Fu-Yin Cherng

Nation Taiwan University

Inheritance and polymorphism

Review

- (Quick) Review of C++
  - syntax, concpets
- From Quiz 0
  - array, reference and pointer, class
- From online materials
  - File I/O, C++ strings, and Header file (Prof. Ling-Chieh Kung)

Outline

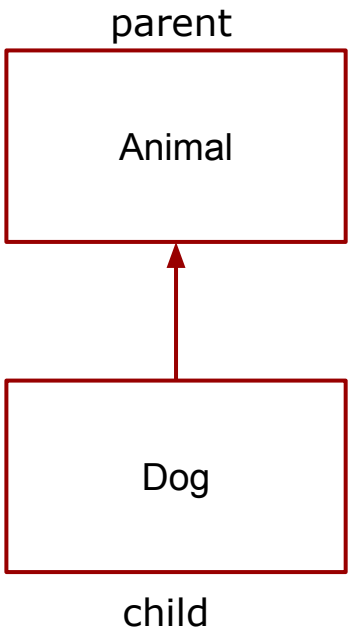
- **Inheritance**
- polymorphism

Inheritance

- We can use classes to define the objects with common abstract features
  - 哈士奇, 吉娃娃 -> dog
  - 麻雀、鸚鵡 -> Bird
- **Dog** and **Bird** are **Animal**
- How to set up and describe the relations between these classes? Inheritance

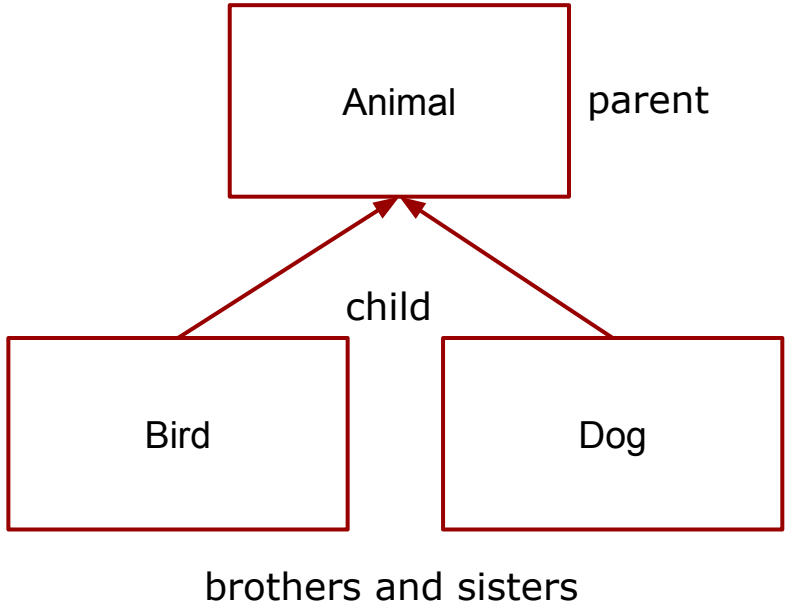
Inheritance

- A derived (child) class inherits a base (parent) class
- A child class has (some) members defined in the parent class.
- XXX is a OOO
  - dog is a creature
  - car is a vheicle
- Reuse code in parent class & Reduce inconsistency



A example

- Let’s create a zoo
- create birds and dog by inheritance
- multiple inheritance
  - a parent class can inherited by multiple child classess



A Example

```
int main() {
  Bird myb;

  cout << myb.bird_type;
  cout << myb.age;
  myb.animalSound();
  cout << myb.weight;

  return 0;
}
```

```
// Base class
class Animal {
  public:
    int age = 1;
    int weight = 4;
    void animalSound() {
      cout << "The animal makes a sound \n" ;
  }
};

// Derived class
class Bird : public Animal {
  public:
    string bird_type = "owl";
};


// Derived class
class Dog : public Animal {
  public:
    string dog_type = "Chihuahua";
};
```

Access modifiers

```
int main() {
  Bird myb;

  cout << myb.bird_type;

  cout << myb.age;
  myb.animalSound();

  cout << myb.getWeight();

  cout << myb.super_type; //error
  return 0;
}
```

public: accesse by anyone
protected: accessed by itself and its childs
private: accessed by itself

```
// Base class
class Animal {
  public:
    int age = 1;
    void animalSound() {
      cout << "The animal makes a sound \n" ;
  }
  protected:
    int weight = 4;
  private:
    string super_type = "Eukaryota";
};

// Derived class
class Bird : public Animal {
  public:
    string bird_type = "owl";
  void setWeight(int w){
      weight = w;
    }
  int getWeight() {
      return weight;
    }
};
```

Constructors of Parent class

- Parent class **constructors** will **not** be inherited
- Constructor
  - special method that is automatically called when an object of a class is created
  - useful for setting initial values for attributes
  - same name as the class, it is always public

```
class Animal {
  public:
    int age = 1;
    int weight = 4;
    void animalSound() {
     cout << "The animal makes a
sound \n" ;
  }
    Animal(){cout<<"Animal";}
    Animal(int a, int w){
     age = a;
     weight = w;
    }
};
```

