제11장 상속

1. 상속의 개념을 이해한다.
2. 상속을 이용하여 자식 클래스를 작성할 수 있다.
3. 상속과 접근 지정자와의 관계를 이해한다.
4. 상속시 생성자와 소멸자가 호출되는 순서를 이해한다.

상속이란?

상속은 객체 단위 코드를 재사용하는 방법이다.

- 상속을 통해 코드를 재사용하거나 확장/개선할 수 있다.
- 상속을 배우는 순간부터 클래스는 최소 2개 이상이 되고 이들 간의 관계를 이해하는 것이 매우 중요하다.
- 상속은 is-a, has-a 관계이다.
- 파생 클래스의 인스턴스가 생성될 때 기본 클래스 생성자도 호출된다.
- 파생 클래스는 기본 클래스의 멤버에 접근할 수 있다. 단, private 접근제어 지시자로 선언된 멤버는 접근할 수 없다.
- 파생 클래스 인스턴스를 통해 기본 클래스 메서드도 호출할 수 있다.

이번 장에서 만들어 볼 프로그램

```
class Circle {
        int x, y;
        int radius;
        ...
}
class Rect {
        int x, y;
        int width, height;
        ...
}
```

중복

11.2 상속의 개요

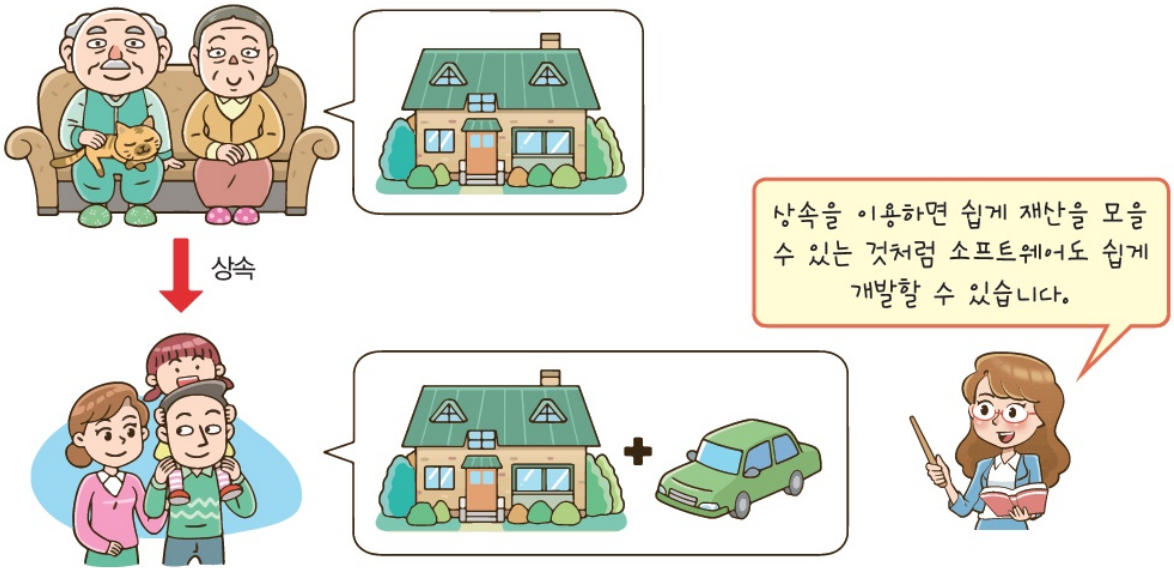


그림 11.1 상속의 개념

상속의 정의

문법 11.3 다중 상속

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

- 부모 클래스의 멤버 변수와 멤버 함수가 자식 클래스로 상속됨.
- 자식 클래스는 필요하면 자신만의 변수와 멤버 함수를 추가 시킬 수도 있고 부모 클래스에 이미 존재하는 멤버 함수를 새롭게 정의하여 사용할 수도 있다.

예제

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

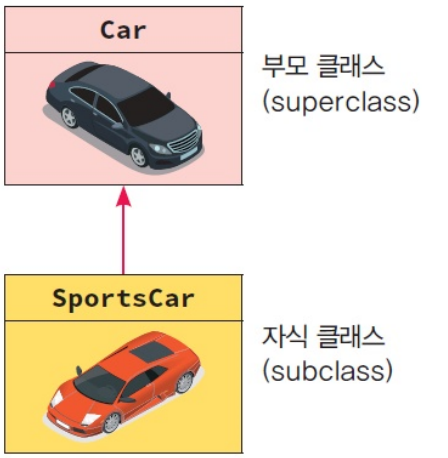


그림 11.2 자동차에서의 상속의 예

자식클래스와 부모클래스

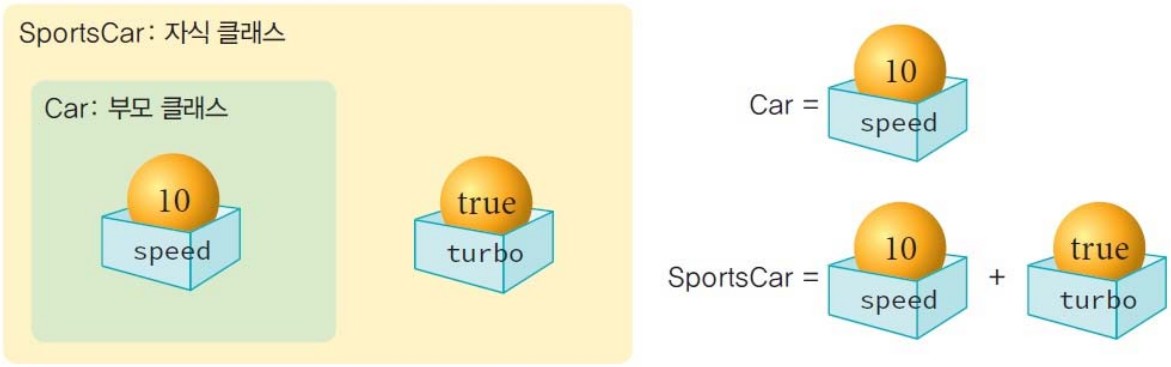
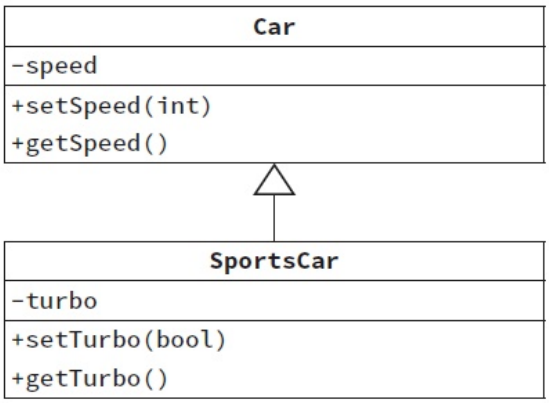


