C++ Espresso

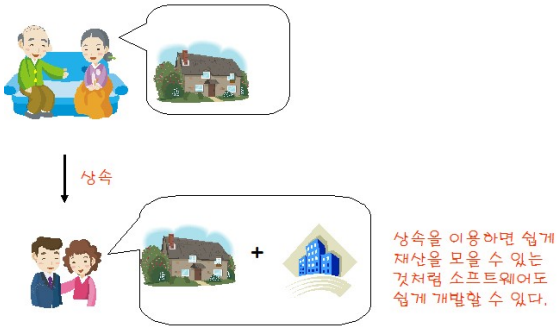
제8장 상속

이번 장에서 학습할 내용

- 상속이란?
- 접근 제어 지정자
- 상속에서의 생성자와 소멸자
- 재정의(오버라이딩)
- 다중 상속

상속이란?

- 상속: 기존에 존재하는 유사한 클래스로부터 속성과 동작을 이어받고 자신이 필요한 기능을 추가하는 기법



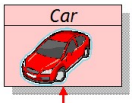
상속의 장점

- 상속의 장점
  - 상속을 통하여 기존 클래스(부모 클래스)의 필드(멤버변수)와 메소드(멤버함수)를 재사용
    - 자식 클래스는 부모 클래스의 멤버변수와 멤버함수를 상속 받아 사용할 수 있다.
  - 기존 클래스(부모 클래스) 내용의 일부 변경도 가능
  - 상속은 이미 작성된 검증된 소프트웨어를 재사용
    - 기존에 작성된 클래스를 상속하면 그 기능을 그대로 사용 가능
  - 신뢰성 있는 소프트웨어를 손쉽게 개발, 유지 보수
  - 코드의 중복을 줄일 수 있다.

상속

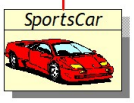
상속한다는 의미

```
class Car
{
        int speed;
}
class SportsCar : public Car
{
        bool turbo;
}
```



수퍼클래스(superclass)

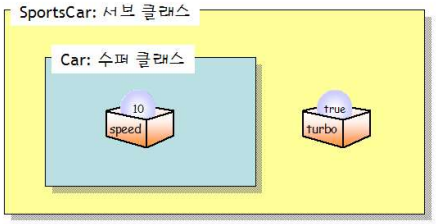
부모 클래스

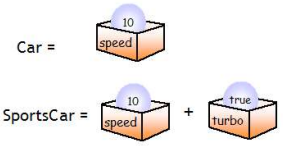


서브클래스(subclass)

자식 클래스

자식 클래스는 부모 클래스를 포함





상속의 예(is-a 관계)

수퍼 클래스	서브 클래스
Animal(동물)	Lion(사자), Dog(개), Cat(고양이)
Bike(자전거)	MountainBike(산악자전거)
Vehicle(탈것)	Car(자동차), Bus(버스), Truck(트럭), Boat(보트), Motocycle(오토바이), Bicycle(자전거)
Student(학생)	GraduateStudent(대학원생), UnderGraduate(학부생)
Employee(직원)	Manager(관리자)
Shape(도형)	Rectangle(사각형), Triangle(삼각형), Circle(원)

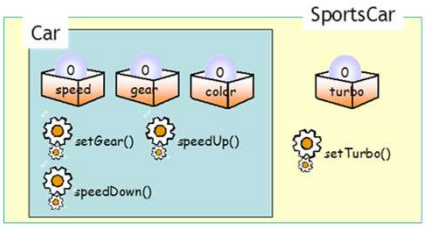
참고

- 수퍼 클래스 == 부모 클래스(parent class) == 베이스 클래스(base class)
- 서브 클래스 == 자식 클래스(child class) == 파생된 클래스(derived class)

Car 클래스, SportsCar 클래스

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

class Car {
public:
        // 3개의멤버변수선언
        int speed; // 속도
        int gear; // 주행거리
        string color; // 색상

        // 3개의멤버함수선언
        void setGear(int newGear) {
                gear = newGear;
        }
        void speedUp(int increment) {
                speed += increment;
        }
        void speedDown(int decrement) {
                speed -= decrement;
        }
};
```

```
// Car를 상속받는다.
class SportsCar : public Car {
    bool turbo;          // 1개의 멤버변수를 추가

public:
        // 1개의 멤버 함수를 추가
        // 터보모드 설정 멤버함수
    void setTurbo(bool newValue) {
        turbo = newValue;  // T or F
    }
};
```



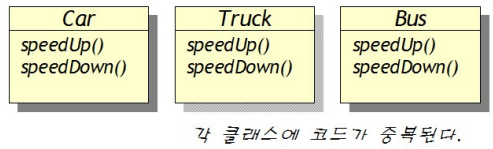
SportsCar 클래스

```
int main()
{
        SportsCar  c;

        c.color = "Red";          // 부모클래스멤버변수접근
        c.setGear(3);             // 부모클래스멤버함수호출
        c.speedUp(100);           // 부모클래스멤버함수호출
        c.speedDown(30);          // 부모클래스멤버함수호출
        c.setTurbo(true);         // 자식 멤버함수호출
        return 0;
}
```

자식 클래스는 부모 클래스의 변수와 함수를 마치 자기 것처럼 사용할 수 있다.

c
speed, gear=3
color = "Red"
setGear(), speedup()
speedDown()
turbo = true
setTurbo()

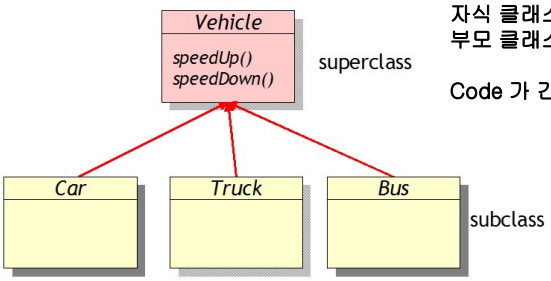
상속은 왜 필요한가?



class

각 클래스에 코드가 중복된다.

상속 관계로 작성



superclass

subclass

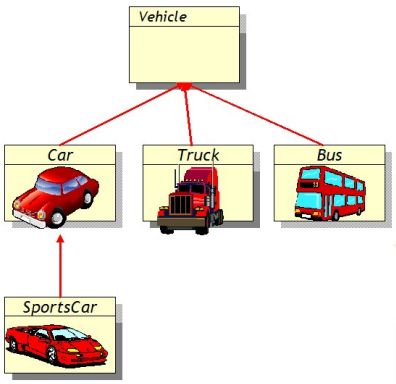
자식 클래스의 공통 속성을
부모 클래스 속성으로 지정

Code 가 간단

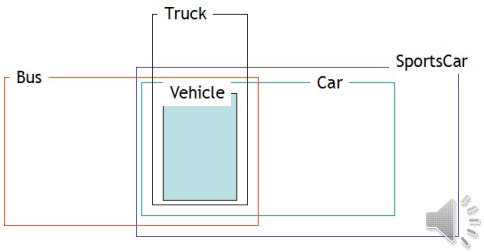
중복되는 코드는 수퍼 클래스에 모은다.

상속 계층도

- 상속은 여러 단계로 이루어질 수 있다.