Constructors of Parent class

- Although not inherited, parent constructor is called before the child constructors is called
- If not specified, the parent's **default** constructor will be invoked.
- Create parent before child

```
int main(){
     Bird b;//print "Animal"
     return 0;
}
```

```
Animal::Animal(){ //default
    cout<<"Animal";
}
Animal::Animal(int a, int w){
     age = a;
     weight = w;
}
Bird::Bird(){
    //call default constructor
}
```

Specify parent constructor

- Use syntax for member initializer
- Pass arguments to control the behavior

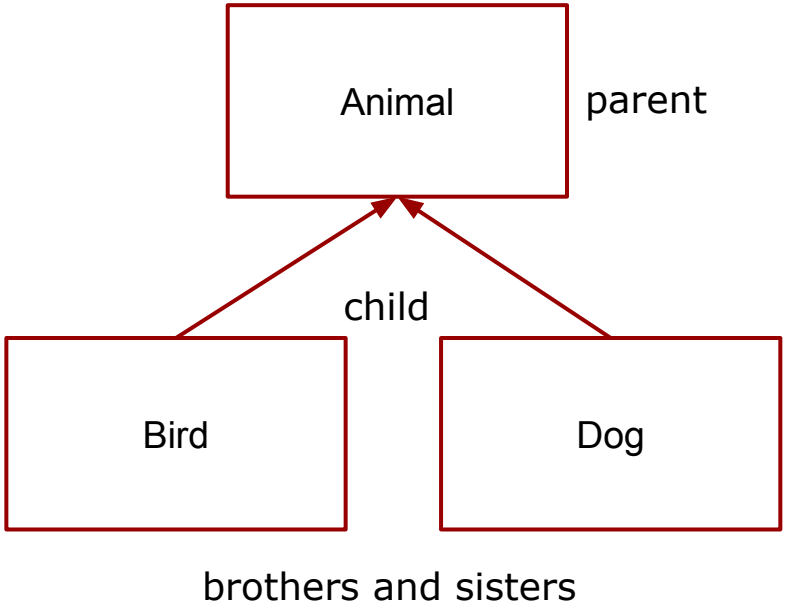
```
Animal::Animal(){ //default
cout<<"Animal";
}
Animal::Animal(int a, int w){
     age = a;
     weight = w;
}
Bird::Bird():Animal(2,2) {
    //call Animal(int a, int w)
}

int main(){
     Bird b;
     return 0;
}
```

Destructor of parent class

- When an object of the child class is to be destroyed
- child destructor -> parent destructor
- parent destructor will be invoked automatically
- directly use parent destrutor for child class

Summary

- What is Inheritance and why use it?
  - saving time & enhance consistency
- use Access modifiers in parent class
- constructor and destructor in parent class and their relation with child class



Function Overriding

- child can use parent's member function
- child can define it's own functions
  - parent can't access child's member

```
// Parent class
class Animal {
  public:
    int age = 1;
    int weight = 4;
    void animalSound() {
      cout << "The animal makes a sound \n" ;
  }
};

// Child class
class Bird : public Animal {
  public:
    string bird_type = "owl";
};


int main(){
      Bird b;
      b.animalSound();
      return 0;
};
```

Function Overriding

- redefine the members inherited from a parent
- child class **override** the member function of the parent
- invoke parent's member function by using ::
  - enhance consistency

```
// Parent class
class Animal {
  public:
    int age = 1;
    int weight = 4;
    void animalSound() {
      cout << "The animal makes a sound \n" ;
  }
};

// Child class
class Bird : public Animal {
  public:
    string bird_type = "owl";
    void animalSound(){
        Animal::animalSound();
        cout << "chuchu \n" ;
    }
};


int main(){
      Bird b;
      b.animalSound();
      return 0;
};
```

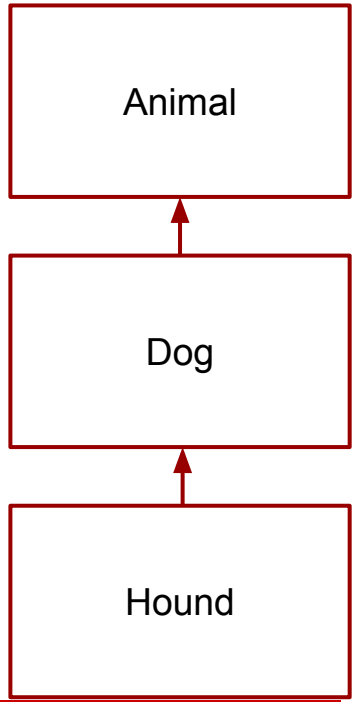
The animal makes a sound
chuchu

When to override function?

