Inheritance

CMSC 202

Warmup

Identify which constructor each of the following use (default, non-default, copy)

```
MyClass a;
MyClass b(a);
MyClass c(2);
MyClass* d = new MyClass;
MyClass* e = new MyClass(*d);
MyClass* f = new MyClass(4);
```

Code Reuse

How have we seen Code Reuse so far?
- Functions
  - Function Libraries
    - Ex: math -> pow, sqrt
- Classes
  - Class Libraries
    - Ex: vector, string
- Aggregation
  - Customer “has-a” DVD
  - RentalSystem “has-a” Customer

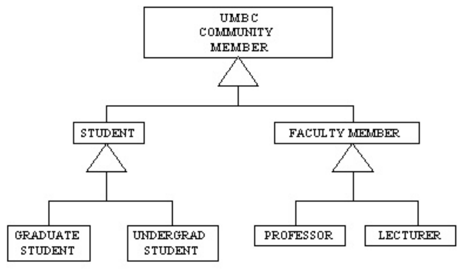
Object Relationships

- "Uses a"
  - Object_A "uses a" Object_B
    - Ex: Student sits in a chair
- "Has a"
  - Object_A "has a" Object_B
    - Ex: Student has a name
- "Is a" or "Is a kind of"
  - Object_A "is a" Object_B
    - Ex: Student is a kind of Person

Inheritance

- What is Inheritance?
  - Unfortunately – not what your parents/grandparents will be giving you…
- Inheritance
  - "is a" or "is a kind of" relationship
  - Code reuse by sharing related methods
  - Specific classes "inherit" methods from general classes
- Examples
  - A student is a person
  - A professor is a faculty member
  - A lecturer is a faculty member

Inheritance Hierarchy



An Inheritance Heirarchy

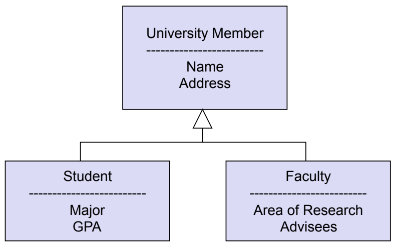
Why Inheritance?

- Abstraction for sharing similarities while retaining differences
- Group classes into related families
  - Share common operations and data
- Multiple inheritance is possible
  - Inherit from multiple base classes
  - Not advisable
- Promotes code reuse
  - Design general class once
    - Extend implementation through inheritance

Inheritance and Classes

- Base class (or superclass)
  - More general class
  - Contains common data
  - Contains common operations
- Derived class (or subclass)
  - More specific class
  - Inherits data from Base class
  - Inherits operations from Base class
  - Uses, modifies, extends, or replaces Base class behaviors

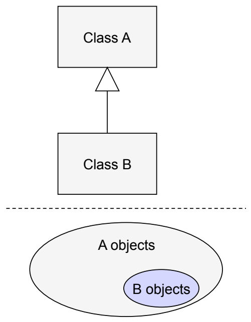
Inheritance Example



Inheritance

Assume the hierarchy on the right…
- A is Base class
- B is derived class
- B derives from A

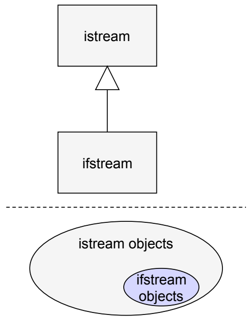
- Every B is an A
- Every A is NOT a B
- Some A' s are B' s



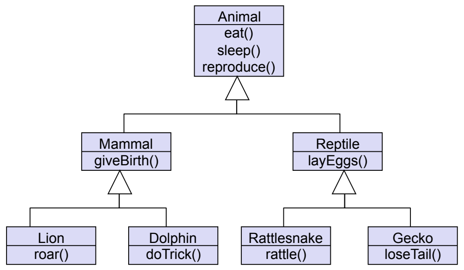
Inheritance

Assume the hierarchy on the right…
- Everywhere an A can be used, a B can be used
  - Parameters
  - Return values
  - Items in vectors
  - Items in arrays
- Reverse is not true…

