# Chapter 06. 상속의 이해


**박 종 혁 교수**

**UCS Lab**

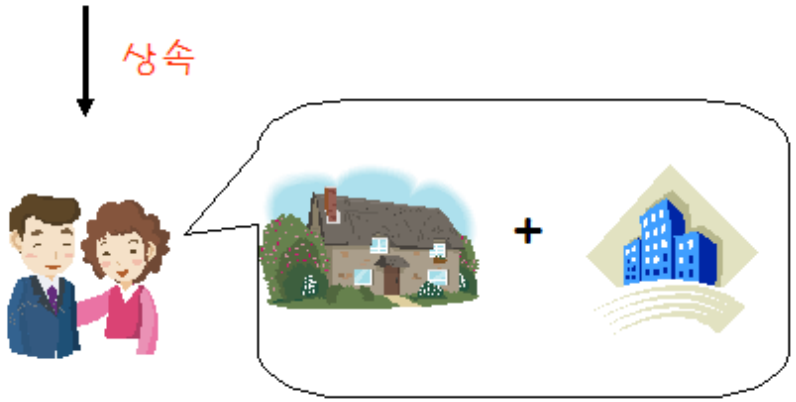
**Tel: 970-6702**

**Email: jhpark1@seoultech.ac.kr**

# 06-1. 상속의 기본개념

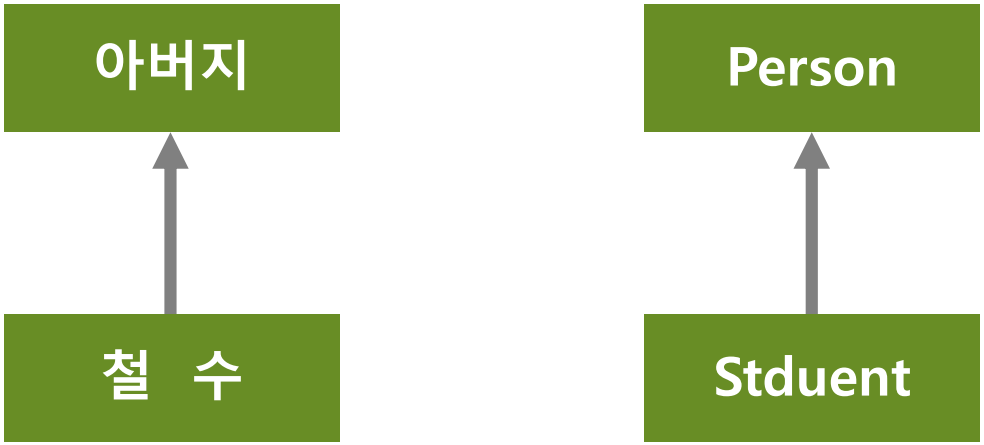
# 상속의 기본 개념

- 상속(inheritance)
  - 한 클래스가 다른 클래스에서 정의된 속성(자료,함수)를 이어받아 그대로 사용
  - 이미 정의된 클래스를 바탕으로 필요한 기능을 추가하여 정의




상속을 이용하면 쉽게 재산을 모을 수 있는 것처럼 소프트웨어도 쉽게 개발할 수 있다.

- 상속의 예1
  - "철수는 아버지로부터 좋은 목소리와 큰 키를 물려 받았다."
- 상속의 예2
  - "Student 클래스가 Person 클래스를 상속한다."



# 상속의 장점

- 상속을 통하여 기존 클래스의 속성(자료,함수) 재사용
- 기존 클래스의 일부 변경도 가능
- 상속을 이용하게 되면 복잡한 GUI 프로그램을 순식간에 작성
- 상속은 이미 작성된 검증된 소프트웨어를 재사용
- 신뢰성 있는 소프트웨어를 손쉽게 개발, 유지 보수
- 코드의 중복 감소

# 용어정의

- 베이스 클래스 (base class)
  - 기존에 이미 만들어진 클래스
  - 부모 클래스(parent class) or 슈퍼 클래스라 고도 함

- 파생 클래스(derived class)
  - 이를 상속 받아 새로 만들어지는 클래스
  - 하위클래스 라고도 함
  - 파생 클래스는 C++에서 각 클래스의 속성을 공유하고 물려받는 객체지향 프로그래밍의 상속(inheritance)을 구현한 것

| 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) |

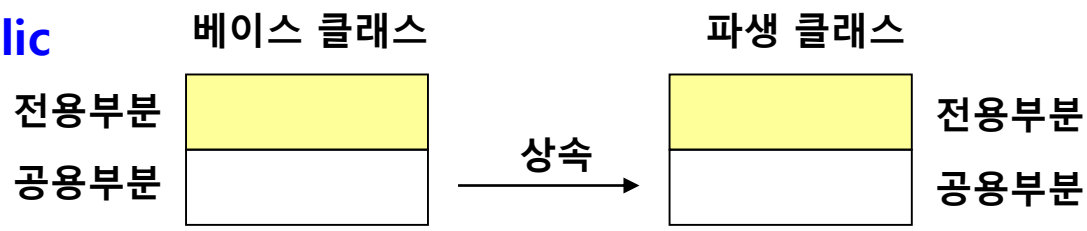
참고

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

# 공용부분 상속

```
class 파생클래스명 : public[private] 베이스클래스명  {
        ……
};
```

**public**



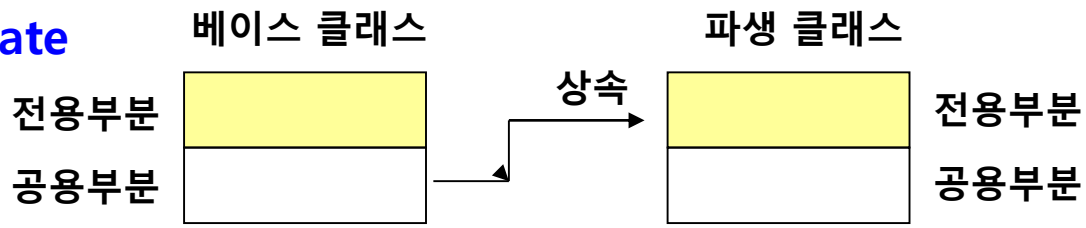

**객체생성 순서**

1.메모리 할당

2.Base 클래스 생성자 실행

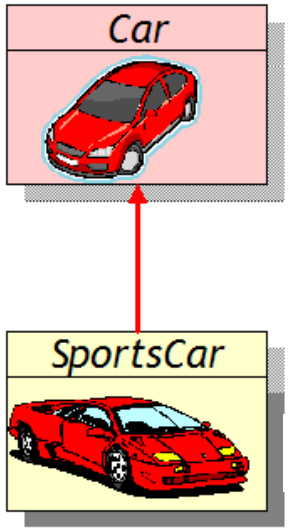
3.Derived클래스 생성자 실행

**private**



# 상속의 표현 예제1)

```
class Car
{
        int speed;
}

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




베이스클래스

파생클래스

상속한다는 의미

# 상속의 표현 예제2)


상속 관계 표현

```
class Phone {
   void call();
   void receive();
};
```

Phone을 상속받는다.

```
class MobilePhone : public Phone {
   void connectWireless();
   void recharge();
};
```

MobilePhone을 상속받는다.

