10. 상속

10.1 상속 개념

10.2 파생 클래스 정의 및 객체 생성 방법

10.3 파생 클래스의 생성자와 소멸자

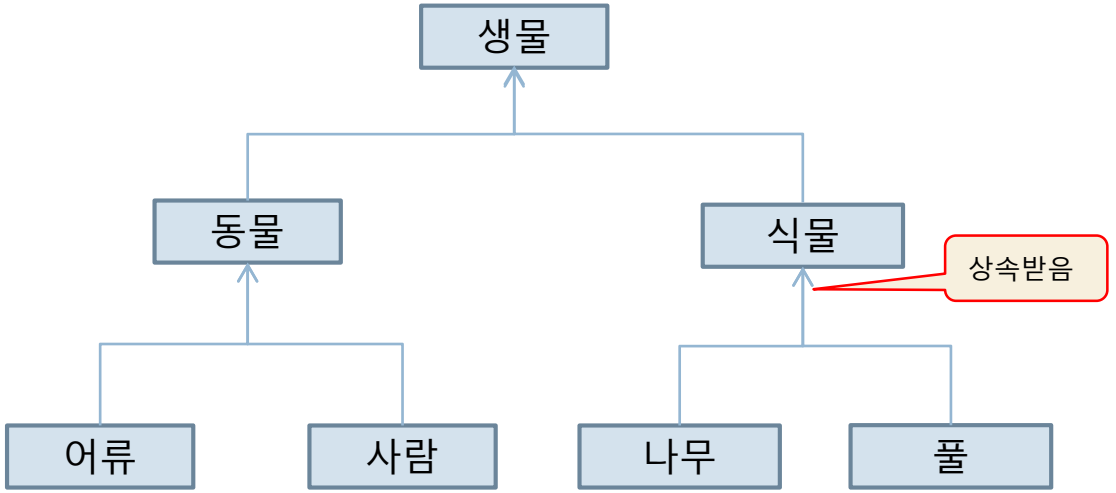
10.4 접근 지정자와 접근 변경자

10.5 다중 상속

10.1 상속 개념

유전적 상속과 객체 지향 상속

유전적 상속 : 객체 지향 상속



유전적 상속과 관계된 생물 분류

C++에서의 상속(Inheritance)

- C++에서의 상속이란?
  - **클래스 사이에서 상속 관계 정의**
    - 객체 사이에는 상속 관계 없음
  - **기존의 클래스가 가진 기능을 이어받아서 새로운 클래스를 정의**하는 것
    - 기본 클래스 : 상속을 해주는 클래스, 일반적인 특징을 제공
    - 파생 클래스 : 상속을 받는 클래스, 일반적인 특징+구체적인 특징
      - 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

다른 클래스를 상속 받아서 새로운 클래스를 정의할 때, 기존의 클래스에 구현되어 있는 기능은 새로운 클래스에 다시 구현할 필요가 없다.

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

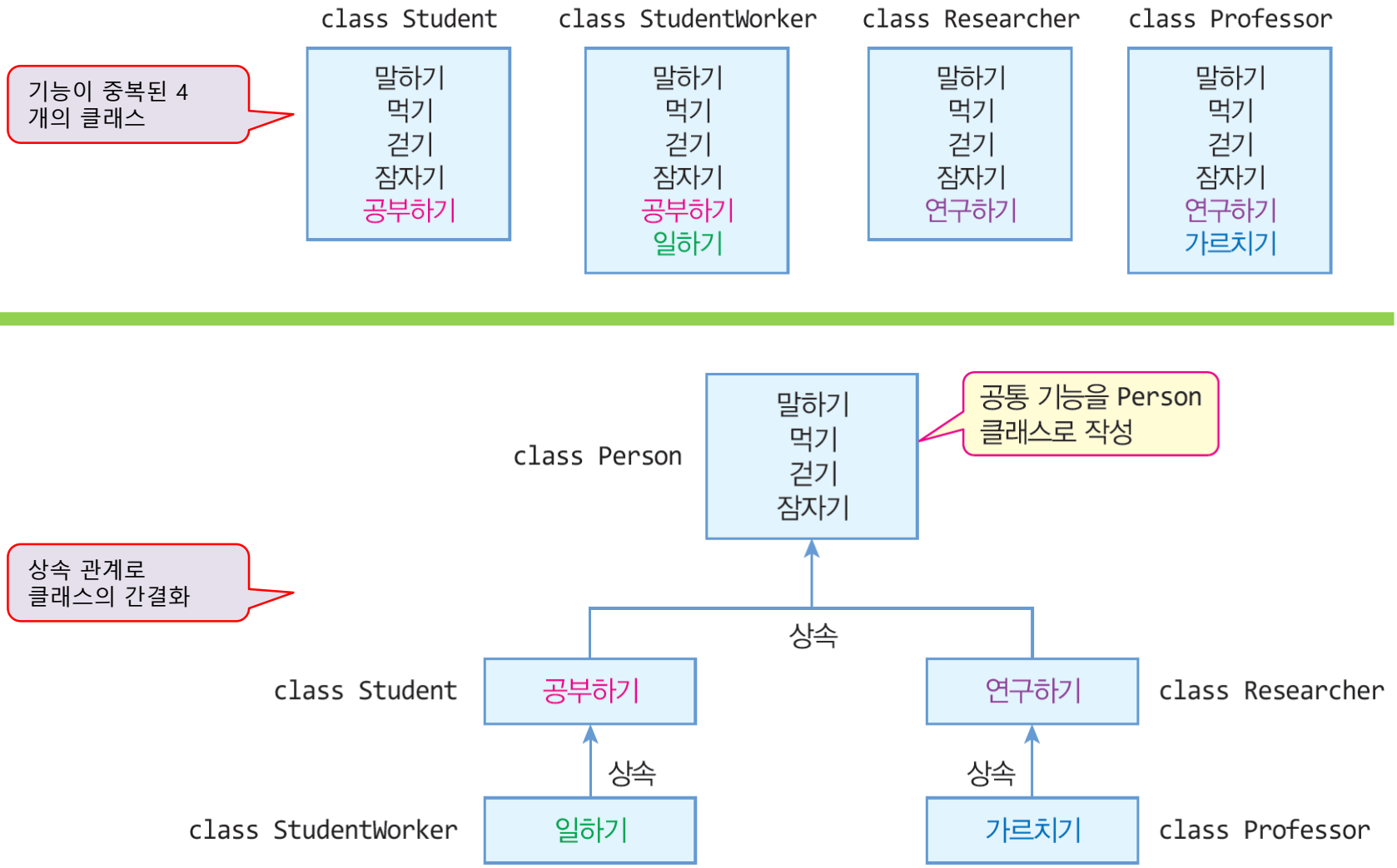
휴대 전화기

음악 기능
전화기

상속의 목적 및 장점

- 간결한 클래스 작성
  - 기본 클래스의 기능을 물려받아 파생 클래스를 간결하게 작성
- 클래스 간의 계층적 분류 및 관리의 용이함
  - 상속은 클래스들의 구조적 관계 파악 용이
- 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 관계로 클래스의 간결화 사례



10.2 파생 클래스 정의 및 객체의 생성 방법

상속 선언

- 파생 클래스의 정의
  - 클래스 이름 다음에 콜론(:)을 쓰고 public 키워드와 함께 기본 클래스 이름을 적어줌

```
class 파생클래스이름 : 접근변경자 기본클래스이름
{
};
```

- 파생 클래스의 멤버
  - **상속받는 멤버 변수는 다시 정의하지 않음**
  - 새로 추가된 멤버 변수나 멤버 함수 정의
  - **상속 받은 멤버 함수 중에서 기본 클래스와 다르게 처리할 멤버 함수를 재정의(overriding)**

Point 클래스를 상속받는 ColorPoint 클래스 만들기

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

// 2차원 평면에서 한 점을 표현하는 클래스 Point 선언
class Point {
   int x, y; //한 점 (x,y) 좌표값
public:
   void set(int x, int y) { this->x = x; this->y = y; }
   void showPoint() {
      cout << "(" << x << "," << y << ")" << endl;
   }
};
```

```
class ColorPoint : public Point { // 2차원 평면에서 컬러 점을 표현하는 클래스 ColorPoint. Point를 상속받음
   string color;// 점의 색 표현
public:
   void setColor(string color)  { this->color = color; }
   void showColorPoint();
};

void ColorPoint::showColorPoint() {
   cout << color << ":";
   showPoint(); // Point의 showPoint() 호출
}