```
class Vehicle { ...        }
class Car : public Vehicle { ...    }
class Truck : public Vehicle { ...  }
class Bus : public Vehicle { ...    }
class SportsCar : public Car { ...  }
```



상속은 is-a/사용 관계

- 상속은 is-a 관계
  - 스포츠카는 승용차이다
    - 승용차(부모 클래스), 스포츠 카(자식 클래스)
    - 스포츠카는 승용차의 기본 기능에 새로운 특성이 추가된 것.
  - 승용차는 차량이다. (Car *is a* Vehicle).
    - 차량(부모 클래스), 자동차(자식 클래스)
  - 사자, 개, 고양이는 동물이다.
    - 동물(부모 클래스), 사자/개/고양이(자식 클래스)
- has-a(포함) 관계(7 장)는 상속으로 모델링을 하면 안 된다.
  - 도서관은 책을 가지고 있다(Library *has a* book).
  - 사각형은 꼭지점을 가지고 있다.
  - 포함 관계로 표현

접근 제어 지정자

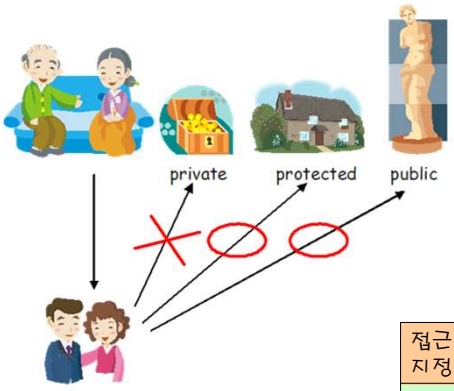


그림 13.9 상속에서의 접근 지정자

부모의 멤버의 접근 지정자에 따라 자식에서 접근 가능성

접근 지정자	부모 클래스 현재 클래스	부모 클래스 자식 클래스	외부
private	○	×	×
protected	○	○	×
public	○	○	○

접근 제어 지정자 예제-미완성(생성자 없음)

```
class  Employee {
// Regident Resgistration Number: 주민번호
        int rrn;          // private

protected:
        int salary;     // 월급

public:
        string name;            // 이름
        void setSalary(int salary);
        int getSalary();
};
// 월급 지정
void Employee::setSalary(int salary) {
        this->salary = salary;
}

// 사용 안함
int Employee::getSalary() {
        return salary;
}
```

```
class Manager : public Employee {
        int bonus;
public:
        Manager(int b=0) : bonus(b) { }
        void modify(int s, int b);
        void display();
};

// 월급, 보너스를 변경
void Manager::modify(int s, int b) {
// 부모클래스의 보호(protected)멤버 사용 가능!
        salary = s; // 부모멤버
        bonus = b;  //자식 멤버
}

// 주민번호, 월급, 보너스 출력
void Manager::display() {

  cout << "봉급: " << salary << " 보너스: "
    << bonus << endl;  // 주민번호는 출력 못함
}
```

예 제

```
int main()
{
        Manager m;          // employee 클래스의 자식 클래스 객체
        m.setSalary(2000);          //부모 멤버함수
        m.display();                // 자식 멤버함수
        m.modify(1000, 500);        // 자식 멤버함수
        m.display();                // 자식 멤버함수
}
```

```
봉급: 2000 보너스: 0
봉급: 1000 보너스: 500
계속하려면 아무 키나 누르십시오 . . .
```

상속에서의 생성자와 소멸자

- 자식 클래스의 객체가 생성될 때에 당연히 자식 클래스의 생성자는 호출된다. → 이때 부모 클래스 생성자도 호출될까?

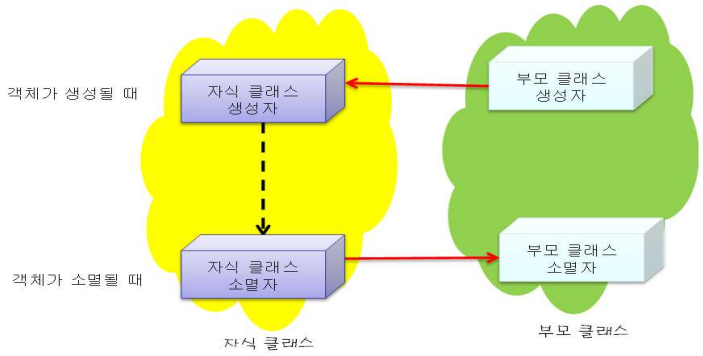


그림 8.10 상속에서 생성자와 소멸자의 호출

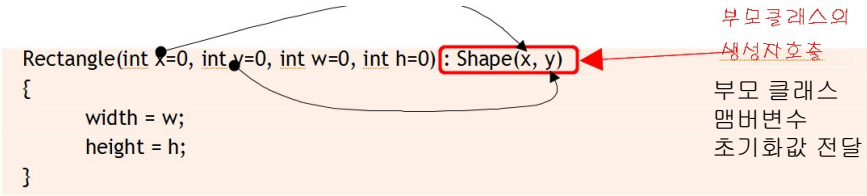
예제) 자식 생성자에서 부모 생성자 호출 없는 경우

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

class Shape {
      int x, y;
public:
   Shape() {
     cout << "Shape 생성자() " << endl;
   }
   ~Shape() {
     cout << "Shape 소멸자() " << endl;
   }
};
```

```
class Rectangle : public Shape {
   int width, height;
public:
   Rectangle() {
            cout << "Rectangle 생성자()" << endl;
   }
// 컴파일러는 위 함수를 다음과 같이 해석
//   Rectangle() : Shape() {          // 위 함수와 동일
//            cout << "Rectangle 생성자()" << endl;
//   }

   ~Rectangle(){
            cout << "Rectangle 소멸자()" << endl;
   }
};
int main() {
            Rectangle r;
            return 0;
}
```

```
Shape 생성자()
Rectangle 생성자()
Rectangle 소멸자()
Shape 소멸자()
계속하려면 아무 키나 누르십시오 . . .
```