```
class MusicPhone : public MobilePhone {
   void downloadMusic();
   void play();
};
```

C++로 상속 선언

전화기

휴대 전화기

음악 기능 전화기

## 정리: 상속을 이용하면 ?

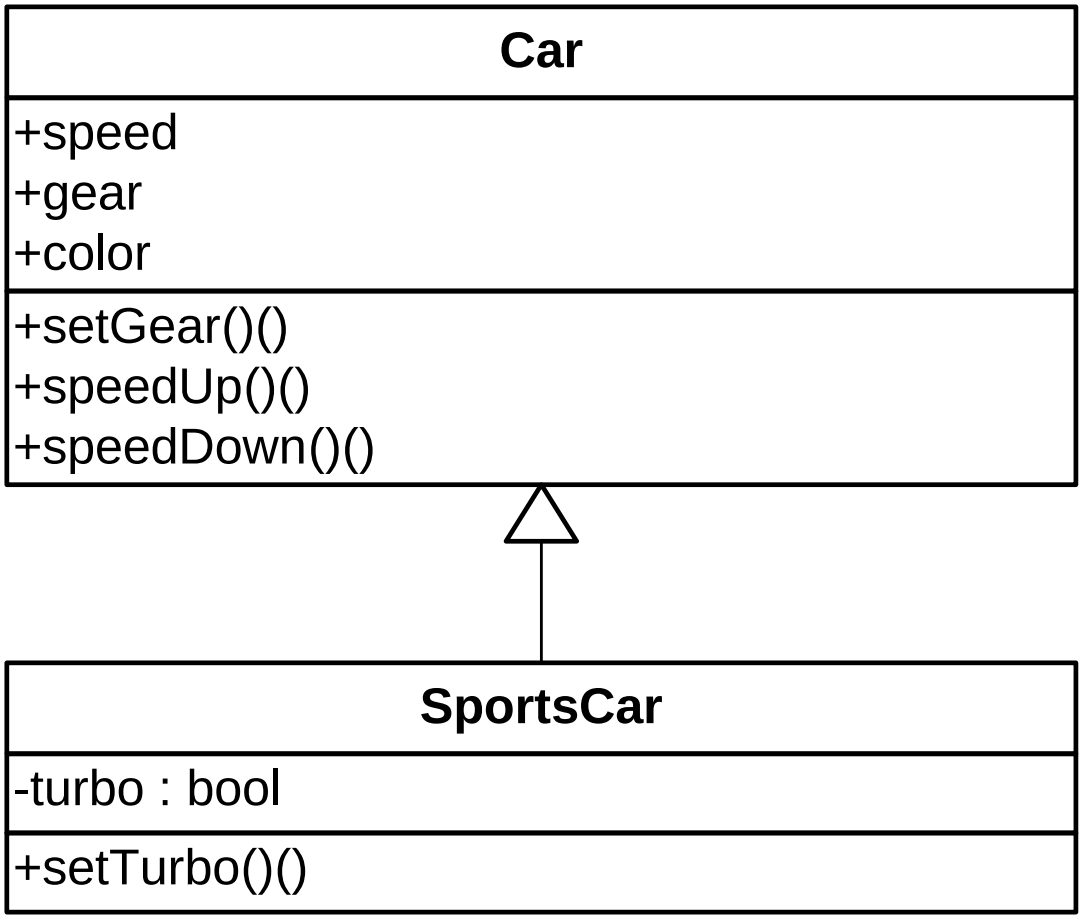
1. 간결한 클래스 작성
   - 기본 클래스의 기능을 물려받아 파생 클래스를 간결하게 작성

2. 클래스 간의 계층적 분류 및 관리의 용이함
   - 상속은 클래스들의 구조적 관계 파악 용이

3. 클래스 재사용과 확장을 통한 소프트웨어 생산성 향상
   - 빠른 소프트웨어 생산 필요
   - 기존에 작성한 클래스의 재사용 – 상속
     - 상속받아 새로운 기능을 확장
   - 앞으로 있을 상속에 대비한 클래스의 객체 지향적 설계 필요

# 06-2. 상속의 문법적인 이해

# 상속의 예제

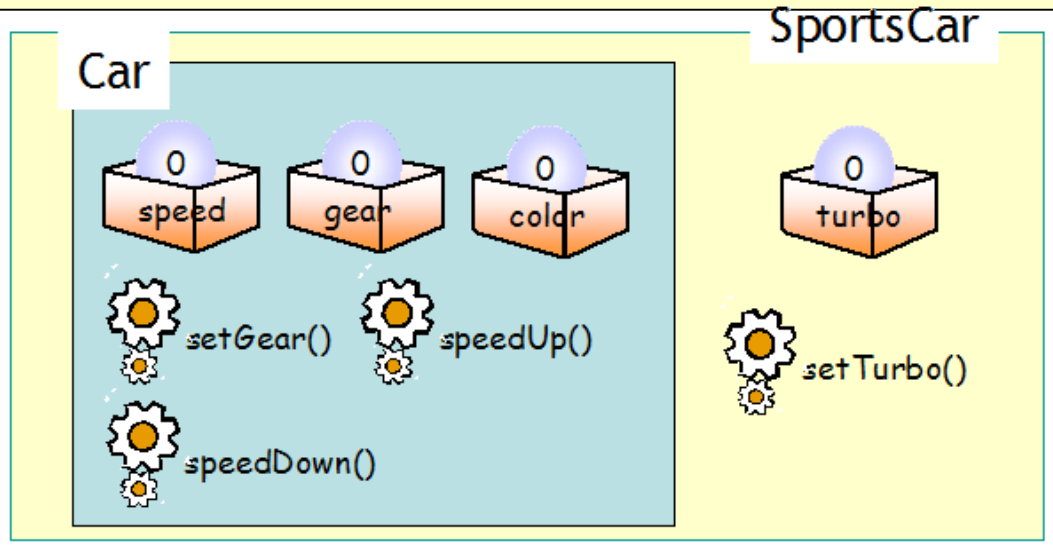

Car
+speed
+gear
+color
+setGear()()
+speedUp()()
+speedDown()()
SportsCar
-turbo : bool
+setTurbo()()

# Car 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) { // 기어설정멤버함수
                gear = newGear;
        }
        void speedUp(int increment) { // 속도증가멤버함수
                speed += increment;
        }
        void speedDown(int decrement) { // 속도감소멤버함수
                speed -= decrement;
        }
};
```

# SportsCar 클래스

```
// Car 클래스를상속받아서다음과같이SportsCar 클래스를작성하여보자.
class SportsCar : public Car {                    // Car를상속받는다.
        // 1개의멤버변수를추가
    bool turbo;

public:
        // 1개의멤버함수를추가
        void setTurbo(bool newValue) { // 터보모드설정멤버함수
                turbo = newValue;
        }
};
```