그림 11.3 자식 클래스는 부모 클래스를 포함한다.

부모 클래스	자식 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)

- 상속의 강점은 부모 클래스로부터 상속된 특징들을 자식 클래스에서 추가, 교체, 상세화시킬 수 있는 능력으로부터 나온다

예제



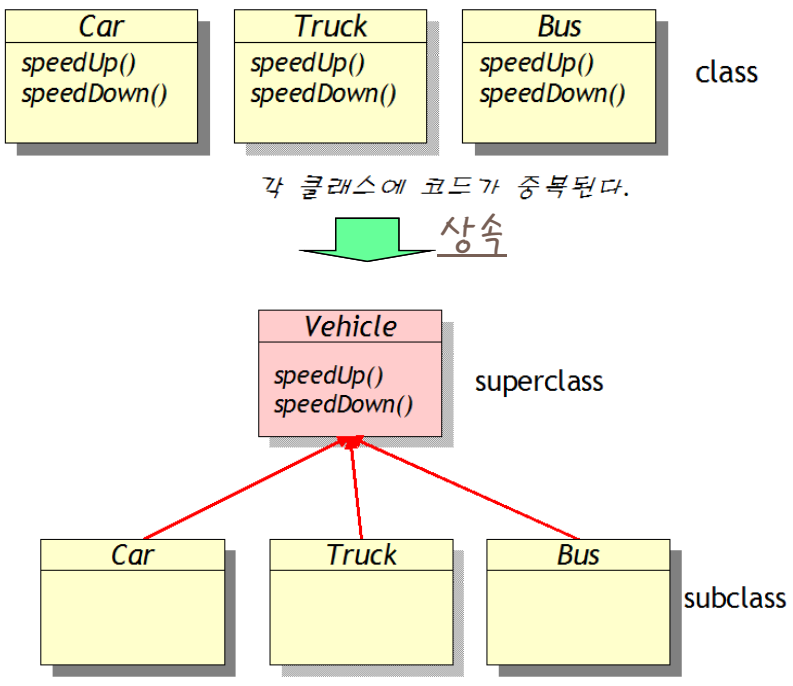
예제

```
#include <iostream>
#include <string>
using namespace std;
class Car {
        int speed; // 속도
public:
        void setSpeed(int s)        {       speed = s;      }
        int getSpeed()              {       return speed;   }
};
// Car 클래스를 상속받아서 다음과 같이 SportsCar 클래스를 작성한다.
class SportsCar : public Car {
        bool turbo;
public:
        void setTurbo(bool newValue) {  turbo = newValue;       }
        bool getTurbo()   {             return turbo;           }
};
```

예제

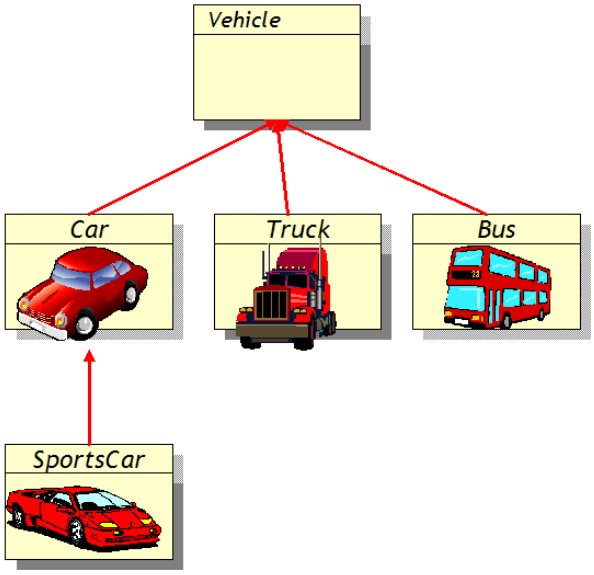
```
int main()
{
        SportsCar  c;

        c.setSpeed(60);          // 부모 클래스 함수 호출
        c.setTurbo(true);        // 자식 클래스 함수 호출
        c.setSpeed(100);
        c.setTurbo(false);
        return 0;
}
```


11.3 상속은 왜 필요한가?



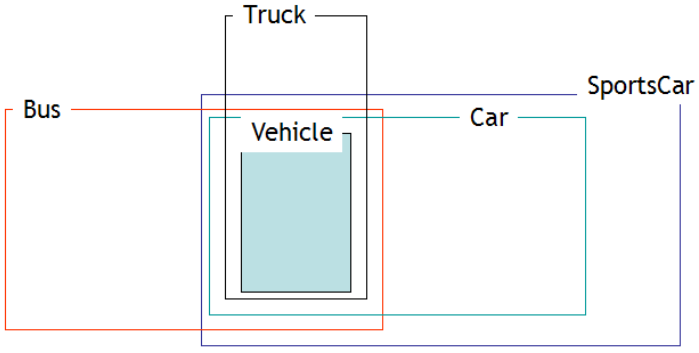
상속 계층도

- 상속은 여러 단계로 이루어질 수 있다.