자식 생성자에서 부모 생성자의 명시적 호출




```
class Rectangle : public Shape {
        ...
public:
  Rectangle() {
        cout << "Rectangle 생성자()" << endl;
  }
  // 위와 동일 내용
  // Rectangle() : Shape() {
  //       cout << "Rectangle 생성자()" << endl;
  // }
};
```

명시적으로 지정하지 않는 경우 → 옆과 같이 인자 없이 부모 생성자 호출

예 제

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

class Shape {
   int x, y;  // 좌상단 좌표, private
public:
   Shape() {
      cout << "Shape 생성자() " << endl;
   }
   Shape(int xloc, int yloc) : x(xloc), y(yloc){
      cout << "Shape 생성자(xloc, yloc) " << endl;
   }
   ~Shape() {
      cout << "Shape 소멸자() " << endl;
   }
};
```

```
class Rectangle : public Shape {
   int width, height;  // private

public:
   Rectangle(int x=0, int y=0, int w=0, int h=0);

   ~Rectangle(){
      cout << "Rectangle 소멸자()" << endl;
   }
};

Rectangle::Rectangle(int x, int y, int w, int h)
                        : Shape(x, y)
{
   width = w;
   height = h;
   cout << "Rectangle 생성자(x, y, w, h)" << endl;
}
```

- **각 변수는 private**
- **각 클래스 생성자는 자신의 멤버변수만 초기화하면 된다.**
- **부모 생성자 수행 → 부모 멤버변수는 부모 생성자에서 초기화**
- **has-a 관계에서도 자기 멤버만 초기화**

예 제

```
Shape::Shape(int xloc, int yloc) : x(xloc), y(yloc){

        cout << "Shape 생성자(xloc, yloc) " << endl;
}
```

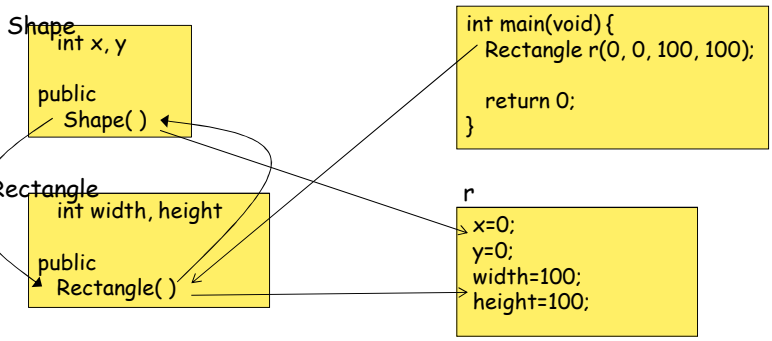
```
Rectangle::Rectangle(int x, int y, int w, int h)
                        : Shape(x, y)
{
   width = w;
   height = h;
   cout << "Rectangle 생성자(x, y, w, h)" << endl;
}
```

```
int main(){
            Rectangle r(0, 0, 100, 100);
            return 0;
}
```

```
Shape 생성자(xloc, yloc)
Rectangle 생성자(x, y, w, h)
Rectangle 소멸자()
Shape 소멸자()
계속하려면 아무 키나 누르십시오 . . .
```

<생성자 호출 순서>
- 부모 클래스(Shape) 생성자 → 자식 클래스(Rectangle) 생성자

<소멸자 호출 순서>
- 자식 클래스 (Rectangle) 소멸자 → 부모 클래스 (Point) 소멸자



- Rectangle 객체 r 생성
- Rectangle 생성자 호출
  - Shape 생성자 호출/수행
    - x=0, y=0
- Rectangle 생성자 수행
  - width=100, height=100

Has-a 관계와 is-a 관계의 생성자 비교

Rectangle 은 Point 를 포함한다.

```
class Point{
   int x, y;
public:
   Point(int a, int b) : x(a), y(b)      { }
};

class Rectangle{
   Point p1, p2;
public:
   Rectangle(int x1, int y1, int x2, int y2)
      : p1(x1, y1), p2(x2, y2)   // 멤버객체
   { }          // Point p1(x1, y1); 으로 해석
                // P1 객체 생성 -> Point 생성자 호출
};

int main() {
   Rectangle r1(10, 10, 100, 100);
   return 0;
}
```

Rectangle 은 shape 이다.

```
class Shape {
   int x, y;  // 좌상단 좌표
public:
  Shape(int xloc, int yloc) : x(xloc), y(yloc){     }
};

class Rectangle : public Shape {
   int width, height;
public:                                  // 부모 생성자 호출
   Rectangle(int x, int y, int w, int h) : Shape(x, y) {
      width = w;          height = h;
   }
};

int main() {
   Rectangle r(0, 0, 100, 100);
   return 0;
}
```

상속 예제 1

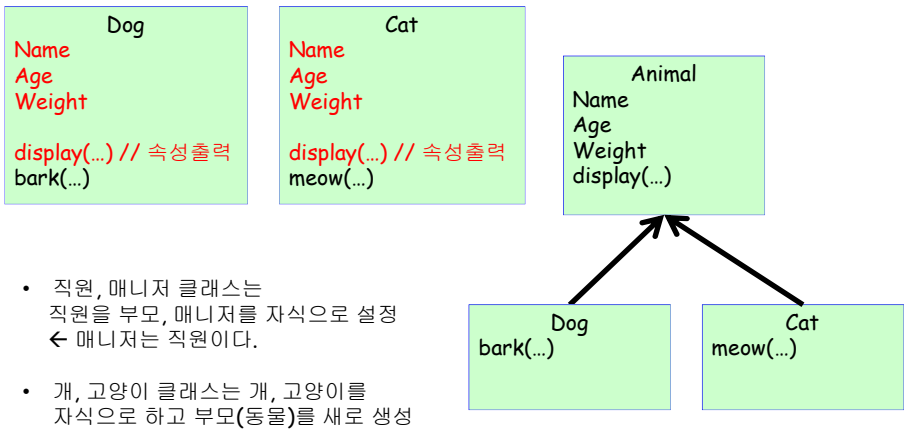
- 개와 고양이