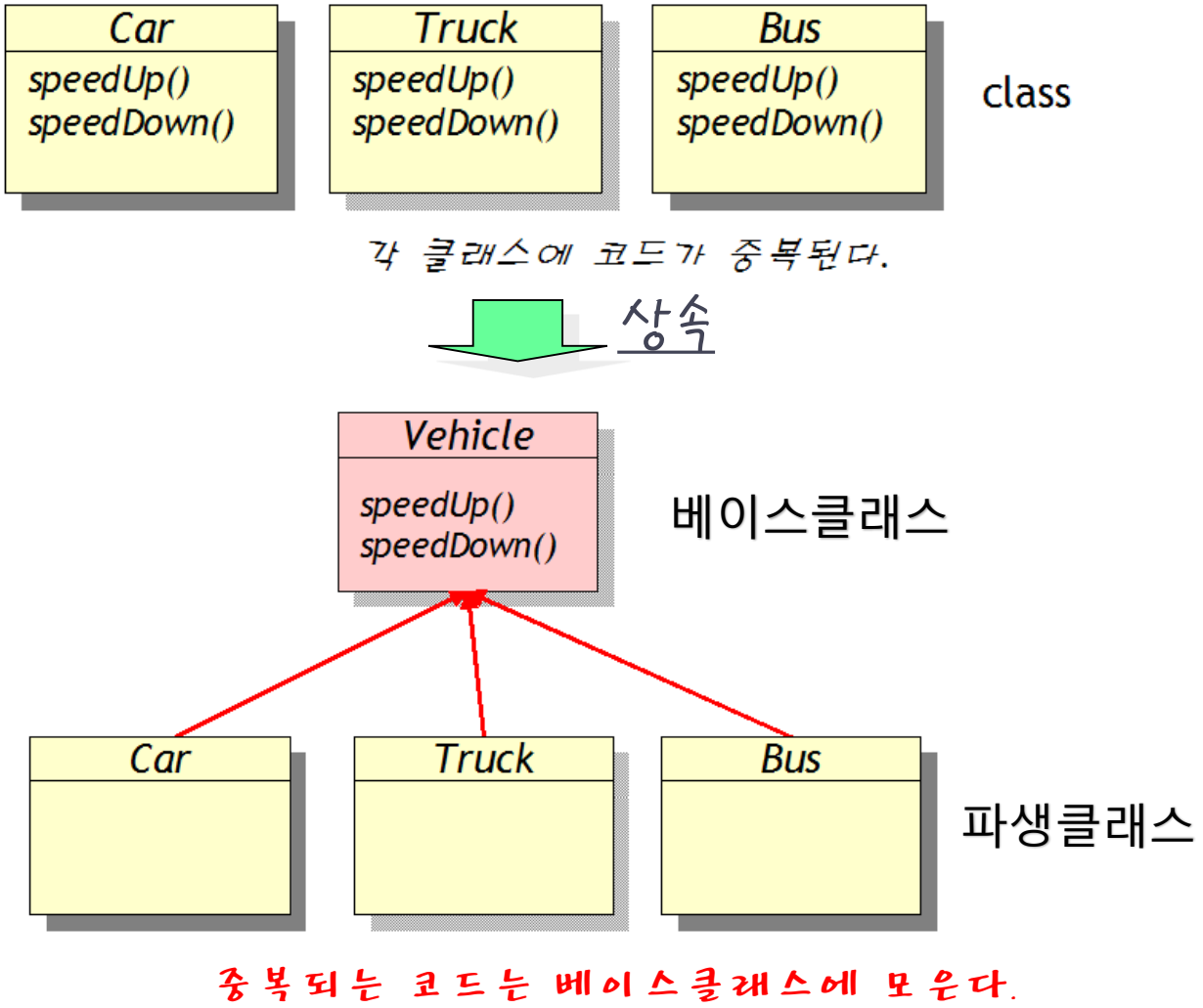


# SportsCar 클래스

```
int main()
{
        SportsCar  c;
        c.color = "Red";                    // 부모클래스멤버변수접근
        c.setGear(3);                               // 부모클래스멤버함수호출
        c.speedUp(100);                     // 부모클래스멤버함수호출
        c.speedDown(30);                    // 부모클래스멤버함수호출
        c.setTurbo(true);                   // 자체멤버함수호출
        return 0;
}
```

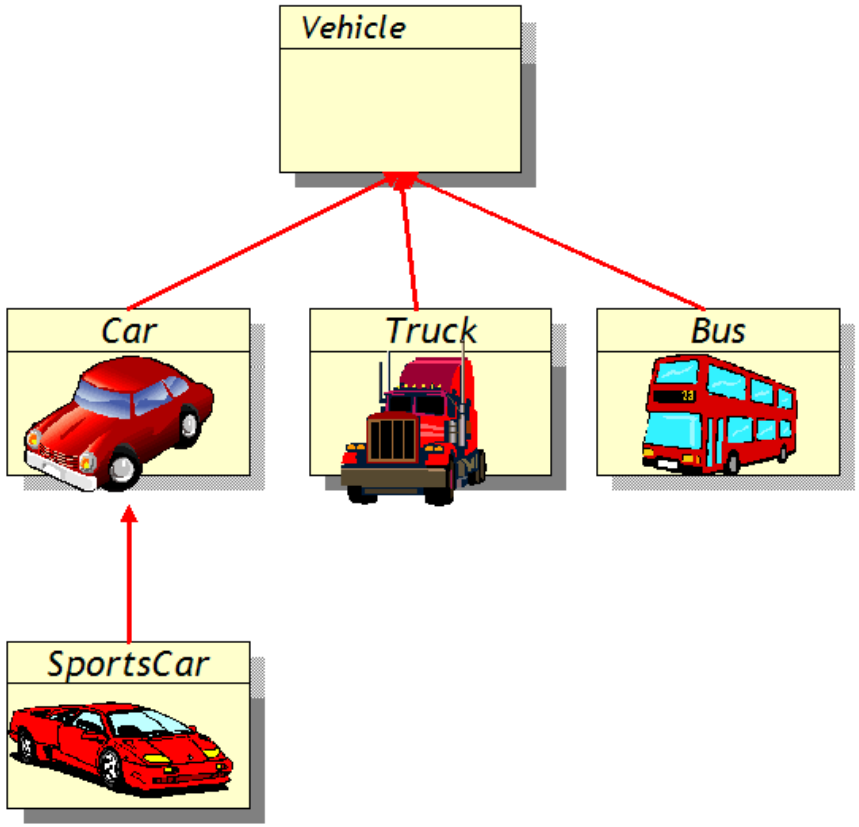
파생클래스는 베이스클래스의 변수와 함수를 마치 자기 것처럼 사용할 수 있다.

# 상속은 왜 필요한가?

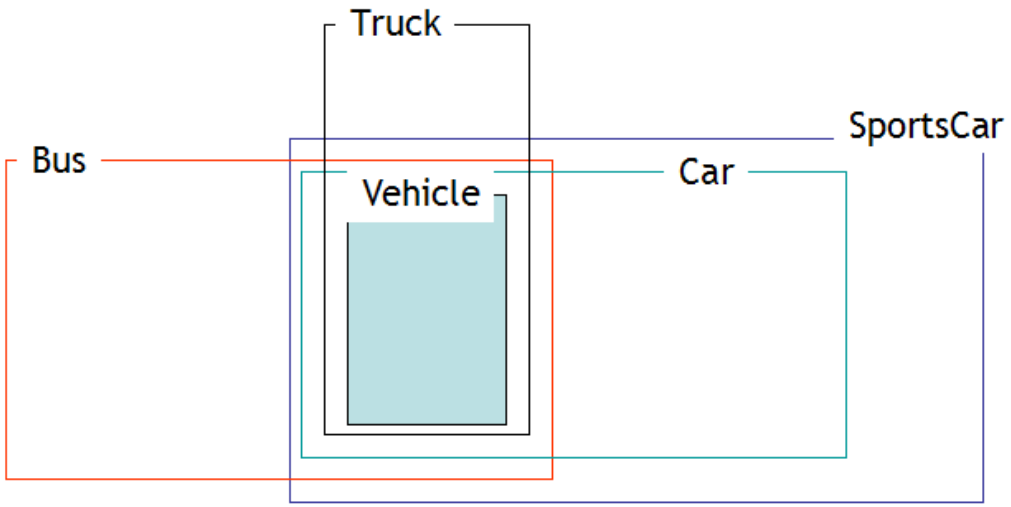

Car
speedUp()
speedDown()
Truck
speedUp()
speedDown()
Bus
speedUp()
speedDown()
class
각 클래스에 코드가 중복된다.
상속
Vehicle
speedUp()
speedDown()
베이스클래스
Car
Truck
Bus
파생클래스
중복되는 코드는 베이스클래스에 모은다.

