C++에서의 상속(Inheritance)

- Is a(~은 ~이다)
  (학생은 사람이다. 사각형은 도형이다.)

- C++에서의 상속이란?
  - 클래스 사이에서 상속관계 정의
    - 객체 사이에는 상속 관계 없음
  - 기본 클래스의 속성과 기능을 파생 클래스에 물려주는 것
    - 기본 클래스(base class) – 상속해주는 클래스. 부모 클래스
    - 파생 클래스(derived class) – 상속받는 클래스. 자식 클래스
      - 기본 클래스의 속성과 기능을 물려받고 자신 만의 속성과 기능을 추가하여 작성
  - 기본 클래스에서 파생 클래스로 갈수록 클래스의 개념이 구체화
  - 다중 상속을 통한 클래스의 재활용성 높임

상속의 표현

상속 관계 표현

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

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

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

Phone을 상속받는다.

MobilePhone을 상속받는다.

C++로 상속 선언

전화기

휴대 전화기

음악 기능
전화기

상속의 목적 및 장점

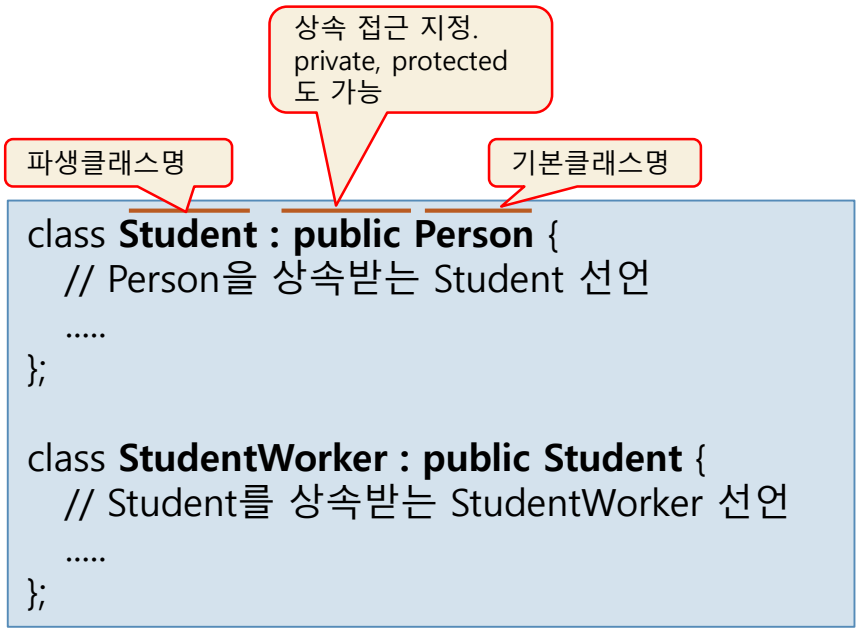
1. 간결한 클래스 작성
   - 기본 클래스의 기능을 물려받아 파생 클래스를 간결하게 작성

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

상속 선언

상속 선언



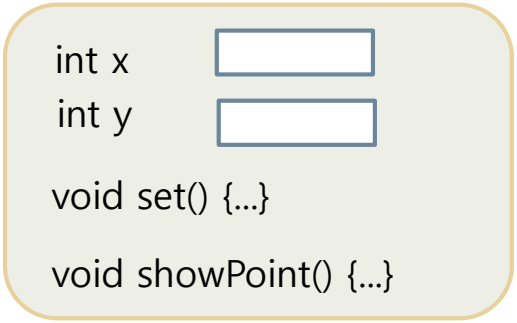
- Student 클래스는 Person 클래스의 멤버를 물려받는다.
- StudentWorker 클래스는 Student의 멤버를 물려받는다.
  - Student가 물려받은 Person의 멤버도 함께 물려받는다.

파생 클래스의 객체 구성

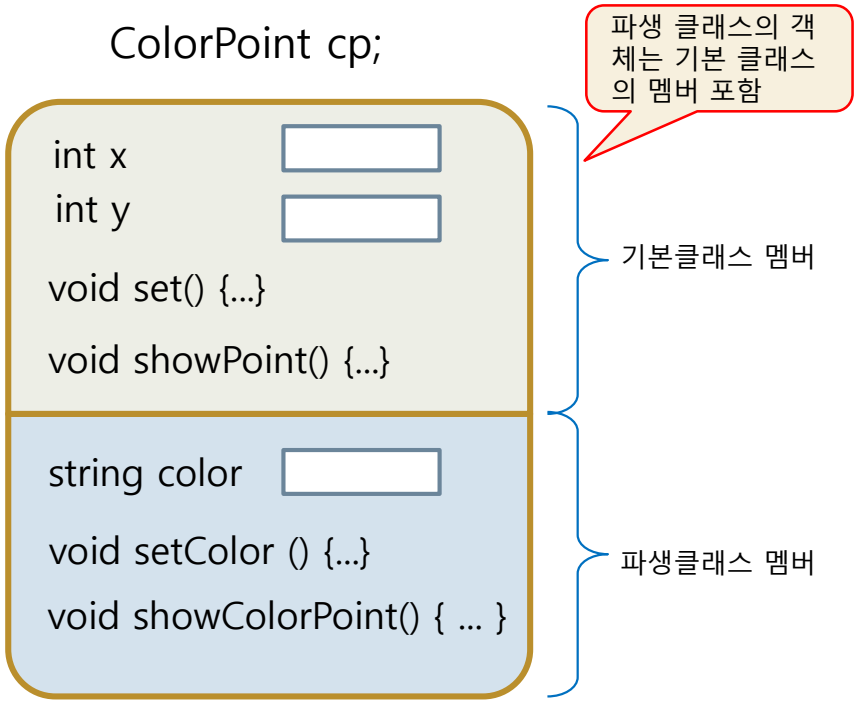

```
class Point {
   int x, y; // 한 점 (x,y) 좌표 값
public:
   void set(int x, int y);
   void showPoint();
};
```

```
class ColorPoint : public Point { // Point를 상속받음
   string color; // 점의 색 표현
public:
   void setColor(string color);
   void showColorPoint();
};
```

