Welcome

Object Oriented Programming with C++
CIS 265

Week 8 – Inheritance

Christopher K. Burns

Schedule

Week		Content
1	1/11	*Review* Chapter 1 Intro to Computers and C++ Programming Chapter 2 Control Structures Chapter 3 Functions Chapter 4 Arrays
2	1/18	Additional Array and Function Topics Chapter 5 Pointers and Strings Lab 1 – Functions, Arrays, and Strings Homework 1 Assigned
3	1/25	Chapter 6 Classes and Data Abstraction
4	2/1	Chapter 7 Classes: Part I Lab 2 – Classes 1 Homework 1 Due
5	2/8	***MID-TERM EXAMINATION*** *(Chapters 1 through 7*)* **only portions of chapter 7 that were covered in class* *Final Project Assigned*
6	2/15	Chapter 7 Classes: Part II Chapter 8 Operator Overloading Homework 2 Assigned
7	2/22	Chapter 8 Operator Overloading Homework 2 Due Lab 3 – Classes 2
8	3/1	Chapter 9 Inheritance: Part I Homework 3 Assigned
9	3/8	Chapter 9 Inheritance: Part II Chapter 10 Polymorphism Homework 3 Due
10	3/15	Chapter 10 Polymorphism Lab 4 – Inheritance and Polymorphism
11	3/22	***FINAL PROJECTS DUE***

Agenda

Tonight's agenda

- Review
- Inheritance

Final Project

Updates on Web Sites

DiskUtil class – methods for getting files from directory also return Time of file creation. Example main() function updated to demonstrate this.

RegUtil class – gets/sets/deletes keys and values from registry.

Object Oriented Programming

Home Work

Read Chapter 10 in text

Implement examples in lecture

Classes - Review

What we've covered about classes up to this point:

- Basic structure of a class
- Constructors and overloading
- Separating Interface from Implementation
- Data encapsulation (get/set)
- Static members
- this pointer
- Operator Overloading
- Standard C++ Library – String and Vector

Inheritance

Data encapsulation, Inheritance and Polymorphism are the three of the main tenets of object oriented programming.

You are already familiar with encapsulating data and functionality into an object.

What is inheritance?

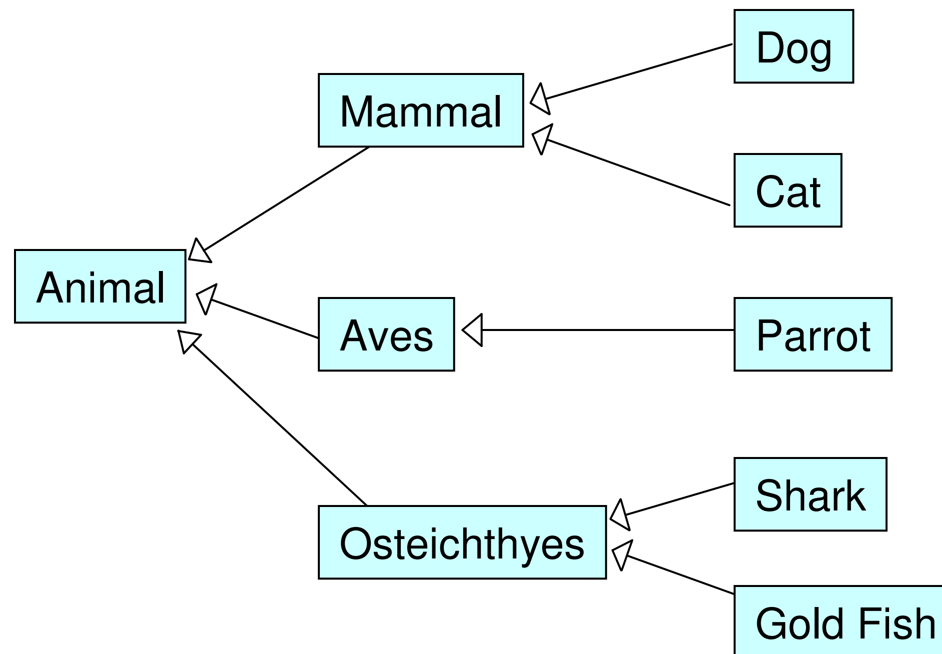
Inheritance

Main Entry: **in·her·i·tance**
Pronunciation: in-'her-&-t&n(t)s
Function: *noun*
1 a : the act of inheriting property **b :** the reception of genetic qualities by transmission from parent to offspring **c :** the acquisition of a possession, condition, or trait from past generations
2 : something that is or may be inherited
3 a : TRADITION b : a valuable possession that is a common heritage from nature
4 *obsolete* **: POSSESSION**

from Merriam-Webster.com

Inheritance

A C++ class can inherit interface and methods from another C++ class.



