

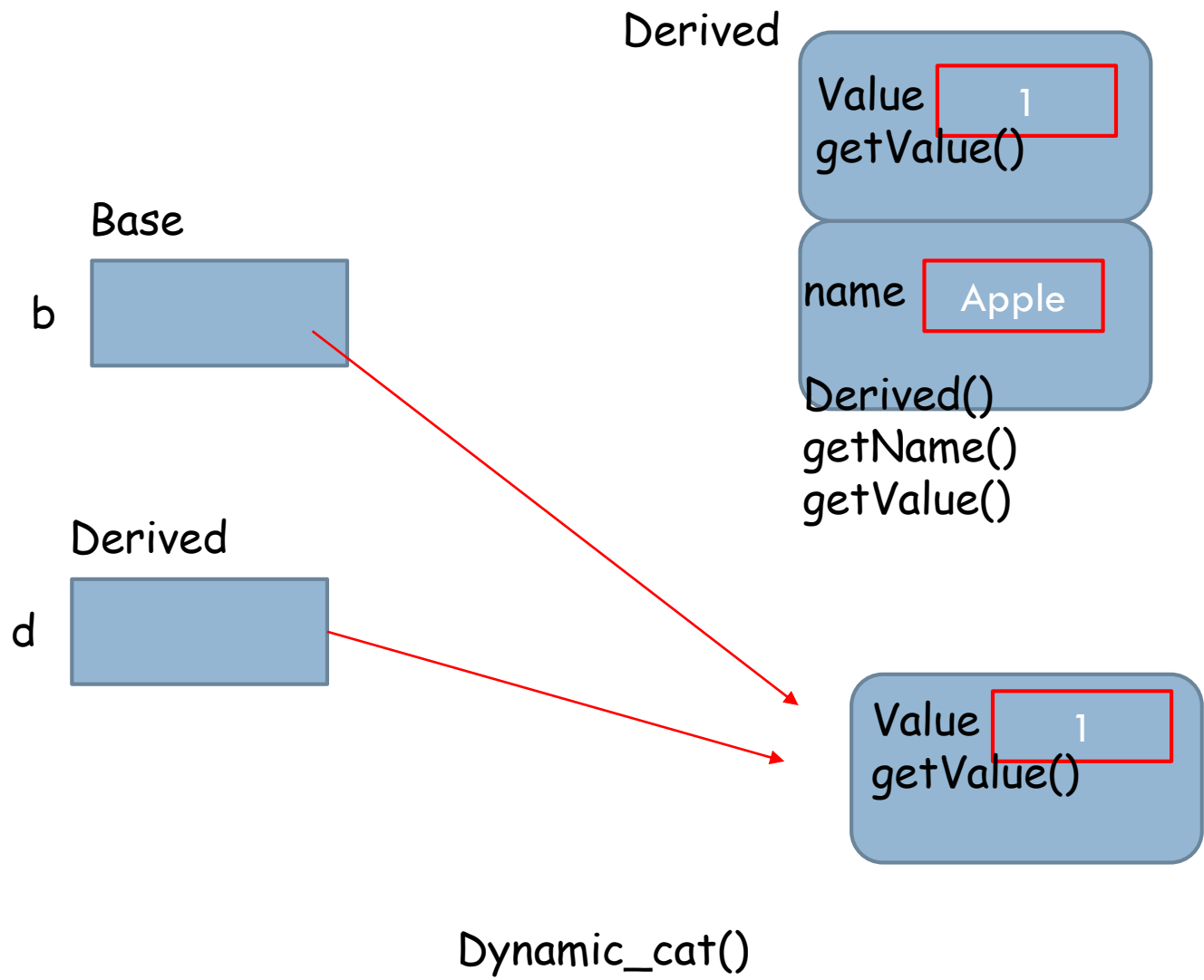
# 12장 다형성과 가상함수

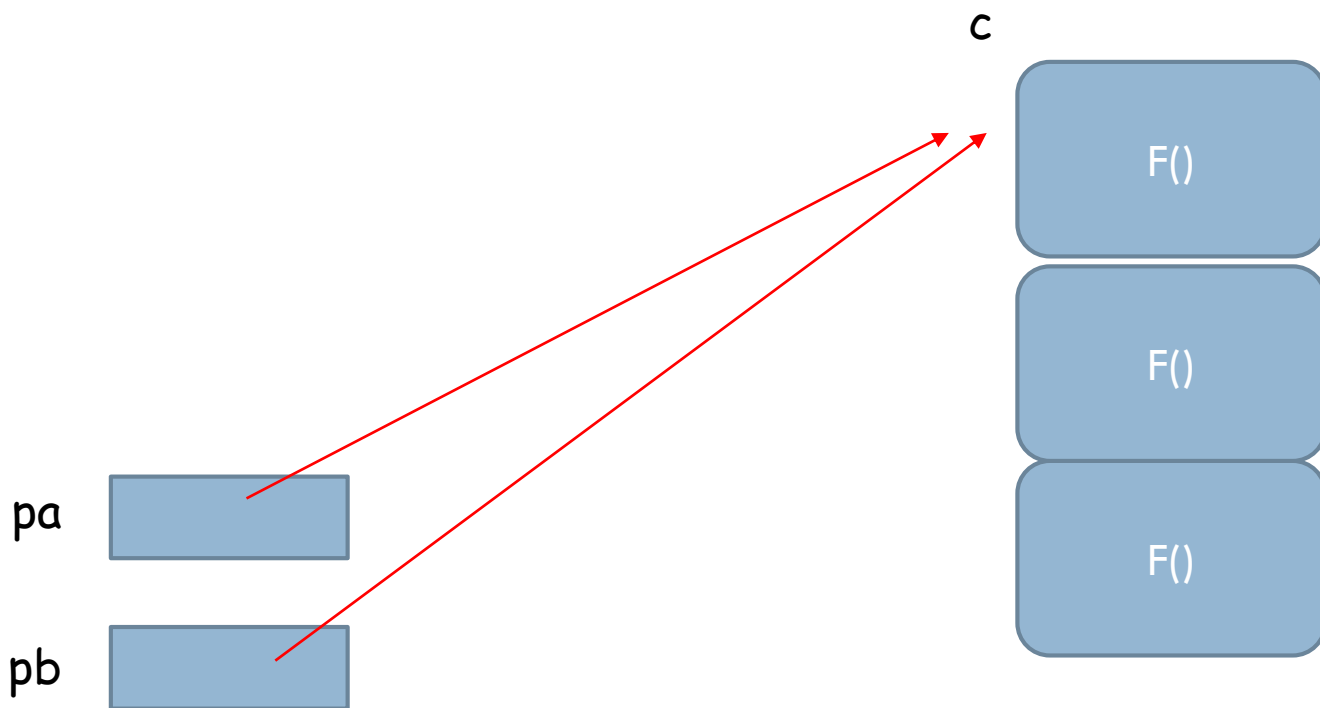
2020. 11. 16

선천향대학교 컴퓨터 공학과  
전 문 공 과

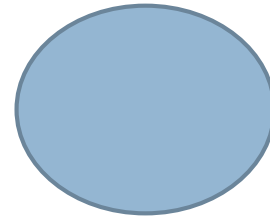
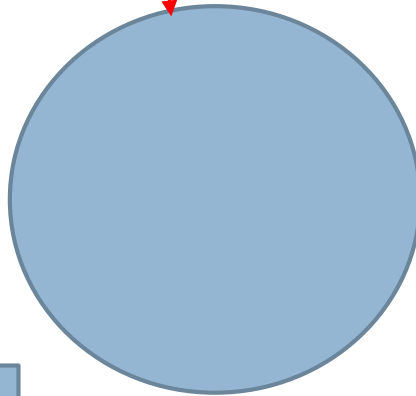
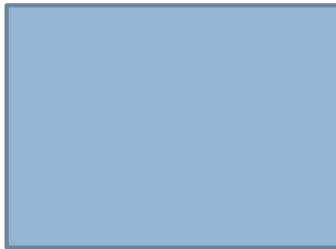
# 내용

- 객체포인터의 타입 변환
- C++의 다형성
- 동적 바인딩
- 가상 함수
- 추상 클래스





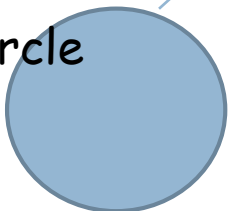