```
int main() {
            // 이름, 나이, 무게
   Dog dol("쫑", 8, 20);
   Cat nabi("나비", 4, 15);

   dol.display();       // 이름, 나이, 무게 출력
   dol.bark();          // 멍멍 출력

   nabi.display();
   nabi.meow();         // 야옹 출력
   return 0;
}
```

  - 등장 객체 → 개, 고양이 → 각각을 클래스로 구현

상속 예제 1

- 각 클래스 공통 부분을 모아 부모 클래스 구성

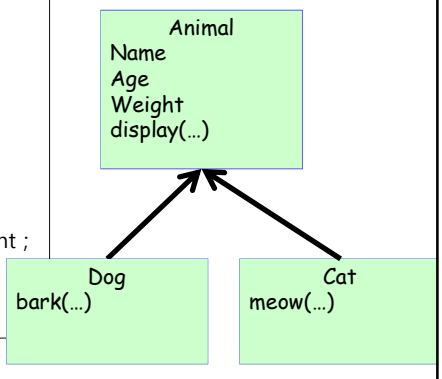


- 직원, 매니저 클래스는
  직원을 부모, 매니저를 자식으로 설정
  ← 매니저는 직원이다.
- 개, 고양이 클래스는 개, 고양이를
  자식으로 하고 부모(동물)를 새로 생성
  ← 개/고양이는 동물이다.

23

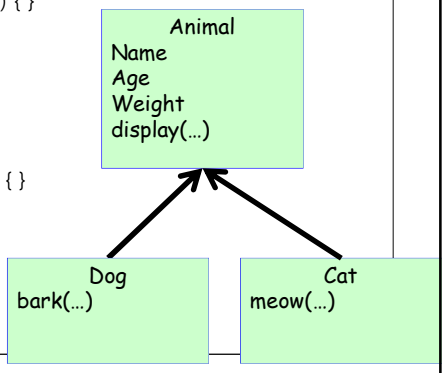
- 각각을 클래스로 구현

```
class Animal {
     string name;
     int age, weight;

public:
      Animal(string n, int a, int w)
     : name(n), age(a), weight(w) { }

     void display() const {
            cout << name << age << weight ;
       }

};
```



24

```
class Dog : public Animal {
public:
      Dog (string n, int a, int w) : Animal (n, a, w) { }

      void bark() { cout << "멍멍" << endl; }
};

class Cat : public Animal {
public:
      Cat (string n, int a, int w) : Animal (n, a, w) { }

      void meow() { cout << "야옹" << endl; }
};
```



25

상속 예제 2

- full-time, part-time 직원

```
int main() {
                        // 이름, 정규직(1)/비정규직(0), 월급, 보너스
   FullTime a("김철수", 1, 200, 120);
                        // 비정규직, 보너스 없음.
   PartTime b("김현빈", 0, 150);

   a.displayF(); // 모든 속성을 출력, 속성 차이로 다른 두 함수 사용 출력
   b.displayP();
   return 0;
}
```

- 등장 객체를 파악하여 각각을 클래스로 작성 → full time, part time

26

상속 예제 2

- 등장 객체를 파악하여 각각을 클래스로 작성 → full time, part time

full-time	part-time
Name Full_time // 정규직 1 Salary // 월급 Bonus // 보너스 … DisplayF(…) // 이름, 월급, 보너스 출력	Name Full_time // 비정규직 0 Salary // 월급 … DisplayP(…) // 이름, 월급 출력

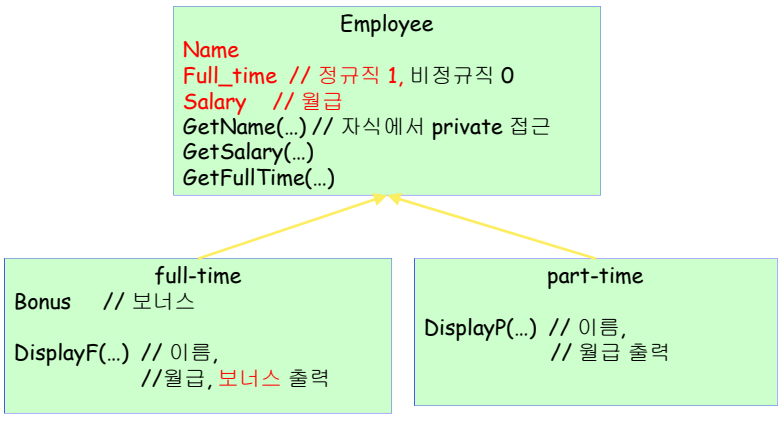
- 각 클래스의 공통 부분을 클래스로 구성하고 부모 클래스로 설정

Employee
Name Full_time // 정규직 1/비정규직 0 Salary // 월급

27

상속 예제 2

- 부모 자식 관계



?? Display(공통 내용 있음) 를 한 함수로 구현하는 것이 좋지 않나 ?
→ 나중에 가상함수(오버라이딩) 사용으로 해결 가능

28

- 각각을 클래스로 구현

```
class Employee {
    string name;
    int salary, full_time;

public:
     Employee(string n, int f , int s)
    : name(n), full_time(f), salary(s) { }

    string getName() const { return name; }

    int getSalary() const { return salary; }

    // full time 이면 1 반환, part time 이면 0 반환
    bool getFullTime() const { return full_time; }
};
```

Employee
Name
Full_time // 정규직 1
Salary // 월급
GetName(…)
GetSalary(…)
GetFullTime(…)

29

```
class FullTime : public Employee {
    int bonus;
public:
     FullTime(string n, int f, int s, int b)
          : Employee(n,  f, s), bonus(b) { }

    void DisplayF() const;
};

void FullTime ::DisplayF() const {
    cout << getName() << endl;
    cout << getSalary() << endl;
    cout << getFullTime() << endl;      // (1)
    cout << bonus << endl;
}

// 다음 내용은 (1) 과 동일
// cout << Employee::getFullTime() << endl;
```

full-time
Bonus // 보너스

DisplayF(…) // 이름, 나이,
//월급, 보너스 출력

30