상속 계층도

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



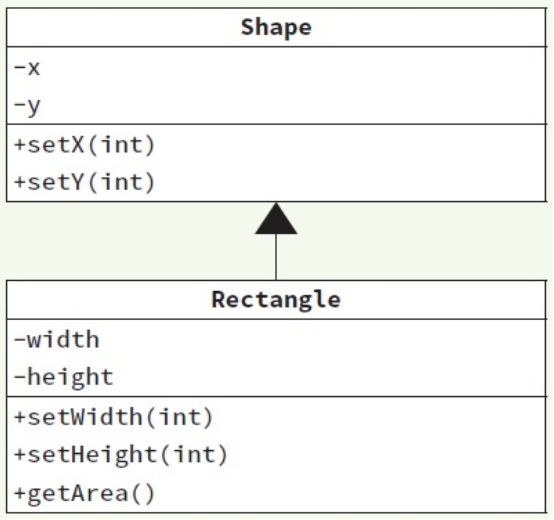
상속은 is-a 관계

- 상속은 is-a 관계
  - 자동차는 탈것이다. (Car *is a* Vehicle).
  - 사자, 개, 고양이는 동물이다.
- has-a(포함) 관계는 상속으로 모델링을 하면 안 된다.
  - 도서관은 책을 가지고 있다(Library *has a* book).
  - 거실은 소파를 가지고 있다.

중간 점검

1. 상속은 왜 필요한가?
2. 사자, 호랑이, 고양이를 상속 계층 구조를 이용하여 표현하여 보자.

Lab: 도형과 사각형



예제

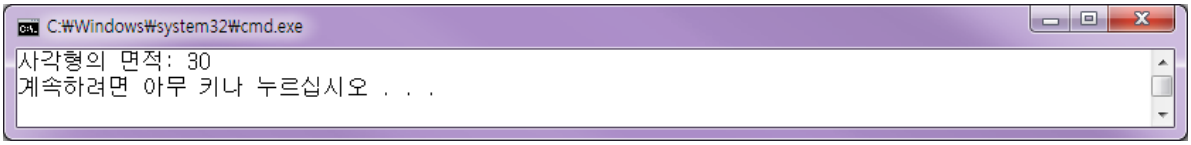
```
#include <iostream>
using namespace std;

class Shape {
        int x, y;
public:
        void setX(int xval) {
                x = xval;
        }
        void setY(int yval) {
                y = yval;
        }
};
```

예제

```
class Rectangle : public Shape {
        int width, height;
public:
        void setWidth(int w) {
                width = w;
        }
        void setHeight(int h) {
                height = h;
        }
        int getArea() {
                return (width * height);
        }
};
```

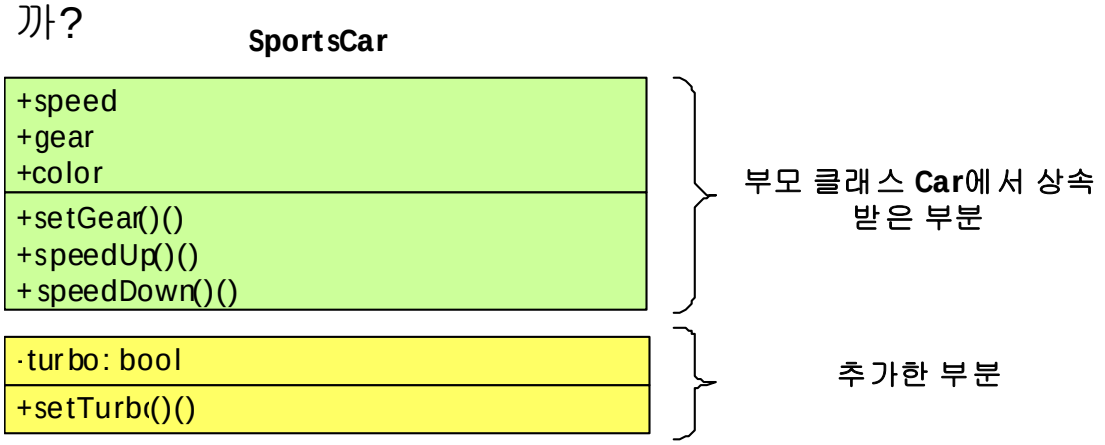

예제

```
int main() {
        Rectangle r;

        r.setWidth(5);
        r.setHeight(6);

        cout << "사각형의 면적: " << r.getArea() << endl;

        return 0;
}
```



11.4 상속에서의 생성자와 소멸자

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



상속에서의 생성자와 소멸자

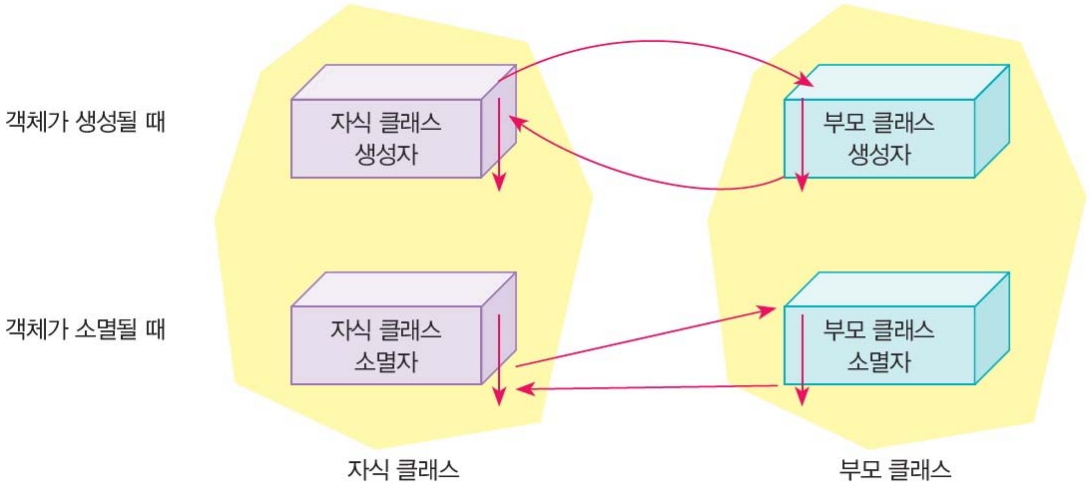


그림 11.8 상속에서 생성자와 소멸자의 호출

예제

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


예제

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

부모 클래스의 생성자를 지정하는 방법

문법 11.2 자식 클래스의 생성자 호출

```
자식클래스의 생성자() : 부모클래스의 생성자()
{
}
```


예제

```
Rectangle(int x=0, int y=0, int w=0, int h=0) : Shape(x, y)
{
                    width = w;
                    height = h;
}
```

```
Rectangle(int x=0, int y=0, int w=0, int h=0) : Shape(x, y), width(w), height(h)
{

}
```