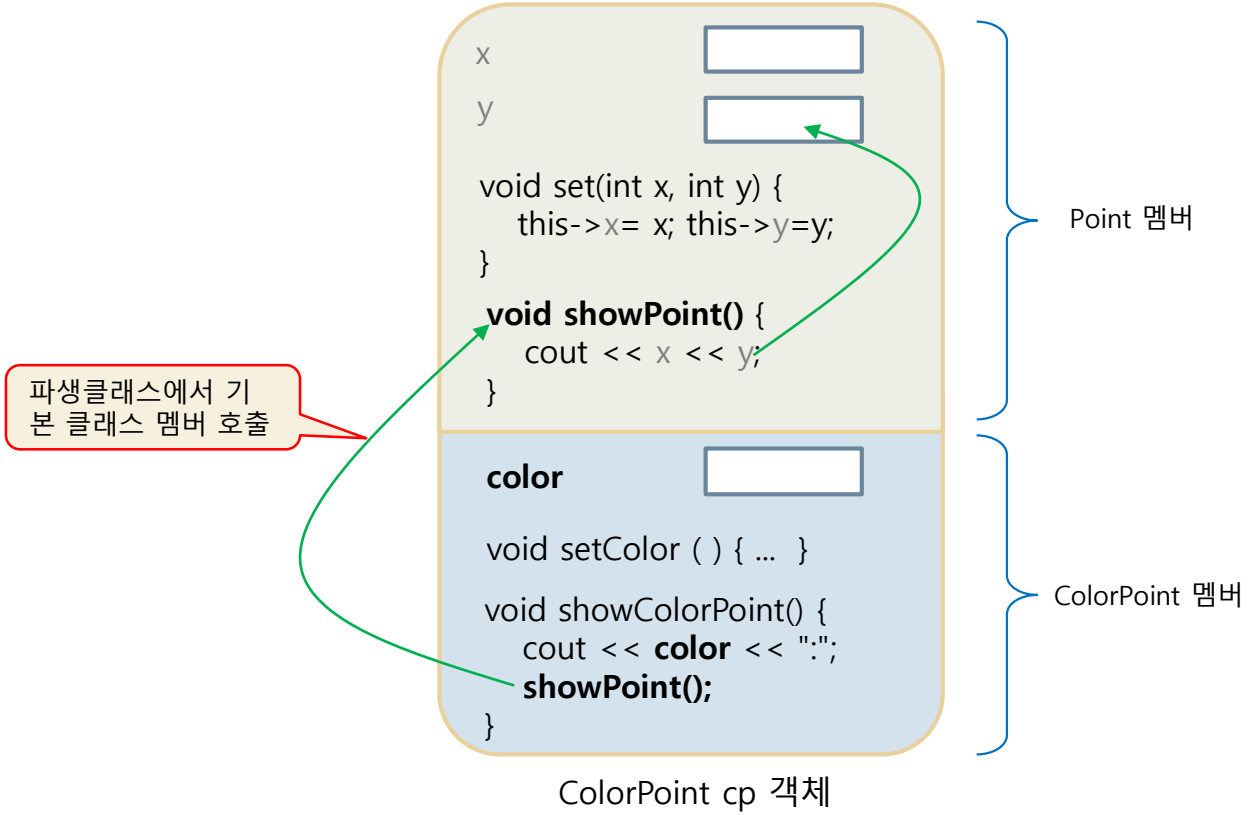
Point p;



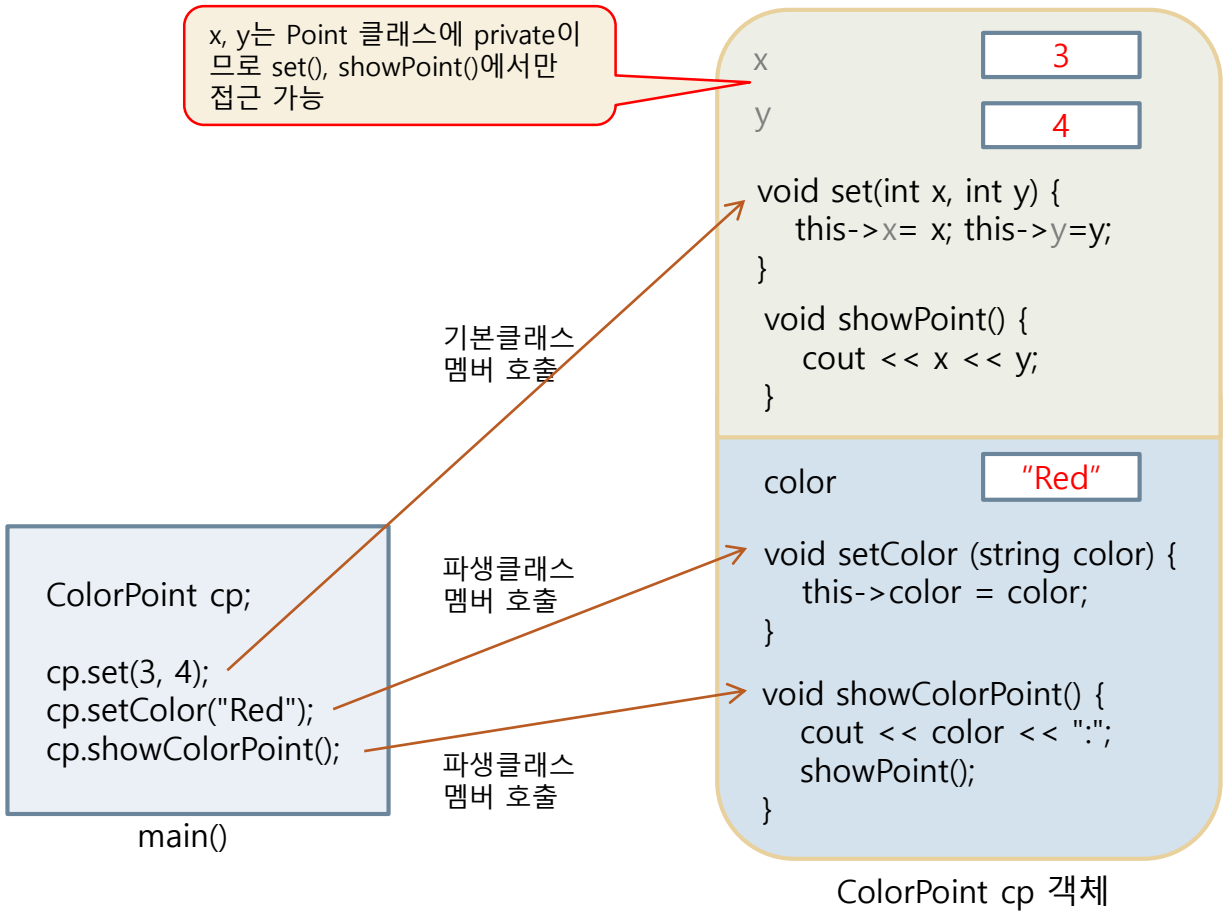
ColorPoint cp;