shapes



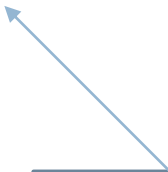
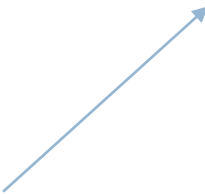
Shape



Circle



Rectangle





# 상속과 객체 포인터

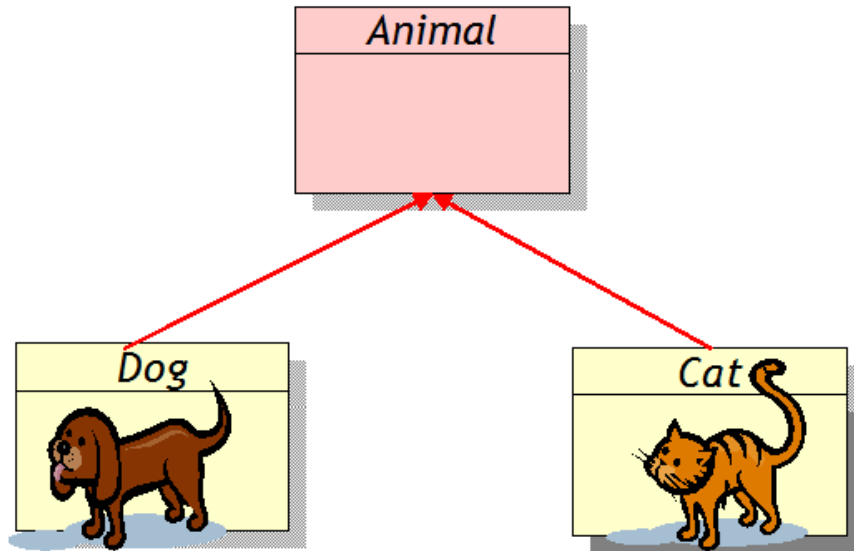


그림 9.2 상속 계층도

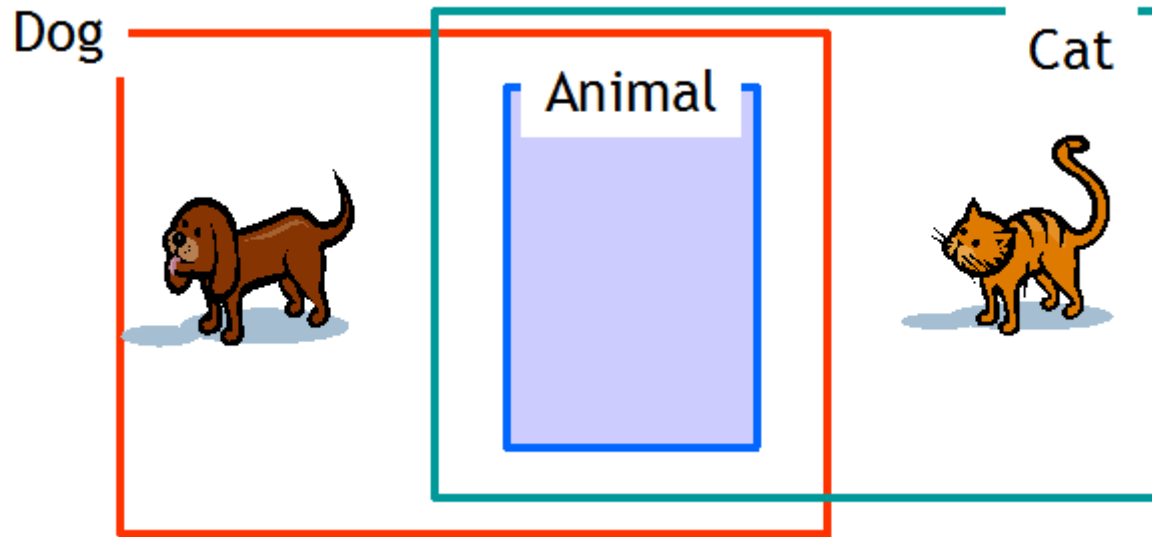
Animal 타입  
포인터로 Dog  
객체를 참조하니  
틀린 거 같지만  
올바른 문장!!

```
Animal *pa = new Dog(); // OK!
```