# 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 { ...   }
```



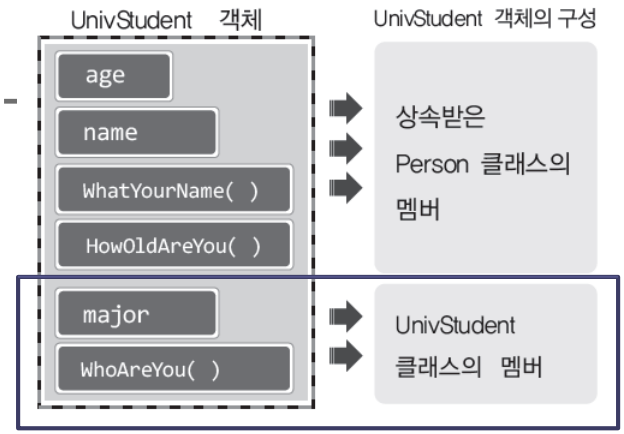
# 상속의 방법과 그 결과

```
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**

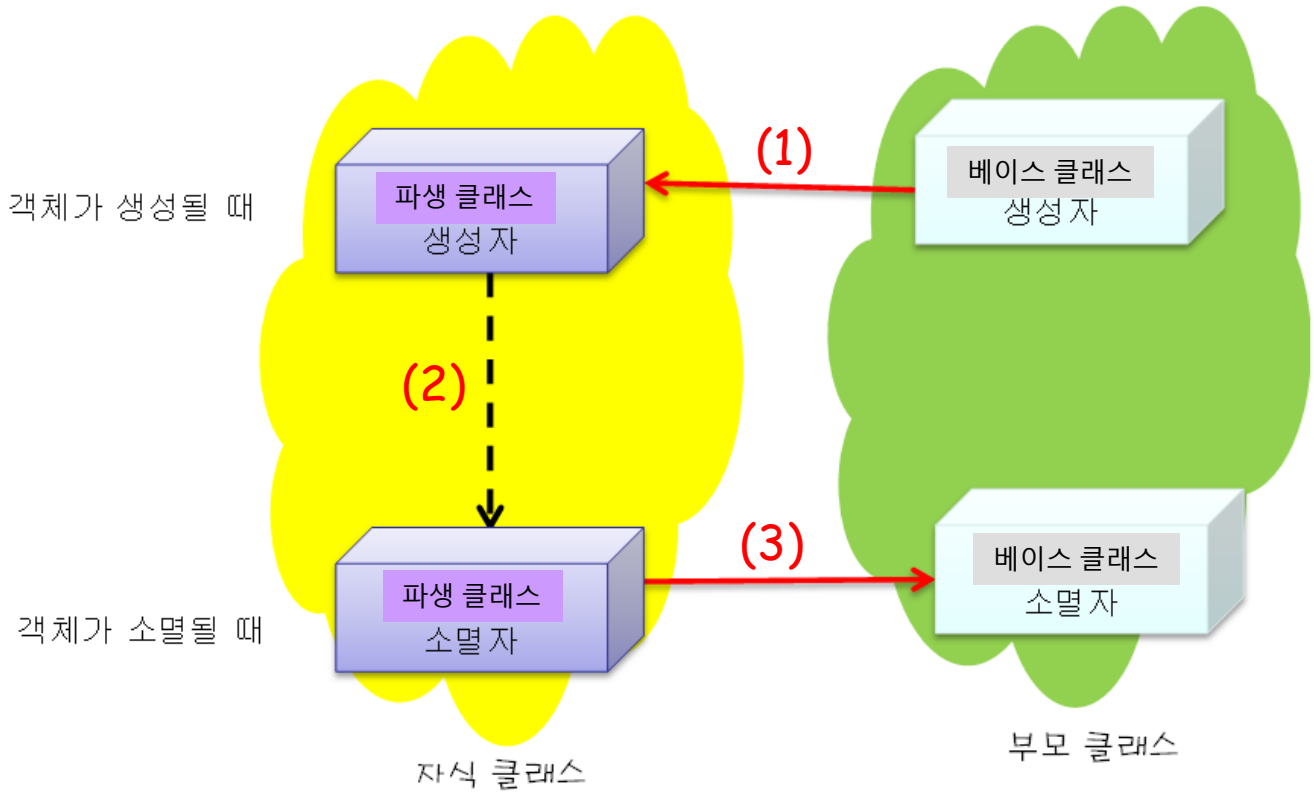
**Person 클래스의 멤버**




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

**용어정리**

# 상속에서의 객체의 생성, 생성자와 소멸자 순서




상속에서 생성자와 소멸자의 호출

# 상속받은 클래스의 생성자 정의

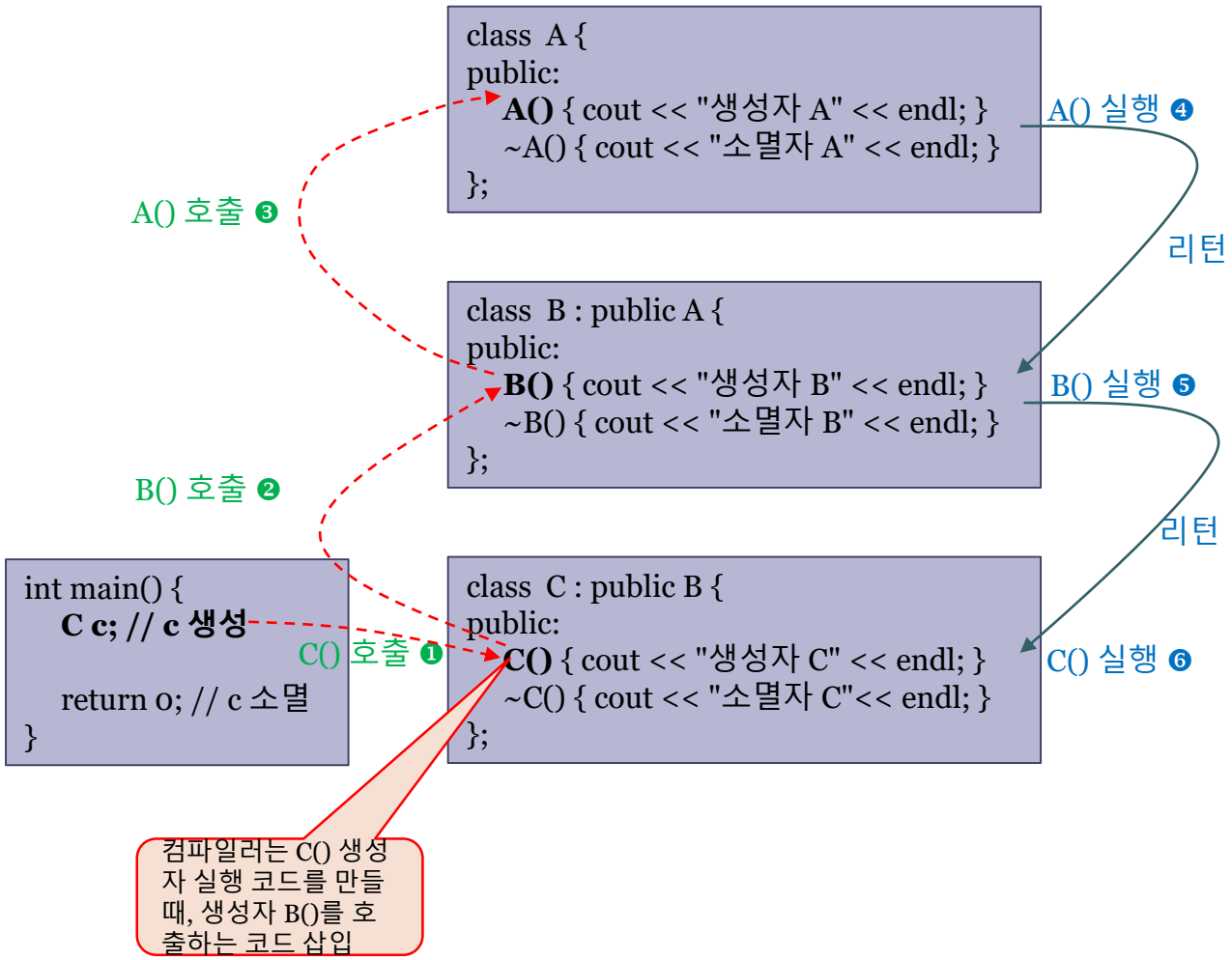
```
UnivStudent(char * myname, int myage, char * mymajor)
    : Person(myage, myname)
{
    strcpy(major, mymajor);
}
```

- 이니셜라이저를 통해서 파생클래스는 베이스 클래스의 생성자를 명시적으로 호출해야 한다.
- 파생클래스의 생성자는 베이스클래스의 멤버를 초기화 할 의무를 갖는다. 단! 베이스 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;
};
```

- 파생 클래스 UnivStudent는 베이스 클래스의 생성자 호출을 위한 인자까지 함께 전달받아야 한다.
- private 멤버는 파생 클래스에서도 접근이 불가능하므로, 생성자의 호출을 통해서 베이스 클래스의 멤버를 초기화해야 한다.

# 생성자 호출 관계 및 실행 순서



# 파생클래스의 객체생성 과정 예제

```
class SoBase// 베이스클래스
{
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
{    // 파생클래스
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;
};
```

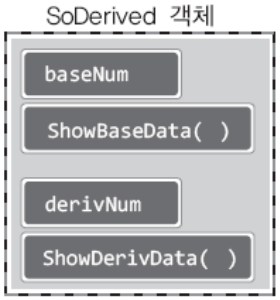
실행결과
?