파생 클래스에서 기본 클래스 멤버 접근



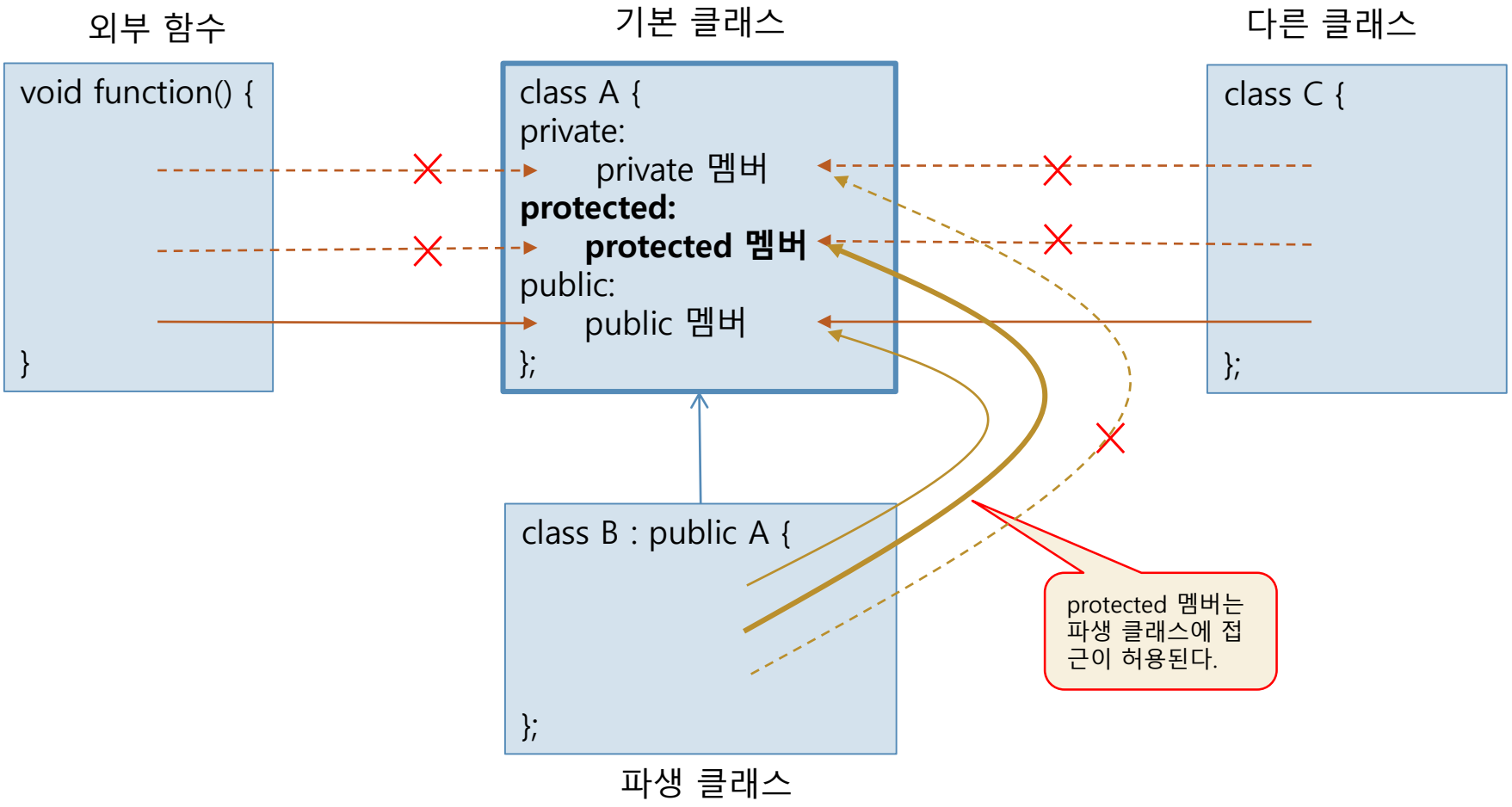
외부에서 파생 클래스 객체에 대한 접근



protected 접근 지정

- 접근 지정자
  - private 멤버
    - 선언된 클래스 내에서만 접근 가능
    - 파생 클래스에서도 기본 클래스의 private 멤버 직접 접근 불가
  - public 멤버
    - 선언된 클래스나 외부 어떤 클래스, 모든 외부 함수에 접근 허용
    - 파생 클래스에서 기본 클래스의 public 멤버 접근 가능
  - protected 멤버
    - 선언된 클래스에서 접근 가능
    - 파생 클래스에서만 접근 허용
      - 파생 클래스가 아닌 다른 클래스나 외부 함수에서는 *protected* 멤버를 접근할 수 없다.