```
class PartTime : public Employee {
public:
      PartTime(string n, int f, int s)
            : Employee(n,  f, s) { }

     void DisplayP() const;
};

void PartTime ::DisplayP() const {
     cout << getName() << endl;
     cout << getSalary() << endl;
     cout << getFullTime() << endl;
}
```

part-time

DisplayP(…) // 이름, 나이,
// 월급 출력

31

Report P8_1, P8_2

- 과제 2개, 각각 제출
  - oop_학번_8_1.txt
  - oop_학번_8_2.txt
- DOOR 과제에 제출(활동일지 아님)

Report P8_1

- 상속을 사용하여 2 입력 and, or, xor gate를 구현(예전 과제는 상속 사용 않음)
- 프로그램 작성시 and, or, xor 연산자는 각각 &&, ||, ^ 사용
- ANDGate, ORGate, XORGate 객체 상태(멤버변수, protected)
  - bool x, y; (-> 입력변수 저장), bool z; (-> gate 연산결과 저장)
- ANDGate, ORGate, XORGate 동작(멤버함수, public)
  - 1) 생성자 -> 인자 없음, 멤버 변수를 모두 false로 저장
  - 2) gate 입력(x, y) 지정 -> void inputSet(bool xx, bool yy)

    : xx 를 멤버변수 x 에 저장, yy 를 멤버변수 y 에 저장

    : main 에서 호출, public
  - 3) gate 입력에 따른 and, or, xor 연산 수행 -> op( )

    : 각 gate 객체에서 멤버변수 (x, y) 사용하여 연산 수행 후 결과를 z 에 저장

    : 입력 / 결과 출력

    : main 에서 호출, public

Report P8_1

```
void p8_1() { // main
    ANDGate gate_and;
    ORGate gate_or;
    XORGate gate_xor;

    gate_and.inputSet(true, false);
    gate_and.op();
    gate_and.inputSet(true, true);
    gate_and.op();
    gate_or.inputSet(true, false);
    gate_or.op();
    gate_or.inputSet(true, true);
    gate_or.op();
    gate_xor.inputSet(true, false);
    gate_xor.op();
    gate_xor.inputSet(true, true);
    gate_xor.op();
}
```

출력

AND input : 1 0 -> 0
AND input : 1 1 -> 1
OR input : 1 0 -> 1
OR input : 1 1 -> 1
XOR input : 1 0 -> 1
XOR input : 1 1 -> 0

Report P8_2

- 상속 사용
- 사용하는 객체들과 속성, 동작
  - Point, Line, Circle => 이들을 class로 구성
- class 간 관계 ← 클래스는 4개 사용
  - Point 클래스 ← has-a 관계 → Line, Circle 클래스
  - Line, Circle 클래스 공통점 모아 부모 클래스(Shape 클래스) 구성
    - Shape 클래스 ← is-a 관계 → Line, Circle 클래스
- 멤버변수는 모두 private

Report P8_2

- Line, Circle => 이들을 class로 구성
  - 속성 : 시작점, 끝점 좌표 저장
    - 원은 사각형의 내접원으로 그릴 수 있음.
    - (직선은 시작점/끝점), (원은 좌상단점/우하단점 좌표 의미)
  - 동작 1 → 생성자 : 시작점, 끝점 좌표를 인자로 받아 속성에 저장
  - 동작 2 : Draw( )
    : Line class 는 속성(시작점, 끝점) 출력, “직선 그린다” 출력
    : Circle class 는 속성(시작점, 끝점) 출력, “원 그린다” 출력

Report P8_2

```
void p8_2( ) {      // main
        Circle a(1, 1, 5, 5);  // 시작점, 끝점 좌표
        Line b(5, 5, 9, 9);    // 시작점, 끝점 좌표

        a.Draw();// "원 그린다" , 시작점, 끝점 좌표 출력

        b.Draw();// "직선 그린다", 시작점, 끝점 좌표 출력
}
```

상속에서 복사 생성자

- 생성자는 항상 자식/부모 모두 작성 → 멤버 초기화
  - 자식 생성자에서 부모 생성자 호출하는 코드 작성
- 복사 생성자는
  - 멤버에 포인터 없으면 복사 생성자는 부모/자식 모두 작성하지 않아도 됨
    - default 복사 생성자 만들어져 자동 수행
  - 멤버에 포인터 있으면 복사 생성자는 부모/자식 모두 작성
    - 자식 복사생성자에서 부모 복사생성자 호출하는 코드 작성
    - cf) 자식 생성자에서 부모 생성자 호출과 동일 방식으로 작성

```
class Parent {      // 복사 생성자 모두 있는 경우
public:
   int a;
   Parent(int x = 10) : a(x) {
      cout << "Parent 생성자" << endl;        // ... (1)
   }
   Parent(const Parent& tt) {
      a = tt.a;
      cout << "Parent 복사 생성자" << endl; // ... (2)
   }
};
class Child : public Parent {
public:
   int c;
   Child(int a, int x = 30)
      : Parent(a), c(x)                          // ...(3)
   {     cout << "Child 생성자" << endl; } // ...(4)

   Child(const Child& tt) : Parent(tt)       // ..(5)
   {  c = tt.c;
      cout << "Child 복사생성자" << endl; // ...(6)
   }
};
```

```
int main() {
   Parent aa(1);
   Parent bb(aa);  // Parent bb = aa 와 동일

   Child cc(3, 4);
   Child dd(cc);
   return 0;
}
```

aa : (1) → Parent 생성자
bb : (2) → Parent 복사 생성자

cc : (3) → (1) → (4)
Parent 생성자
Child 생성자

dd : (5) → (2) → (6)
Parent 복사 생성자
Candy 복사 생성자

aa(1) b(1)
cc(3, 4) dd(3, 4)

```
// 복사 생성자 모두 없는 경우(없어도 무관)
class Parent {
public:
   int a;
   Parent(int x = 10) : a(x) {
      cout << "Parent 생성자" << endl;        // ... (1)
   }
};

class Child : public Parent {
public:
   int c;
   Child(int a, int x = 30)
      : Parent(a), c(x)                          // ...(3)
   {
      cout << "Child 생성자" << endl;    // ...(4)
   }
};
```

- 디폴트 복사 생성자 : 모든 멤버변수의 값들을 복사