int main() {
   Point p; // 기본 클래스의 객체 생성
   ColorPoint cp; // 파생 클래스의 객체 생성
   cp.set(3,4); // 기본 클래스의 멤버 호출
   cp.setColor("Red"); // 파생 클래스의 멤버 호출
   cp.showColorPoint(); // 파생 클래스의 멤버 호출
}
```

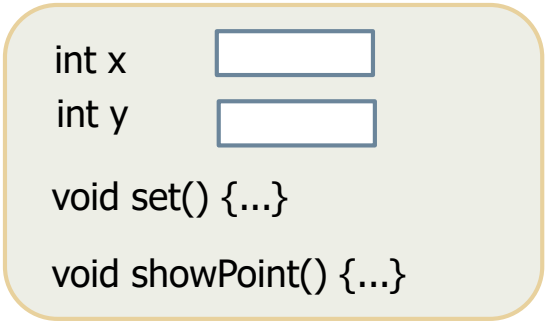
```
Red:(3,4)
```

파생 클래스의 객체 구성

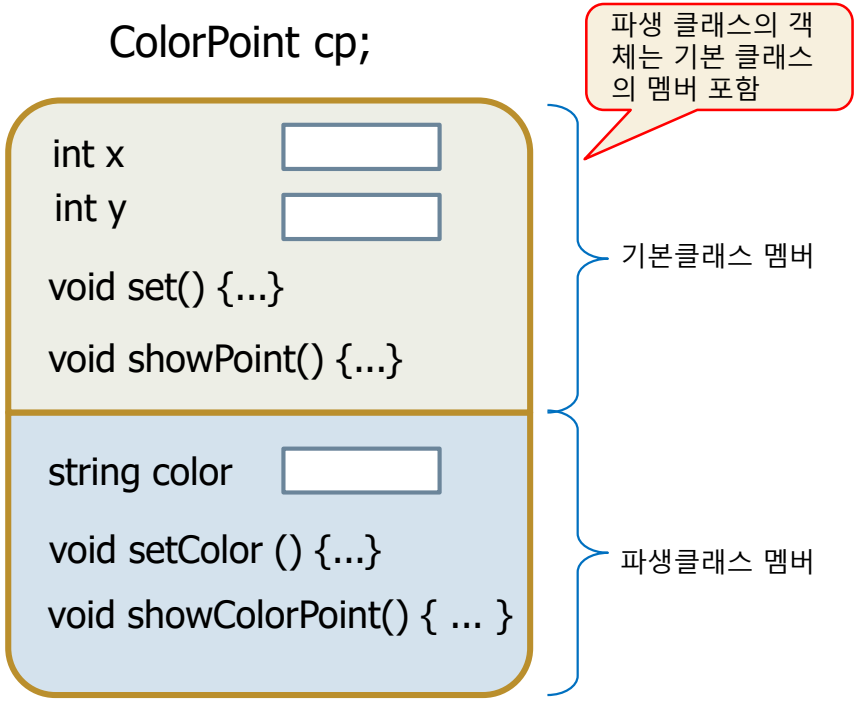
```
class Point {
   int x, y; // 한 점 (x,y) 좌표 값
public:
   void set(int x, int y);
   void showPoint();
};
```

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

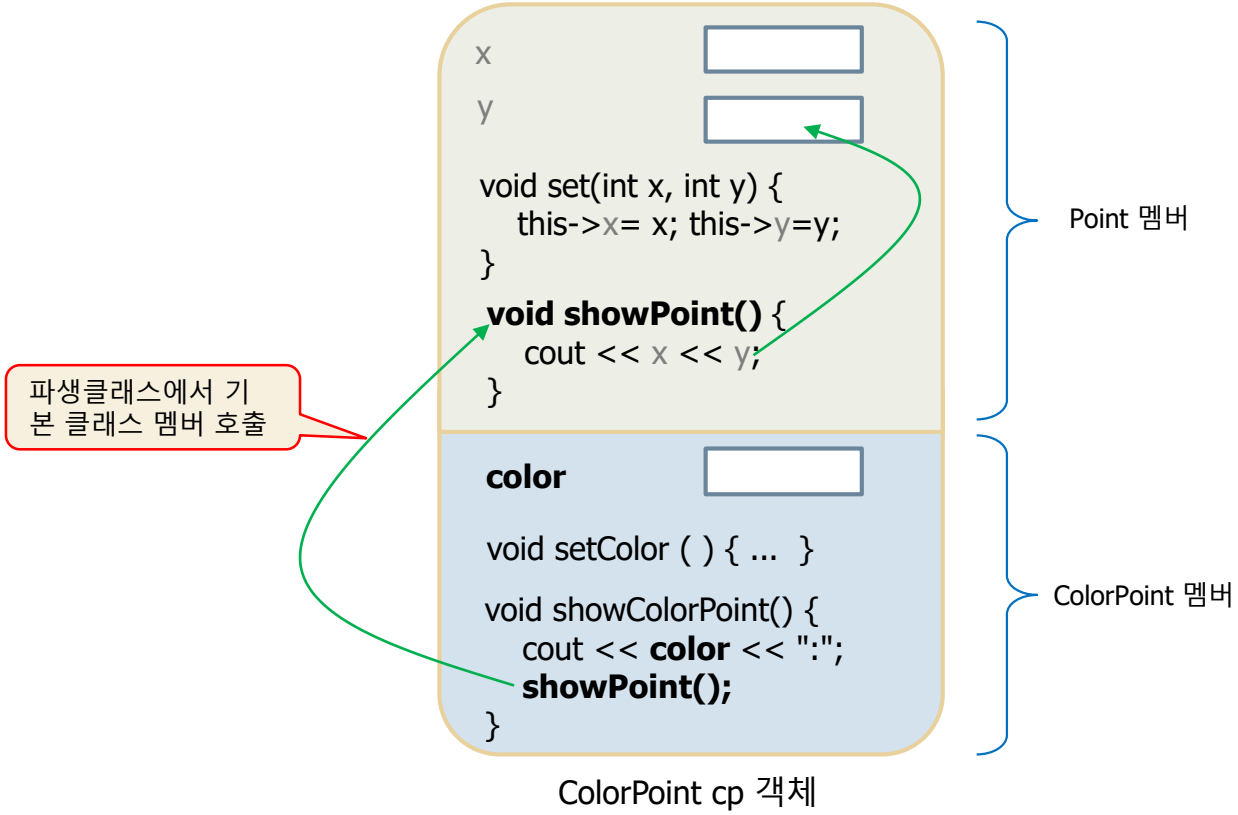
Point p;



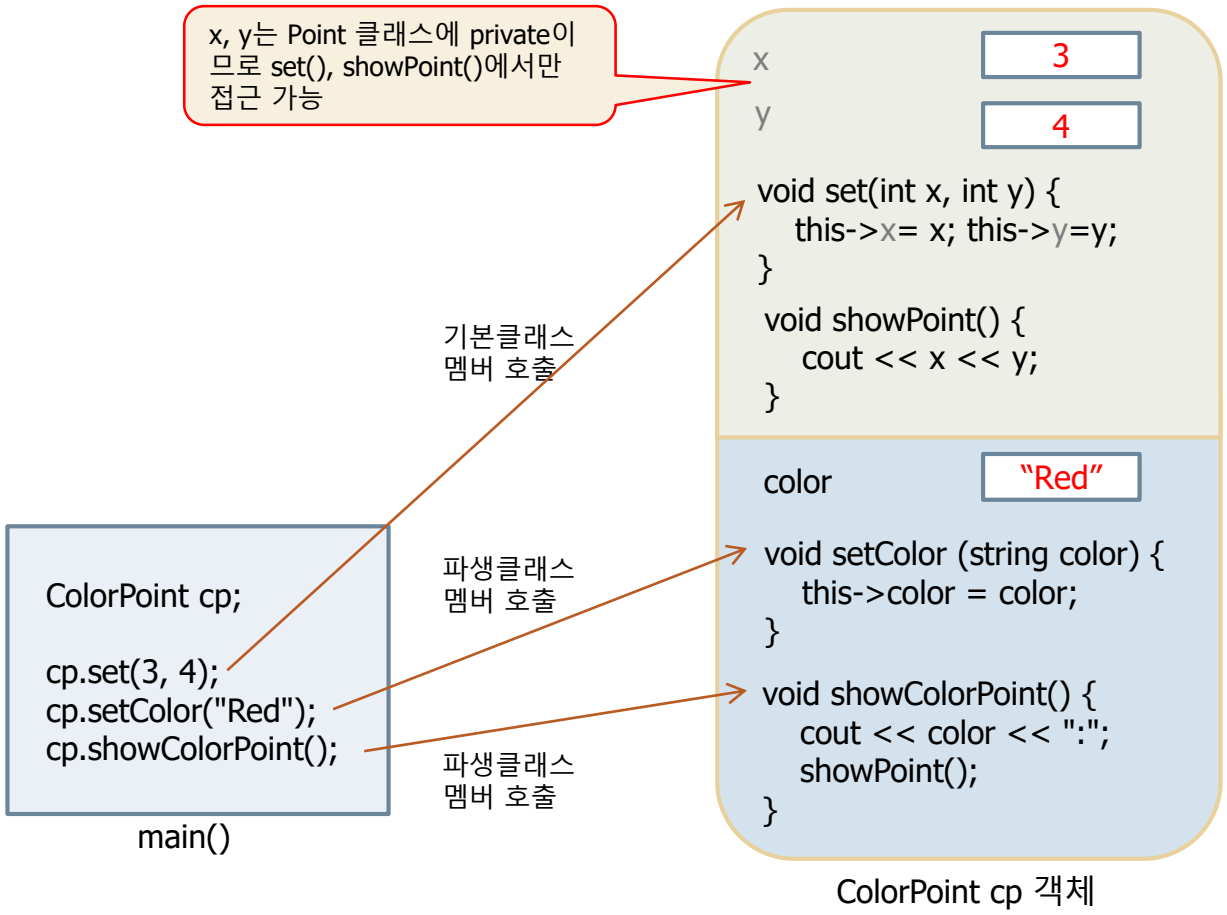
ColorPoint cp;



파생 클래스에서 기본 클래스 멤버 접근

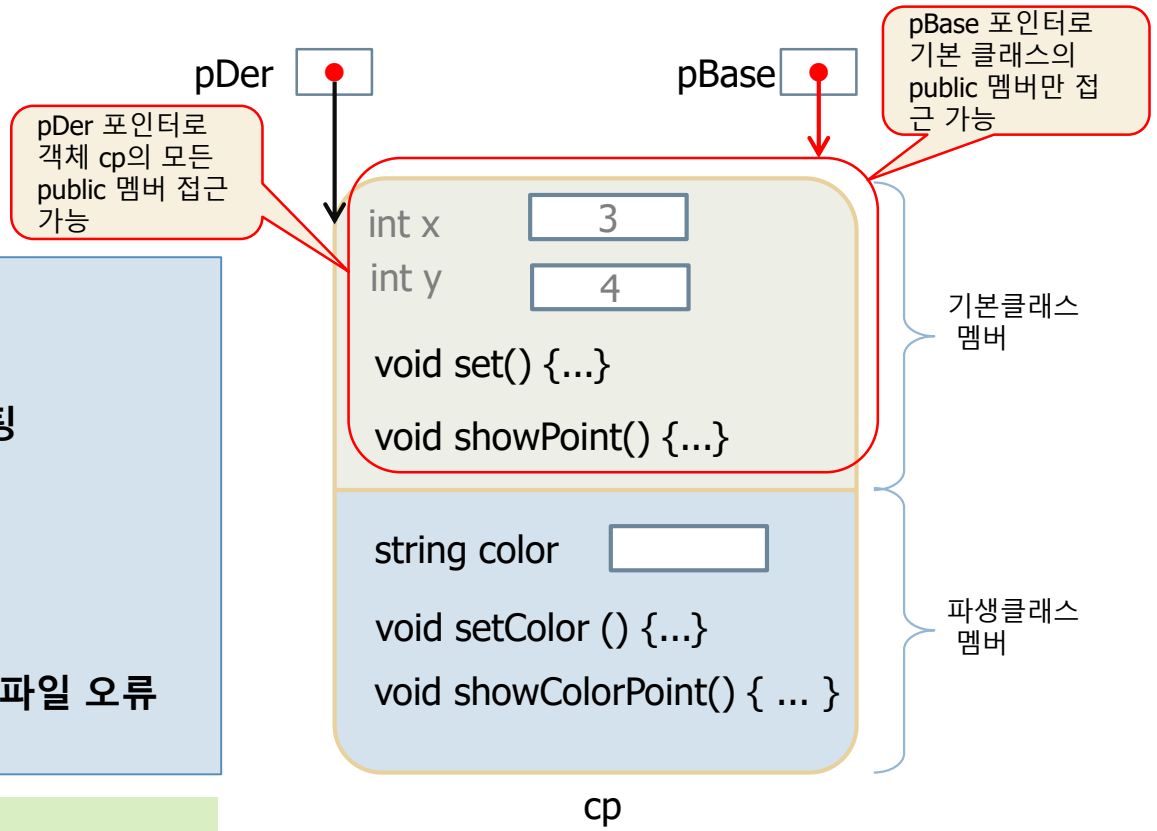


외부에서 파생 클래스 객체에 대한 접근



상속과 객체 포인터 – 업 캐스팅

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

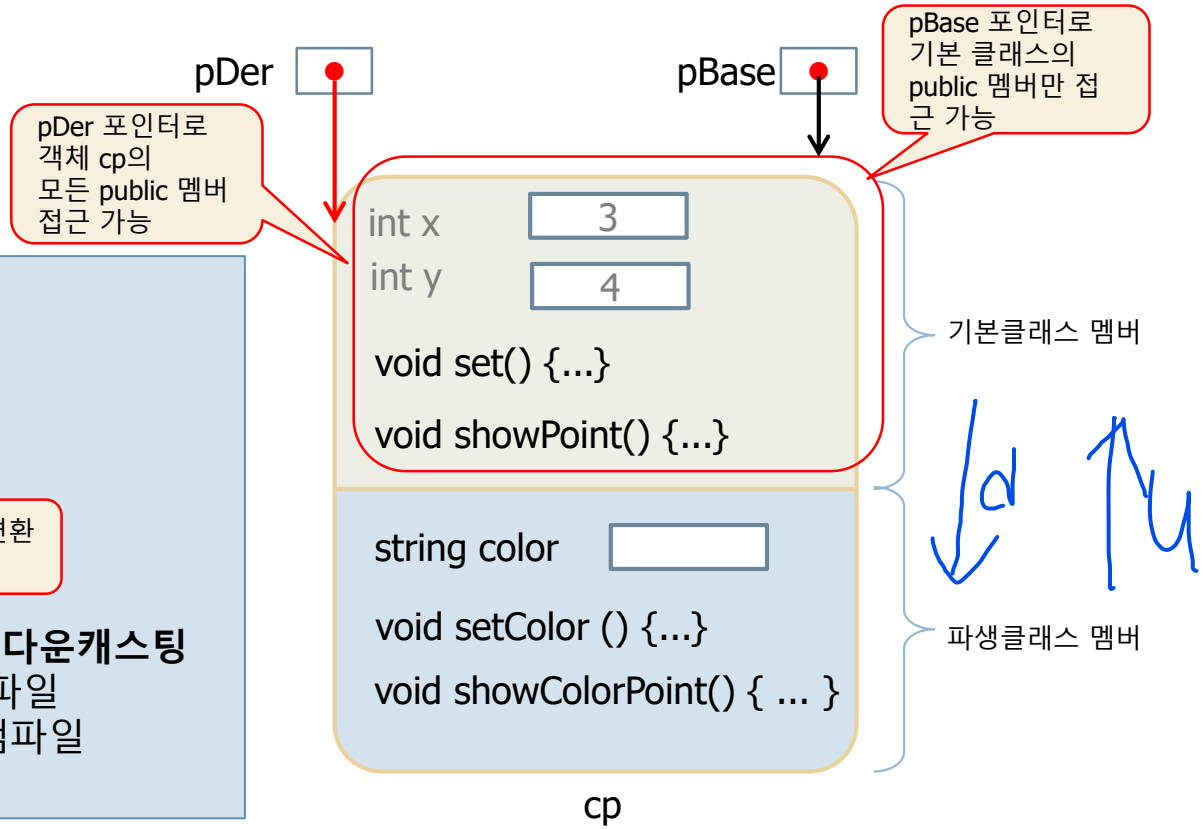