```
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;
};
```

실행결과

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

# 파생클래스의 객체생성 과정 case1

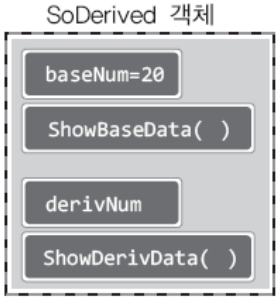



**순서 1. 메모리 공간의 할당**

SoDerived dr1


```
SoDerived( ) : derivNum(30)
{
    cout<<"SoDerived()"<<endl;
}
```

**순서 2. 파생클래스의 void 생성자 호출**



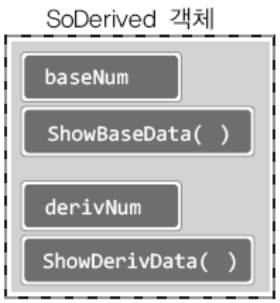

```
SoDerived( ) : derivNum(30)
{
    cout<<"SoDerived()"<<endl;
}
```

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

```
SoBase( ) : baseNum(20)
{
   cout<<"SoBase()"<<endl;
}
```

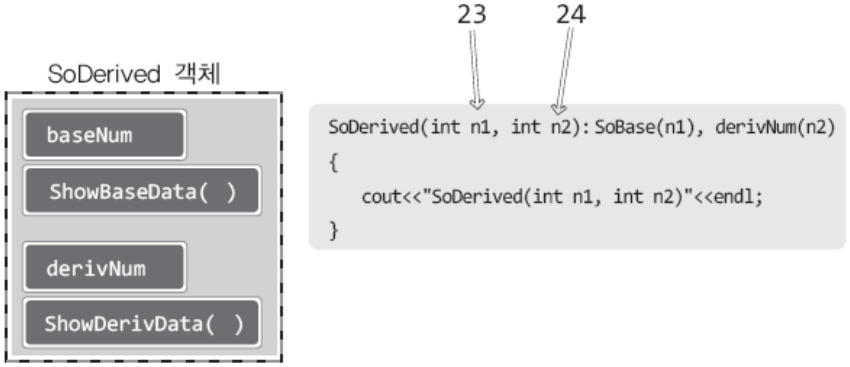
**순서 4. 파생클래스의 실행**

# 파생클래스의 객체생성 과정 case3

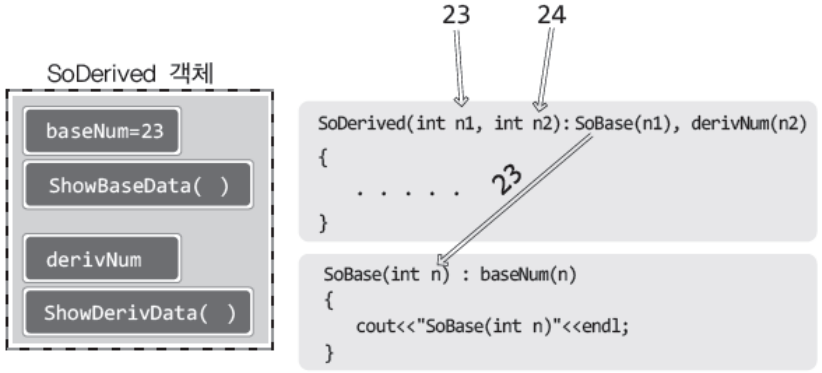



**순서 1. 메모리 공간의 할당**

```
SoDerived dr3(23, 24);
```




**순서 2. 파생클래스의 생성자 호출**




**순서 3. 베이스클래스의 생성자 호출 및 실행**

**순서 4. 파생 클래스의 생성자 실행**

# 파생클래스 객체의 소멸과정

```
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;
    }
};
```

실행결과

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

파생클래스의 소멸자가 실행된 후 베이스클래스의 소멸자가 실행된다.

스택에 생성된 객체의 소멸순서는 생성순서와 반대이다.

# 상속의 소멸자 예

```
#include <iostream>
using std::endl;
using std::cout;

class AAA    //Base 클래스
{
public:
   AAA() {
      cout<<"AAA() call!"<<endl;
  }
   ~AAA() {
      cout<<“~AAA(int i) call!"<<endl;
   }
};

class BBB : public AAA    //Derived 클래스
{
public:
   BBB(){
      cout<<"BBB() call!"<<endl;
  }
   ~BBB() {
      cout<<"~BBB() call!"<<endl;
   }
};
```

**객체소멸 순서**

---

**1. 파생클래스 소멸자 실행**

**2. 베이스 클래스 소멸자 실행**

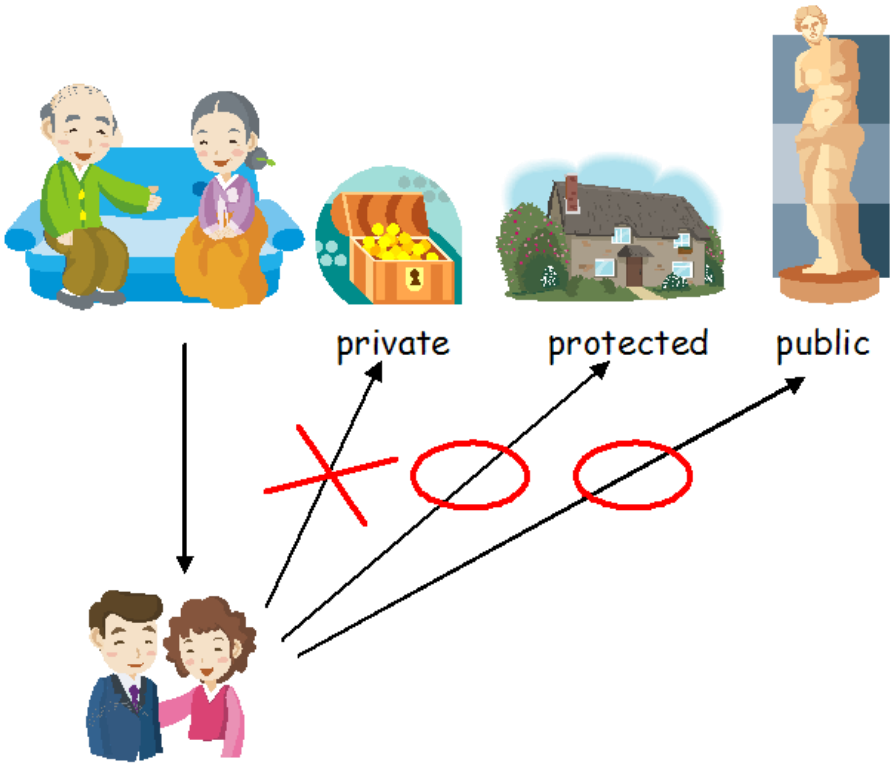
**3. 메모리 반환(해제)**