# 왜 그럴까?

- 자식 클래스 객체는 부모 클래스 객체를 포함하고 있기 때문이다.



```
Animal *pa = new Dog(); // OK!
```

# 객체 포인터의 타입 변환

## 객체 포인터의 타입변환

상향 타입변환(upcasting):

자식 클래스 타입을 부모 클래스타입으로  
변환

하향 타입변환(downcasting):

부모 클래스 타입을 자식 클래스타입으로  
변환



# 타입 상향변환(업 캐스팅)

- 자식 클래스 포인터가 기본 클래스 포인터에 치환되는 것

객체 cp의 모든 public  
멤버 접근 가능

pDer



cp

기본 클래스의 public  
멤버만 접근 가능

pBase



```
int main() {  
    ColorPoint cp;  
    ColorPoint *pDer = &cp;  
    Point* pBase = pDer; // 업캐스팅  
  
    pDer->set(3,4);  
    pBase->showPoint();  
    pDer->setColor("Red");  
    pDer->showColorPoint();  
    pBase->showColorPoint(); // ?  
}
```

int x 3  
int y 4  
void set() {...}  
void showPoint() {...}

기본클래스  
멤버

string  
color  
void setColor () {...}  
void showColorPoint() { ... }

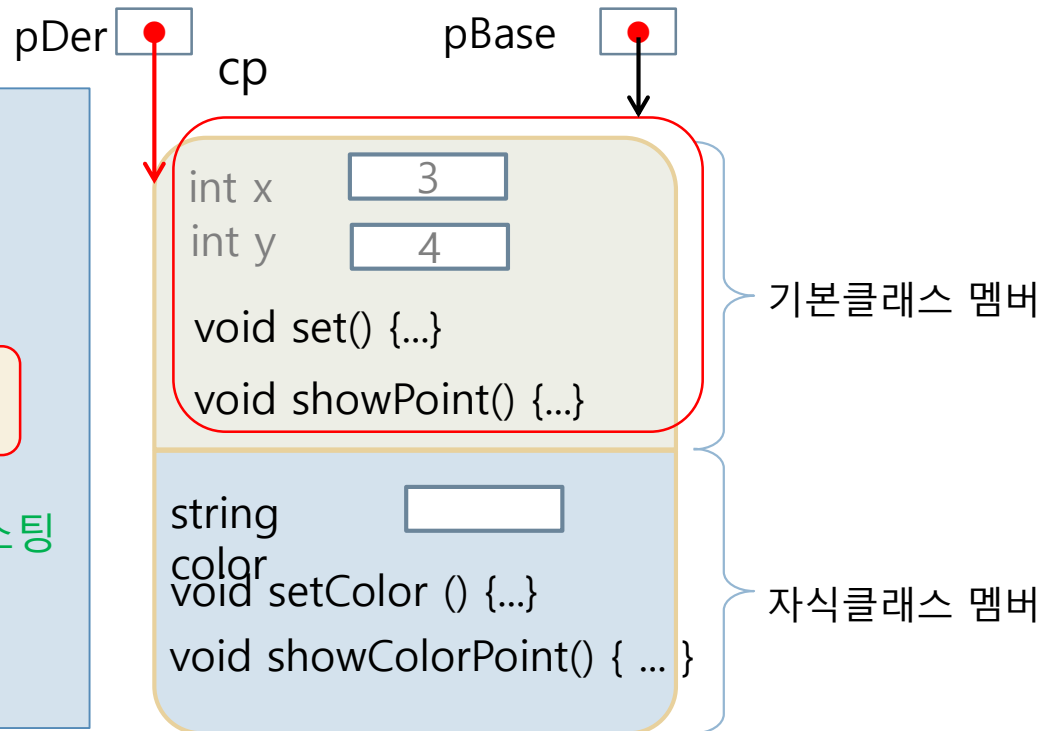
자식클래스  
멤버

# 타입 하향변환(다운 캐스팅)

- 기본 클래스의 포인터가 자식 클래스의 포인터에 치환되는 것

```
int main() {  
    ColorPoint cp;  
    ColorPoint *pDer;  
    Point* pBase = &cp; // 업캐스팅  
  
    pBase->set(3,4);  
    pBase->showPoint();  
  
    pDer = (ColorPoint *)pBase; //다운캐스팅  
    pDer->setColor("Red");  
    pDer->showColorPoint();  
}
```

강제 타입 변환  
반드시 필요



```
class Animal {
public:
    void speak() { cout << "Animal speak()" << endl; }
};

class Dog : public Animal {
public:
    int age;
    void speak() { cout << "멍멍" << endl; }
};

class Cat : public Animal {
public:
    void speak() { cout << "야옹" << endl; }
};
```

```
int main() {
    Animal *a1 = new Dog();
    a1->speak(); // 출력?

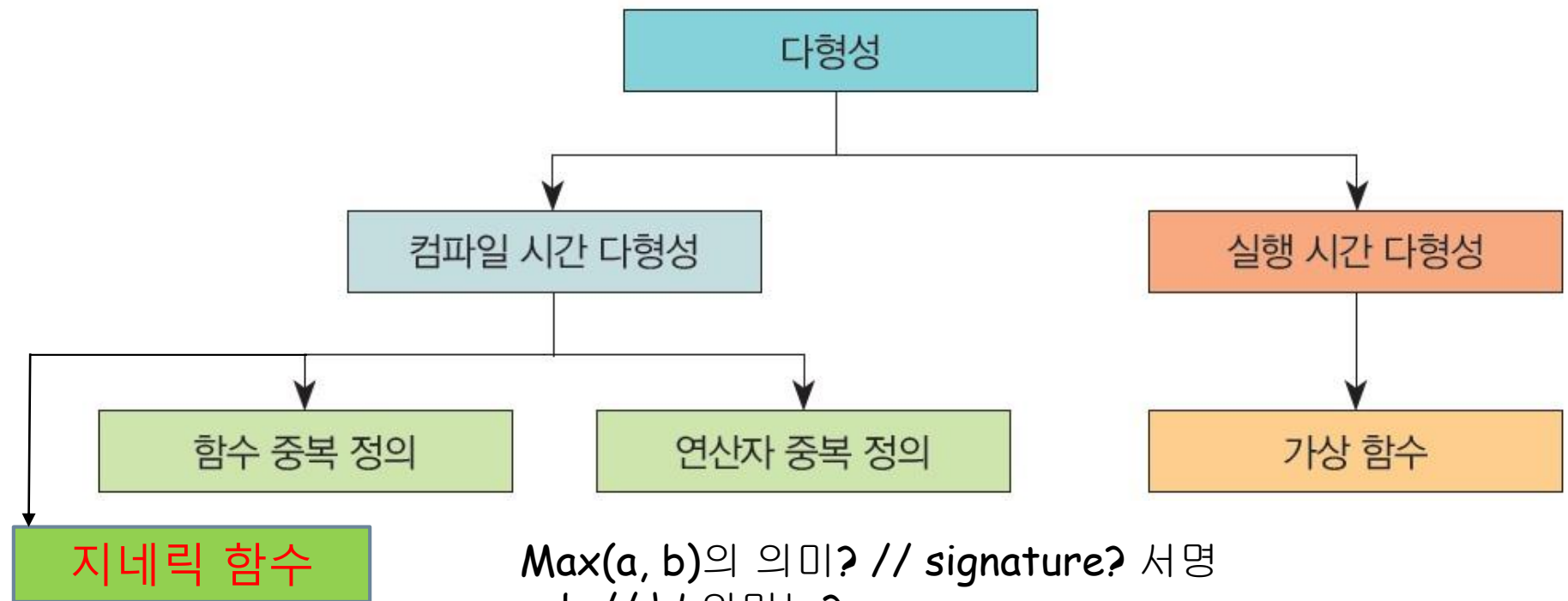
    Animal *a2 = new Cat();
    a2->speak(); // 출력?

    a1->age = 10; // ?
    return 0;
}
```