멤버의 접근 지정에 따른 접근성



상속하는 클래스의 객체 생성

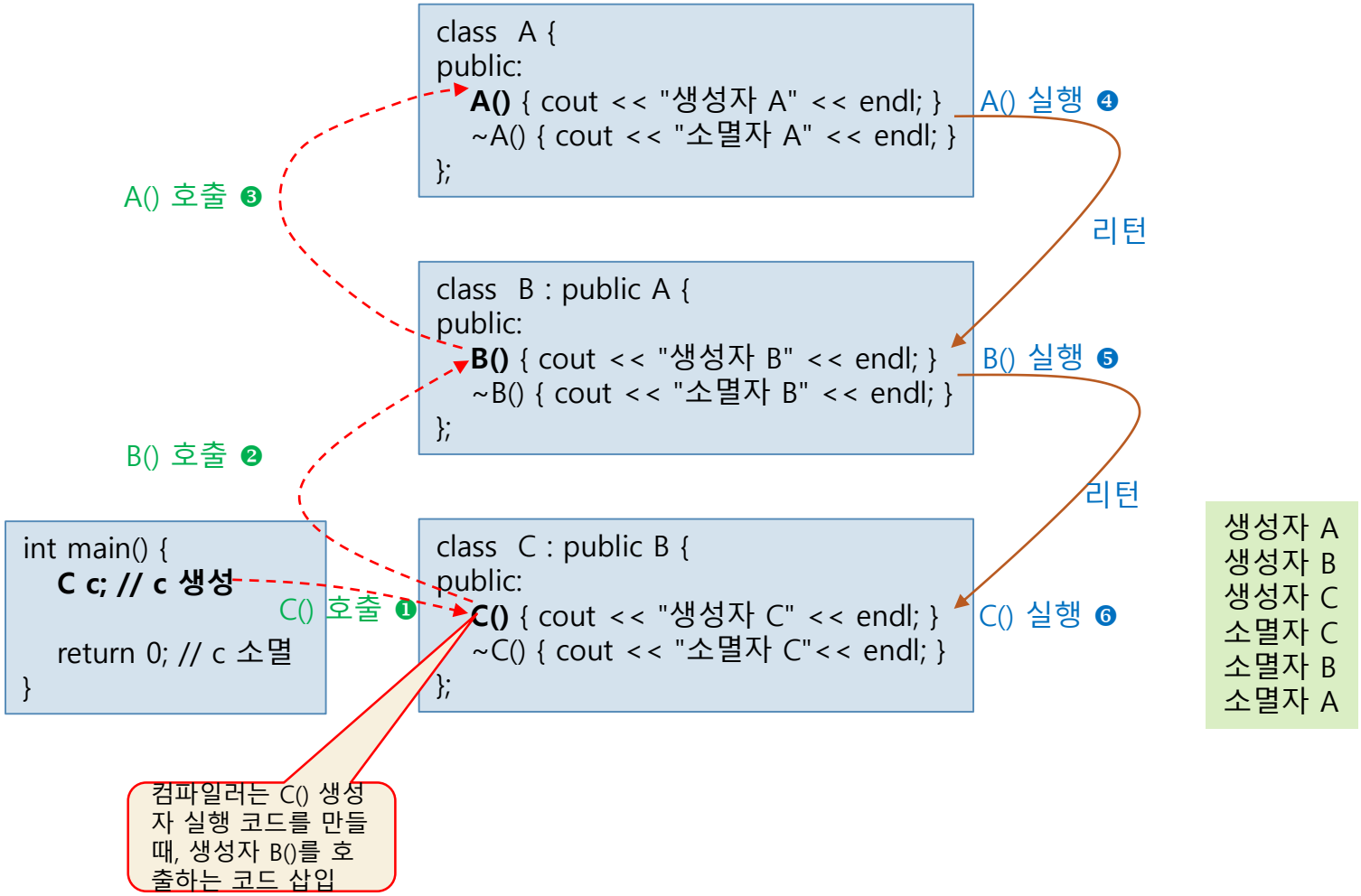
- 상속하는 클래스의 객체 생성

  - 메모리 공간 할당
    (상속되는 클래스를 감안하여 공간할당)

  - 기반클래스 생성자 호출

  - 파생클래스 생성자 호출

생성자 호출 관계 및 실행 순서

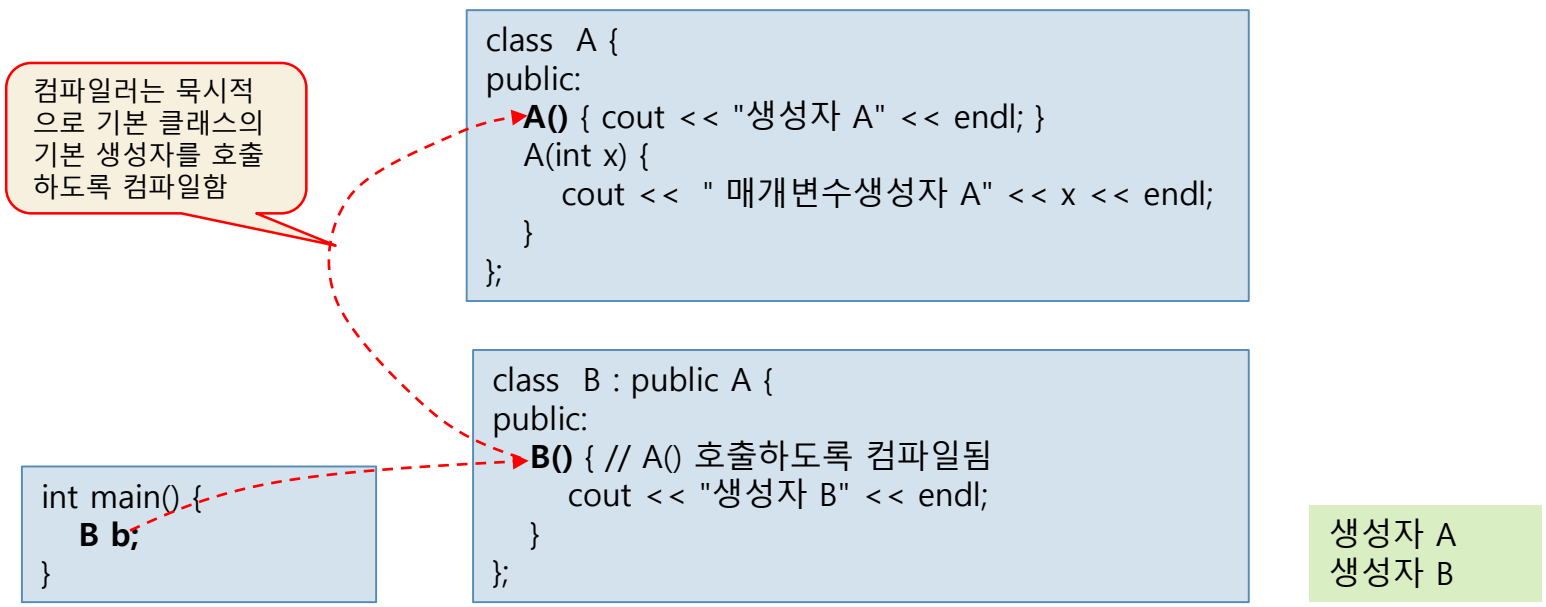


소멸자의 실행 순서

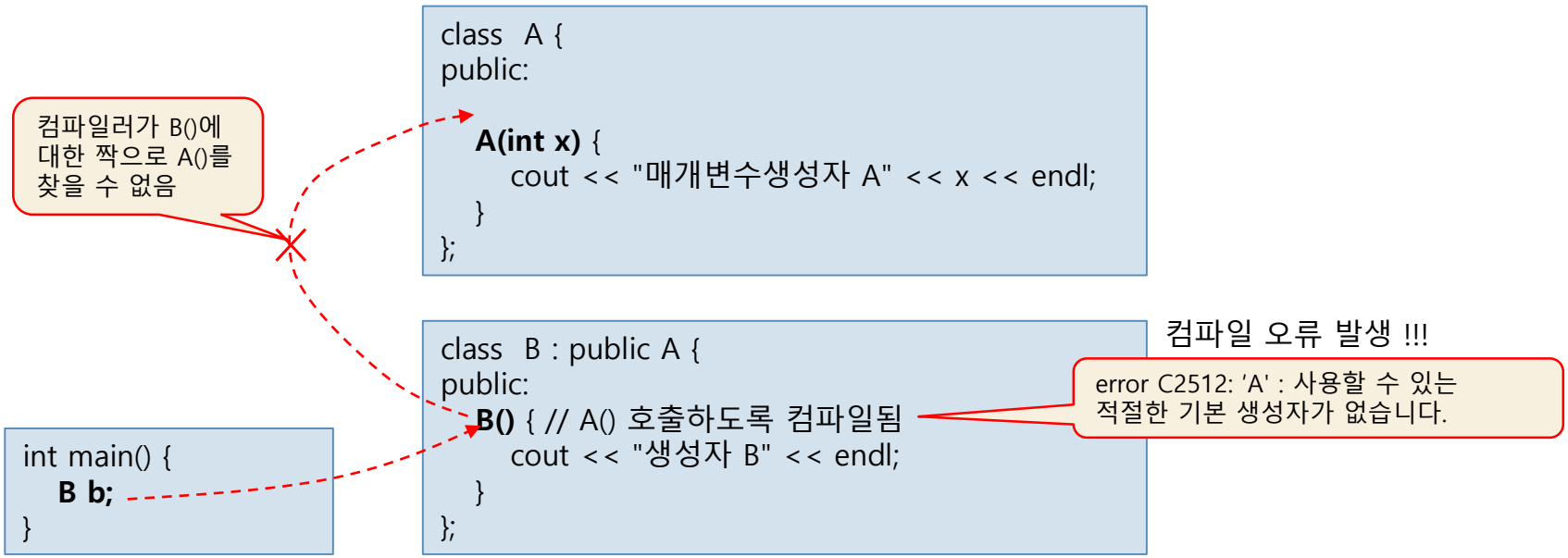
- 파생 클래스의 객체가 소멸될 때
  - 파생 클래스의 소멸자가 먼저 실행되고
  - 기본 클래스의 소멸자가 나중에 실행