예제

```
class Shape {
        int x, y;
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;
        }
};
```


예제

```
class Rectangle : public Shape {
        int width, height;
public:
        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;
        }
        ~Rectangle() {
                cout << "Rectangle 소멸자()" << endl;
        }
};
```

예제

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

상속의 방법과 그 결과 -> UnivStudentInheri.cpp실행

```
class Person
{
private:
    int age;          // 나이
    char name[50];    // 이름
public:
    Person(int myage, char * myname) : age(myage)
    {
        strcpy(name, myname);
    }
    void WhatYourName() const
    {
        cout<<"My name is "<<name<<endl;
    }
    void HowOldAreYou() const
    {
        cout<<"I'm "<<age<<" years old"<<endl;
    }
};
```

```
class UnivStudent : public Person
{
private:
    char major[50];   // 전공과목
public:
    UnivStudent(char * myname, int myage, char * mymajor)
        : Person(myage, myname)
    {
        strcpy(major, mymajor);
    }
    void WhoAreYou() const
    {
        WhatYourName();
        HowOldAreYou();
        cout<<"My major is "<<major<<endl<<endl;
    }
};
```

Person 클래스를 public 상속함

Person 클래스의 멤버

Person	↔	UnivStudent
상위 클래스	↔	하위 클래스
기초(base) 클래스	↔	유도(derived) 클래스
슈퍼(super) 클래스	↔	서브(sub) 클래스
부모 클래스	↔	자식 클래스


```
int main(void)
{
    UnivStudent ustd1("Lee", 22, "Computer eng.");
    ustd1.WhoAreYou();

    UnivStudent ustd2("Yoon", 21, "Electronic eng.");
    ustd2.WhoAreYou();
    return 0;
```

상속받은 클래스의 생성자 정의

```
UnivStudent(char * myname, int myage, char * mymajor)
    : Person(myage, myname)
{
    strcpy(major, mymajor);
}
```

이니셜라이저를 통해서 유도 클래스는 기초 클래스의 생성자를 명시적으로 호출해야 한다.
유도 클래스의 생성자는 기초 클래스의 멤버를 초기화 할 의무를 갖는다. 단! 기초 클래스의 생성자를 명시적으로 호출해서 초기화해야 한다.

```
int main(void)
{
    UnivStudent ustd1("Lee", 22, "Computer eng.");
    ustd1.WhoAreYou();

    UnivStudent ustd2("Yoon", 21, "Electronic eng.");
    ustd2.WhoAreYou();
    return 0;
};
```

때문에 유도 클래스 UnivStudent는 기초 클래스의 생성자 호출을 위한 인자까지 함께 전달받아야 한다.

private 멤버는 유도 클래스에서도 접근이 불가능하므로, 생성자의 호출을 통해서 기초 클래스의 멤버를 초기화해야 한다.

유도 클래스의 객체생성 과정 -> DerivCreOrder.cpp실행

```
class SoBase
{
private:
    int baseNum;
public:
    SoBase() : baseNum(20)
    {
        cout<<"SoBase()"<<endl;
    }
    SoBase(int n) : baseNum(n)
    {
        cout<<"SoBase(int n)"<<endl;
    }
    void ShowBaseData()
    {
        cout<<baseNum<<endl;
    }
};
```

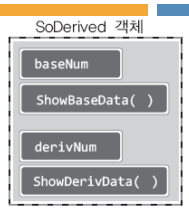
```
class SoDerived : public SoBase
{
private:
    int derivNum;
public:
    SoDerived() : derivNum(30)
    {
        cout<<"SoDerived()"<<endl;
    }
    SoDerived(int n) : derivNum(n)
    {
        cout<<"SoDerived(int n)"<<endl;
    }
    SoDerived(int n1, int n2) : SoBase(n1), derivNum(n2)
    {
        cout<<"SoDerived(int n1, int n2)"<<endl;
    }
    void ShowDerivData()
    {
        ShowBaseData();
        cout<<derivNum<<endl;
    }
};
```

```
int main(void)
{
    cout<<"case1..... "<<endl;
    SoDerived dr1;
    dr1.ShowDerivData();
    cout<<"--------------------"<<endl;
    cout<<"case2..... "<<endl;
    SoDerived dr2(12);
    dr2.ShowDerivData();
    cout<<"--------------------"<<endl;
    cout<<"case3..... "<<endl;
    SoDerived dr3(23, 24);
    dr3.ShowDerivData();
    return 0;
};
```

```
case1.....
SoBase()
SoDerived()
20
30
--------------------
case2.....
SoBase()
SoDerived(int n)
20
12
--------------------
case3.....
SoBase(int n)
SoDerived(int n1, int n2)
23
24
```