```
int main() {
   ColorPoint cp;
   ColorPoint *pDer = &cp;
   Point* pBase = pDer; // 업캐스팅

   pDer→set(3,4);
   pBase→showPoint();
   pDer→setColor("Red");
   pDer→showColorPoint();
   pBase→showColorPoint(); // 컴파일 오류
}
```

```
(3,4)
Red(3,4)
```

상속과 객체 포인터 – 다운 캐스팅

- 다운 캐스팅(down-casting) 강제
  - 기본 클래스의 포인터가 파생 클래스의 포인터에 치환되는 것




```
int main() {
   ColorPoint cp;
   ColorPoint *pDer;
   Point* pBase = &cp; // 업캐스팅

   pBase→set(3,4);
   pBase→showPoint();

   pDer = (ColorPoint *)pBase; // 다운캐스팅
   pDer→setColor("Red"); // 정상 컴파일
   pDer→showColorPoint(); // 정상 컴파일
}
```

강제 타입 변환 반드시 필요

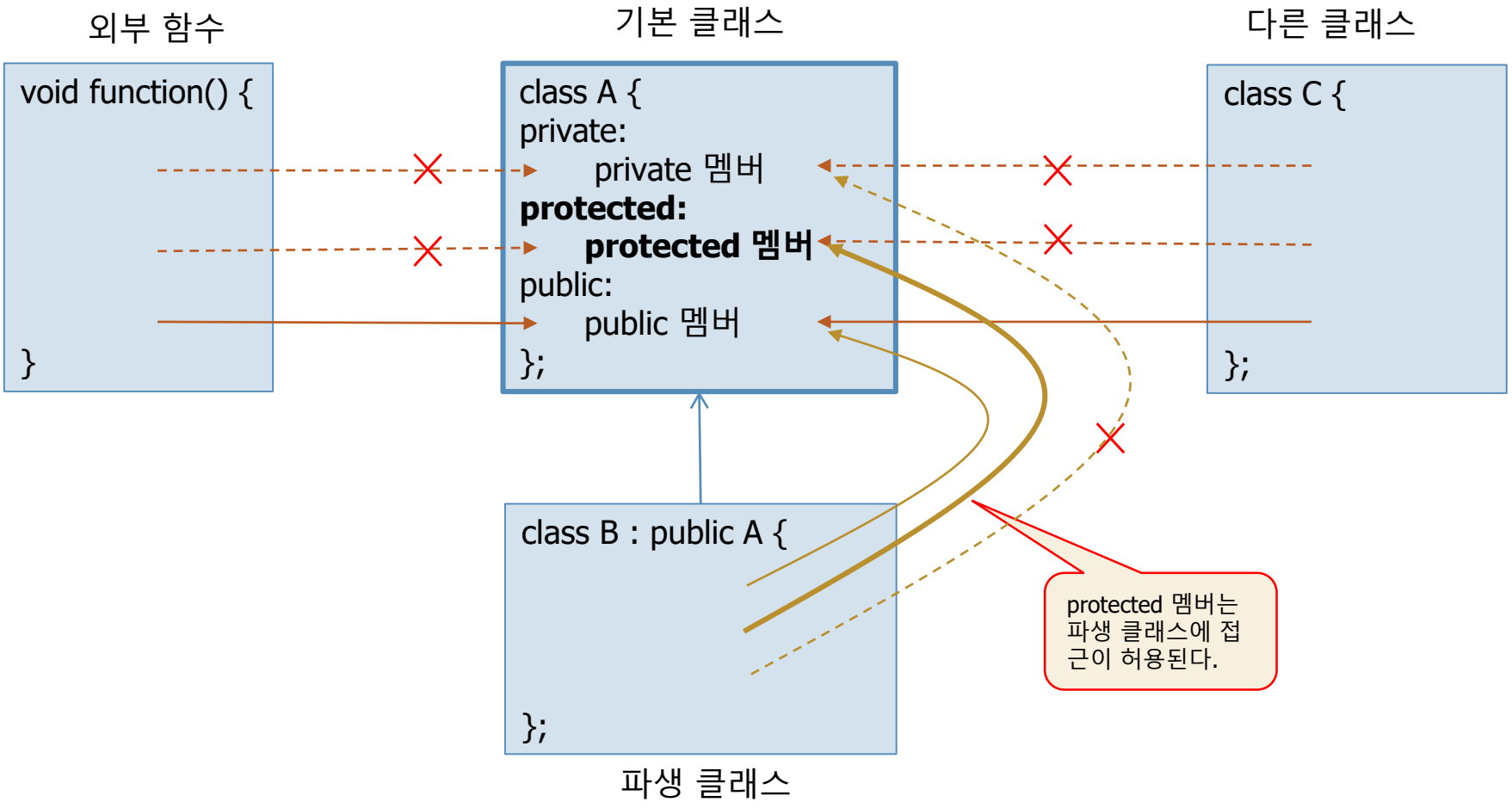
```
(3,4)
Red(3,4)
```

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

이거 중요



protected 멤버에 대한 접근

public만 접근 가능,,,

protected, private X

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

class Point {
protected:
    int x, y; //한 점 (x,y) 좌표값
public:
    void set(int x, int y);
    void showPoint();
};
void Point::set(int x, int y) {
    this→x = x;
    this→y = y;
}
void Point::showPoint() {
    cout << "(" << x << "," << y << ")" << endl;
}

class ColorPoint : public Point {
    string color;
public:
    void setColor(string color);
    void showColorPoint();
    bool equals(ColorPoint p);
};

void ColorPoint::setColor(string color) {
    this→color = color;
}
```

```
void ColorPoint::showColorPoint() {
    cout << color << ":";
    showPoint(); // Point 클래스의 showPoint() 호출
}

bool ColorPoint::equals(ColorPoint p) {
    if (x == p.x && y == p.y && color == p.color) // ①
        return true;
    else
        return false;
}

int main() {
    Point p; // 기본 클래스의 객체 생성
    p.set(2,3);                                    // ②
    p.x = 5;                                       // ③
    p.y = 5;                                       // ④
    p.showPoint();

    ColorPoint cp; // 파생 클래스의 객체 생성
    cp.x = 10;                                     // ⑤
    cp.y = 10;                                     // ⑥
    cp.set(3,4);
    cp.setColor("Red");
    cp.showColorPoint();

    ColorPoint cp2;
    cp2.set(3,4);
    cp2.setColor("Red");
    cout << ((cp.equals(cp2))?"true":"false"); // ⑦
}
```