# 다형성이란?

- 다형성(polymorphism)은 “다양한 형태를 갖는 다”는 그리스어의 어원
  - Ex. 한 기호가 여러 가지 의미로 사용
- 다형적 함수(polymorphic function)란 함수 호출 시에 다른 타입의 매개변수를 가질 수 있는 함수를 말함
  - `Max(a, b) //a, b: int`  
`// a, b: float`
  - `Max(a,b,c)`

# C++에서의 다형성



Max(a, b)의 의미? // signature? 서명  
a+b // '+' 의미는?

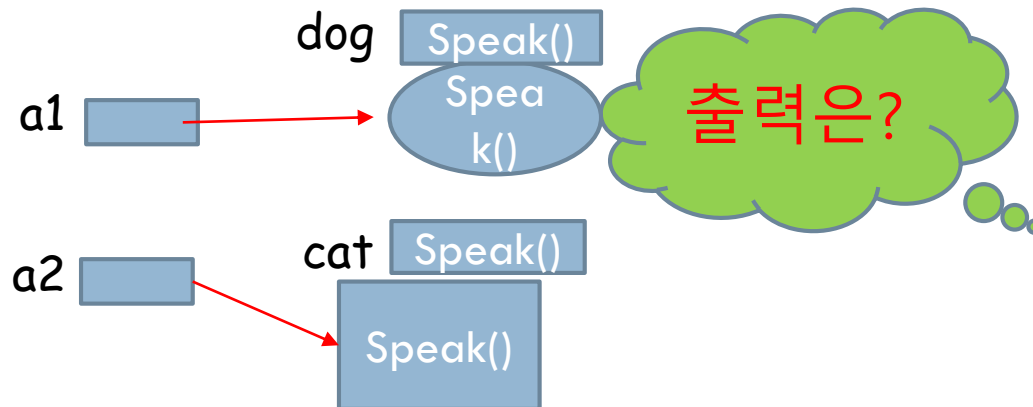
f (T a) {...}  
F(n)  
F(f)

```
class Animal {
public:
    void speak() { cout << "Animal speak()" << endl; }
};
```

```
class Dog : public Animal {
public:
    int age;
    void speak() { cout << "멍멍" << endl; }
};
```

```
class Cat : public Animal {
public:
    void speak() { cout << "야옹" << endl; }
};
```

정적 바인딩 vs  
동적 바인딩



```
int main() {
    Animal *a1 = new Dog();
    a1->speak();

    Animal *a2 = new Cat();
    a2->speak();

    return 0;
}
```

```
class Animal
{
public:
    virtual void speak() { cout << "Animal speak()" << endl; }
};
```

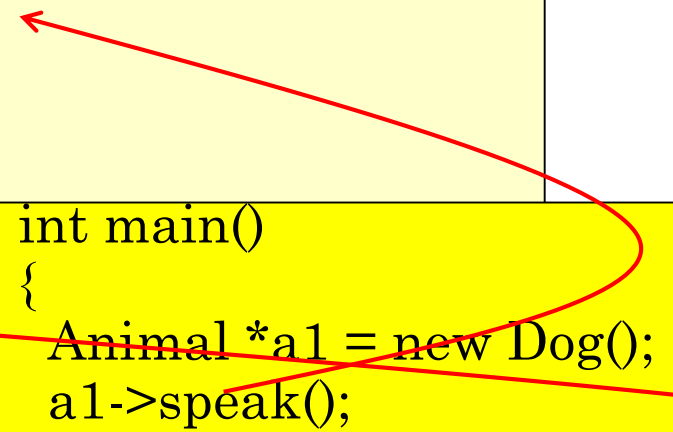
```
class Dog : public Animal
{
public:
    int age;
    void speak() { cout << "멍멍" << endl; }
};
```

```
class Cat : public Animal
{
public:
    void speak() { cout << "야옹" << endl; }
};
```

```
int main()
{
    Animal *a1 = new Dog();
    a1->speak();

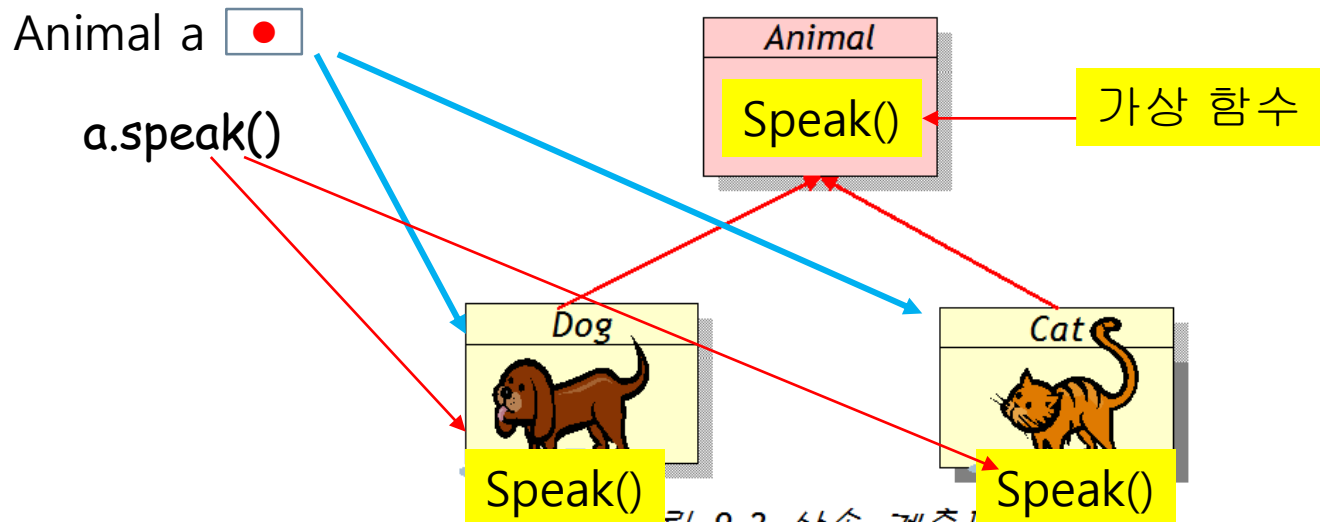
    Animal *a2 = new Cat();
    a2->speak();

    return 0;
}
```