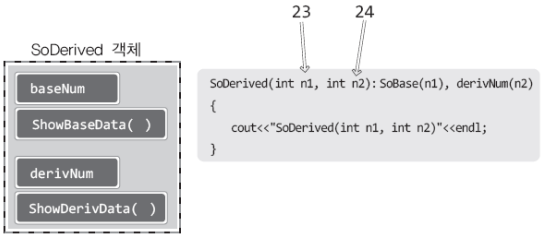
실행결과

유도 클래스의 객체생성 과정 case1

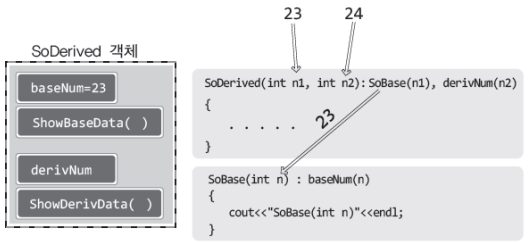


순서 1. 메모리 공간의 할당

SoDerived dr3(23, 24);



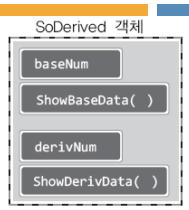
순서 2. 유도 클래스의 생성자 호출



순서 3. 기초 클래스의 생성자 호출 및 실행

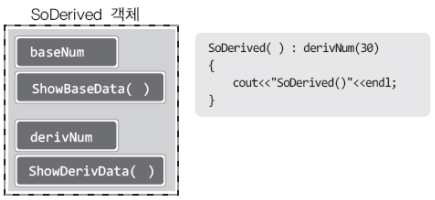
순서 4. 유도 클래스의 생성자 실행

유도 클래스의 객체생성 과정 case2

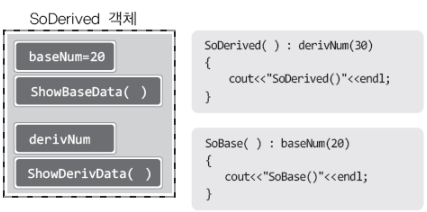


순서 1. 메모리 공간의 할당

SoDerived dr1



순서 2. 유도 클래스의 void 생성자 호출



순서 3. 이니셜라이저를 통한 기초 클래스의 생성자 호출이 명시적으로 정의되어 있지 않으므로 void 생성자 호출

순서 4. 유도 클래스의 실행

유도 클래스 객체의 소멸과정 -> DerivDestOrder.cpp실행

```
class SoBase
{
private:
    int baseNum;
public:
    SoBase(int n) : baseNum(n)
    {
        cout<<"SoBase() : "<<baseNum<<endl;
    }
    ~SoBase()
    {
        cout<<"~SoBase() : "<<baseNum<<endl;
    }
};

class SoDerived : public SoBase
{
private:
    int derivNum;
public:
    SoDerived(int n) : SoBase(n), derivNum(n)
    {
        cout<<"SoDerived() : "<<derivNum<<endl;
    }
    ~SoDerived()
    {
        cout<<"~SoDerived() : "<<derivNum<<endl;
    }
};
```

```
int main(void)
{
SoDerived drv1(15);
SoDerived drv2(27);
return 0;
};
```

```
SoBase() : 15
SoDerived() : 15
SoBase() : 27
SoDerived() : 27
~SoDerived() : 27
~SoBase() : 27
~SoDerived() : 15
~SoBase() : 15
```

실행결과

유도 클래스의 소멸자가 실행된 이후에 기초 클래스의 소멸자가 실행된다.

스택에 생성된 객체의 소멸순서는 생성순서와 반대이다.

유도 클래스 정의 모델 DestModel.cpp 실행

```
class Person
{
private:
    char * name;
public:
    Person(char * myname)
    {
        name=new char[strlen(myname)+1];
        strcpy(name, myname);
    }
    ~Person()
    {
        delete []name;
    }
    void WhatYourName() const
    {
        cout<<"My name is "<<name<<endl;
    }
};
```

```
class UnivStudent : public Person
{
private:
    char * major;
public:
    UnivStudent(char * myname, char * mymajor)
        : Person(myname)
    {
        major=new char[strlen(mymajor)+1];
        strcpy(major, mymajor);
    }
    ~UnivStudent()
    {
        delete []major;
    }
    void WhoAreYou() const
    {
        WhatYourName();
        cout<<"My major is "<<major<<endl<<endl;
    }
};
```

기초 클래스의 멤버 대상의 동적 할당은 기초 클래스의 생성자를 통해서, 소멸 역시 기초 클래스의 소멸자를 통해서

실습 [상속과 생성자의 호출]

- 다음 클래스에 적절한 생성자, 소멸자를 삽입하고 이의 확인을 위한 main 함수를 정의하여 실행하라.

```
class Car {// 기본 연료 자동차
        private: int gasolineGauge;
        public:
        int GetGasGauge()       { return gasolineGauge;   }
};
class HybridCar : public Car {// 하이브리드 자동차
        private: int electricGauge;
        public:
        int GetElecGauge()      { return electricGauge;   }
};
class HybridWaterCar : public HybridCar {// 하이브리드 워터카
        private: int waterGauge;
        public:  void ShowCurrentGauge()
        {        cout<<"잔여 가솔린: "<<GetGasGauge()<<endl;
                 cout<<"잔여 전기량: "<<GetElecGauge()<<endl;
                 cout<<"잔여 워터량: "<<waterGauge<<endl;
        } };
```


실습 [상속과 생성자의 호출]

- 다음 두 클래스에 적절한 생성자와 소멸자를 정의하고 이의 확인을 위한 main 함수를 정의하여 실행하라.

```
class MyFriendInfo {
private: char * name;
         int age;
public:  void ShowMyFriendInfo()
         {       cout<<"이름: "<<name<<endl;
                 cout<<"나이: "<<age<<endl;}
};

class MyFriendDetailInfo : public MyFriendInfo {
private: char * addr;
         char * phone;
public:  void ShowMyFriendDetailInfo()
         {       ShowMyFriendInfo();
                 cout<<"주소: "<<addr<<endl;
                 cout<<"전화: "<<phone<<endl<<endl;}
};
```

