

# CS202: Programming Systems

---

Week 5

Virtual functions and polymorphism

11/2018

# CS202 – What will be discussed?

---

- **IS-A** and **HAS-A** relationship
- Virtual functions and polymorphism
- Pure virtual functions
- Abstract class

# IS-A and HAS-A relationship

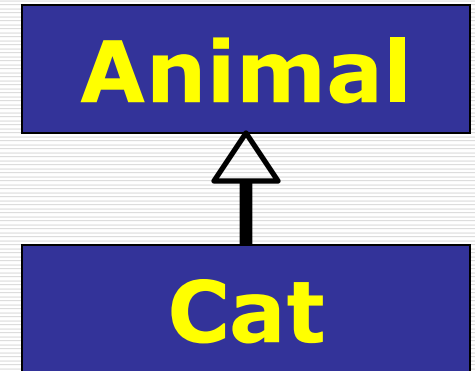
---

- The inheritance is only applied when there is a “**IS-A**” relationship between classes
  - E.g. Dog **is an** animal.  
Or, employer **is an** employee.
  
- When two classes have a “**HAS-A**” relationship, we declare one class as a member of the other.

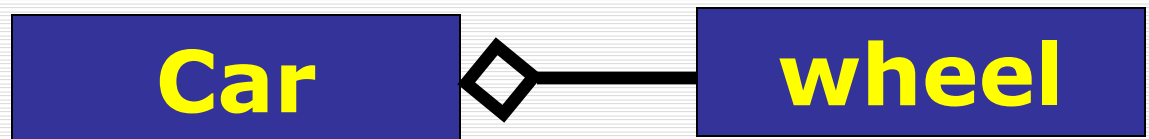
# An example

---

- Cat “is an” animal.



- Car “has a” wheel.



# An example (cont.)

---

```
class Animal
{
    ...
};
```

```
class Cat: public Animal
{
    ...
};
```

```
class Wheel
{
    ...
};
```

```
class Car
{
private:
    Wheel wh;
};
```

# A pointer to Base class

---

- ❑ A pointer to base class can be assigned with the address of an object of the derived class.
- ❑ For example:

```
Animal* pAni;  
Cat c;  
pAni = &c; //OK
```

# Implicit type conversion in inheritance

---

- It is normal to pass a derived class variable to a function with an argument of base class data type. The compiler will do an implicit conversion.

```
Animal::process(const Animal &a);  
  
int main() {  
    Cat c;  
    Animal::process(c); //OK  
}
```

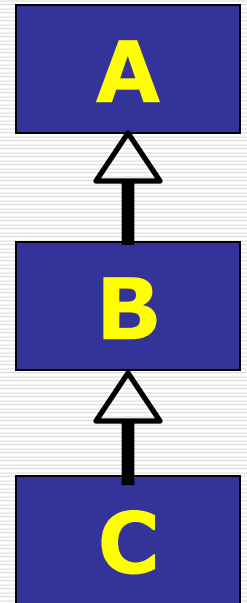
# Static binding

---

Consider the following situation:

- ❑ `class A` has `void print()`
- ❑ `class B` also has `void print()`
- ❑ `class C` has `void print()` too

```
int main() {  
    C varC;  
    B varB;  
    varB.print();    // print() of B  
    varC.print();    // print() of C  
    varC.B::print(); // print() of B  
}
```



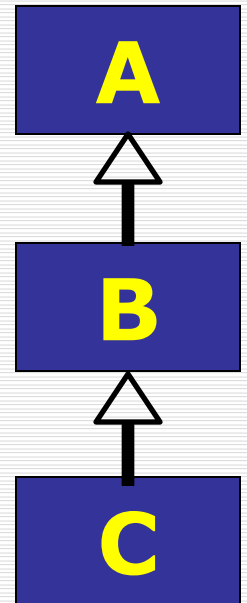


# Static binding

---

## □ Another example:

```
int main() {  
    A varA;  
    B varB;  
    C varC;  
    A* var1, *var2;  
  
    var1 = &varC;  
    var2 = &varB;  
    var1->print(); // print() of A  
    var2->print(); // print() of A  
}
```