# 가상 함수

- Animal 포인터를 통하여 객체의 멤버 함수를 호출하더라도 객체의 타입에 따라서 서로 다른 `speak()`가 호출된다면 상당히 유용할 것이다.
- 이를 위해서, C++에서 멤버 함수가 **가상 함수 (virtual function)**로 설정되어야 함







```
class Shape {  
protected:  
    int x, y;  
public:  
    Shape(int x, int y) : x(x), y(y) {  
        virtual void draw() {  
            cout << "Shape Draw" << endl;  
        }  
};
```

```
class Rect: public Shape {  
private:  
    int width, height;  
public:  
    Rect(int x, int y, int w, int h) : Shape(x, y), width(w), height(h) {}  
    void draw() {  
        cout << "Rectangle Draw" << endl;  
    }  
};
```

출력은?

```
int main() {  
    Shape *ps = new Rect(0, 0, 100, 100);  
    ps->draw();  
  
    delete ps;  
    return 0;  
}
```

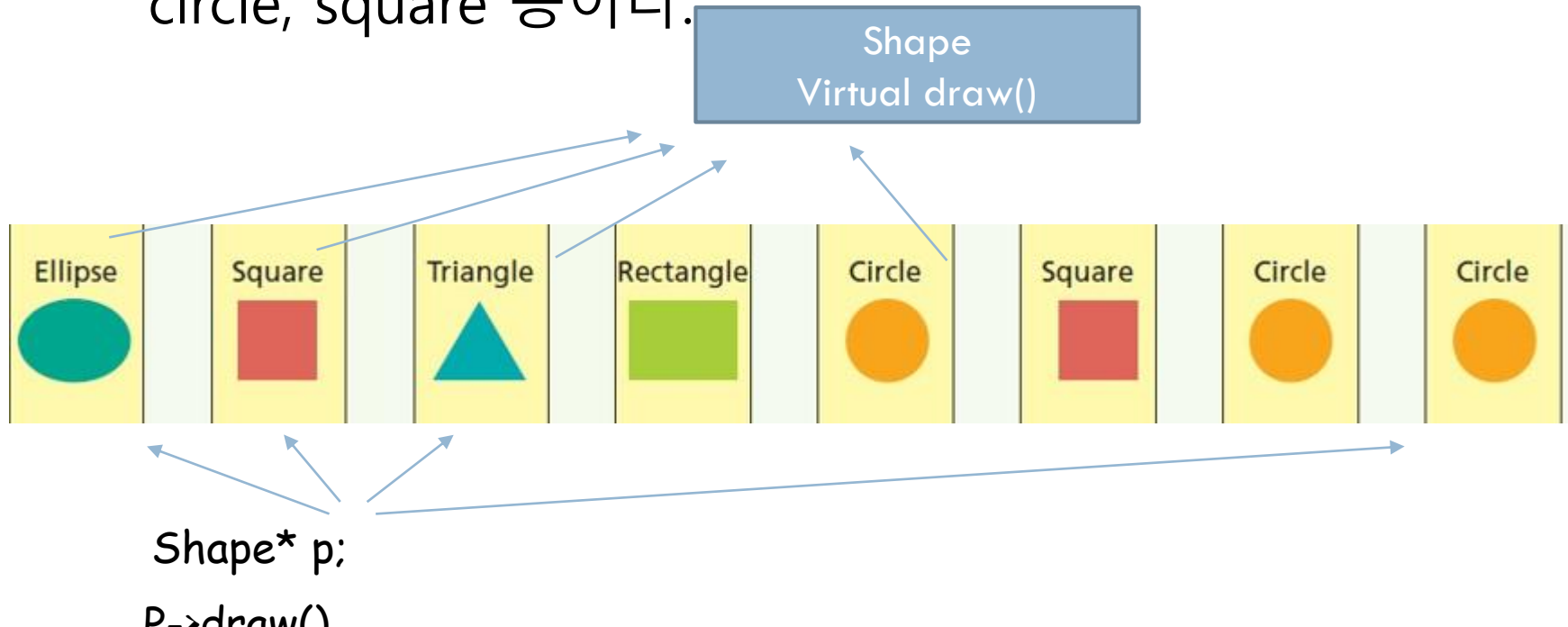
# 동적 바인딩 vs 정적 바인딩

- 바인딩(binding)은 함수 호출을 함수 정의와의 연결
- C++에서 정적 바인딩은 실행 전에 이루어지고, 가상 함수 아닌 일반 함수 대상이고,
- 동적 바인딩은 실행 중에 이루어지고, 가상 함수 대상