컴파일러에 의해 묵시적으로 기본 클래스의 생성자 자동 호출

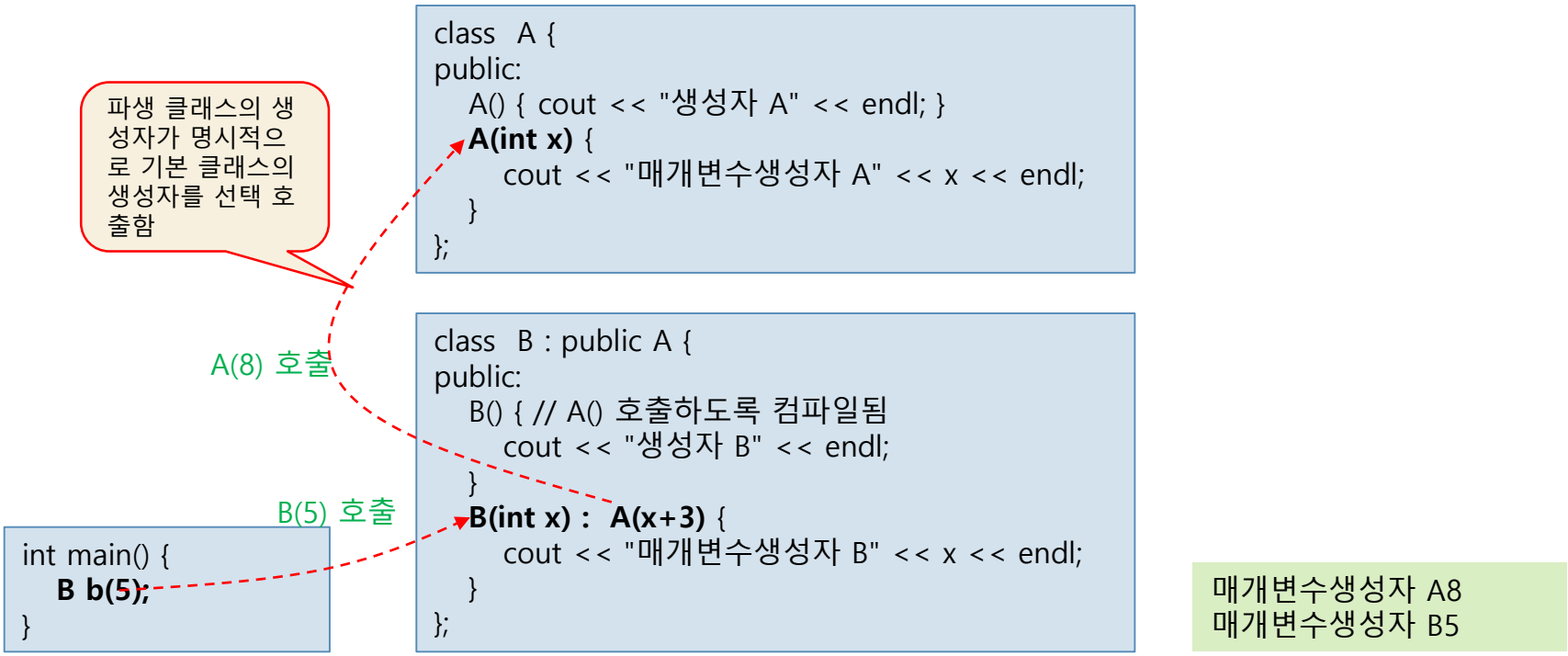
파생 클래스의 생성자에서 기본 클래스의 기본 생성자 자동 호출



기본 클래스에 기본 생성자가 없는 경우



파생 클래스의 생성자에서 명시적으로 기본 클래스의 생성자 선택



컴파일러의 기본 생성자 호출 코드 삽입

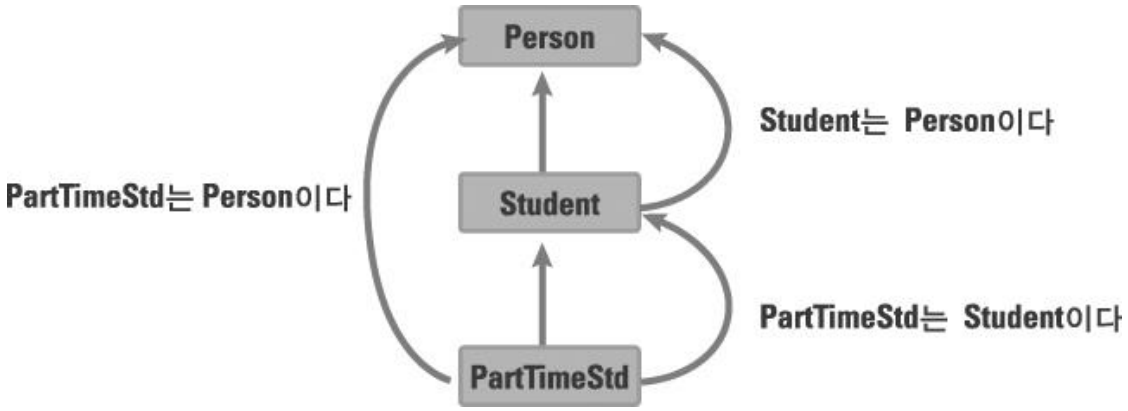

```
class B {
   B() : A() {
      cout << "생성자 B" << endll;
   }

   B(int x) : A() {
      cout << "매개변수생성자 B" << x << endll;
   }
};
```

컴파일러가 묵시적으로 삽입한 코드

컴파일러가 묵시적으로 삽입한 코드

상속된 객체와 포인터 관계

- 객체 포인터
  - 객체의 주소 값을 저장할 수 있는 포인터
  - AAA 클래스의 포인터는 AAA 객체 뿐만 아니라, AAA 클래스를 상속하는 Derived 클래스 객체의 주소 값도 저장 가능



상속된 객체와 포인터 관계

- **객체 포인터의 권한**
  - **포인터를 통해서 접근할 수 있는 객체 멤버의 영역.**
  - **AAA 클래스의 객체 포인터는 가리키는 대상에 상관없이 AAA 클래스 내에 선언된 멤버에만 접근**