- overriding parent's member function is typical
- since child class is a **special case** for parent class, so need some **customizations** according to this special case
- depend on how you design the classes, but try to reuse code as much as possible

Cascade/multilevel inheritance

- inheritance can be multilevel
- Animal class is parent for Dog, Dog is parent for Houd, and Animal is grandparent for Hound.
- Houd can access the public & proctected members in ancestors class

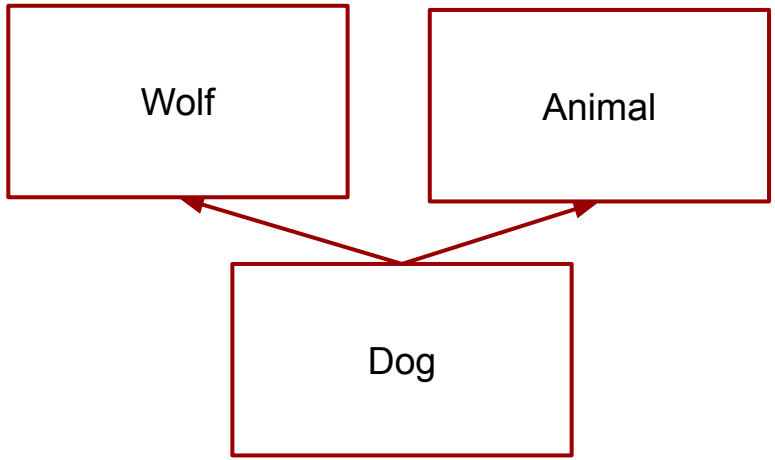


Inheritance visibility

- child can make the public members of parent class into private
- but **can’t** make private members into public
- we can only **narrow down** the visibility
- manage class visibility (access specifier) is crucial in a big and collaborative project

Multiple Inheritance

- a class can also be derived from more than one base class
  - a child can have multipel parent
- relationship between classes will be pretty messy and hard to maintain and debug...




```
// Derived class
class Dog : public Animal, public Wolf {
  public:
    string dog_type = "Chihuahua";
};
```

Outline

- Inheritance
- **polymorphism**

Polymorphism

- "many forms"
- occurs when we have many classes that are related to each other by inheritance.
- use a ***variable*** of a **parent** type to **store** a ***value*** of a **child** type.
- related to
  - early and late binding
  - virtual functions

```
int main(){
     int i = 5; //int variable store int value
     Animal a;
     Bird b;
     a = b; //parent variable store child obj/value
     return 0;
};
```

Example

```
void getWidth(Shape s){
      cout << s.width;
}

int main(){
      Shape shape;
      Rectangle rec(10,7,90);
      Triangle  tri(8,5);

      shape = rec; //ok, 90 discarded

      Shape *shape_p;
      shape_p = &rec //store address of rec

      getWidth(rec); //10
      getWidth(tri); //8
      return 0;
}
```

```
class Shape { // base (parent) class
   public:
      int width, height;
      Shape(int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape { // derived (child) class
   public:
       int angle;
      Rectangle(int a, int b, int c):Shape(a, b) {
             angle = c;
       }
};
class Triangle: public Shape { // derived (child) class
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }
      };
```

Early & Late Binding

```
int main(){
      Shape shape;
      Rectangle rec(10,7,90);
      shape = rec; //early binding

      Shape *shape_p;
      shape_p = &rec //late binding

      return 0;
}
```

```
class Shape {// base (parent) class
   public:
      int width, height;
      Shape(int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {// derived (child) class
   public:
      int angle;
      Rectangle(int a, int b, int c):Shape(a, b) {
            angle = c;
      }
      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
```

Early & Late Binding

```
int main(){
      Shape shape;
      Rectangle rec(10,7,90);
      shape = rec; //early binding

      Shape *shape_p;
      shape_p = &rec //late binding

      shape.area();
      shape_p->area();

      return 0;
}
```

```
Parent class area :
Parent class area :
```

```
class Shape {// base (parent) class
   public:
      int width, height;
      Shape(int a = 0, int b = 0){
         width = a;
         height = b;
      }
      int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {// derived (child) class
   public:
      int angle;
      Rectangle(int a, int b, int c):Shape(a, b) {
            angle = c;
      }
      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
```

Virtual Functions

```
int main(){

      Shape *shape_p;
      Rectangle rec(10,7,90);
      shape_p = &rec //late binding

      shape_p->area();
      return 0;
}
```

Rectangle class area :70

- define what functions is virtual when design parent class
- use pointer to do ploymorphism to enable late binding

```
class Shape {// base (parent) class
   public:
      int width, height;
      Shape(int a = 0, int b = 0){
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
class Rectangle: public Shape {// derived (child) class
   public:
      int angle;
      Rectangle(int a, int b, int c):Shape(a, b) {
            angle = c;
      }
      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};
```

Summary

- polymorphism is based on inheritance
- *use a variable of a parent type to store a value of a child type.*
- function overriding and virtual function
- make program clearer, more flexible, and less inconsistency
  - easy to extend and maintain