바인딩의 종류	특징	속도	대상
정적 바인딩 (static binding)	컴파일 시간에 호출 함수가 결정된다. (포인터 타입 기준)	빠르다	일반 함수
동적 바인딩 (dynamic binding)	실행 시간에 호출 함수가 결정된다. (포인터가 가리키는 콘텐츠(객체))	늦다	가상 함수

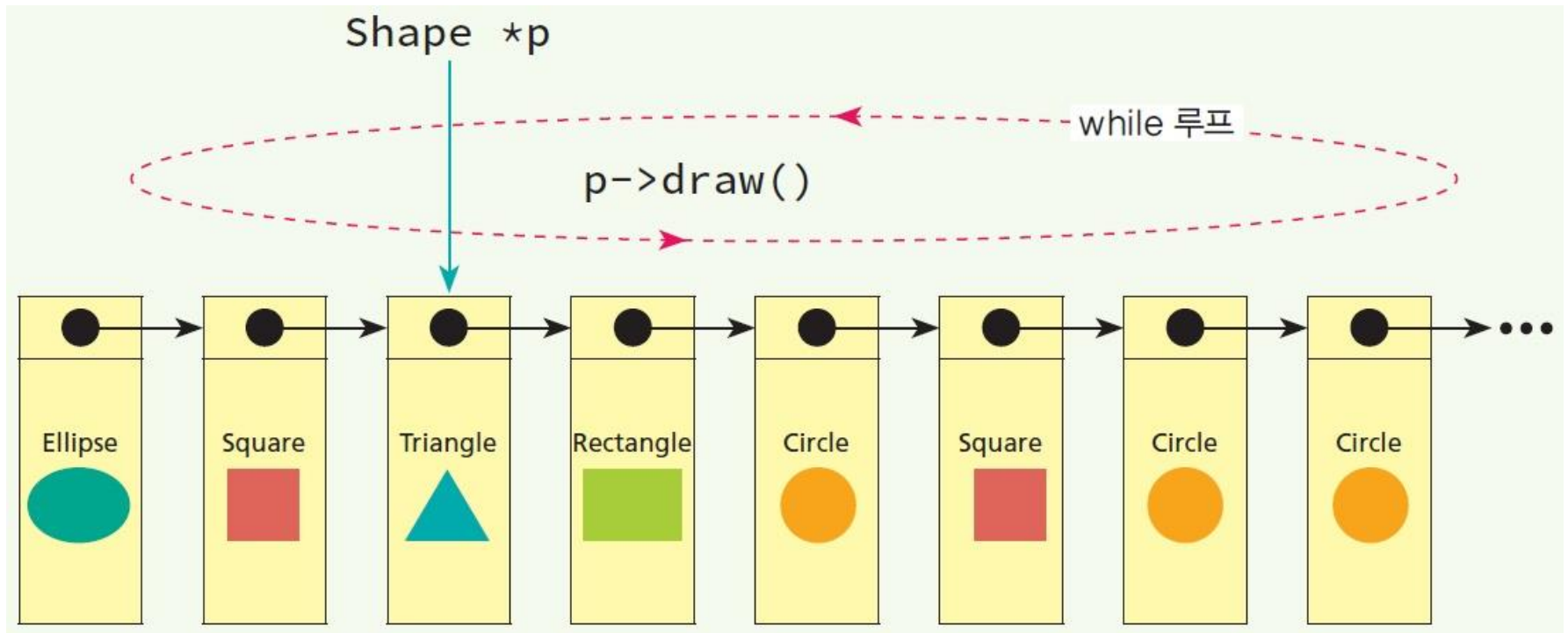
# 예제

- 다양한 도형을 배열에 저장하고, 배열에 포함된 각 도형을 적절하게 그리고자 한다. 어떻게 해결할 것인가?
- 도형 타입은 ellipse, shquare, triangle, rectangle, circle, square 등이다.



# 예제(계속)

- 다양한 도형을 포함하는 배열 표현은?



# 예제 (계속)

- Shape 클래스를 설계하고, Rectangle, Triangle, Circle 등의 클래스를 그 자식 클래스로 설계
- Shape의 draw() 함수를 가상 함수로 명세