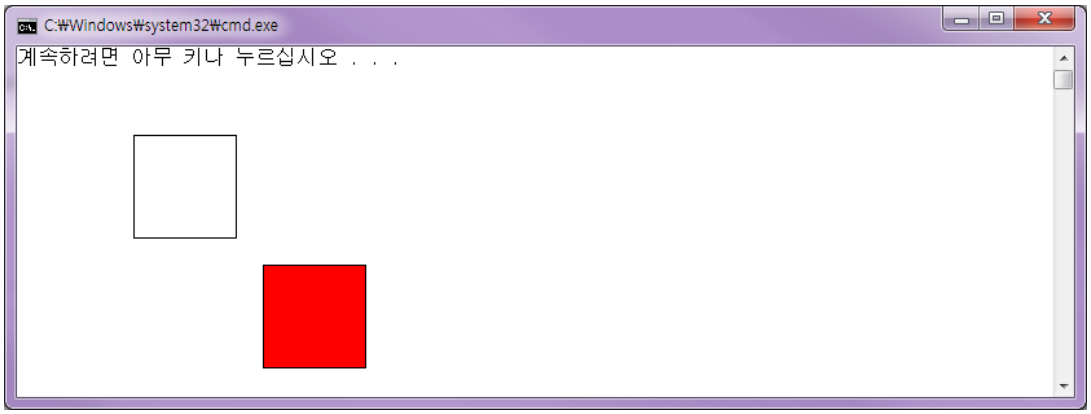
상속 생성자

The using statement brings into scope all constructors from the base class except those that have an identical signature to constructors in the derived class.

```
class CMyDataEx : public CMyData
{
    public:
    using CMyData::CMyData;
    // 단, 파라미터 없는 버전은 파라미터 있는 버전이 있을 때
    // using으로 부모것을 쓸수 없고 직접 만들어 줘야 함
    ...
};
```


Lab: 컬러 사각형

- 사각형을 Rect 클래스로 나타내자. 이 클래스를 상속받아서 컬러 사각형 ColoredRect을 정의해보자. ColoredRect 클래스를 이용하여 화면에 다음과 같은 색깔있는 사각형을 그려보자.



예제

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


예제

```
class ColoredRect : public Rect {
        int red, green, blue;
public:
        ColoredRect(int x, int y, int w, int h, int r, int g, int b) :
                Rect(x, y, h, w), red(r), green(g), blue(b) { }
        void draw()
        {
                HDC hdc = GetWindowDC(GetForegroundWindow());
                SelectObject(hdc, GetStockObject(DC_BRUSH));
                SetDCBrushColor(hdc, RGB(red, green, blue));
                Rectangle(hdc, x, y, x + width, y + height);
        }
};
```

예제

```
int main()
{
        Rect r1(100, 100, 80, 80);
        ColoredRect r2(200, 200, 80, 80, 255, 0, 0);

        r1.draw();
        r2.draw();
        return 0;
}
```


11.5 접근 지정자

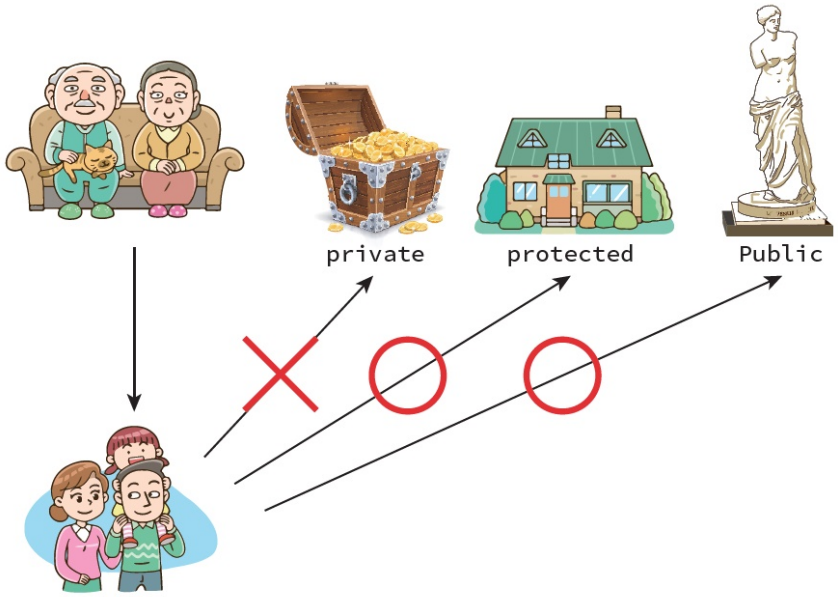


그림 11.9 상속에서의 접근 지정자

접근 지정자

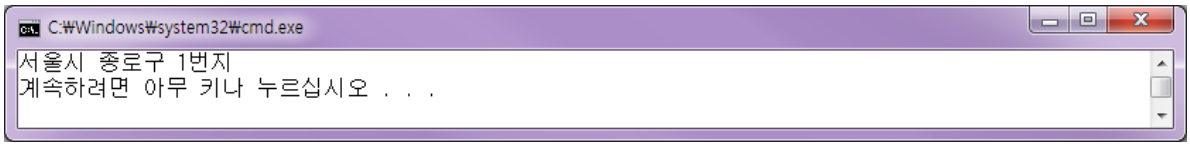
접근 지정자	자기 클래스	자식 클래스	외부
private	○	×	×
protected	○	○	×
public	○	○	○

예제

```
class Person {
        string name;
protected:
        string address;
};

class Student : public Person {
public:
        void setAddress(string add) {
                address = add;
        }
        string getAddress() {
                return address;
        }
};
```

예제

```
int main() {
        Student obj;

        obj.setAddress("서울시 종로구 1번지");
        cout << obj.getAddress() << endl;

        return 0;
}
```



PROTECTED 선언과 세 가지 형태의 상속

protected로 선언된 멤버가 허용하는 접근의 범위

```
class Base
{
private:
    int num1;
protected:
    int num2;
public:
    int num3;
    void ShowData()
    {
        cout<<num1<<", "<<num2<<", "<<num3;
    }
};
```

```
class Derived : public Base
{
public:
    void ShowBaseMember()
    {
        cout<<num1;        // 컴파일 에러
        cout<<num2;        // 컴파일 OK!
        cout<<num3;        // 컴파일 OK!
    }
};
```

private < protected < public

private을 기준으로 보면,
protected는 private과 달리
상속관계에서의 접근을
허용한다!