Inheritance

More efficient approach to building software.

Create a base set of functionality and build classes upon that.

The base functionality is called the "base class"

Inheritance

Sometimes the relationships between classes, bases, and members can become ambiguous.

A phonetic trick to determine bases from members:

`“is-a”` `“has-a”`

Inheritance

is-a: Inheritance

Derived class is a Base class

– Car is a Vehicle

has-a: Contains

Class has a member

– Car has a Engine

Inheritance

```
class Animal
{
public:
  string name;
  string sound;
};

class Dog : public Animal
{
public:
  Dog()
  {
    name = "Dog";
    sound = "Bark";
  }
};
```

Inheritance

```
#include <iostream>
#include <string>
using namespace std;

class Animal
{
public:
  string name;
  string sound;
};

class Dog : public Animal
{
public:
  Dog()
  {
    name = "Dog";
    sound = "Bark";
  }
};
```

```
class Cat : public Animal
{
public:
  Cat()
  {
    name = "Cat";
    sound = "Meow";
  }
};

void main()
{
  Dog dog;
  Cat cat;
  cout << dog.name << endl;
  cout << cat.name << endl;
}
```

Inheritance

members of a class can be assigned access of:

- public – anybody can access
- private – only class members and friends can access
- protected – same as private, but includes derived classes

```
class Animal
{
private:
  int count;
protected:
  string name;
  string sound;
public:
  string getName() { return this->name; }
};
```

Protected Members

Class members declared as **protected** can be used only by the following:

- Member functions of the class that originally declared these members.
- Friends of the class that originally declared these members.
- Classes derived with public or protected access from the class that originally declared these members.
- Direct privately derived classes that also have private access to protected members.

(from MSDN: http://msdn.microsoft.com/library/en-us/vccelng/htm/cntrl_3.asp?frame=true)

Protected Members

```
#include <iostream>
#include <string>
using namespace std;


class Animal
{
private:
  int count;
protected:
  string name;
  string sound;
public:
  string getName() {return this->name;}
};
```

```
class Dog : public Animal
{
public:
  Dog()
  {
    name = "Dog";
    sound = "Bark";
  }
};

void main()
{
  Dog dog;
  //cout << dog.name << endl;
  cout << dog.getName() << endl;
}
```

Types of Inheritance

How Dog sees Animal

```
class Dog : public Animal { Dog(){ Animal.name = “Dog” }};
  // Dog can access public members of Animal
  // Dog can access protected members of Animal
  // Dog cannot access private members of Animal

class Dog : private Animal
  // Dog can access public members of Animal
  // Dog can access protected members of Animal
  // Dog cannot access private members of Animal

class Dog : protected Animal
  // Dog can access public members of Animal
  // Dog can access protected members of Animal
  // Dog cannot access private members of Animal
```

Types of Inheritance

How users of Dog see Animal

```
class Dog : public Animal {}; Dog dog; dog.name = "Ralph";
  // public members of Animal stay publicly accessible in Dog
  // private and protected members of Animal remain the same

class Dog : private Animal {}; Dog dog; ~~dog.name = "Ralph";~~
  // public and protected members of Animal become privately
  //   accessible in Dog.
  // private members of Animal remain the same.
  // technically, Dog is no longer an animal in this case
  //   now relationship is "has-a"

class Dog : protected Animal {};
  // public and protected members of Animal become protected
  //   accessible in Dog.
  // private and protected members of Animal remain the same
```

Inheritance

```
class Animal
{
public:
  string name;
  string sound;
};
class Dog : private Animal
{
public:
  Dog()
  {
    name = "Dog";
    sound = "Bark";
  }
  string printName() { return name; }
};
void main()
{
  Dog dog;
  //cout << dog.name << endl;
  cout << dog.printName() << endl;
}
```

publicly accessible members of Animal become private

Inheritance

Which type of inheritance should I use?

Generally, if the relationship is a “is-a” then you should public.

Private inheritance is essentially aggregation, or containment; “has-a”. This is because the base class is not accessible through the derived class.

Inheritance

Okay, I've decided to use private inheritance but still need to let users of Dog get to a public member of Animal, how can this be done?

```
class Animal
{
public:
  string name;
  string sound;
};
```

```
void main()
{
  Dog dog;
  cout << dog.name << endl;
}
```

```
class Dog : private Animal
{
public:
  using Animal::name;
  Dog()
  {
    name = "Dog";
    sound = "Bark";
  }
};
```

Data Abstraction

data abstraction is a term used in object oriented languages to describe the process of refining data down to its essential parts.