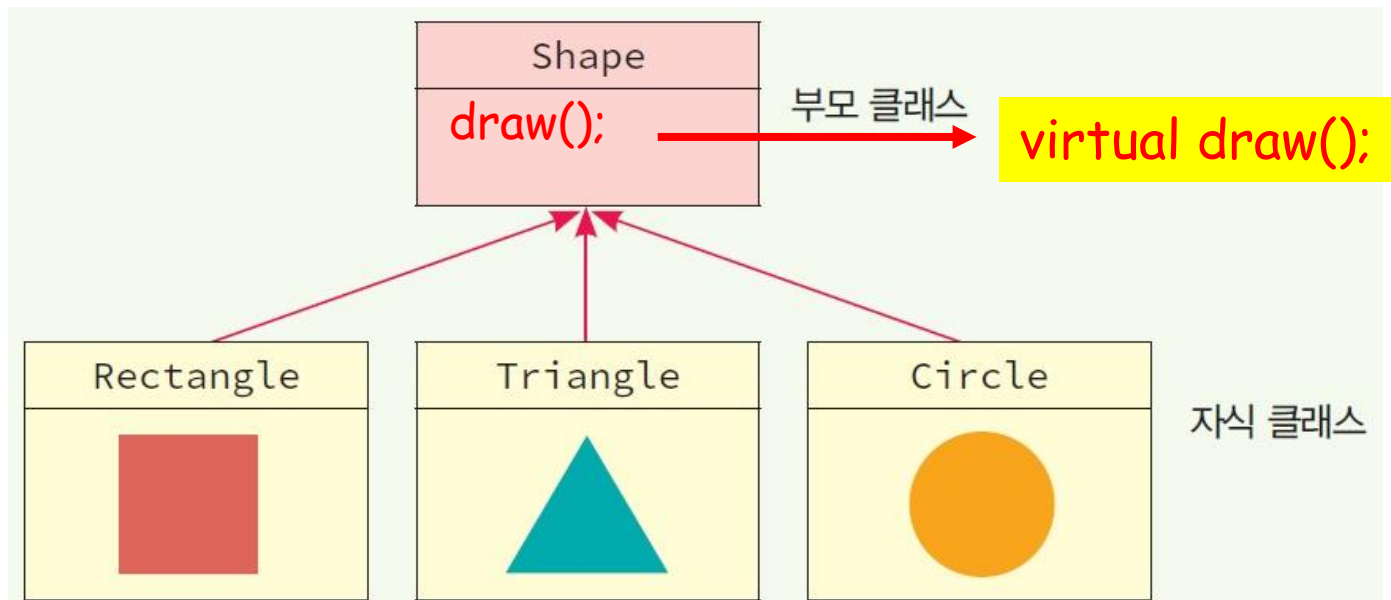


그림 12.3 도형의 상속 구조

# 코드

```
class Shape {
protected:
    int x, y;
public:
    Shape(int x, int y) : x(x), y(y) {    }
    virtual void draw() {
        cout << "Shape Draw" << endl;
    }
};

class Rect : public Shape {
private:
    int width, height;
public:
    Rect(int x, int y, int w, int h) : Shape(x, y), width(w), height(h) {
    }
    void draw() {
        HDC hdc = GetWindowDC(GetForegroundWindow());
        Rectangle(hdc, x, y, x + width, y + height);
    }
};
```

```

class Circle : public Shape {
private:
    int radius;
public:
    Circle(int x, int y, int r) : Shape(x, y), radius(r) { }
    void draw() {
        HDC hdc = GetWindowDC(GetForegroundWindow());
        Ellipse(hdc, x - radius, y - radius, x + radius, y + radius);
    }
};

```

```

int main()
{

```

```

    Shape *shapes[3];

```

```

    shapes[0] = new Rect(rand() % 600, rand() % 300, rand() % 100, rand() % 100);

```

```

    shapes[1] = new Circle(rand() % 600, rand() % 300, rand() % 100);

```

```

    shapes[2] = new Circle(rand() % 600, rand() % 300, rand() % 100);

```

```

    for (int i = 0; i < 3; i++) {

```

```

        shapes[i]->draw();

```

```

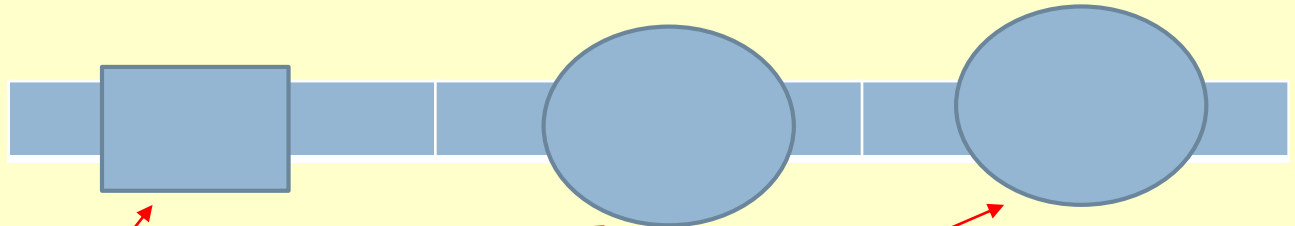
    }

```

```

}

```



# 참조자와 가상함수

- 참조자도 포인터와 마찬가지로 모든 것이 동일하게 적용된다. 즉 부모 클래스의 참조자로 자식 클래스를 참조할 수 있으며 가상 함수의 동작도 동일하다.



```
class Animal
{
public:
    virtual void speak() { cout << "Animal speak()" << endl; }
};
```

```
class Dog : public Animal
{
public:
    int age;
    void speak() { cout << "멍멍" << endl; }
};
```

```
class Cat : public Animal
{
public:
    void speak() { cout << "야옹" << endl; }
};
```

출력은?

```
int main()
{
    Dog d;
    Animal &a1 = d;
    a1.speak();

    Cat c;
    Animal &a2 = c;
    a2.speak();
}
```

# 가상 소멸자

- 부모 클래스의 소멸자 가상이 아니고, 이 클래스의 포인터 p가 자식 클래스 객체를 가리킬 때, delete p는 문제를 야기할 수 있다.
  - Parent 클래스 타입 포인터 p로 Child 클래스 타입의 객체를 가리킬 경우에, delete p는 단지 parent 클래스의 소멸자만 호출
  - Parent 클래스의 소멸자를 virtual로 설정하면, Child 클래스 소멸자 먼저 호출되고, 다음에 Parent 클래스의 소멸자가 호출

# 예제

```
class Parent
{
public:
    ~Parent() { cout << "Parent 소멸자" << endl; }
};
class Child : public Parent
{
public:
    ~Child() { cout << "Child 소멸자" << endl; }
};

int main()
{
    Parent* p = new Child();
    delete p; // 이 문장이 생략될 경우는?
}
```

출력은?

# 예제(계속)

- 부모 클래스의 소멸자를 가상 함수로 선언

```
class Parent
{
public:
    virtual ~Parent() { cout << "Parent 소멸자" << endl; }
};
class Child : public Parent
{
public:
    ~Child() { cout << "Child 소멸자" << endl; }
};

int main()
{
    Parent* p = new Child();
    delete p;
}
```

출력은?

# 순수 가상 함수

- 기본 클래스의 가상 함수 목적
  - 파생 클래스에서 재정의할 함수를 알려주는 역할
  - 실행할 코드를 작성할 목적이 아님
  - 기본 클래스의 가상 함수를 굳이 구현할 필요가 있을까?
- 순수 가상 함수(pure virtual function)
  - 함수의 코드가 없고 선언만 있는 가상 멤버 함수
  - 선언 방법: '멤버 함수의 원형 =0;'으로 선언

```
class Shape {  
public:  
    virtual void draw()=0; // 순수 가상 함수 선언  
};
```

# 추상 클래스

30

- 추상 클래스 : 적어도 하나의 순수 가상 함수를 가진 클래스

```
class Shape { // Shape은 추상 클래스
    Shape *next;
public:
    void paint() {
        draw();
    }
    virtual void draw() = 0; // 순수 가상 함수
};

void Shape::paint() {
    draw();
}
```

# 추상 클래스 특징

31

- 온전한 클래스가 아니므로 객체 생성 불가능

```
Shape shape; // 컴파일 오류  
Shape *p = new Shape(); // 컴파일 오류
```