```
int main(void)
{
    BBB bbb;

    return 0;
}
```

```
AAA() call!
BBB() call!
~BBB() call!
~AAA() call!
```

# 06-3. 상속의 형태 및 protected 선언

# 접근 제어 지정자




상속에서의 접근 지정자

# 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!**

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

**private 상속**

**private보다 접근의 범위가 넓은 멤버는 protected로 상속한다.**

**단, private은 접근불가로 상속한다!**

# 세 가지 형태의 상속

- 접근권한 변경
  - Base 클래스의 멤버는 상속되는 과정에서 접근 권한 변경

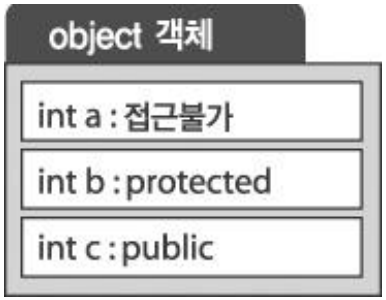
| 상속 형태 / Base 클래스 | public 상속 | protected 상속 | private 상속 |
|---|---|---|---|
| public 멤버 | public | protected | private |
| Protected 멤머 | protected | protected | private |
| Private 멤버 | 접근 불가 | 접근 불가 | 접근 불가 |

```
class 클래스명 {
[private:]              // 전용부분 멤버 정의
        .....          // 클래스 외부 접근 및 상속 안됨
protected:             // 보호부분 멤버 정의
        .....          // 클래스 외부 접근 안됨, 상속 됨
public:                // 공용부분 멤버 정의
        .....          // 클래스 외부 접근 및 상속 됨
};
```

# 세 가지 형태의 상속 예

```
class Base
{
private:
    int a;
protected:
    int b;
public:
    int c;
};

class Derived : public Base  // public 상속 →
{
   // EMPTY
};
```

```
int main(void)
{
  Derived object;
  return 0;
};
```

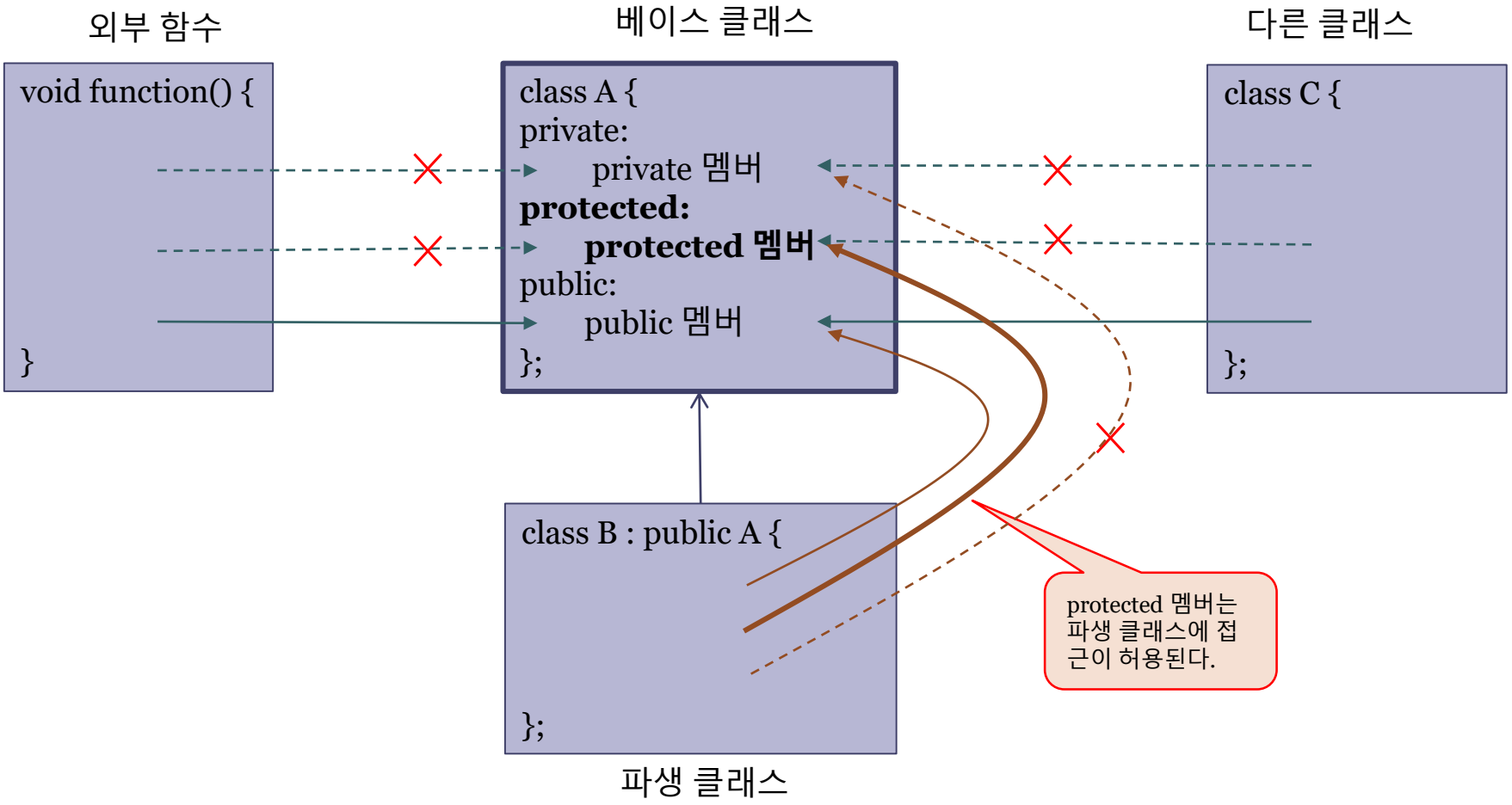



**protected** ?

**private** ?

** 사용된 상속보다 넓은 범위는 그 범위로 맞춤!!

# 멤버의 접근 지정에 따른 접근성



# protected 멤버 상속

- 파생 클래스의 멤버함수는 베이스 클래스의 공용멤버에 직접 접근이 가능하지만 베이스 클래스의 private멤버에 대해서는 접근할 수 없음

- 파생 클래스의 멤버함수가 베이스 클래스의 private멤버를 상속받아 자유로이 사용을 위해서는 다음 두 가지 방식 사용
  - 프렌드 함수를 사용
  - 베이스 클래스 정의 시 protected 키워드를 사용한 보호부분을 정의

- 보호부분에 있는 멤버는 private멤버와 같이 외부에서는 직접 접근할 수 없지만 파생 클래스의 멤버함수에서는 직접 접근이 가능
  - 보호부분 멤버가 파생 클래스에서 public으로 상속 받으면 파생 클래스의 protected 멤버가 됨
  - private으로 상속 받으면 파생 클래스에서 private부분 멤버가 됨

# protected 멤버 상속 예

- 상속관계에 놓여있는 경우 접근 허용
- 그 이외는 private 멤버와 동일

```
class AAA
{
private:
    int a;
protected:
    int b;
};

class BBB: public AAA
{
public:
    void SetData() {
        a=10;  // private 멤버, 따라서 에러
          b=10; // protected 멤버, OK!
       }
};
```

```
int main(void)
{
    AAA aaa;
    aaa.a=10; // private 멤버, 따라서 에러 !!
    aaa.b=20; // protected 멤버, 따라서 에러!!