Inheritance so far?
- ifstream is an istream
- ofstream is an ostream



Trip to the Zoo



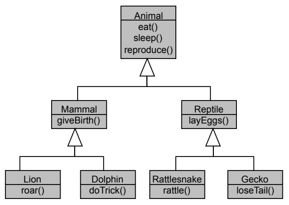
Inheritance

```
class BaseClass
{
    public:
        // operations
    private:
        // data
};

class DerivedClass : public BaseClass
{
    public:
        // operations
    private:
        // data
};
```

Indicates that this derived class inherits data and operations from this base class

Inheritance in Action

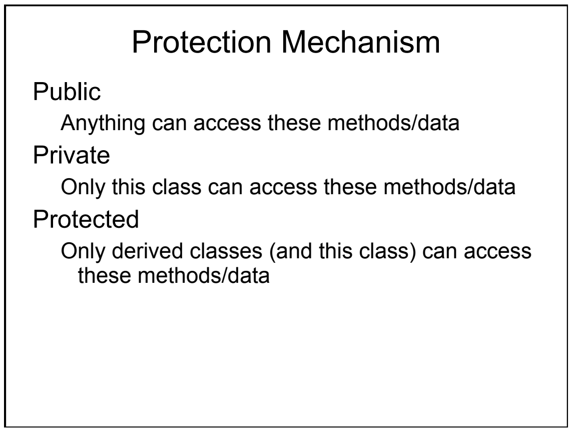
```
class Animal
{ };

class Mammal : public Animal
{ };

class Lion : public Mammal
{ };

class Dolphin : public Mammal
{ };

class Reptile : public Animal
{ };

class Gecko : public Reptile
{ };

class Rattlesnake : public Reptile
{ };
```



Inherited Functionality

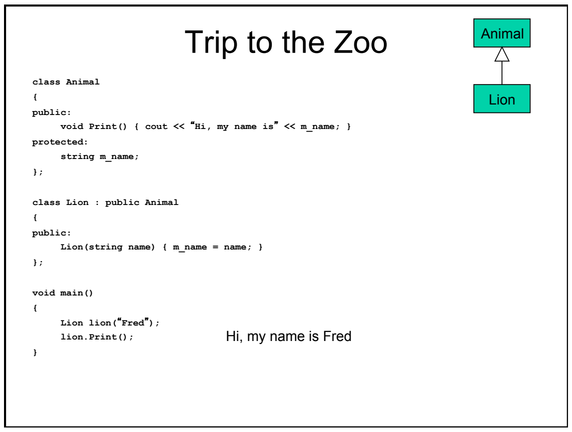
Derived class
- Has access to all public methods of base class
- “Owns” these public methods
  - Can be used on derived class objects!

```
BaseClass b;
b.BaseClassMethod();
~~b.DerivedClassMethod();~~

DerivedClass d;
d.BaseClassMethod();
d.DerivedClassMethod();
```

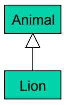


Protection Mechanism

- Public
  - Anything can access these methods/data
- Private
  - Only this class can access these methods/data
- Protected
  - Only derived classes (and this class) can access these methods/data