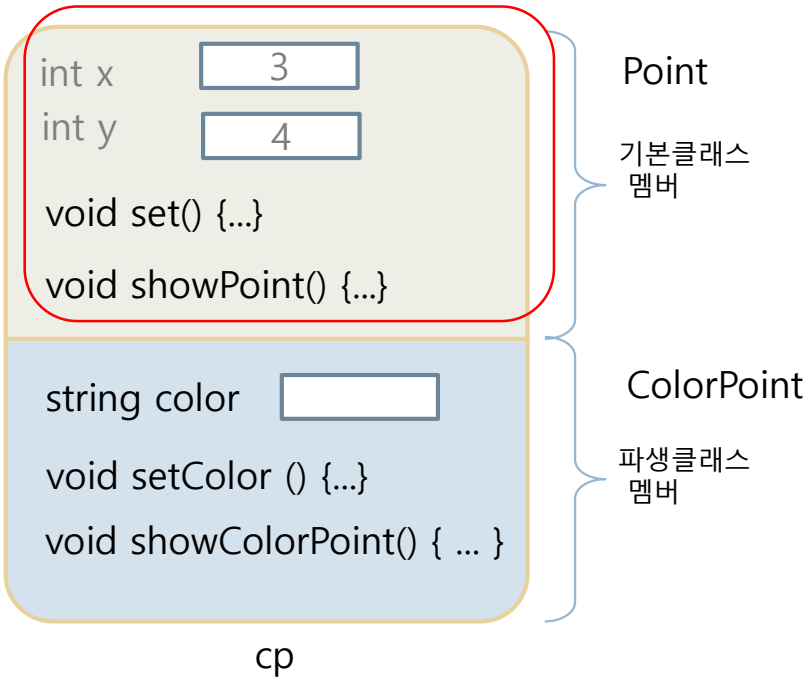
- **기반 클래스의 포인터는 파생클래스의 오브젝트의 주소를 저장할 수 있다(UpCase). 그러나 파생클래스의 포인터는 기반클래스의 오브젝트 주소를 저장할 수 없다(DownCast).**

- **기반클래스의 포인터가 파생클래스의 오브젝트를 가리킬 때 그 포인터를 통해 멤버함수를 호출하면, 기반클래스로 부터 상속받은 멤버나, 멤버함수만 호출할 수 있다.**

상속과 객체 포인터 – 업 캐스팅

- 업 캐스팅(up-casting)
  - 파생 클래스 포인터가 기본 클래스 포인터에 치환되는 것
    - 예) 사람을 동물로 봄

```
ColorPoint cp;
ColorPoint *pDer = &cp;
Point* pBase = pDer; // 업캐스팅
```



상속된 객체와 참조 관계

- **객체 레퍼런스의 권한**
  - **객체를 참조하는 레퍼런스의 권한**
  - **클래스 포인터의 권한과 일치!**

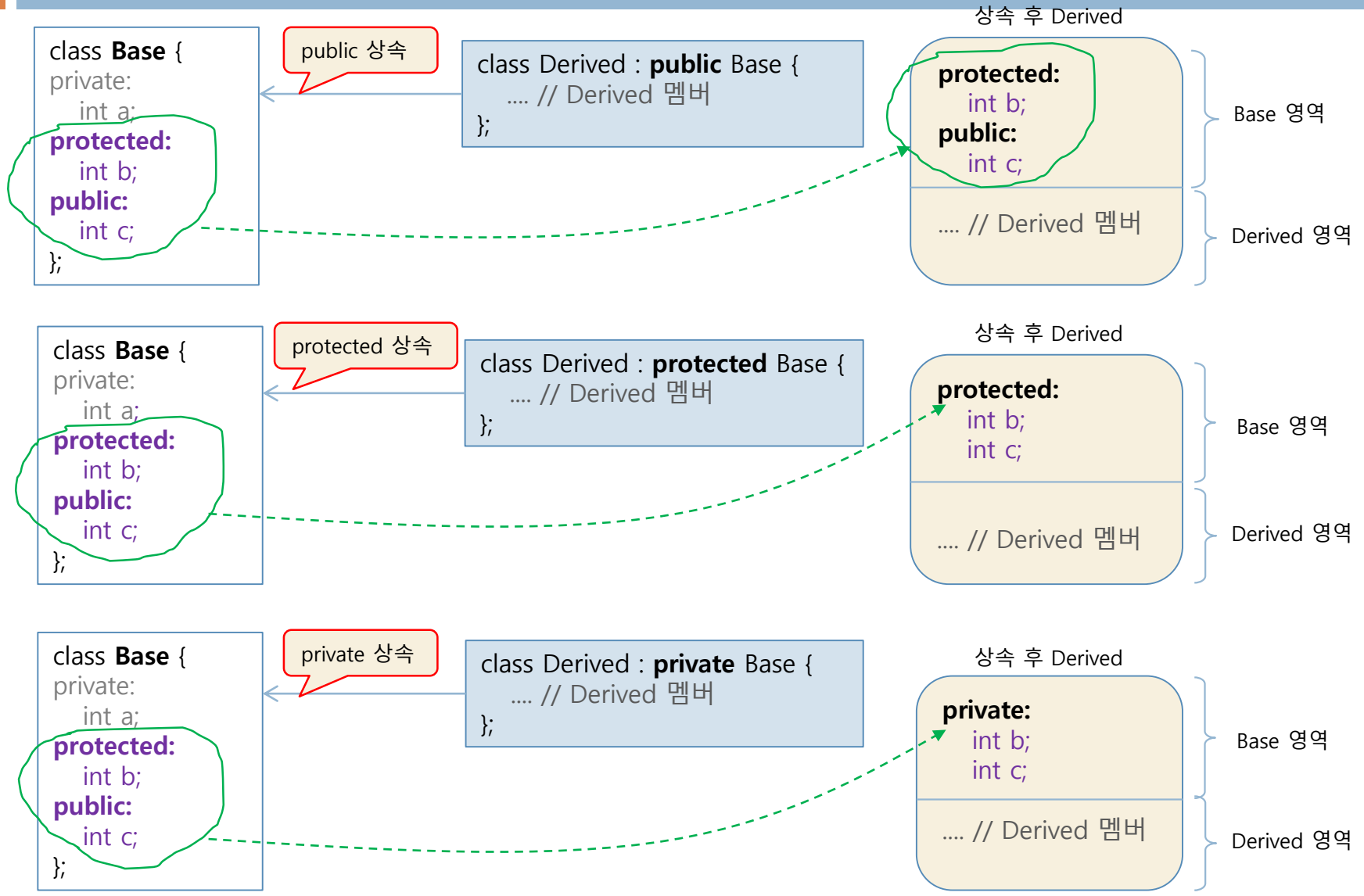
- A 클래스의 레퍼런스를 참조하는 대상이 어떠한 객체이건,

  A 클래스 타입내에 선언된 멤버와 상속된 클래스에만 접근이 가능하다.