- 추상 클래스의 포인터는 선언 가능

```
Shape *p;
```

# 추상 클래스의 목적

- 추상 클래스의 인스턴스를 생성할 목적 아님
  - 추상 클래스는 상속 계층구조의 상위 레벨에 위치
  - 상위 레벨 클래스는 일반적이어서 사례화되기보다는 하위 레벨 클래스의 기반으로 사용
  - 상위 레벨 클래스의 메소드는 구현 의미가 없고, 하위 레벨 클래스에서 구현되는 용도의 역할을 함
- 상속에서 기본 클래스의 역할을 하기 위함
  - 순수 가상 함수를 통해 파생 클래스에서 구현할 함수의 형태(원형)을 보여주는 인터페이스 역할
  - 추상 클래스의 모든 멤버 함수를 순수 가상 함수로 선언할 필요 없음



# 추상 클래스의 상속과 구현

- 추상 클래스의 상속

- 추상 클래스를 단순 상속하면 자식 클래스는 자동으로 추상 클래스가 됨

- 추상 클래스의 구현

- 추상 클래스를 상속받아 순수 가상 함수를 오버라이딩
- 자식 클래스는 추상 클래스가 아님

# 예제

## 추상 클래스의 단순 상속

```
class Shape {  
public:  
    virtual void draw() = 0;  
};  
  
class Circle : public Shape { // 추상 클래스  
public:  
    string toString() { return "Circle 객체"; }  
};  
  
Shape* shape;  
Circle* waffle;
```

## 추상 클래스의 구현

```
class Shape {  
public:  
    virtual void draw() = 0;  
};  
  
class Circle : public Shape { // 추상 클래스 아님  
public:  
    void draw() {  
        cout << "Circle";  
    }  
    string toString() { return "Circle 객체"; }  
};  
  
Shape* shape;  
Circle waffle;
```

```
class Shape {
protected:
    int x, y;
public:
    Shape(int x, int y) : x(x), y(y) {
        virtual void draw() = 0;
    };
class Rect : public Shape {
private:
    int width, height;
public:
    Rect(int x, int y, int w, int h) : Shape(x, y), width(w), height(h) {
        void draw() {
            cout << "Rectangle Draw" << endl;
        }
    };
};
```

```
int main() {
    Shape *ps = new Rect(0, 0, 100, 100);

    ps->draw();
    delete ps;

    return 0;
}
```

출력은?

# Lab: 동물 예제

move();  
speak();  
eat();

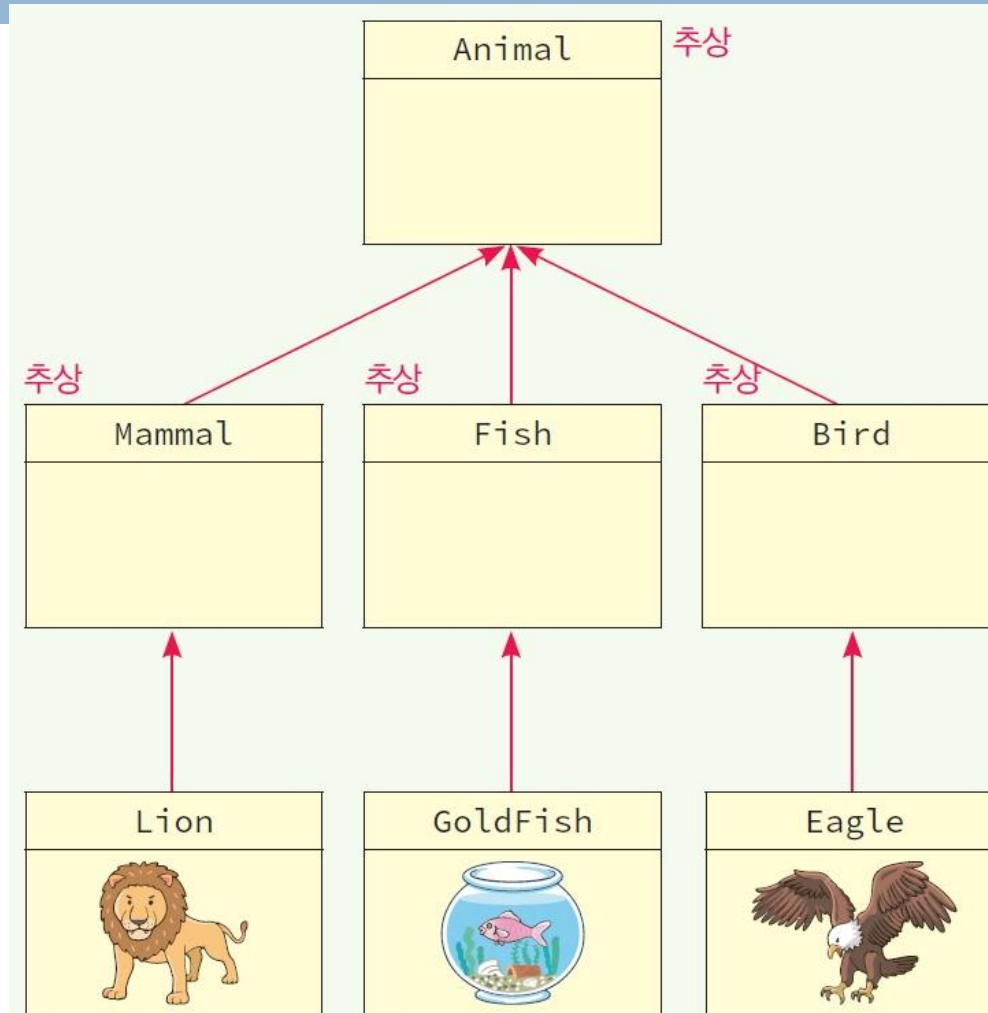


그림 12.4 동물의 상속 계층도에서 추상 클래스

# 코드

```
class Animal {  
    virtual void move() = 0;  
    virtual void eat() = 0;  
    virtual void speak() = 0;  
};  
class Lion : public Animal {  
    void move(){  
        cout << "사자의 move()" << endl;  
    }  
    void eat(){  
        cout << "사자의 eat()" << endl;  
    }  
    void speak(){  
        cout << "사자의 speak()" << endl;  
    }  
};
```