세 가지 형태의 상속

```
class Derived : public Base
{
    . . . . .
}
```

public 상속
접근 제어 권한을 그대로 상속한다!
단, private은 접근불가로 상속한다!

```
class Derived : protected Base
{
    . . . . .
}
```

protected 상속
protected보다 접근의 범위가 넓은 멤버는
protected로 상속한다.
단, private은 접근불가로 상속한다!

ProtectedHeri.cpp

```
class Derived : private Base
{
    . . . . .
}
```

private 상속
private보다 접근의 범위가 넓은 멤버는 private로
상속한다.
단, private은 접근불가로 상속한다!

protected 상속과 private 상속

```
class Base
{
private:
    int num1;
protected:
    int num2;
public:
    int num3;
};
```

```
class Derived : private Base
{
    // empty!
};
```

private 상속의 결과

```
class Derived : private Base
{
접근불가:
    int num1;
private:
    int num2;
private:
    int num3;
};
```

때문에 이 이상의 상속은 무의미할 수 있다.

```
class Derived : protected Base
{
    // empty!
};
```

protected 상속의 결과

```
class Derived : protected Base
{
접근불가:
    int num1;
protected:
    int num2;
protected:
    int num3;
};
```


11.6 멤버 함수 재정의

- 자식 클래스가 필요에 따라 상속된 멤버 함수를 재정의하여 사용하는 것을 의미한다.

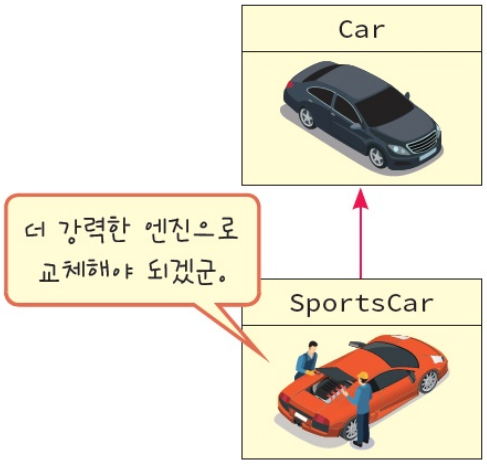


그림 11.10 멤버 함수 재정의

예제

```
#include <iostream>
#include <string>
using namespace std;
class Animal {
public:
        void speak()
        {
                cout << "동물이 소리를 내고 있음" << endl;
        }
};
class Dog : public Animal {
public:
        void speak()
        {
                cout << "멍멍!" << endl;
        }
};
```


예제

```
int main()
{
        Dog obj;
        obj.speak();
        return 0;
}
```

중복정의와 재정의

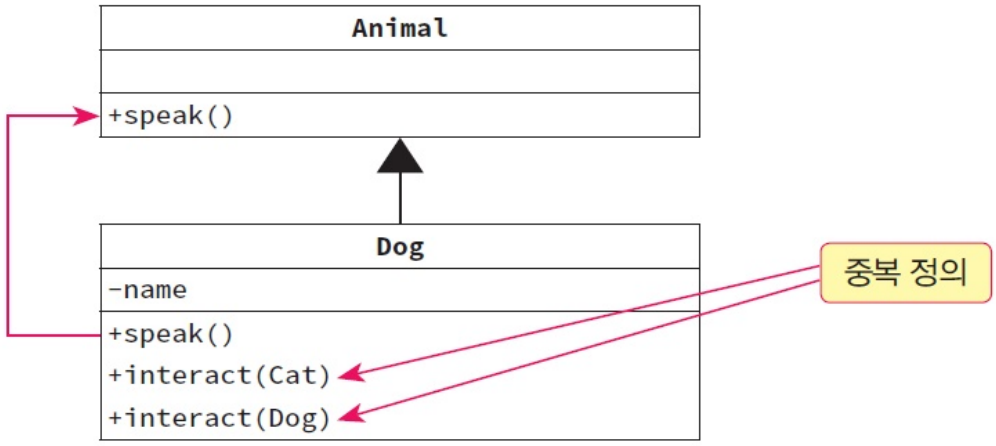
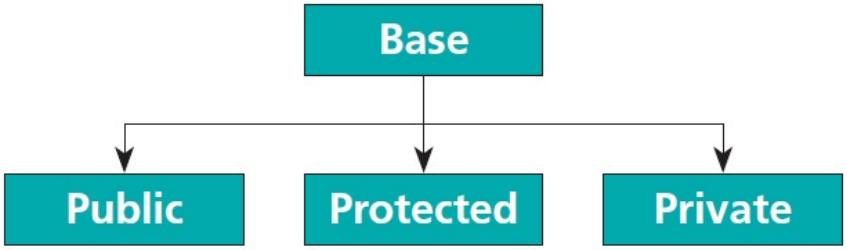


그림 11.11 재정의와 중복정의의 비교

부모클래스의 멤버함수 호출

```
#include <iostream>
#include <string>
using namespace std;
class ParentClass {
public:
        void print() {
        cout << "부모 클래스의 print() 멤버 함수" << endl;    }
};
class ChildClass : public ParentClass {
        int data;
public:
        void print() { //멤버 함수 재정의
                ParentClass::print();
                cout << "자식 클래스의 print() 멤버 함수 " << endl;
        }
};
```

부모 클래스를 상속받는 3가지 방법



	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

예제

```
#include <iostream>
using namespace std;
class Base {
        public: int x;
        protected: int y;
        private: int z;
};
class Derived : private Base{
        // x는 자식 클래스에서 사용가능하지만 private로 지정된다.
        // y는 자식 클래스에서 사용가능하지만 private로 지정된다.
        // z는 자식 클래스에서도 사용할 수 없다.
};
int main()
{
        Derived obj;
        ~~cout << obj.x;~~
}
```

객체 포인터의 참조관계

객체의 주소 값을 저장하는 객체 포인터 변수

" C++에서, **AAA형 포인터 변수는 AAA 객체 또는 AAA를 직접 혹은 간접적으로 상속하는 모든 객체를 가리킬 수 있다**(객체의 주소 값을 저장할 수 있다)."

```
class Student : public Person
{
    . . . . .
};

class PartTimeStudent : public Student
{
    . . . . .
};
```

Person * ptr=new Student();

Person * ptr=new PartTimeStudent();

Student * ptr=new PartTimeStudent();

유도 클래스의 객체도 가리키는 포인터!

