

Overriding Functions

- Overridden Functions
  - Derived Class has member function with same function name and signature as Base Class
- The Derived Class implementation overrides the Base Class implementation and takes precedence
  - Resolve bottom-up



Overridden Functions

```
class Animal {
public:
  virtual void speak() const { cout << "..." << endl; }
};
class Dog : public Animal {
public:
  void speak() const override { cout << "bark" << endl; }
};
// ...
Dog odie;
odie.speak();             // prints bark -- odie is a Dog
((Animal)odie).speak(); // prints ...  -- odie is an Animal
odie.Animal::speak();   // prints ...  -- odie is a Dog, explicitly call Animal
odie.Dog::speak();      // prints bark -- odie is a Dog, explicitly call Dog
```

- To call a specific form, either
  - Cast object type
  - Use scope resolution

More Polymorphism Concerns

```
void hearAnimal(Animal animal) {
  animal.speak();
}

void hearDog(Dog dog) {
  dog.speak();
}
// ...
Cat garfield;
hearDog(garfield);          // error! -- garfield is a Cat, not a Dog
hearAnimal(garfield);       // prints ...  -- garfield is an Animal
```

- Class Cast Error → Compiler Error!
- Polymorphism checked at compile-time

More Concerns

```
Animal john;
Dog odie;
Cat garfield;


vector<Animal> animals(3);

animals[0] = john;      // assign Animal form of john
animals[1] = odie;      // assign Animal form of odie
animals[2] = garfield; // assign Animal form of garfield
                        // implicit casting occurs


for(int i = 0; i < animals.size(); i++) {
  // animals[i] is an Animal, use Animal::speak()
  animals[i].speak();
}
```

Want

```
Animal john;
Dog odie;
Cat garfield;


vector<Animal> animals(3);

animals[0] = john;      // assign Animal form of john
animals[1] = odie;      // assign Dog form of odie
animals[2] = garfield; // assign Cat form of garfield

for(int i = 0; i < animals.size(); i++) {
  // if animals[i] is a Dog, use Dog::speak()
  // if animals[i] is a Cat, use Cat::speak()
  animals[i].speak();
}
```

On Tap For Today

- Polymorphism
  - Compile Time
- Virtual Functions
  - Run Time Polymorphism
- Practice

Compile Time Polymorphism

- Implementation tied to type of object
  - Type is static
  - Known at compile time

```
vector<Animal> animals(3);
animals[0] = john;      // copy Animal john
animals[1] = odie;      // copy Animal odie
animals[2] = garfield; // copy Animal garfield
for(int i = 0; i < animals.size(); i++) {
  // animals[i] is an Animal, use Animal::speak()
  animals[i].speak();
}
```

Run Time Polymorphism

- Implementation is resolved by type of object
  - Type is dynamic
  - Known at run time

```
vector<Animal*> animals(3);
animals[0] = &john;     // point to Animal john
animals[1] = &odie;     // point to Dog odie
animals[2] = &garfield; // point to Cat garfield
for(int i = 0; i < animals.size(); i++) {
  // if animals[i] points to Animal, use Animal::speak()
  // if animals[i] points to Dog, use Dog::speak()
  // if animals[i] points to Cat, use Cat::speak()
  animals[i]->speak();
}
```

Virtual Classes

- Classes with virtual functions need a virtual destructor
  - When deleting pointer, need to delete subtype object
- Explicitly declare parent destructor as virtual
  - Typically don't mark child destructor as override (names don't match)

```
class Animal {
public:
  virtual ~Animal() { cout << "Destroying an animal" << endl; }
  virtual void speak() const { cout << "..." << endl; }
};

class Dog : public Animal {
public:
  ~Dog() { cout << "Destroying a dog" << endl; }
  void speak() const override { cout << "bark" << endl; }
};
```

On Tap For Today

- Polymorphism
  - Compile Time
- Virtual Functions
  - Run Time Polymorphism
- Practice

To Do For Next Time

- Start Set5
- Be working on Final Project