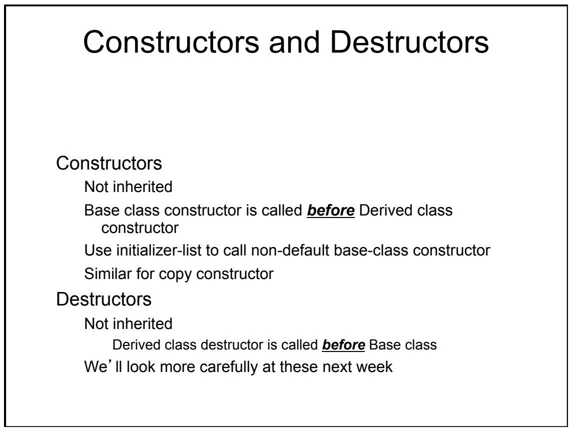


Trip to the Zoo



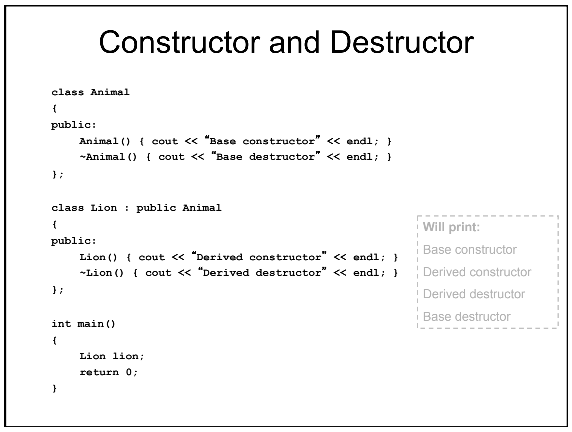
```
class Animal
{
public:
     void Print() { cout << "Hi, my name is" << m_name; }
protected:
     string m_name;
};

class Lion : public Animal
{
public:
     Lion(string name) { m_name = name; }
};

void main()
{
     Lion lion("Fred");
     lion.Print();
}
```

Hi, my name is Fred



Constructors and Destructors

- Constructors
  - Not inherited
  - Base class constructor is called ***before*** Derived class constructor
  - Use initializer-list to call non-default base-class constructor
  - Similar for copy constructor
- Destructors
  - Not inherited
    - Derived class destructor is called ***before*** Base class
  - We'll look more carefully at these next week

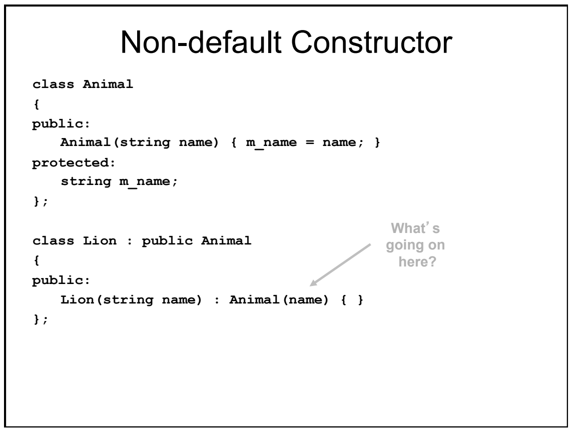


Constructor and Destructor

```
class Animal
{
public:
    Animal() { cout << "Base constructor" << endl; }
    ~Animal() { cout << "Base destructor" << endl; }
};

class Lion : public Animal
{
public:
    Lion() { cout << "Derived constructor" << endl; }
    ~Lion() { cout << "Derived destructor" << endl; }
};

int main()
{
    Lion lion;
    return 0;
}
```

Will print:

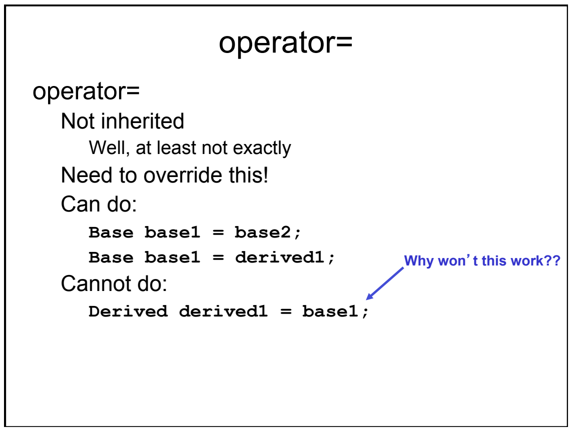
```
Base constructor
Derived constructor
Derived destructor
Base destructor
```



Non-default Constructor

```
class Animal
{
public:
    Animal(string name) { m_name = name; }
protected:
    string m_name;
};

class Lion : public Animal
{
public:
    Lion(string name) : Animal(name) { }
};
```

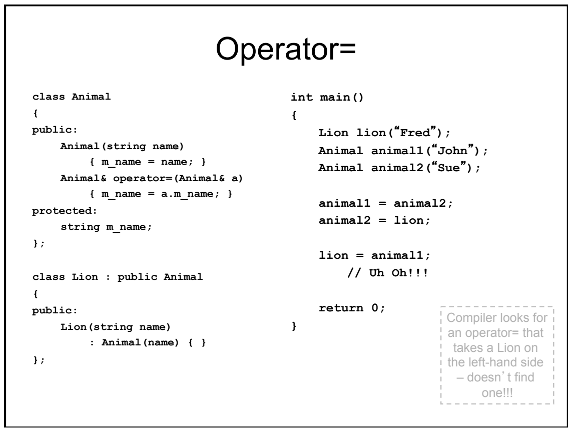
What's going on here?



operator=

- operator=
  - Not inherited
    - Well, at least not exactly
  - Need to override this!
  - Can do:
    - `Base base1 = base2;`
    - `Base base1 = derived1;`
  - Cannot do:
    - `Derived derived1 = base1;`

Why won't this work??