파생 클래스에서 protected는 접근 가능
private -> 내 함수(선언된 클래스) 안에서만 접근 가능

오류

오류

오류

오류

10.3 파생클래스의 생성자 및 소멸자

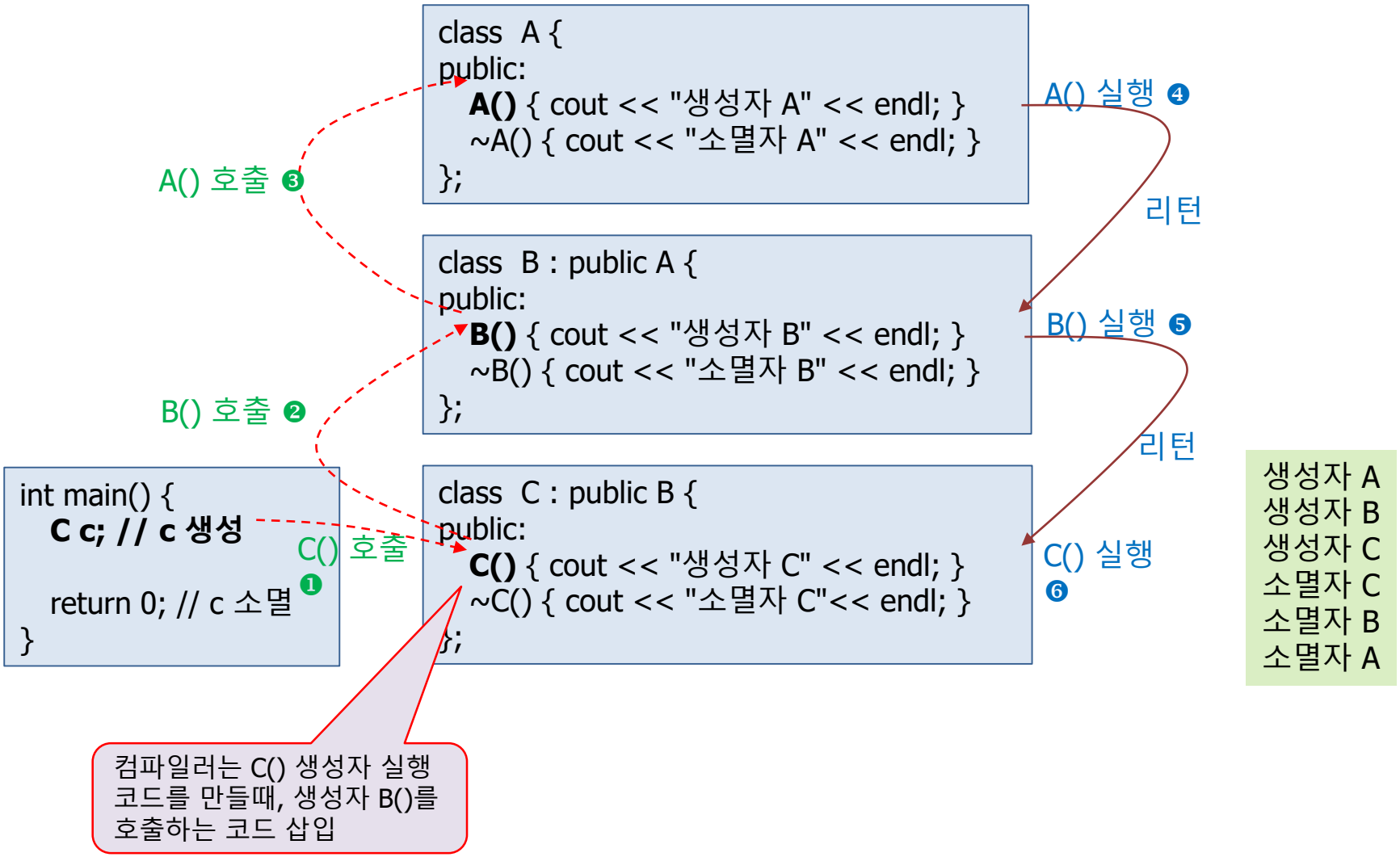
상속 관계의 생성자와 소멸자 실행

- 질문 1
  - 파생 클래스의 객체가 생성될 때 파생 클래스의 생성자와 기본 클래스의 생성자가 모두 실행되는가? 아니면 파생 클래스의 생성자만 실행되는가?
    - 답 - 둘 다 실행된다.

- 질문 2
  - 파생 클래스의 생성자와 기본 클래스의 생성자 중에서 어떤 생성자가 먼저 실행되는가?
    - 답 - 기본 클래스의 생성자가 먼저 실행된 후 파생 클래스의 생성자가 실행된다.