상속 지정

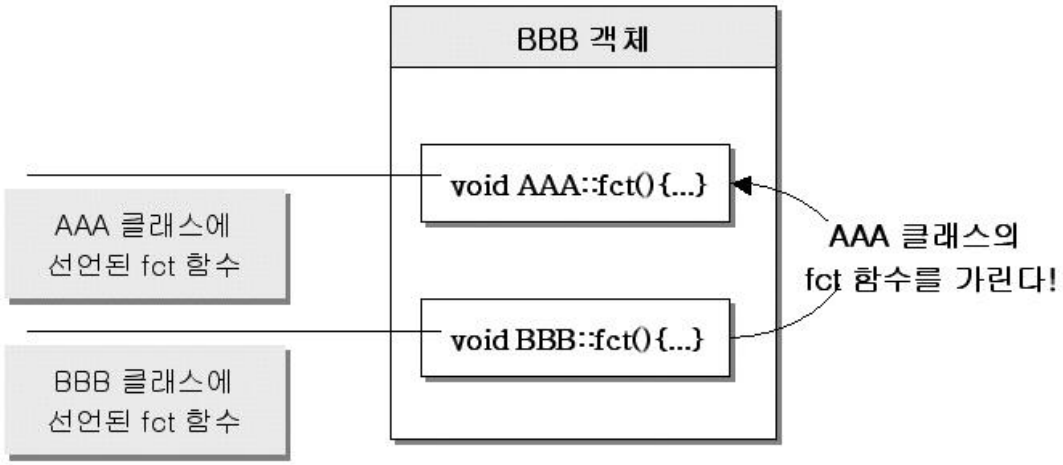
- 상속 지정
  - 상속 선언 시 public, private, protected의 3가지 중 하나 지정
  - 기본 클래스의 멤버의 접근 속성을 어떻게 계승할지 지정
    - *public* – 기본 클래스의 *protected, public* 멤버 속성을 그대로 계승
    - *private* – 기본 클래스의 *protected, public* 멤버를 *private*으로 계승
    - *protected* – 기본 클래스의 *protected, public* 멤버를 *protected*로 계승

상속 시 접근 지정에 따른 멤버의 접근 지정 속성 변화



함수 재정의

- 오버라이딩(Overriding)의 이해
  - Base 클래스에 선언된 멤버와 같은 형태의 멤버를 Derived 클래스에서 선언
  - Base 클래스의 멤버를 가리는 효과!
  - 보는 시야(Pointer)에 따라서 달라지는 효과!



오버라이딩(Overriding)의 이해

- **** 함수 오버라이딩(Overriding) : 함수 재정의**

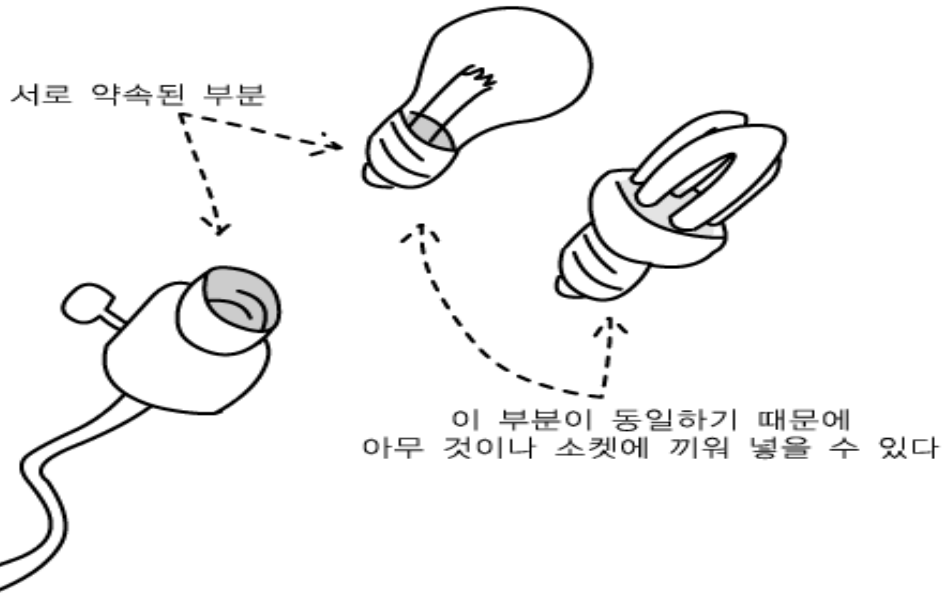
- 함수 재정의는 기반클래스의 멤버함수와 원형이 같은 함수를 파생클래스에서 다시 만들 수 있는 기능이다. **재정의한 멤버함수는 기반클래스의 멤버함수와 함수명, 인자, 리턴형이 모두 같아**야 한다.

- **이때 파생클래스에서 재정의한 멤버함수에 대해 기반클래스의 멤버함수는 숨겨진다**.

- 따라서 기반클래스의 멤버함수에 접근하려면 범위연산자와 함께 기반클래스를 명시적으로 써 주어야 한다.

다형성(Polymorphism)(1)

- 다형성이란 서로 다른 객체를 동일한 방식으로 명령을 내릴 수 있는 성질을 말한다. 이 때 서로 다른 객체들은 같은 명령을 받지만 제각기 다른 방식으로 명령을 수행할 수 있다.



  - 인터페이스는 하나지만 서로 다른 기능을 처리할 수 있다.
  - 프로그래밍 언어에서는 가상함수를 통하여 다형성을 구현한다.
  - 다형성은 상속관계에서만 이루어 진다.

Dynamic Binding(가상함수, Virtual)

- 사용된 오브젝트에 의해 실행코드가 결정되는 함수로 지정자 virtual 을 갖는다. **vitual은 선언에 둔다**.(가상함수)

- 정적결합 : 함수의 실행코드 결정이 컴파일 시에 이루어 진다.
  실행시 속도는 빠르지만 융통성이 적다. 일반 함수 사용.

- 동적결합 : 함수의 실행코드 결정이 실행 시에 이루어 진다.
  실행 시 속도는 느리지만 융통성이 많다. 가상 함수 사용.
  가상함수를 사용하면 실행속도 면에서는 손해를 보지만
  실행 시 다형성(runtime polymorphism)이라는 융통성을
  갖게 된다

Dynamic Binding(가상함수, Virtual)

- 가상함수(Virtual) : 실행시 파생클래스 주소로 변경

- vtable : 가상함수를 포함한 클래스의 오브젝트를 생성하면 vtable(virtual function table)이 함께 만들어진다.

vtable은 가상함수에 대한 주소값이 들어있는 포인터 배열이다.

- 파생클래스의 오브젝트는 기반클래스의 오브젝트로 부터 vtable을 물려받아 수정, 확장한다. 만일 파생클래스에서 재정의한 가상함수가 있으면 **기반클래스의 가상함수에 대한 포인터가 들어 있던 곳에 파생클래스에서 재정의한 가상함수에 대한 포인터를 넣는다**. 따라서 파생클래스에서 수정한 vtable을 사용하게 되므로 파생클래스에서 재정의된 함수가 호출된다.

추상클래스(Abstract class)

- **추상클래스(Abstract class)**란 **순수 가상 함수를 포함하고** 있는 클래스를 가르켜 추상클래스라 한다. 추상 클래스는 순수가상 함수를 통해 인터페이스를 제공, **파생클래스에서 그 인터페이스를 구현**한다.