Data Abstraction

When developing a program that will contain a set of domesticated pets, you can determine a set of properties common to all the pets. In our example, we have created an abstract data type called Animal which contains these commonalities.

An abstract data type for a program that contains Managers, Engineers and Sales staff might be Employee.

Data Abstraction

Our abstract data type can contain an interface and methods common to all derived classes.

We then only have to write the common functionality once and it can be shared by all derived classes.

Overriding

Sometimes our derived class needs to change the functionality defined in the abstract base class. This is called overriding.

Overriding

```
#include <iostream>
#include <string>
using namespace std;

class Animal
{
public:
  string name;
  void produceSound()
  {
    cout << "no sound defined\n";
  }
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound()
  {
    cout << "woof\n";
  }
};
```

overrode produceSound

```
class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
};

void main()
{
  Dog dog;
  Cat cat;
  dog.produceSound();
  cat.produceSound();
}
```

oops, Cat didn’t override produceSound

Virtual Methods

Virtual Methods differ from static (standard) methods in that they are dynamically bound to the object.

If a function may be overridden by a derived class it should be defined as virtual. We will see why when we begin discussing polymorphism.

```
class Animal
{
public:
  string name;
  virtual void produceSound() { };
};
```

Pure Virtual Methods

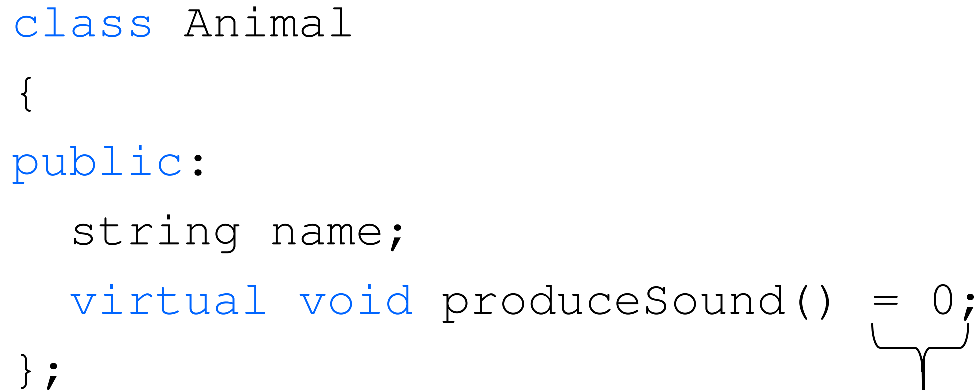
Often, it is known what the interface of the derived classes should look like, but the implementation cannot always be defined.

We can use pure virtual methods to define common functions that the derived classes must implement.

This would avoid problems as with the Cat class in the previous example.

Pure Virtual Methods

A Pure Virtual Method is defined by:



```
class Animal
{
public:
   string name;
   virtual void produceSound() = 0;
};
```

"= 0" means that there is no function defined. No class with a pure virtual function can be instantiated. Thus, before a derived class can be used, this function must be implemented.

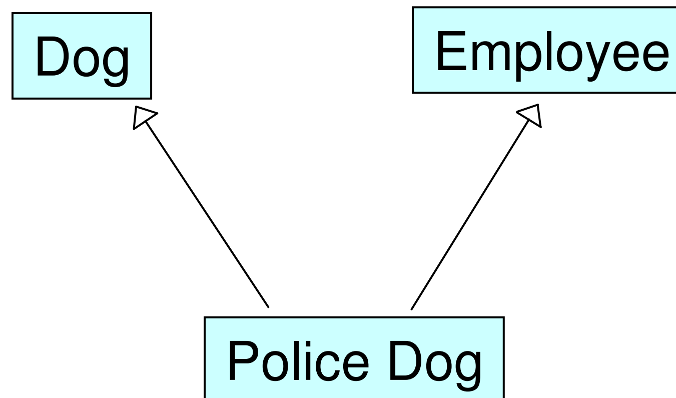
Overriding

```
#include <iostream>
#include <string>
using namespace std;

class Animal
{
public:
  string name;
  virtual void produceSound() = 0;
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound()
  {
    cout << "woof\n";
  }
};
```

```
class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
  void produceSound()
  {
    cout << “meow\n";
  }
};

void main()
{
  Dog dog;
  Cat cat;
  dog.produceSound();
  cat.produceSound();
}
```