생성자 호출 관계 및 실행 순서

A생 B생 C생, A소 B소 C소 순 실행됨

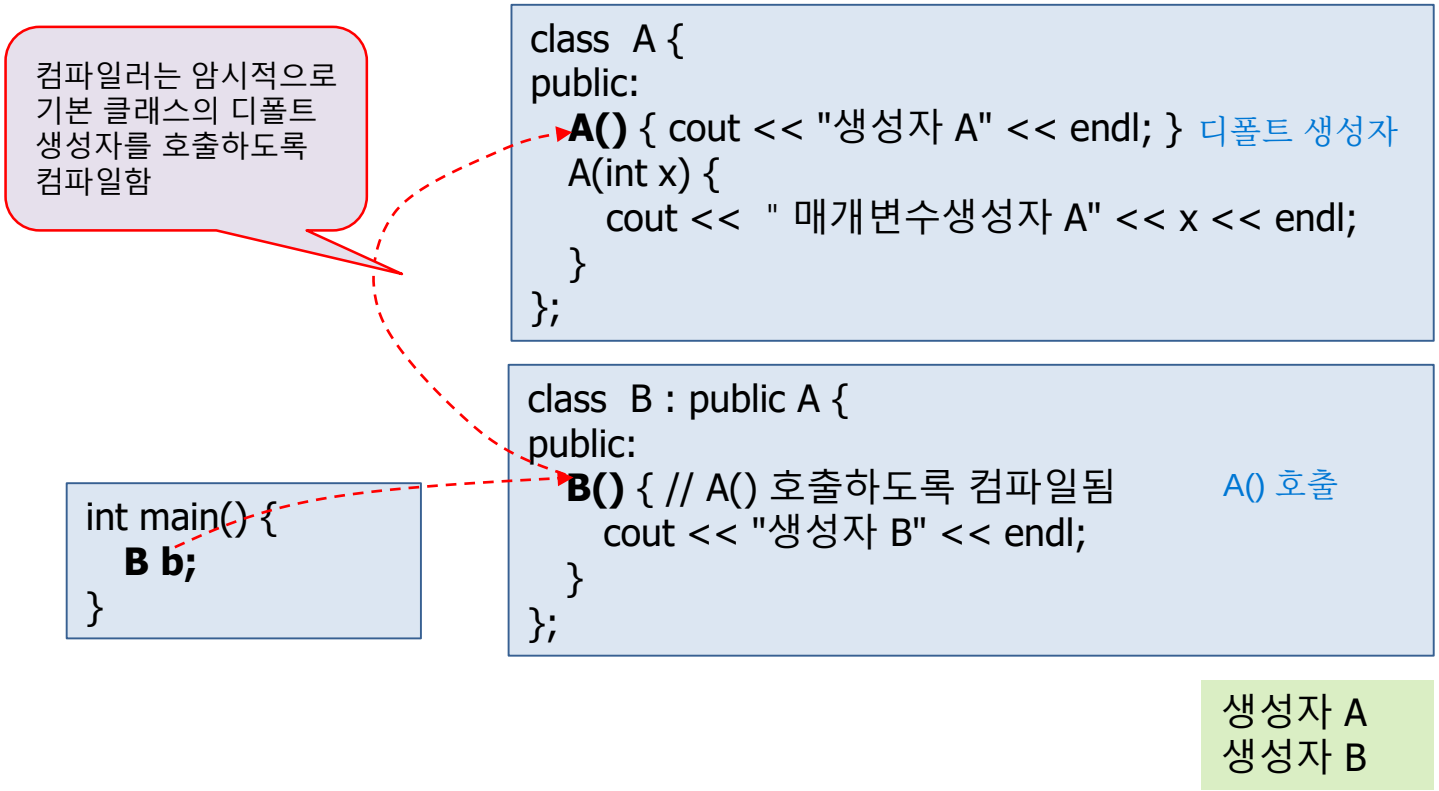


소멸자의 실행 순서

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

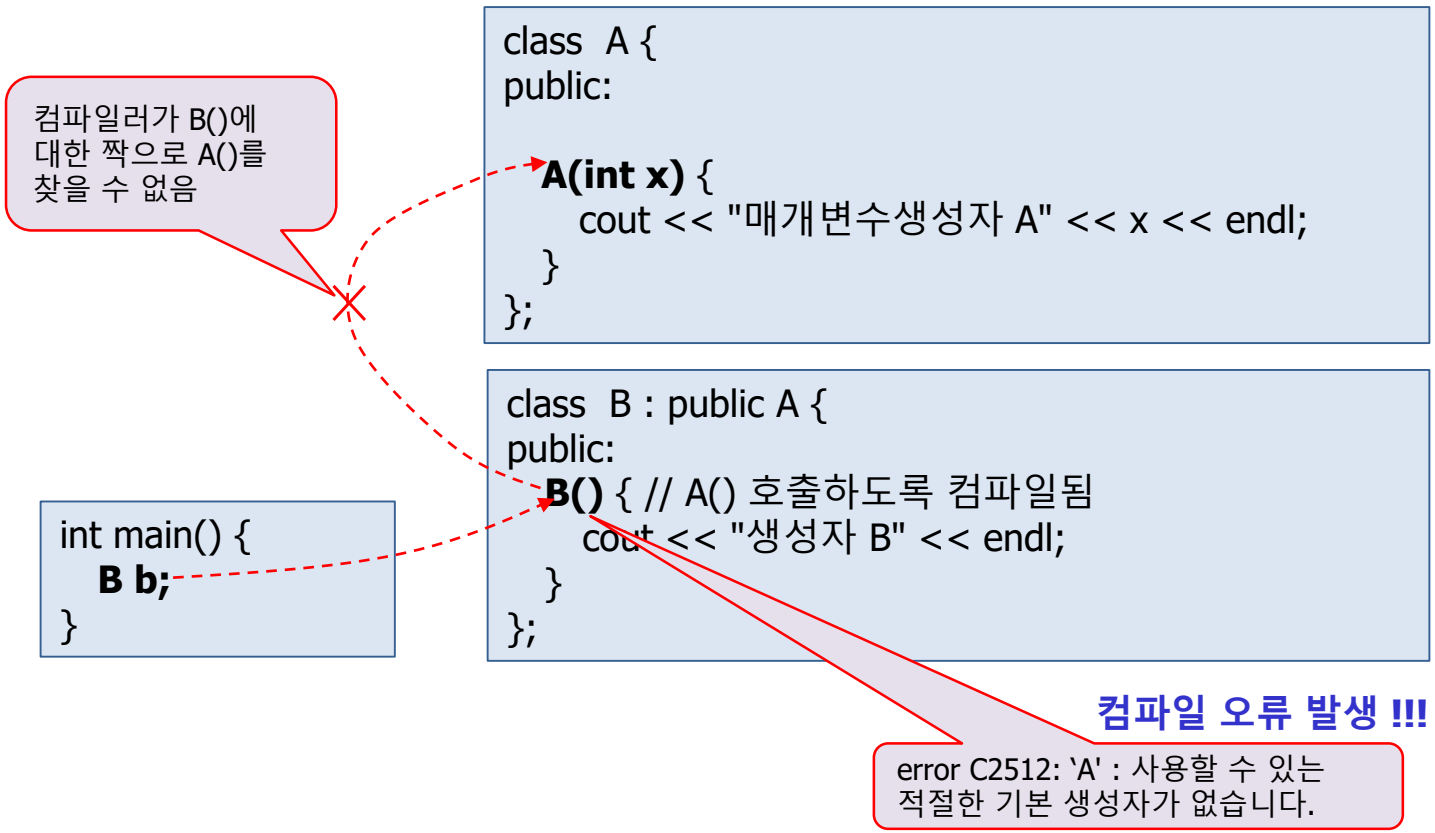
파생 클래스의 생성자 호출 사례1

- 기본 클래스의 디폴트 생성자의 암시적 호출



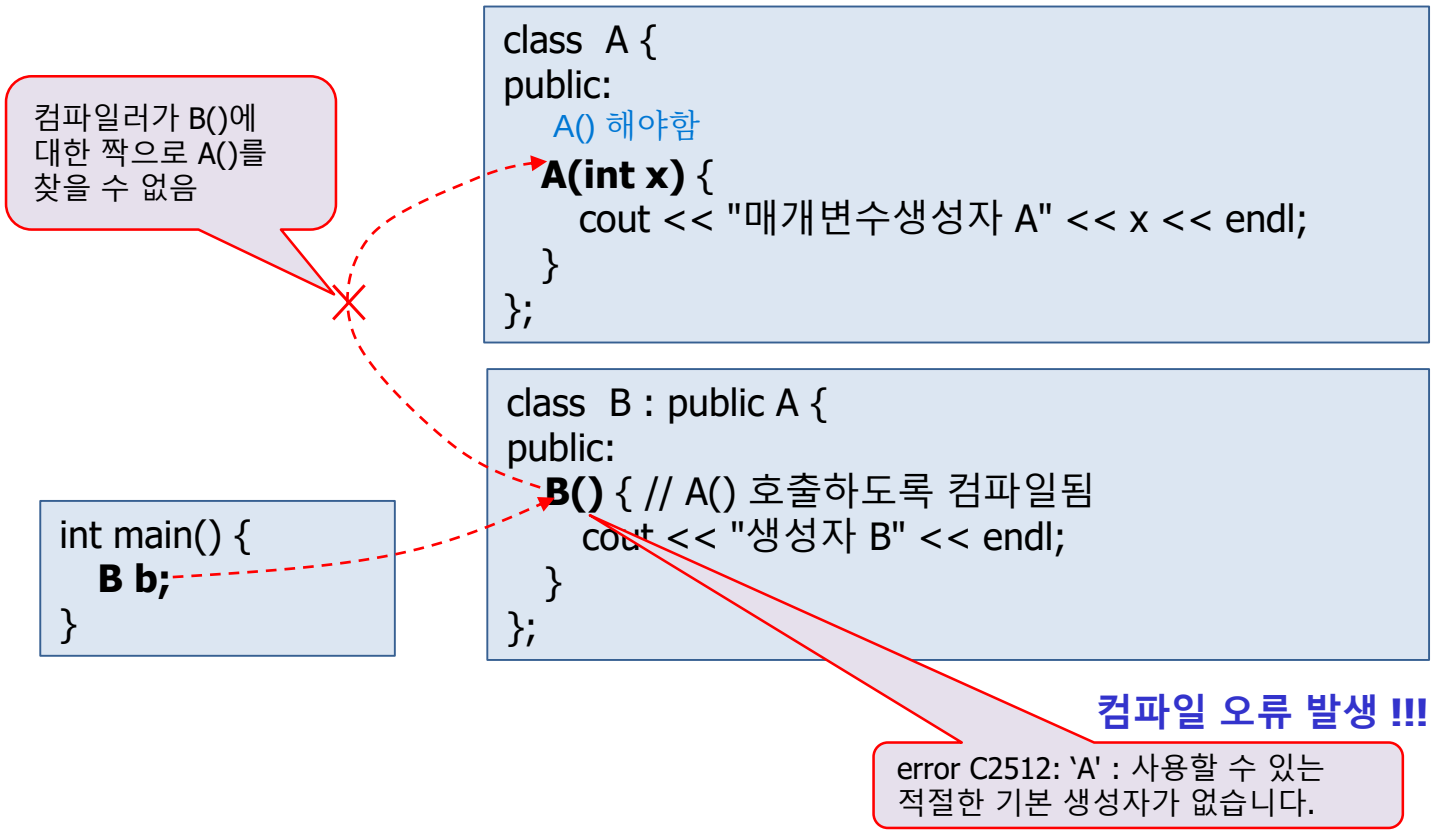
파생 클래스의 생성자 호출 사례2

- 기본 클래스에 디폴트 생성자가 없는 경우



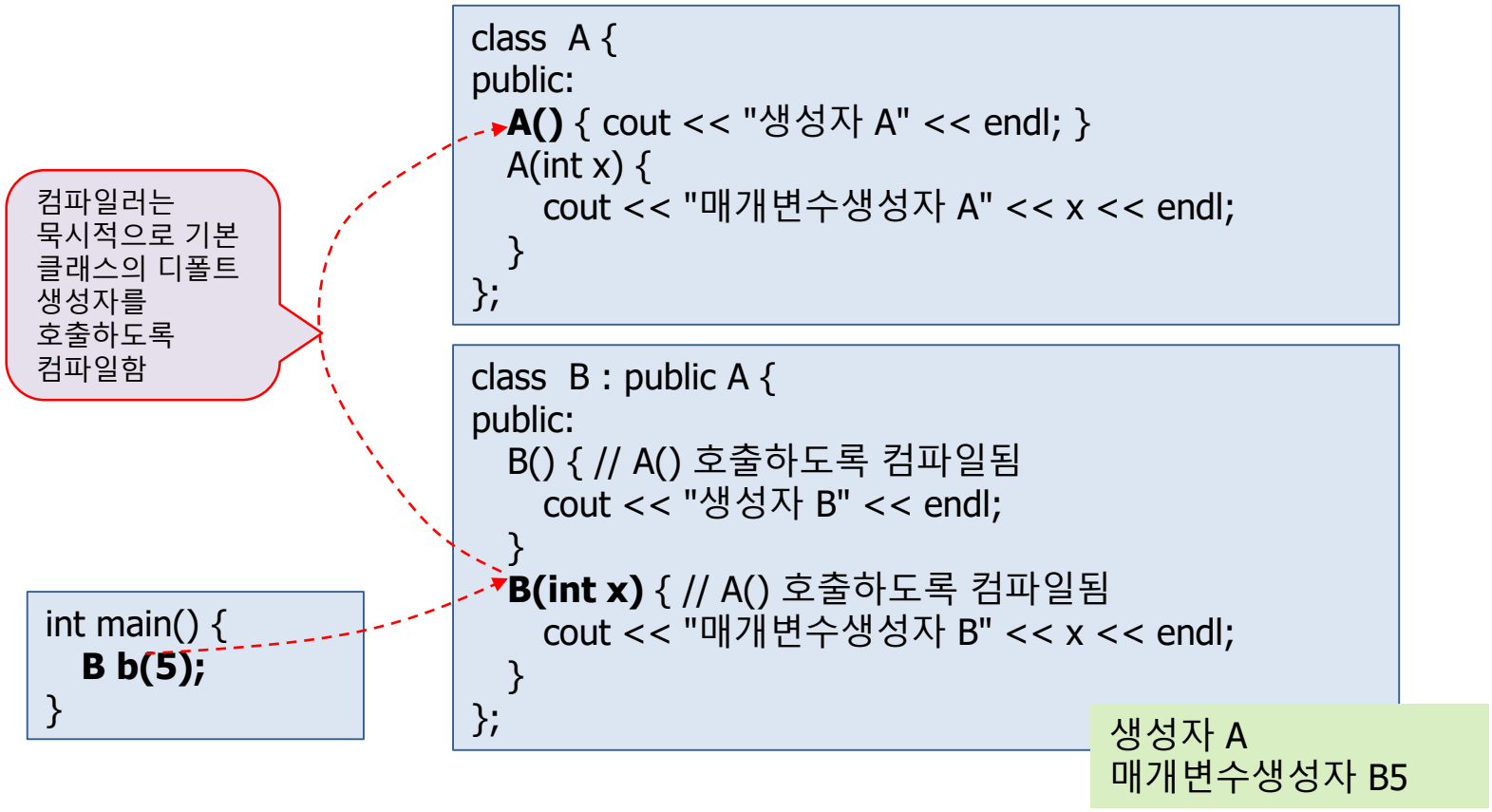
파생 클래스의 생성자 호출 사례2

- 기본 클래스에 디폴트 생성자가 없는 경우



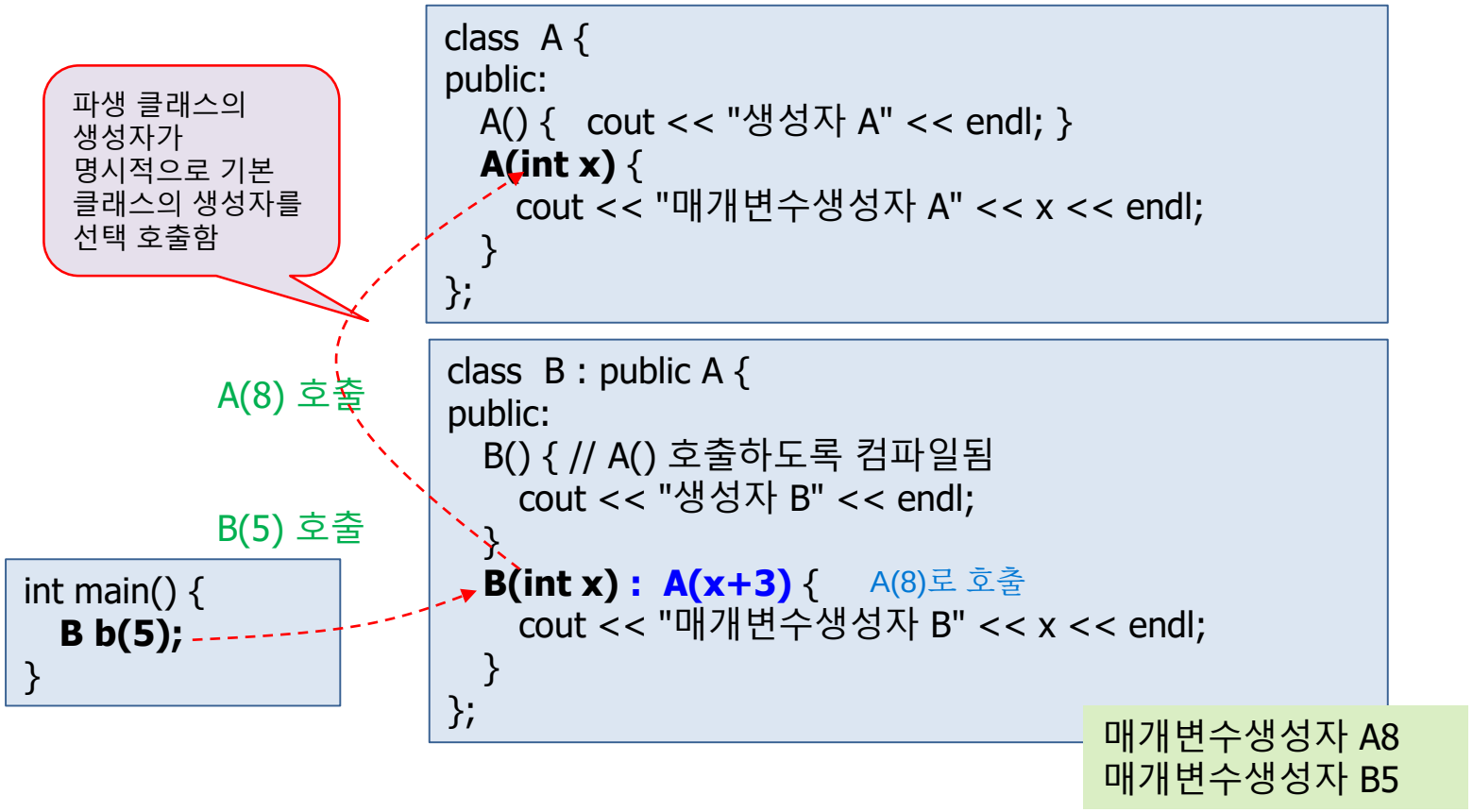
파생 클래스의 생성자 호출 사례3

- 파생 클래스의 인자있는 생성자가 기본 클래스의 디폴트 생성자 호출



파생 클래스의 생성자 호출 사례4

- 파생 클래스의 생성자에서 명시적으로 기본 클래스의 특정한 생성자의 명시적 호출



컴파일러의 디폴트 생성자 호출 코드 삽입

: public A

컴파일러가 묵시적으로 삽입한 코드

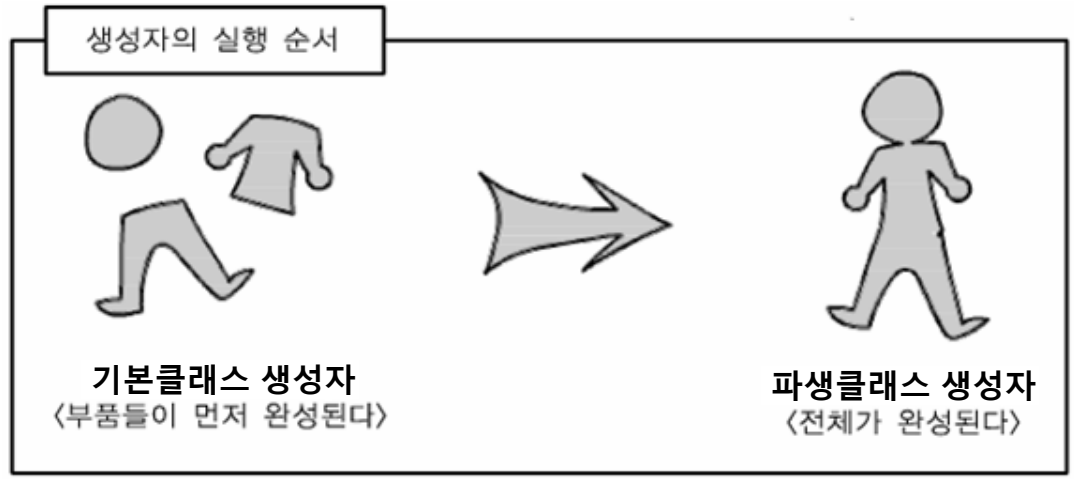
```
class B {
   B() : A() {
      cout << "생성자 B" << endl;