```
int main() {
   Parent aa(1);
   Parent bb(aa);
   Child cc(3, 4);
   Child dd(cc);
   return 0;
}
```

aa : (1) → Parent 생성자
bb :

cc : (3) → (1) → (4)
Parent 생성자
Child 생성자

dd :

aa(1) b(1)
cc(3, 4) dd(3, 4)

```
class Parent {
public:
   int a;
   Parent(int x = 10) : a(x) {}
   Parent(const Parent& x) : a(x.a) { }
};
class Child : public Parent {
public:
   int c;
   Child(int a, int x = 30): Parent(a), c(x) { }
   Child(const Child& tt) : c(tt.c) { }
};
void main() {
   Child cc(3, 4), dd(cc);
   cout << cc.a << " " << cc.c << endl;
   cout << dd.a << " " << dd.c << endl;
}
```

3 4

10 4

- 자식 클래스에서 부모 클래스의 복사 생성자를 명시적으로 호출해주지 않을 경우(옆 코드와 동일 결과)

→ 부모 클래스의 복사 생성자가 호출되지 않고 디폴트 생성자가 호출

```
class Parent {
public:
   int a;
   Parent(int x = 10) : a(x) { }
   Parent(const Parent& x) : a(x.a) { }
};
class Child : public Parent {
public:
   int c;
   Child(int a, int x = 30) : Parent(a), c(x) { }
   Child(const Child& tt) : c(tt.c), Parent( ) { }
};
void main() {
   Child cc(3, 4), dd(cc);
   cout << cc.a << " " << cc.c << endl;
   cout << dd.a << " " << dd.c << endl;
}
```

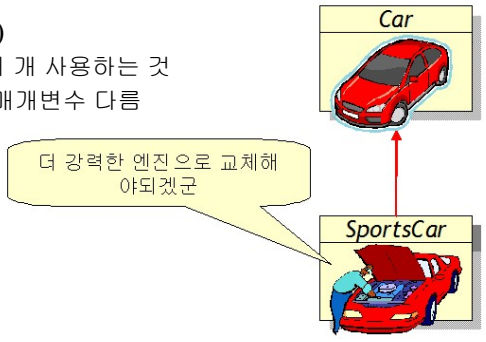
3 4

10 4

- 부모의 디폴트 생성자 호출

재정의 - 중요

- 재정의(overriding): 필요에 따라 상속된 부모 멤버 함수를 자식 클래스가 다시 정의하여 사용하는 것 → 자식 클래스에서 부모의 함수 내용을 수정하여 사용
  - 함수 이름, 매개변수, 반환형 동일
- 참고) 중복정의(overloading)
  - 같은 이름의 함수를 여러 개 사용하는 것
  - 이름만 동일, 함수마다 매개변수 다름



예제

```
class Car {
public:
  int getHP() {
    // 100마력반환
    return 100;
  }
};

class SportsCar : public Car {
public:
  int getHP() {
        // 300마력반환
        return 300;
  }
};
```

재정의

```
int main()
{
        SportsCar sc;
        cout << "마력: " << sc.getHP() << endl;
        return 0;
}
```

```
마력: 300
계속하려면 아무 키나 누르십시오 . . .
```

sc 는 SportsCar 클래스 객체

→ sc.getHP( ) 하면 SportsCar 클래스 멤버함수 수행

→ 본인(자식)의 함수들을 먼저 찾고 없으면 부모의 함수를 찾음.

→ 부모의 함수를 무시(overriding)

재정의의 조건

- 부모 클래스의 멤버 함수와 동일한 시그니처(원형)를 가져야 한다.
- 즉 멤버 함수의 이름, 반환형, 매개 변수의 개수와 매개 변수의 데이터 타입이 일치하여야 한다.

```
class Animal {
        void makeSound()
        {
        }
};
```

재정의가 아님

```
class Dog : public Animal {
        int makeSound()
        {
        }
};
```

예제

```
class Car {
public:
  int getHP() {
    // 100마력반환
    return 100;
  }
};

class SportsCar : public Car {
public:

  float getHP() {
     // 300마력반환
     return 300.0;
  }
};
```

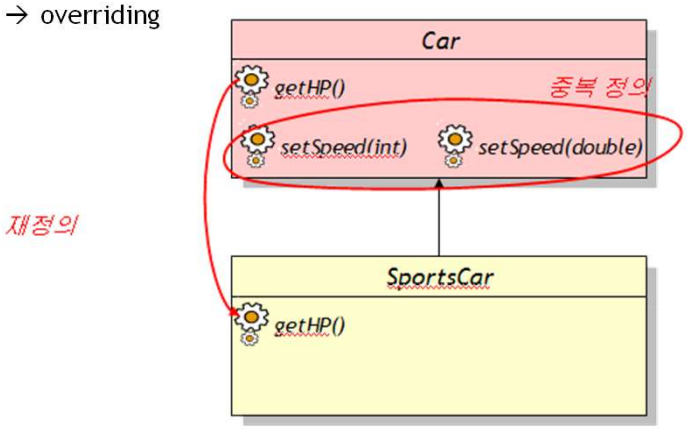
```
int main()
{
        SportsCar sc;
        cout << "마력: " << sc.getHP() << endl;
        return 0;
}
```

```
마력: 300.0
계속하려면 아무 키나 누르십시오 . . .
```

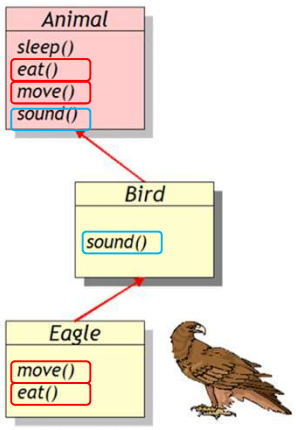
- 재정의 아님 → 반환형 틀림
- 결과는 앞과 동일하지만 재정의 아님
- 함수 호출시 본인(자식)의 함수들을 먼저 찾고 없으면 부모의 함수를 찾음.

재정의와 중복 정의

- 중복 정의: 한 클래스에서 같은 이름의 멤버 함수를 여러 개 정의하는 것 → overloading
- 재정의: 부모 클래스에 있던 상속받은 멤버 함수를 다시 정의하는 것 → overriding



재정의된(overriding) 멤버 함수의 호출 순서