- 추상클래스의 **오브젝트를 생성하는 것은 불가능하다**. 그러나 추상클래스의 포인터를 선언하는 것은 가능하다. 추상클래스의 포인터는 흔히 파생클래스의 오브젝트를 가리키게 된다. **파생클래스의 오브젝트를 가리킬 때 그 포인터를 통해 가상함수를 호출하면 파생클래스에서 재정의한 함수가 호출**된다.

- 순수가상 함수 재정의 문제 : **순수가상 함수는 사용하기 전에 반드시 재정의** 되어야 한다. 만일 기반클래스가 순수가상 함수를 포함하고 있는데, 파생클래스에서 재정의 하지 않으면 파생클래스도 추상 클래스가 된다. 그러면 기반클래스 뿐만 아니라 파생클래스의 오브젝트를 생성하는 것이 불가능하다 .

순수(pure) 가상 함수와 추상 클래스

- 추상 클래스: 순수 가상 함수 지니는 클래스
- 추상 클래스는 객체화될 수 없다.

```
class Employee
{
protected:
    char name[20];
public:
     const char* GetName();     // 일반 메서드
•    virtual int GetPay()=0;       // 순수 가상 함수
};
```

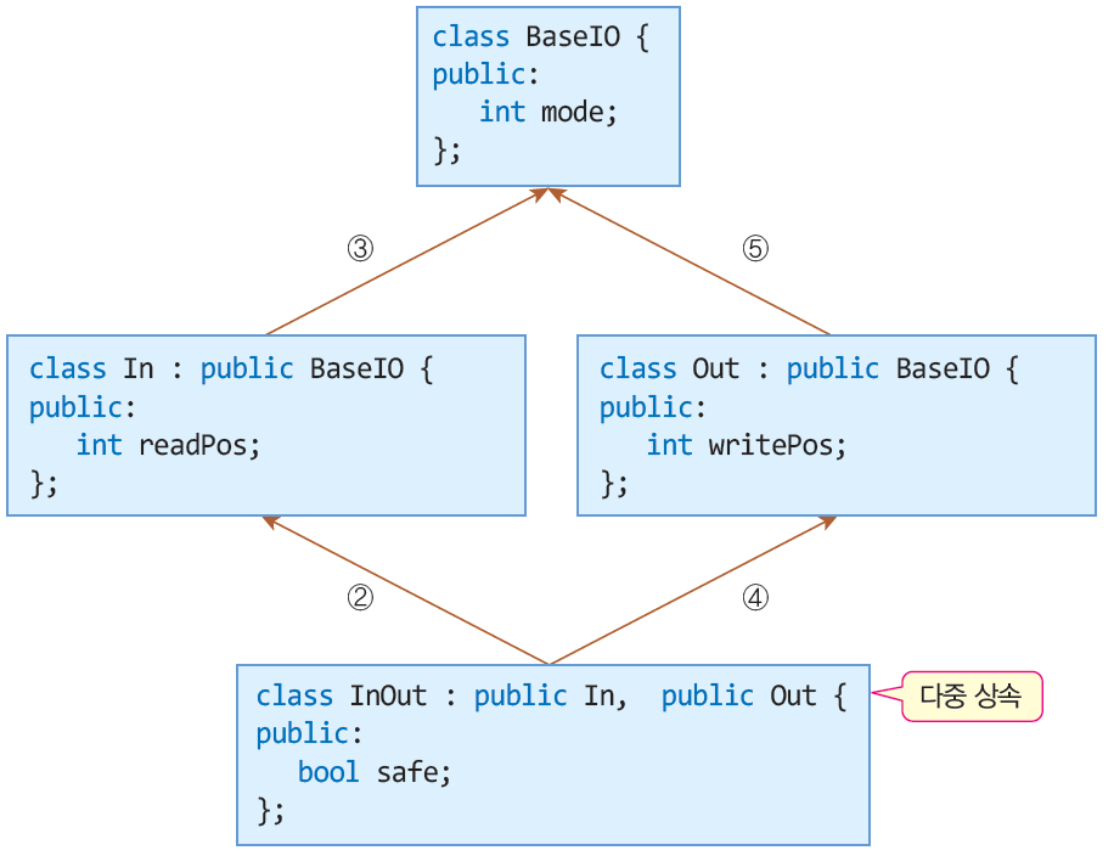
가상 상속(다중 상속을 받을 때)

- 다중 상속으로 인한 기본 클래스 멤버의 중복 상속 해결
- 가상 상속
  - 파생 클래스의 선언문에서 기본 클래스 앞에 **virtual**로 선언
  - 파생 클래스의 객체가 생성될 때 기본 클래스의 멤버는 오직 한 번만 생성
    - 기본 클래스의 멤버가 중복하여 생성되는 것을 방지

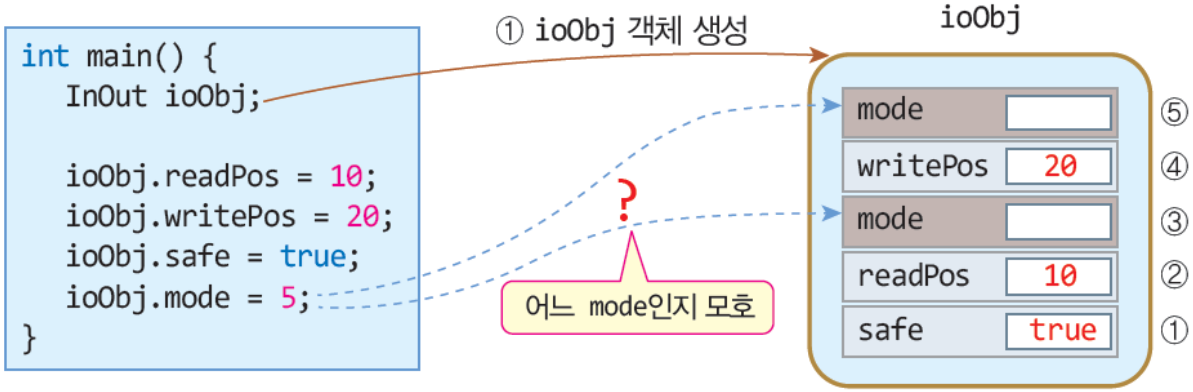
```
class In : virtual public BaseIO { // In 클래스는 BaseIO 클래스를 가상 상속함
...
};

class Out : virtual public BaseIO { // Out 클래스는 BaseIO 클래스를 가상 상속함
...
};
```

다중 상속의 문제점 - 기본 클래스 멤버의 중복 상속

- Base의 멤버가 이중으로 객체에 삽입되는 문제점.
- 동일한 x를 접근하는 프로그램이 서로 다른 x에 접근하는 결과를 낳게되어 잘못된 실행 오류가 발생된다.



(a) 클래스 상속 관계



(b) ioObj 객체 생성 과정 및 객체 내부

가상 상속으로 다중 상속의 모호성 해결