   }

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

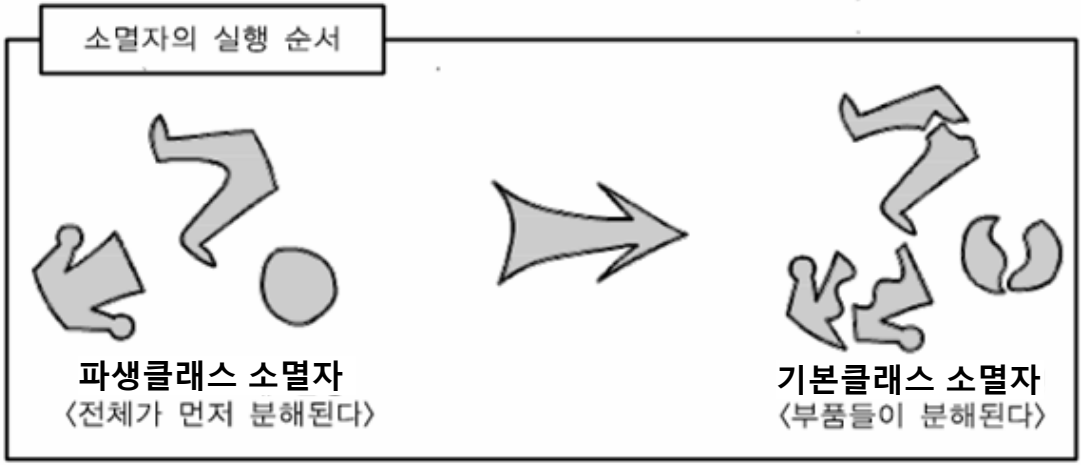
컴파일러가 묵시적으로 삽입한 코드

파생 클래스의 소멸자

- 항상 내부적으로 기본 클래스의 소멸자 호출
  - 파생 클래스의 객체가 소멸될 때 파생 클래스 소멸자는 내부적으로 기본 클래스의 소멸자를 호출함.
  - 소멸자는 오버로딩이 되지 않으므로 인자 없는 소멸자 사용

생성자와 소멸자의 실행 순서





10.4 접근변경자

접근변경자

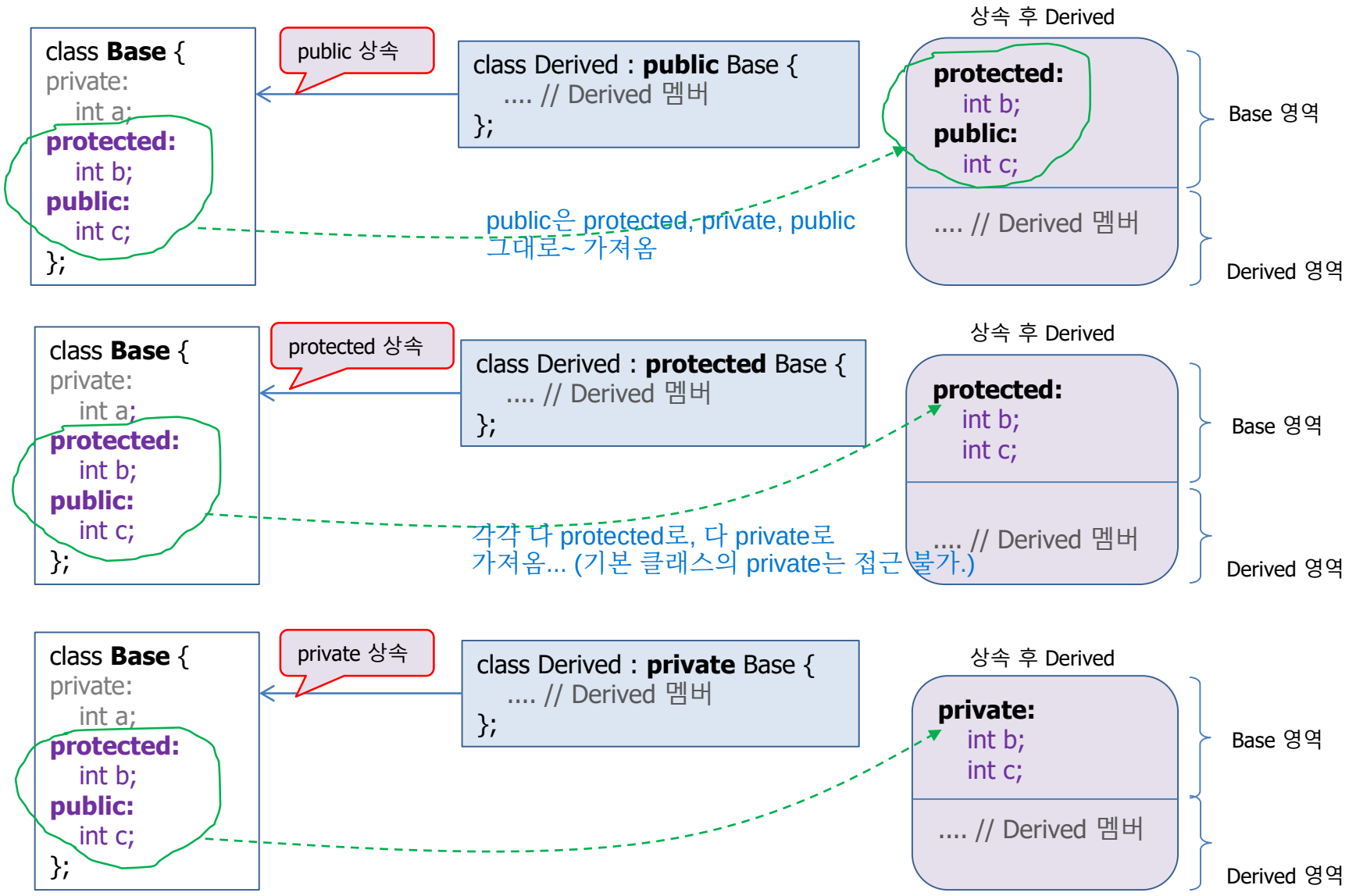
접근변경자

- 기본 클래스의 멤버의 접근 속성을 어떻게 계승할지 지정
  - **public** – 기본 클래스의 protected, public 멤버 속성을 그대로 계승
  - private – 기본 클래스의 protected, public 멤버를 private으로 계승
  - protected – 기본 클래스의 protected, public 멤버를 protected로 계승

접근변경자	기본 클래스의 **private** 멤버	기본 클래스의 **protected** 멤버	기본 클래스의 **public** 멤버
private 상속	파생클래스에서 접근 불가	파생클래스의 private 멤버로 간주	파생클래스의 private 멤버로 간주