```
Eagle e;

e.sleep();  // Animal의sleep() 호출

e.eat();    // Eagle의eat() 호출

e.sound(); // Bird의sound() 호출
```

➔ 상속계층구조에서
해당 객체의 클래스부터 멤버함수를 호출
➔ 자기 것 있으면 자기 것 호출,
없으면 부모로 차례로 올라가며 찾음

```
Bird g;
            // Animal의sleep() 호출
g.sleep();
            // Animal 의eat() 호출
g.eat();
            // Bird의sound() 호출
g.sound();
```

자식 클래스에서 부모 클래스의 멤버 호출

```
class ParentClass {
public:
   void print() {
      cout << "부모클래스의 print() 멤버함수" << endl;
   }
};

class ChildClass : public ParentClass {
     int data;
public:

   void print() { //멤버함수 오버라이딩
      ParentClass::print();
      cout << "자식클래스의print() 멤버함수" << endl;
   }
};
int main()
{
   ChildClass obj;
   obj.print();
            return 0;
}
```

```
부모 클래스의 print() 멤버 함수
자식 클래스의 print() 멤버 함수
계속하려면 아무 키나 누르십시오 . . .
```

부모 클래스의 함수(print( )) 호출!
이름이 같기 때문에 부모클래스 이름을 사용
이름이 다르면 클래스 이름 없어도 가능

자식에서 부모의 함수를
보완하여 사용(일반적)

멤버 변수 재정의(하지 말 것)

```
class Car {

public:
    int speed;
    int gear;
    string color;
    Car(): speed(0), gear(1), color("white") { }
    void setSpeed(int s){ speed = s; }
    int getSpeed(){ return speed; }
};

class SportsCar : public Car {
public:
    int speed;
    int gear;
    string color;
    SportsCar(): speed(100), gear(3), color("blue") { }
};
```

가능하지만 혼란을 일으킴!

```
int main()
{
   SportsCar sc;
// 자식 클래스의 speed 변수
   cout << "스피드: " << sc.speed << endl;
// 부모 클래스의 speed 변수
   cout << "스피드: " << sc.Car::speed << endl;
// 부모 클래스의 speed 반환
   cout << "스피드: " << sc.getSpeed() << endl;
   return 0;
}
```

```
스피드: 100
스피드: 0
스피드: 0
계속하려면 아무 키나 누르십시오 . . .
```

상속의 3가지 유형(대부분 public 으로 상속)

```
class 자식클래스 : public 부모클래스
{
        ...
}
```

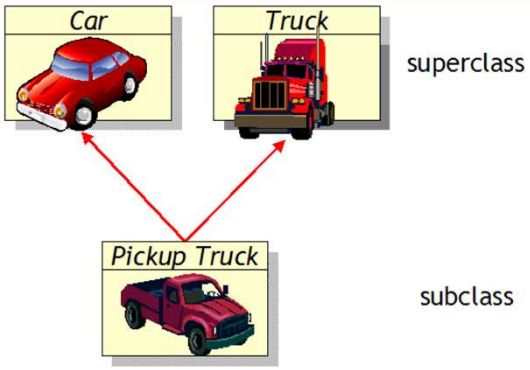
public으로 상속

자식 클래스에서

	public으로 상속	protected로 상속	private로 상속
부모 클래스의 public 멤버	->public	->protected	->private
부모 클래스의 protected 멤버	->protected	->protected	->private
부모 클래스의 private 멤버	접근 안됨	접근 안됨	접근 안됨

다중 상속 - 잘 사용 안함.

```
class Sub : public Sup1, public Sup2
{
        ...// 추가된 멤버
        ...// 재정의된 멤버
}
```



예제

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

class PassangerCar { // 승용차
public:
        int seats; // 정원
        void set_seats(int n){ seats = n; }
};

class Truck {
public:
        int payload; // 적재하중
        void set_payload(int load){ payload = load; }
};

class Pickup : public PassangerCar, public Truck {
public:
        int tow_capability; // 견인능력
        void set_tow(int capa){ tow_capability = capa; }
};
```

```
int main()
{
        Pickup my_car;
        my_car.set_seats(4);
        my_car.set_payload(10000);
        my_car.set_tow(30000);
        return 0;
}
```

다중 상속의 문제점

```
class SuperA
{
public:
  int x;
  void sub(){
          cout << "SuperA의 sub()" << endl;
  }
};
class SuperB
{
public:
  int x;
  void sub(){
          cout << "SuperB의 sub()" << endl;
  }
};
```

```
class Sub : public  SuperA, public SuperB
{
};

int main(){
  Sub obj;


  obj.x = 10;          // error
// obj.x는 어떤 부모클래스의 x를 참조하는가?

  return 0;
}
```

obj.SuperA::x=10;
obj 멤버 x 인데 SuperA 의 것

```
1>.\multi_inheri.cpp(27) : error C2385: 'x' 액세스가 모호합니다.
1>        기본 'SuperA'의 'x'일 수 있습니다.
1>        또는 기본 'SuperB'의 'x'일 수 있습니다.
```

열거형

- 열거형 : 프로그램의 가독성을 높인다.
  - 열거형은 예약어 enum을 사용한다.
  - 열거형은 숫자, 색깔과 같이 일정한 패턴이 있는 상수들을 집합으로 갖도록 선언.
  - 열거형으로 선언된 변수는 집합에 포함되는 상숫값 만을 갖게 된다.

    enum type_이름 {이 type의 변수가 이 가질 수 있는 값들}

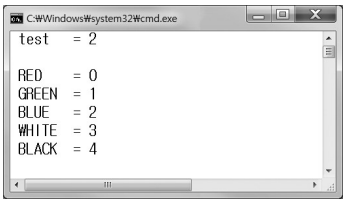
    enum COLOR { RED, GREEN, BLUE, WHITE, BLACK };

  - 열거형 COLOR 는 새로운 자료형이다.
  - { } 안에 기술된 구성원을 '열거 상수'라고 한다.
  - 처음에 기술된 열거 상수의 값은 0 이고 값이 차례로 1씩 증가한다.