    BBB bbb;
    bbb.SetData();

    return 0;
}
```

# 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!
};
```

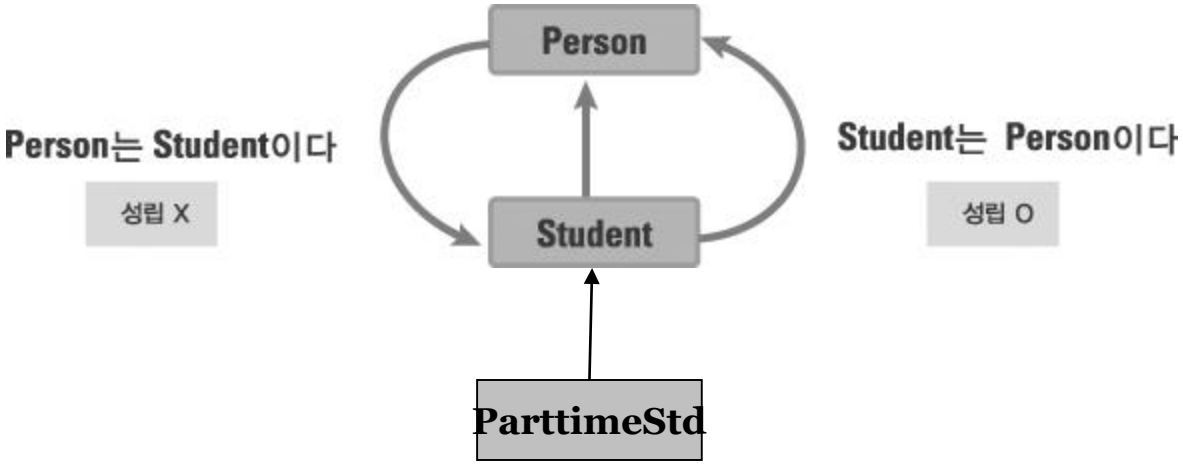
protected
상속의 결과

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

# 06-4. 상속을 위한 조건

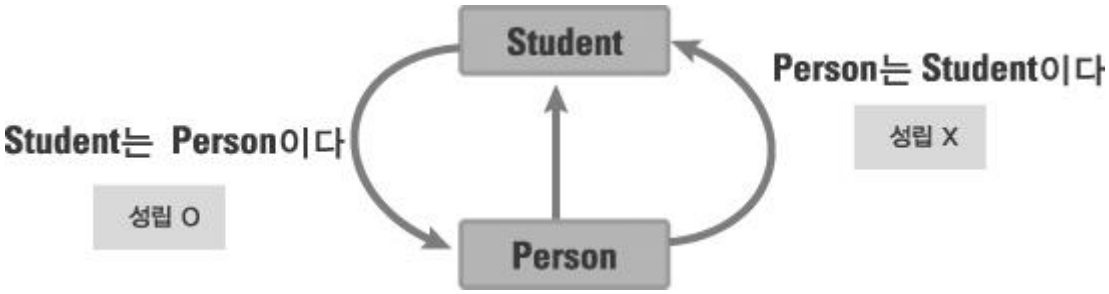
# 상속의 조건

- public 상속은 is-a 관계가 성립되도록 하자.



# 상속의 조건

- 잘못된 상속의 예



# 상속의 조건

- HAS-A(소유) 관계에 의한 상속!
  - 경찰은 몽둥이를 소유한다
  - The Police have a cudgel.

```
#include <iostream>
using std::endl;
using std::cout;

class Cudgel  //몽둥이
{
public:
        void Swing(){ cout<<"Swing a cudgel!"<<endl; }
};

class Police : public Cudgel  //몽둥이를 소유하는 경찰
{
public:
        void UseWeapon(){ Swing(); }
};
```

```
int main()
{
        Police pol;
        pol.UseWeapon();

        return 0;
}
```

Police is a Cudgel **(X)**
Police has a Cudgel **(O)**

# 상속의 조건

- HAS-A에 의한 상속 그리고 대안!
  - **포함 관계를** 통해서 소유 관계를 표현
  - **객체 멤버에** 의한 포함 관계의 형성
  - **객체 포인터 멤버에** 의한 포함 관계의 형성


Police 객체는 Cudgel 객체를 포함한다.

# 상속의 조건

```
/*   객체 멤버 예제 */
class Cudgel  //몽둥이
{
public:
      void Swing(){ cout<<"Swing a cudgel!"<<endl; }
};
class Police //몽둥이를 소유하는 경찰
{
      Cudgel cud;
public:
      void UseWeapon(){ cud.Swing(); }
};

int main()
{
      Police pol;
      pol.UseWeapon();
      return 0;
}
```

# 상속의 조건

```
/*  객체 포인터 멤버 예제 */
class Cudgel  //몽둥이
{
public:
	void Swing(){ cout<<"Swing a cudgel!"<<endl; }
};
class Police //몽둥이를 소유하는 경찰
{
	Cudgel* cud;
public:
	Police(){ cud=new Cudgel; }
	~Police(){ delete cud; }
	void UseWeapon(){ cud->Swing(); }
};
int main()
{
	Police pol;
	pol.UseWeapon();
	return 0;
}
```

# 참고문헌


- 뇌를 자극하는 C++ 프로그래밍, 이현창, 한빛미디어1
- 열혈 C++ 프로그래밍(개정판), 윤성우, 오렌지미디어
- C++ ESPRESSO, 천인국 저, 인피니티북스
- 명품 C++ Programming, 황기태 , 생능출판사

# 추가 자료

# 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!**

# 상속 - 접근 예제

```
#include <iostream>
#include <string>

using namespace std;
class  Employee {
        int rrn;          // Regident Resgistration Number: 주민등록번호

protected:
        int salary;       // 월급

public:
        string name;              // 이름
        void setSalary(int salary);
        int getSalary();
};

void Employee::setSalary(int salary) {
        this->salary = salary;
}
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();
};

void Manager::modify(int s, int b) {
        salary = s; // 베이스클래스의 보호멤버 사용 가능!
        bonus = b;

}

void Manager::display()
{
        cout << "봉급: " << salary << " 보너스: " << bonus << endl;
        // cout << "주민등록번호: " << rrn << endl;//
}
```

```
int main()
{
		Manager m;
		m.setSalary(2000);
		m.display();
		m.modify(1000, 500);
		m.display();
}
```

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

# private 상속 예

다음에서 컴파일 오류가 발생하는 부분을 찾아라.

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

class Base {
   int a;
protected:
   void setA(int a) { this->a = a; }
public:
   void showA() { cout << a; }
};

class Derived : private Base {
   int b;
protected:
   void setB(int b) { this->b = b; }
public:
   void showB() { cout << b; }
};
```

```
int main() {
   Derived x;
   x.a = 5;          // ①
   x.setA(10);       // ②
   x.showA();        // ③
   x.b = 10;         // ④
   x.setB(10);       // ⑤
   x.showB();        // ⑥
}
```

컴파일 오류
①, ②, ③, ④, ⑤

# protected 상속 예

다음에서 컴파일 오류가 발생하는 부분을 찾아라.

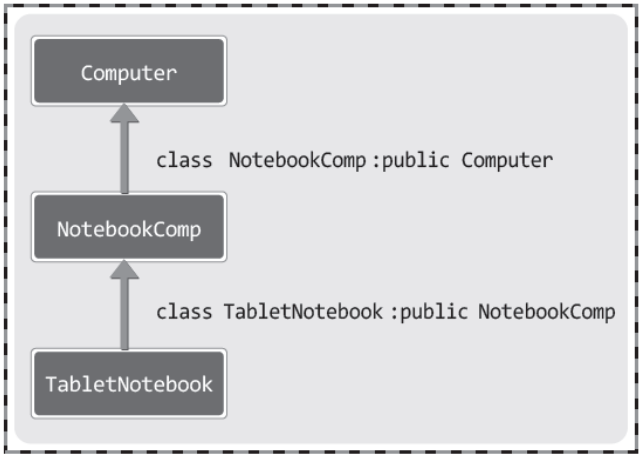
```
#include <iostream>
using namespace std;

class Base {
   int a;
protected:
   void setA(int a) { this->a = a; }
public:
   void showA() { cout << a; }
};

class Derived : protected Base {
   int b;
protected:
   void setB(int b) { this->b = b; }
public:
   void showB() { cout << b; }
};
```