protected 상속	파생클래스에서 접근 불가	파생클래스의 protected 멤버로 간주	파생클래스의 protected 멤버로 간주
public 상속	파생클래스에서 접근 불가	파생클래스의 protected 멤버로 간주	파생클래스의 public 멤버로 간주

- 주로 public 상속 사용
- 접근 변경자도 접근 지정자처럼 디폴트로 private을 사용하므로 반드시 public을 명시적으로 지정해야 함.

상속 시 접근 지정에 따른 멤버의 접근 지정 속성 변화



private 상속 사례

```
#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; }
};
```

main 함수에서는 public으로 접근하는 경우만 보셈

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

protected 상속 사례

```
#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
}
```

상속이 중첩될 때 접근 지정 사례

```
#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() {
      setA(5);              // ①
      showA();              // ②
      cout << b;
   }
};
```

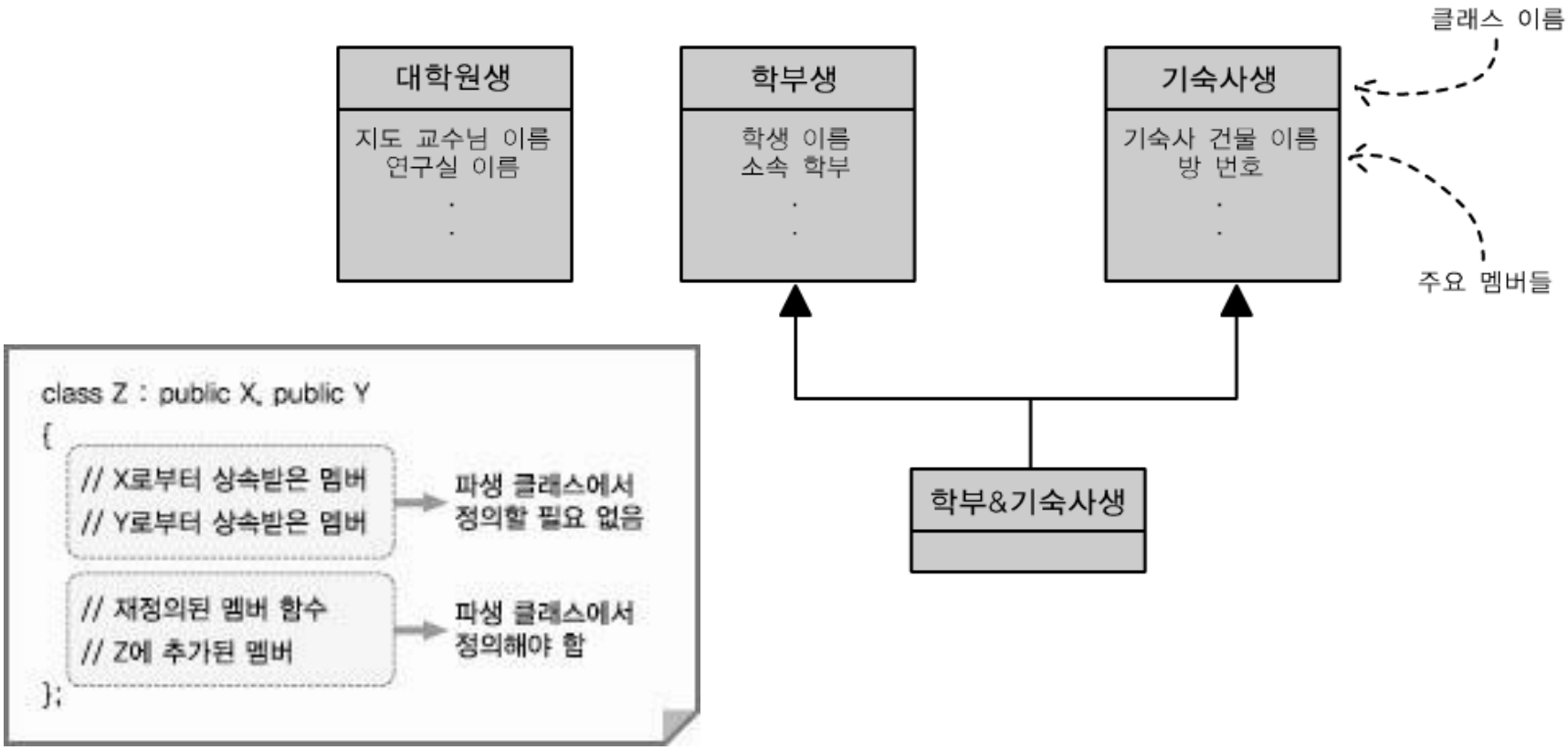
private이어도 클래스 안이라 가능!