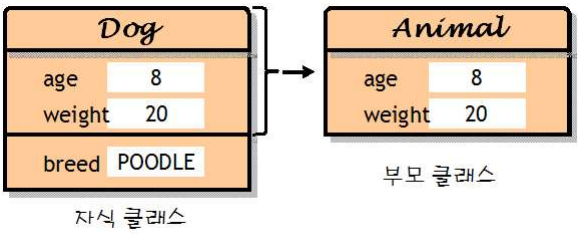
```
enum COLOR a;
a = RED;                    // a = COLOR.RED;
```

예제 8-12. 열거형 사용하기(08_12.cpp)

```
01 #include <iostream>
02 using namespace std;
03
04 enum COLOR { RED, GREEN, BLUE, WHITE, BLACK };
05 void main()
06 {
07 enum COLOR test;      // test는 COLOR 안의 값만 가능

09 test = BLUE;
10 cout<<" test = "<<test <<"₩n₩n"; // test는 정수값 2로 정의되어 있다.
11
12 cout<<" RED = "<< RED<<"₩n";
13 cout<<" GREEN = "<< GREEN<<"₩n";
14 cout<<" BLUE = "<< BLUE<<" ₩n";
15 cout<<" WHITE = "<<WHITE <<" ₩n";
16 cout<<" BLACK = "<< BLACK<<" ₩n";
17 }
```



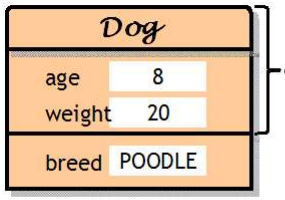
예제 1(책 내용... 생략)



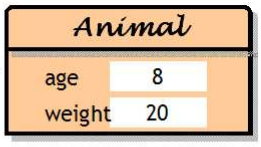

```
int main()
{
        Dog dog;

        dog.eat();
        dog.sleep();
        dog.speak();
        dog.wag();
        return 0;
}
```

클래스 1 종류이지만
상속으로 작성(불필요)

예제 1(책 내용 수정) - 객체 1 종류이지만 상속으로 작성



자식 클래스



부모 클래스

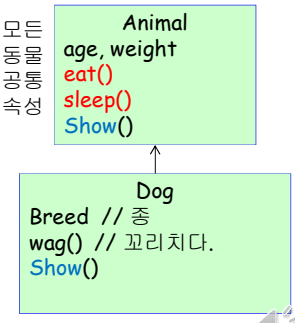
age, weight 는 부모의 private → 자식에서 접근 못함
→ 부모만이 접근 가능

```
int main()
{
        Dog dog(8, 20, "Poodle");

        // 나이, 체중, 종 출력
        dog.Show();

        dog.eat();  // 모든 동물 공통
        dog.sleep();
        dog.wag();
        return 0;
}
```

출력

```
8, 20, Poodle

냠냠
쿨쿨
살랑~
```

모든 동물 공통 속성

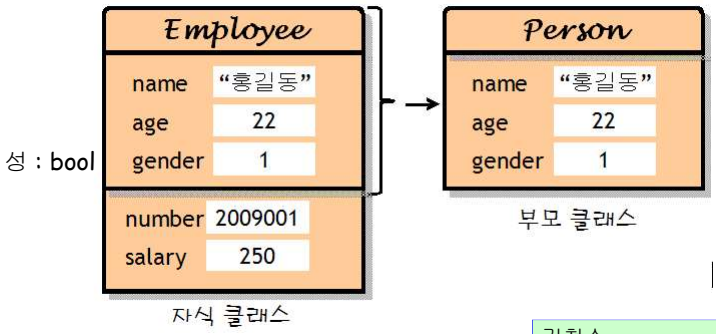


예제

```
class Animal {
private:
  int age, weight;
public:
  Animal(int a, int w) : age(a), weight(w) {}
  void eat() {
          cout << "냠냠" << endl;
  }
  void sleep() {
          cout << "쿨쿨" << endl;
  }
  void Show() {
   cout << "나이:" << age << " 체중 :" << weight;
  }
};
```

```
class Dog : public Animal {
private:
  string Breed;  // 품종
public:
  Dog(int a, int w, string b) : Animal(a, w), Breed(b) {}

  void wag() {
          cout << "살살~~" << endl;
  }
  void Show() {  // 나이, 체중, 품종을 출력
          Animal::Show();
          cout << "품종 : " << Breed << endl;
  }
};
```

예제 #2(생략)




```
int main() {
  Employee e("김철수", 26, true, 2010001, 2800);
  e.display();
  return 0;
}
```

```
김철수
26
1
2010001
2800
계속하려면 아무 키나 누르십시오 . . .
```

예제 #2

모든 사람 공통 속성

```
Person
Name
Age
Gender // 성
...
```

bool gender
// true : 남
// false : 여

```
Person
Name
Age
Gender // 성
...
```

```
Employee
Name
Age
Gender  // 성
Number  // 사번
Salary  // 월급
...
Display(...)  // 이름, 나이, 성, 월급 출력
```

```
Employee
Number  // 사번
Salary  // 월급
...
Display(...)  // 이름, 나이, 성, 월급 출력
```

```
int main() {
  Employee e("김철수", 26, true, 2010001, 2800);
  e.display();
  return 0;
}
```

예 제

```
class Person {
        string name;
        int age;
        bool gender;
public:
        Person(string n="", int a=0, bool g=true): name(n), age(a), gender(g) { }

        string getName() const { return name; }          // 자식이 호출

        void setName(string s) { name = s; }             // 사용 안함
        void setAge (int a) { age = a; }                  // 사용 안함
        int getAge() const { return age; }

        void setGender (bool g) { gender = g; }           // 사용 안함
        bool getGender() const { return gender; }
};
```

예제 1 에서는 부모 클래스 **private** 멤버 출력하는 함수 사용
예제 2 에서는 부모 클래스 **private** 멤버 반환하는 함수 사용

예 제

```
class Employee : public Person {
        int number, salary;
public:
        Employee(string n="", int a=0, bool g=true, int num=0, int s=0)
                : Person(n,  a, g), number(num), salary(s) { }
        void display() const;

        void setNumber (int n) { number = n; }           // 사용 안함
        int getNumber() const { return number; }

        void setSalary (int s) { salary = s; }           // 사용 안함
        int getSalary() const { return salary; }
};

void Employee::display() const {
        cout << this->getName() << endl;
        cout << this->getAge() << endl;
        cout << this->getGender() << endl;
        cout << this->getNumber() << endl;
        cout << this->getSalary() << endl;
}
```