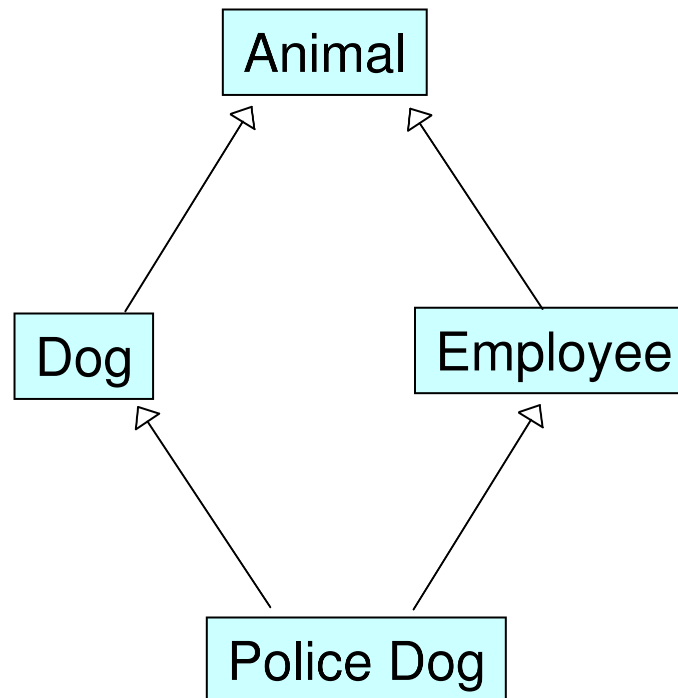
Multiple Inheritance

C++ allows a class to inherit from multiple classes. This is a very powerful feature, but is very dangerous.



Multiple Inheritance

Does this look like it could be a problem?

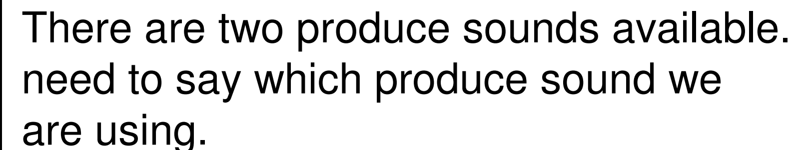


This is often called the “Dreaded Diamond”

Overriding

```
#include <iostream>
#include <string>
using namespace std;

class Animal
{
protected:
  string name;
public:
  virtual void produceSound() = 0;
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound() { cout << "woof\n"; }
};

class Employee : public Animal
{
public:
  float salary;
  void produceSound() {
    cout << "let's have a meeting\n"; }
};
```

```
class PoliceDog :
  public Dog,
  public Employee
{
public:
  void heal() { };
  void attack() { cout << "attack\n"; };
  using Dog::produceSound;
};

void main()
{
  PoliceDog policeDog;
  policeDog.produceSound();
  policeDog.attack();
}
```



There are two produce sounds available. need to say which produce sound we are using.

Multiple Inheritance

Multiple Inheritance can lead to ambiguities.

In the previous example, there were two produceSound() methods that could be called.

Multiple Inheritance

The authors of Java and C# saw the problems that could be induced by using Multiple Inheritance incorrectly. They decided not to allow it explicitly.

Java and C# do support a form of Multiple Inheritance, where by a subclass can only inherit the implementation of one class but the interface of many others.

The Java and C# solution can be applied to C++ by ensuring that your base classes are pure abstract (thus no implementation)

Multiple Inheritance

Should I use multiple inheritance?

Multiple inheritance is a useful tool of many OO languages, but like any tool it should only be used where appropriate.

If your solution can be simplified by using multiple inheritance, then by all means use it; but be cautious.

A (solution) should be a simple as possible, but no simpler – A. Einstein

Polymorphism

…but wait, in some of the earlier examples the dog class is derived from animal, and the cat class is also derived from animal…

…since they are both "animals", can't they be treated as such…?

Yes, this is called polymorphism

Polymorphism

Main Entry: **poly·mor·phism**

Pronunciation: "pä-lE-'mor-"fi-z&m

Function: *noun*

: the quality or state of being able to assume different forms: as **a :** existence of a species in several forms independent of the variations of sex **b :** the property of crystallizing in two or more forms with distinct structure

- **poly·mor·phic** /-fik/ *adjective*
- **poly·mor·phi·cal·ly** /-fi-k(&-)lE/ *adverb*

from Merriam-Webster.com

Polymorphism

```
#include <iostream>
#include <string>
using namespace std;

class Animal
{
public:
  string name;
  virtual void produceSound() = 0;
};

class Dog : public Animal
{
public:
  Dog() { name = "Dog"; }
  void produceSound()
  {
    cout << "woof\n";
  }
};
```

```
class Cat : public Animal
{
public:
  Cat() { name = "Cat"; }
  void produceSound()
  {
    cout << "meow\n";
  }
};

void main()
{
  Animal* animals[2];
  animals[0] = new Dog();
  animals[1] = new Cat();

  for( int i = 0; i < 2; i++ )
  {
    animals[i]->produceSound();
  }
}
```