```
class GrandDerived : private Derived {
   int c;
protected:
   void setAB(int x) {
      setA(x);              // ③
      showA();              // ④
      setB(x);              // ⑤
   }
};
```

10.5 다중 상속

다중 상속의 필요성

- 2개의 기본 클래스를 상속 받을 필요가 있는 경우
  - 모든 기본 클래스의 멤버를 상속 받음



다중 상속 선언 및 멤버 호출

다중 상속 선언

```
class MP3 {
public:
    void play();
    void stop();
};

class MobilePhone {
public:
    bool sendCall();
    bool receiveCall();
    bool sendSMS();
    bool receiveSMS();
};

class MusicPhone : public MP3, public MobilePhone { // 다중 상속 선언
public:
    void dial();
};
```

상속받고자 하는 기본 클래스를 나열한다.

다중 상속 활용

```
void MusicPhone::dial() {
    play();       // mp3 음악을 연주시키고
    sendCall(); // 전화를 건다.
}
```

MP3::play() 호출

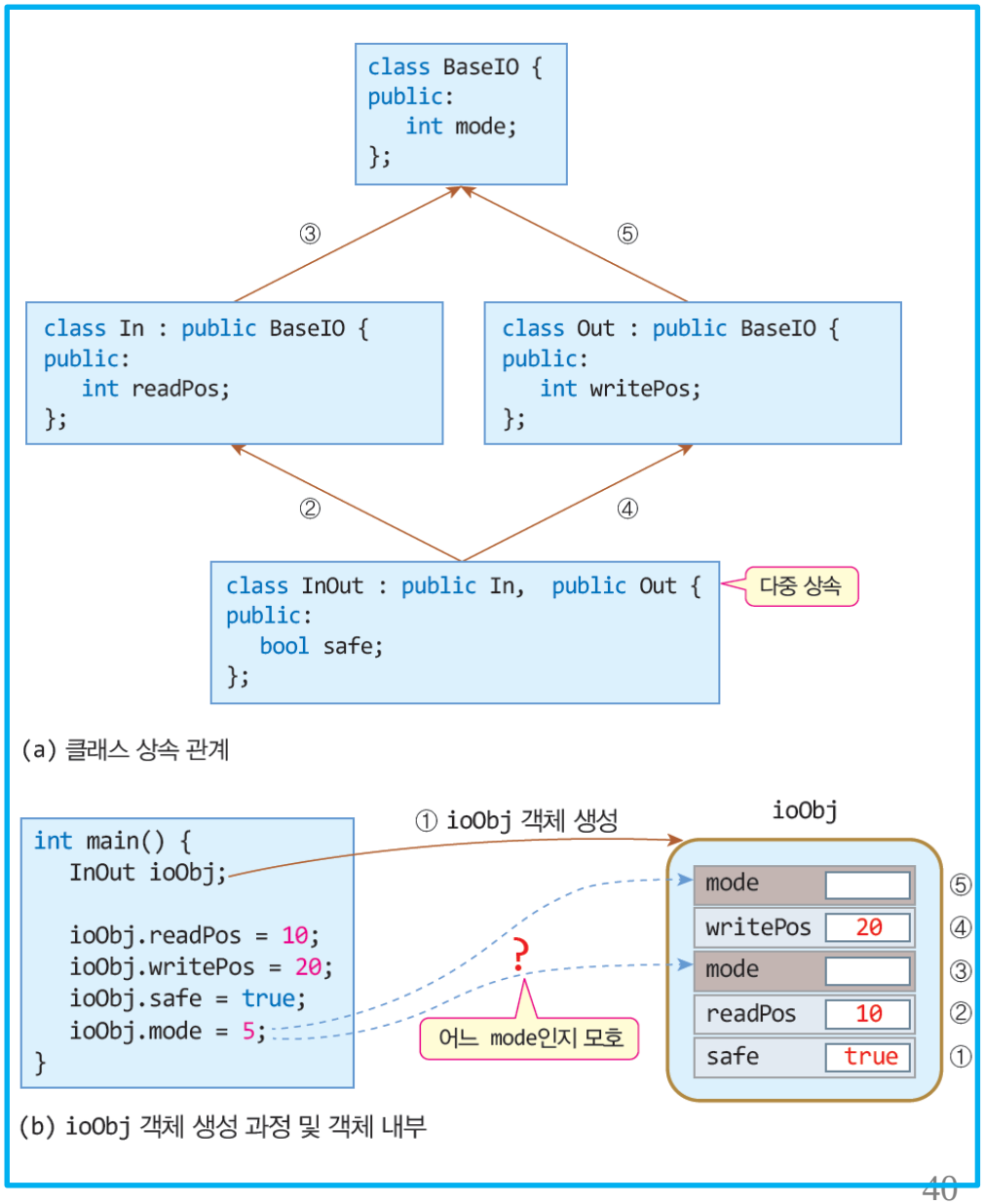
MobilePhone::sendCall() 호출

다중 상속 활용

```
int main() {
    MusicPhone hanPhone;
    hanPhone.play();      // MP3의 멤버 play() 호출
    hanPhone.sendSMS(); // MobilePhone의 멤버 sendSMS() 호출
}
```

다중 상속의 문제점 : 기본 클래스 멤버 접근의 모호성

- 기본 클래스 멤버의 중복 상속
  - Base의 멤버가 이중으로 객체에 삽입되는 문제점
  - 동일한 x를 접근하는 프로그램이 서로 다른 x에 접근하는 결과를 낳게 되어 잘못된 실행 오류가 발생.

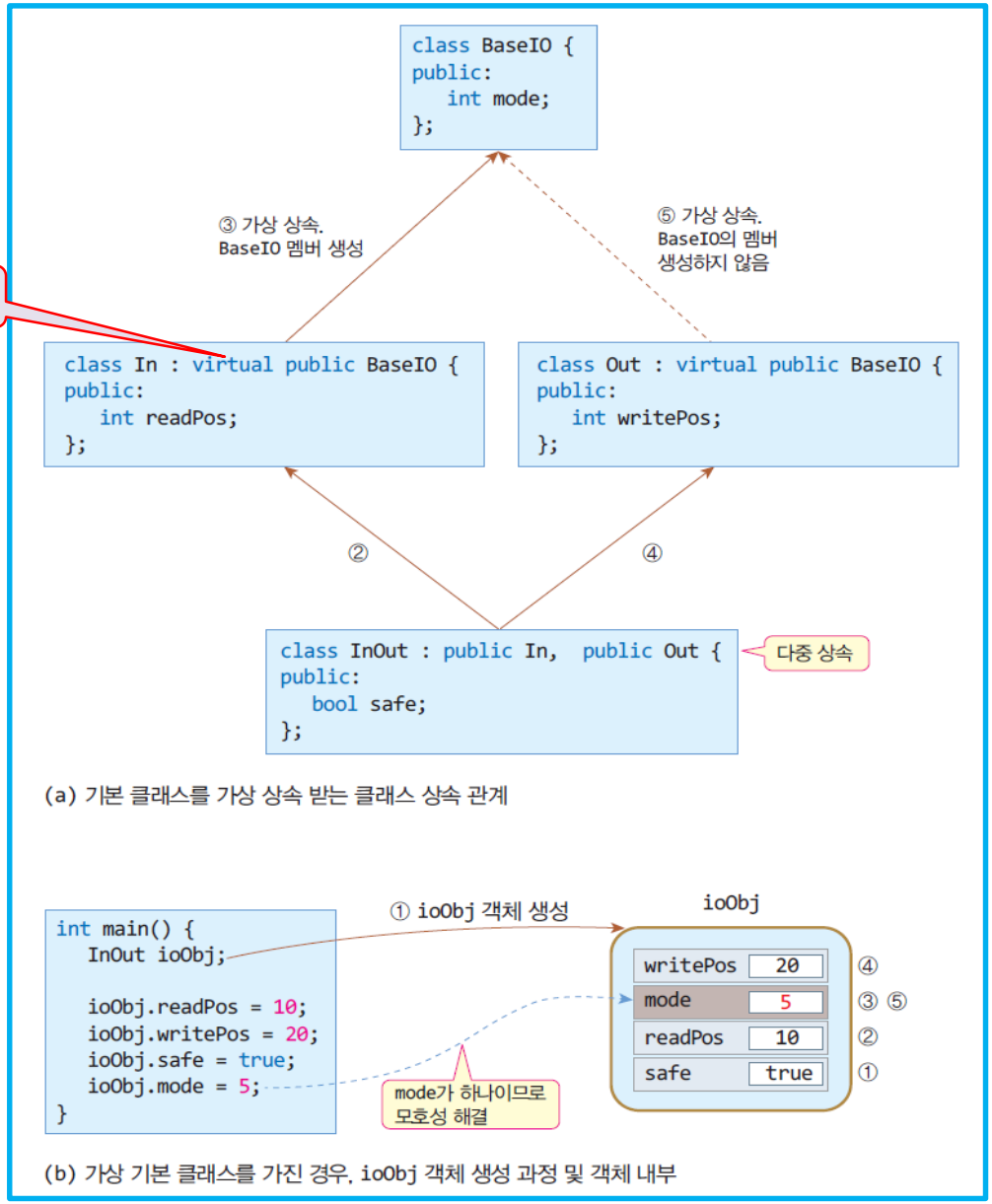


가상 상속

- 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 멤버 공간을 한번만 할당. 이미 할당되어 있다면 그 공간을 공유
    - 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
...
};
```

가상 상속으로 다중 상속의 모호성 해결



충돌의 가능성

- 2 개의 부모 클래스에 같은 이름의 멤버가 있는 경우

Conflict.cpp

```
class UnderGradStudent
{
        …
        void Warn();  // 학사 경고
};

class DormStudent
{
        …
        void Warn();  // 벌점 부여
};

// 중간 생략

UnderGrad_DormStudent std;

std.Warn();                 // Error – 어떤 Warn() 을 말하는 지 알 수 없다.
std.UnderGradStudent::Warn();               // OK
```

정리

- 기본 클래스의 특징을 이어받아 구체적인 특징을 추가로 제공하는 파생 클래스를 정의하는 기능이 바로 상속이다.
- 파생 클래스는 기본 클래스로부터 상속받은 멤버와 새로 추가된 멤버, 재정의된 멤버 함수를 갖는다.
- 파생 클래스 객체를 생성하면 항상 기본 클래스로부터 상속받은 멤버 변수부터 메모리에 할당하고 파생 클래스에 추가된 멤버 변수가 그 다음에 할당된다.
- 파생 클래스의 생성자는 기본 클래스로부터 상속받은 멤버를 초기화하기 위해서 기본 클래스 생성자를 이용한다. 파생 클래스 소멸자는 기본 클래스로부터 상속받은 멤버를 정리하기 위해서 기본 클래스 소멸자를 이용한다.
- 기본 클래스를 상속받을 때 기본 클래스 이름 앞에 private, protected, public 키워드를 사용하는데, 이것을 접근 변경자라고 한다.
- 기본 클래스가 둘 이상인 경우 다중 상속이라고 한다.

다음 수업

- 가상함수와 추상클래스
  - 1_ 가상함수
  - 2_ 가상함수와 오버라이딩 활용사례
  - 3_ 추상클래스와 인터페이스 상속