# Virtual function

---

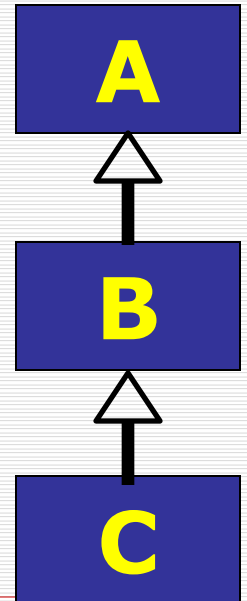
- ❑ Virtual function: in the base class, member functions will become virtual functions if we add the **virtual** keyword in front of their declarations.
- ❑ For example: **virtual void print() ;**
- ❑ In the derived class, the definition of virtual member function can be re-define according to the requirement of that class.
- ❑ Note: it is allowed to have or not have the **virtual** keyword in front of the virtual functions of the derived class.

# Dynamic binding

---

- When using virtual functions, compiler will make sure which member function will be invoked according to which object is calling. For example `print()` is a virtual function

```
int main() {  
    A varA;  
    B varB;  
    C varC;  
    A* var1, *var2;  
    var1 = &varC;  
    var2 = &varB;  
    var1->print(); // print() of C  
    var2->print(); // print() of B  
}
```



# Polymorphism:

## Virtual function & Dynamic binding

---

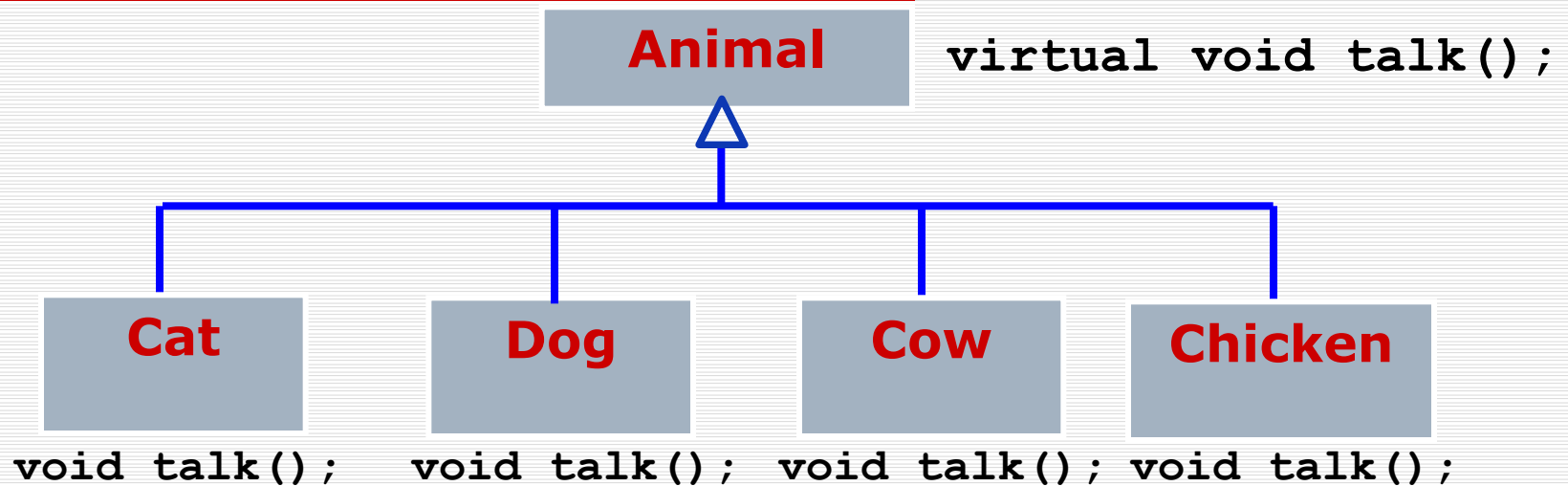
- ❑ Beside the **virtual** keyword, the compiler will use dynamic binding only when the calling object is manipulated via a pointer or a reference.
- ❑ Calling a function using the scope resolution operator `::` ensures that the virtual mechanism is not used.
- ❑ In the derived class, if the virtual function is not re-defined, compiler will look for the latest definition of this function in the inheritance hierarchical structure.