```
int main() {
   Derived x;
   x.a = 5;          // ①
   x.setA(10);       // ②
   x.showA();        // ③
   x.b = 10;         // ④
   x.setB(10);       // ⑤
   x.showB();        // ⑥
}
```

컴파일 오류
①, ②, ③, ④, ⑤

# 상속의 기본 조건인 IS-A 관계의 성립

- 전화기 → 무선 전화기
- 컴퓨터 → 노트북 컴퓨터




- 무선 전화기는 일종의 전화기입니다.
- 노트북 컴퓨터는 일종의 컴퓨터입니다.

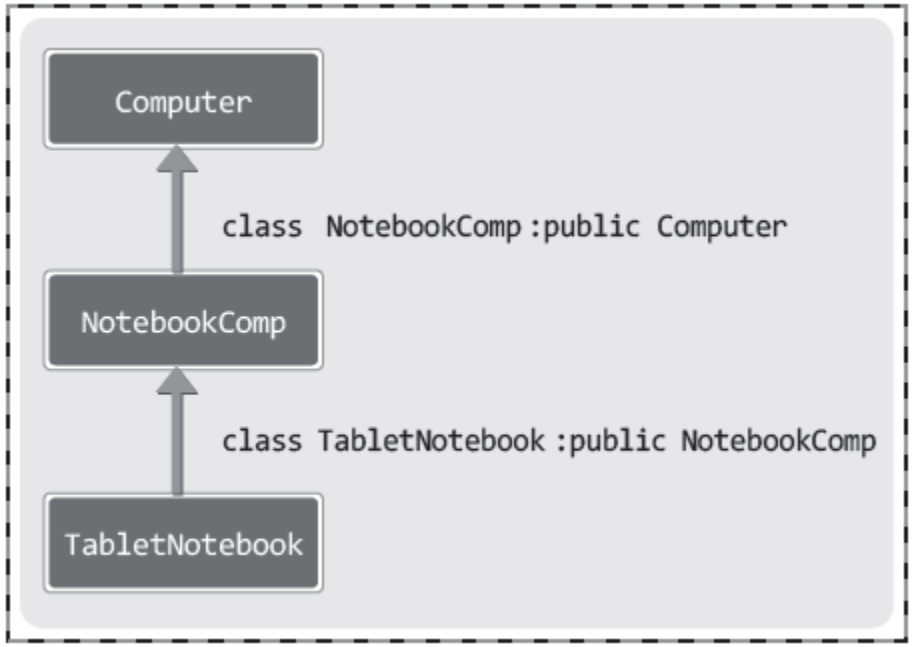
- 무선 전화기 is a 전화기
- 노트북 컴퓨터 is a 컴퓨터

**무선 전화기는 전화기의 기본 기능에 새로운 특성이 추가된 것이다.**

**노트북 컴퓨터는 컴퓨터의 기본 기능에 새로운 특성이 추가된 것이다.**

**이렇듯 is-a 관계는 논리적으로 상속을 기반으로 표현하기에 매우 적절하다.**

# IS-A 기반의 예제




**예제는 도서 본문을 참조합시다!**

- NotebookComp(노트북 컴퓨터)는 Computer(컴퓨터)이다.
- TabletNotebook(타블렛 컴퓨터)는 NotebookComp(노트북 컴퓨터)이다.
- TabletNotebook(타블렛 컴퓨터)는 Computer(컴퓨터)이다.

# ISA 상속 예제

```
#include <iostream>
#include <cstring>
using namespace std;
class Computer
{
private:
    char owner[50];
public:
    Computer(char * name)
    {
        strcpy(owner, name);
    }
    void Calculate()
    {
        cout<<"요청 내용을 계산합니다."<<endl;
    }
};
```

```
class NotebookComp : public Computer
{
private:
    int battary;
public:
    NotebookComp(char * name, int initChag)
        : Computer(name), battary(initChag)
    { }
    void Charging() { battary+=5; }
    void UseBattary() { battary-=1; }
    void MovingCal()
    {
        if(GetBattaryInfo()<1)
        {
            cout<<"충전이 필요합니다."<<endl;
            return;
        }
        cout<<"이동하면서 ";
        Calculate();
        UseBattary();
    }
    int GetBattaryInfo() { return battary; }
};
```

```
class TabletNotebook : public NotebookComp
{
private:
    char regstPenModel[50];
public:
    TabletNotebook(char * name, int initChag, char * pen)
        : NotebookComp(name, initChag)
    {
        strcpy(regstPenModel, pen);
    }
    void Write(char * penInfo)
    {
        if(GetBattaryInfo()<1)
        {
            cout<<"충전이 필요합니다."<<endl;
            return;
        }
        if(strcmp(regstPenModel, penInfo)!=0)
        {
            cout<<"등록된 펜이 아닙니다.";
            return;
        }
        cout<<"필기 내용을 처리합니다."<<endl;
        UseBattary();
    }
};
```

```
int main(void)
{
    NotebookComp nc("이수종", 5);
    TabletNotebook tn("정수영", 5, "ISE-241-242");
    nc.MovingCal();
    tn.Write("ISE-241-242");
    return 0;
}
```

# HAS-A 관계를 상속으로 구성하면

```
class Gun
{
private:
    int bullet;     // 장전된 총알의 수
public:
    Gun(int bnum) : bullet(bnum)
    { }
    void Shut()
    {
        cout<<"BBANG!";
        bullet--;
    }
};
```

```
class Police : public Gun
{
private:
    int handcuffs;      // 소유한 수갑의 수
public:
    Police(int bnum, int bcuff)
        : Gun(bnum), handcuffs(bcuff)
    { }
    void PutHandcuff()
    {
        cout<<"SNAP!";
        handcuffs--;
    }
};
```

**경찰은 총을 소유한다.**
**경찰 has a 총!**

**has a 관계도 상속으로 구현이 가능하다. 하지만 이러한 경우 Police와 Gun은 강한 연관성을 띠게 된다. 따라서 총을 소유하지 않은 경찰이나, 다른 무기를 소유하는 경찰을 표현하기가 쉽지 않아진다.**

# HAS 상속 예제

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

class Gun
{
private:
    int bullet;             // 장전된 총알의 수
public:
    Gun(int bnum) : bullet(bnum)
    { }
    void Shut()
    {
            cout<<"BBANG!"<<endl;
            bullet--;
    }
};
```

```
class Police : public Gun
{
private:
    int handcuffs;          // 소유한 수갑의 수
public:
    Police(int bnum, int bcuff)
            : Gun(bnum), handcuffs(bcuff)
    { }
    void PutHandcuff()
    {
            cout<<"SNAP!"<<endl;
            handcuffs--;
    }
};

int main(void)
{
    Police pman(5, 3); // 총알 5, 수갑 3
    pman.Shut();
    pman.PutHandcuff();
    return 0;
}
```

# HAS-A 관계는 포함으로 표현한다

```
class Gun
{
private:
    int bullet;     // 장전된 총알의 수
public:
    Gun(int bnum) : bullet(bnum)
    { }
    void Shut()
    {
        cout<<"BBANG!"<<endl;
        bullet--;
    }
};
```

```
class Police
{
private:
    int handcuffs;      // 소유한 수갑의 수
    Gun * pistol;       // 소유하고 있는 권총
public:
    Police(int bnum, int bcuff)
        : handcuffs(bcuff)
    {
        if(bnum>0)
            pistol=new Gun(bnum);
        else
            pistol=NULL;
    }
    void PutHandcuff()
    {
        cout<<"SNAP!"<<endl;
        handcuffs--;
    }
    void Shut()
    {
        if(pistol==NULL)
            cout<<"Hut BBANG!"<<endl;
        else
            pistol->Shut();
    }
    ~Police()
    {
        if(pistol!=NULL)
            delete pistol;
    }
};
```

has a의 과계를 포함의 형태로 표현하면, 두 클래스 간 연관성은 낮아지며, 변경 및 확장이 용이해진다.

즉, 총을 소유하지 않은 경찰의 표현이 쉬워지고, 추가로 무기를 소유하는 형태로의 확장도 간단해진다.