IS-A 관계

"학생(Student)은 사람(Person)의 일종이다."

"근로학생(PartTimeStudent)은 학생(Student)의 일종이다."

"근로학생(PartTimeStudent)은 사람(Person)의 일종이다."

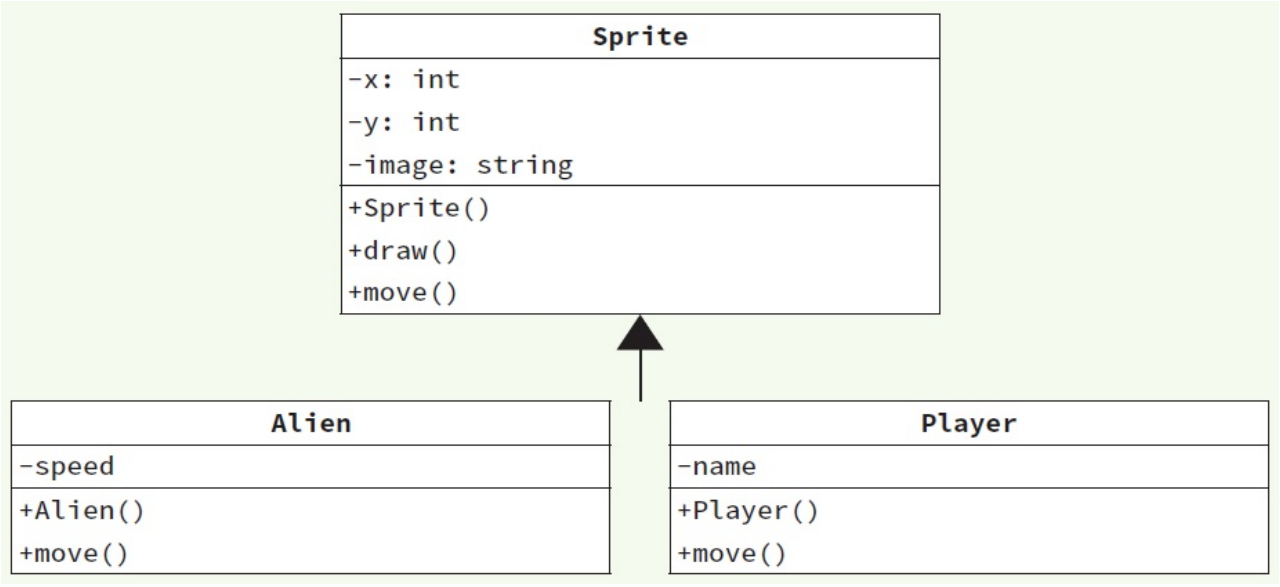
유도 클래스 객체를 기초 클래스 객체로 바라볼 수 있는 근거

"학생(Student)은 사람(Person)이다."

"근로학생(PartTimeStudent)은 학생(Student)이다."

"근로학생(PartTimeStudent)은 사람(Person)이다."

Lab: 게임에서의 상속



예제

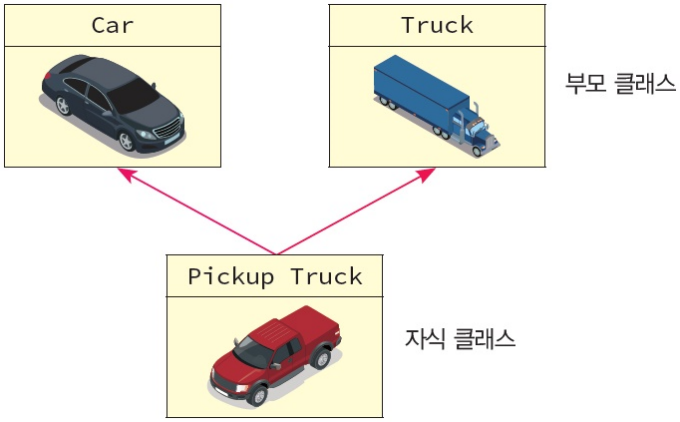
```
class Sprite
{
        int x, y;
        string image;
public:
        Sprite(int x, int y, string image) : x(x), y(y), image(image) {}
        void draw() {}
        void move() {}
};
class Alien : public Sprite
{
        int speed;
public:
        Alien(int x, int y, string image) : Sprite(x, y, image) {}
        void move() {}
};
```


예제

```
class Player : public Sprite
{
        string name;
public:
        Player(int x, int y, string image) : Sprite(x, y, image) {}
        void move() {}
};
int main()
{
        Alien a( 0, 100, "image1.jpg" );
        Player p(0, 100, "image1.jpg");
        return 0;
}
```

11.7 다중 상속

- 다중 상속(multiple inheritance)이란 하나의 자식 클래스가 두개 이상의 부모 클래스로부터 멤버를 상속받는 것을 의미한다.



예제

```
class PassengerCar {
public:
        int seats; // 정원
        void set_seats(int n) { seats = n; }
};
class Truck {
public:
        int payload; // 적재 하중
        void set_payload(int load) { payload = load; }
};
```

예제

```
class Pickup : public PassengerCar, public Truck {
public:
        int tow_capability; // 견인 능력
        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;
}
```


Q & A