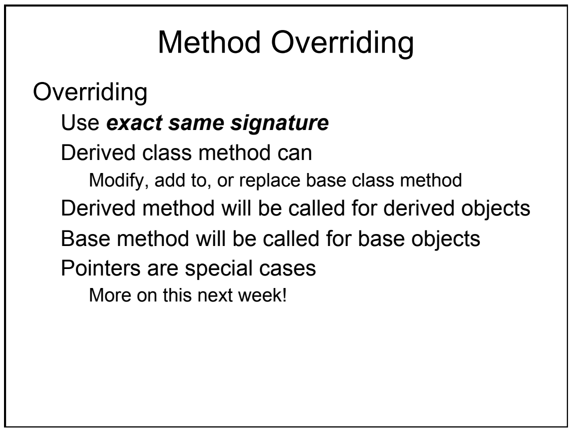


Operator=

```
class Animal
{
public:
    Animal(string name)
        { m_name = name; }
    Animal& operator=(Animal& a)
        { m_name = a.m_name; }
protected:
    string m_name;
};

class Lion : public Animal
{
public:
    Lion(string name)
        : Animal(name) { }
};
```

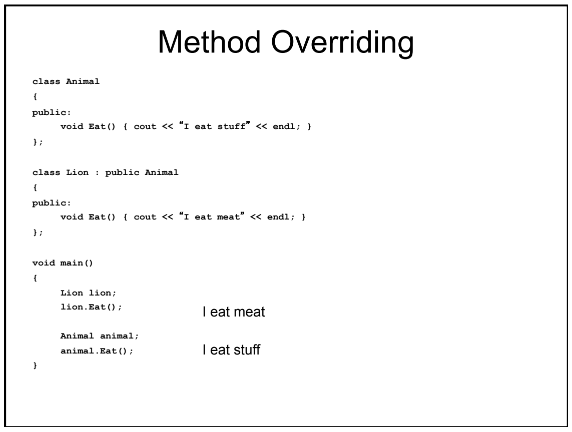
```
int main()
{
    Lion lion("Fred");
    Animal animal1("John");
    Animal animal2("Sue");

    animal1 = animal2;
    animal2 = lion;

    lion = animal1;
        // Uh Oh!!!

    return 0;
}
```

Compiler looks for an operator= that takes a Lion on the left-hand side – doesn' t find one!!!



Method Overriding

- Overriding
  - Use ***exact same signature***
  - Derived class method can
    - Modify, add to, or replace base class method
  - Derived method will be called for derived objects
  - Base method will be called for base objects
  - Pointers are special cases
    - More on this next week!



Method Overriding

```
class Animal
{
public:
    void Eat() { cout << "I eat stuff" << endl; }
};

class Lion : public Animal
{
public:
    void Eat() { cout << "I eat meat" << endl; }
};

void main()
{
    Lion lion;
    lion.Eat();                 I eat meat

    Animal animal;
    animal.Eat();               I eat stuff
}
```

Method Overloading

Overloading
- Use ***different signatures***
- Derived class has access to both…
- Not usually thought of as an inheritance topic
- Pointers are tricky
  - More on this next week!

Method Overloading

```
class Animal
{
public:
     void Eat() { cout << “I eat stuff” << endl; }
};

class Lion : public Animal
{
public:
     void Eat(string food) { cout << “I ate a(n) ” << food << endl; }
};

void main()
{
     Lion lion;
     lion.Eat(“steak”);              I ate a(n) steak

     lion.Eat();                     I eat stuff
}
```

Challenge

Complete the Giraffe and Mammal classes
- Implement at least one overloaded method
- Implement at least one protected data member
- Implement a constructor
- Implement a destructor
- Implement a non-default constructor