# Polymorphism

---

- ❑ Clearly, to implement polymorphism, the compiler must store some kind of type information in each object of class A and use it to call the right version of the virtual function `print()`. In a typical implementation, the space taken is just enough to hold a pointer.
- ❑ Typical implementation of virtual functions is to add to every object of a class with at least one virtual function a pointer to the virtual function table
- ❑ This table contains pointers to all the virtual functions of the classes the object belongs to

# An example



```
int main() {
    Animal *pAni;
    Cat c;
    Dog d;
    pAni = &c; pAni->talk(); // talk() of Cat
    pAni = &d; pAni->talk(); // talk() of Dog
}
```

# Virtual Constructor? **NO!!!**

# Virtual Destructor? YES!

---

☐ **NO** virtual constructor!

- Each constructor is used to initialize the class itself.
- Constructors are designed to run from Base to Derived classes.

☐ Virtual destructor: YES!

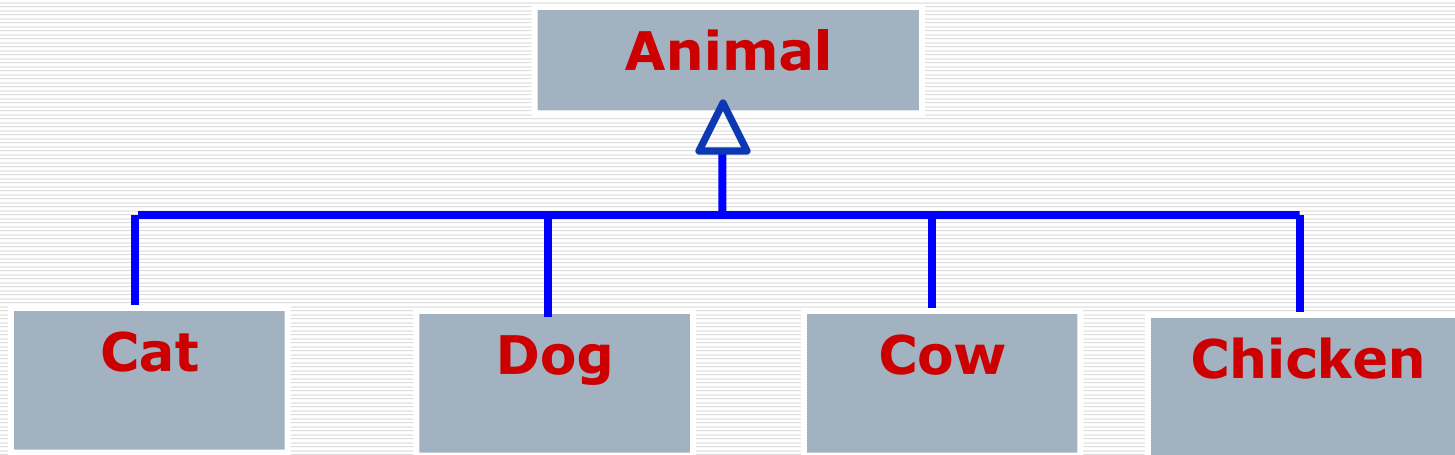
- The destructor should be virtual in order to free the memory/resource of the correct object.

# Pure virtual function

---

- Consider the `class Animal` again

```
virtual void talk();
```





# Pure virtual function

---

## □ Then

```
int main()
{
    Animal a; //What animal?
    a.talk(); //non-sense: how can it talk()
}
```

## □ `talk()` can be changed to pure virtual:

```
virtual void talk() = 0;
```

# Abstract class

---

- ❑ A class which has one or more virtual functions becomes abstract class.
- ❑ No objects of the abstract class can be created.

```
Animal a; //Error!
```

# Abstract class

---

- ❑ A pure virtual function that is not overridden in a derived class remains a pure virtual function, so the derived class is also an abstract class.
- ❑ An important use of abstract classes is to provide an interface without exposing any implementation details.
- ❑ Every class having at least one virtual function should have the virtual destructor.