




# 고급프로그래밍

연산자 중복  
상속

Professor Jeong, Mun-Ho

Robot Vision & Intelligence Laboratory  
Kwangwoon University  
(02-940-5625, mhjeong@kw.ac.kr)

# Schedule

week	Topics		Homework	Quiz
1	과목소개	교과목 소개 (1), C++ 시작 (2)		
2	C++ 	C++ 프로그래밍의 기본(3, 3/12), 클래스와 객체(4, 3/14)	1	1
3		휴강(3/17), 객체생성과 사용(5, 3/19)	2	
4		함수와 참조(6, 3/26), 복사 생성자와 함수중복(7. 3/28)	3	2, 3
5		static friend 연산자중복(8, 4/2), 연산자중복 상속(9, 4/4)	4	4
6		템플릿과 STL, 표준 입출력	5	5
7		파일 입출력		
8	중간고사			
9	C++	예외처리 및 C 사용, 람다식	6	6
10		멀티스레딩	7	7
11		멀티스레딩, 고급문법	8	8
12		고급문법	9	9
13	병렬 프로그래밍	병렬프로그래밍		
14		병렬프로그래밍		
15	기말고사			

# 오늘의 학습내용

- 연산자 중복 - 계속
- 상속

# + 연산자를 외부 프렌드 함수로 구현

```
c = a + b;
```

컴파일러에 의한 변환

```
c = operator+ ( a , b );
```

```
Power operator+ (Power op1, Power op2)
{
    Power tmp;
    tmp.kick = op1.kick + op2.kick;
    tmp.punch = op1.punch + op2.punch;
    return tmp;
}
```

# 예제

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick=0, int punch=0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    friend Power operator+(Power op1, Power op2);
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch
        << endl;
}

Power operator+(Power op1, Power op2) {
    Power tmp; // 임시 객체 생성
    tmp.kick = op1.kick + op2.kick; // kick 더하기
    tmp.punch = op1.punch + op2.punch; // punch 더하기
    return tmp; // 임시 객체 리턴
}
```

```
int main( )
{
    Power a(3,5), b(4,6), c;
    c = a + b; // 파워 객체 + 연산
    a.show();
    b.show();
    c.show();
}
```

```
kick=3,punch=5
kick=4,punch=6
kick=7,punch=11
```

# 단항 연산자 ++를 프렌드로 작성하기

(a) 전위 연산자

**++a**

컴파일러에 의한 변환

**operator++ ( a )**

```
Power& operator++ (Power& op) {  
    op.kick++;  
    op.punch++;  
    return op;  
}
```

0은 의미 없는 값으로  
전위 연산자와 구분하  
기 위함

(b) 후위 연산자

**a++**

컴파일러에 의한 변환

**operator ++ ( a, 0 )**

```
Power operator++ (Power& op, int x) {  
    Power tmp = op;  
    op.kick++;  
    op.punch++;  
    return tmp;  
}
```

# 예제

```
class Power {  
    int kick;  
    int punch;  
public:  
    Power(int kick=0, int punch=0) { this->kick = kick; this->punch = punch; }  
    void show();  
    friend Power& operator++(Power& op); // 전위 ++ 연산자 함수  
    friend Power operator++(Power& op, int x); // 후위 ++ 연산자 함수  
};
```

```
void Power::show() {  
    cout << "kick=" << kick << ', ' << "punch=" << punch << endl;  
}  
Power& operator++(Power& op) { // 전위 ++ 연산자 함수 구현  
    op.kick++;  
    op.punch++;  
    return op; // 연산 결과 리턴  
}
```

```
Power operator++(Power& op, int x) { // 후위 ++ 연산자 함수 구현  
    Power tmp = op; // 변경하기 전의 op 상태 저장  
    op.kick++;  
    op.punch++;  
    return tmp; // 변경 이전의 op 리턴  
}
```

```
int main()  
{  
    Power a(3,5), b;  
    b = ++a; // 전위 ++ 연산자  
    a.show(); b.show();  
  
    b = a++; // 후위 ++ 연산자  
    a.show(); b.show();  
}
```

```
kick=4,punch=6  
kick=4,punch=6  
kick=5,punch=7  
kick=4,punch=6
```

# 예제 - << 연산자

- Power 객체의 kick과 punch에 정수를 더하는 << 연산자를 멤버 함수로 작성하라

```
class Power {
    int kick;
    int punch;
public:
    Power(int kick = 0, int punch = 0) {
        this->kick = kick; this->punch = punch;
    }
    void show();
    Power& operator << (int n); // 연산 후 Power 객체의 참조 리턴
};

void Power::show() {
    cout << "kick=" << kick << ',' << "punch=" << punch << endl;
}

Power& Power::operator <<(int n) {
    kick += n;
    punch += n;
    return *this; // 이 객체의 참조 리턴
}
```

```
int main() {
    Power a(1, 2);
    a << 3 << 5 << 6;
    a.show();
}
```

객체 a에 3, 5, 6이  
순서대로 더해진다

kick=15,punch=16



# 연산자 중복 : [ ]

```
#include <iostream>
using namespace std;
class Complex
{
private :
    int real, image;

public :
    Complex(int r=0, int i=0) : real(r), image(i) { };
    ~ Complex( );
    void show(Complex& b);
    Complex operator-(const Complex& C) const;
    Complex operator-( ) const;
    int operator [ ](const int& idx) const;
    int& operator [ ](const int& idx);
};

void Complex::show(Complex& b)
{
    cout << b.real<<"+"<< b.image<<"j" << endl;
}

Complex Complex::operator -(const Complex& C) const
{
    return Complex(real-C.real, image-C.image);
}
```

```
Complex Complex::operator -( ) const {
    return Complex(-real, -image);
}

int Complex::operator [ ](const int& idx) const {
    return ( idx == 0 ? real : image);
}

int& Complex::operator [ ](const int& idx) {
    return ( idx == 0 ? real : image);
}

int main()
{
    Complex a(10,10), b;
    b[0] = 1.0; b[1] = 2.0;
    show(b);

    b = -a; b[0] = a[0];
    show(b);
}
```

# 실습

- 다음 프로그램이 가능하도록 `Circle` 클래스의 연산자를 프렌드 함수로 작성하시오

```
class Circle {
    int radius;
public:
    Circle(int radius = 0) { this->radius = radius; }
    void show() {
        cout << "radius = " << radius << " circle" << endl;
    }
};

int main() {
    Circle a(5), b(4);
    ++a; // 반지름을 1 증가 시킨다.
    b = a++; // 반지름을 1 증가 시킨다.
    a.show();
    b.show();
}
```

## 실습 - 답

- 다음 프로그램이 가능하도록 `Circle` 클래스의 연산자를 프렌드 함수로 작성하시오

```
class Circle {
    int radius;
public:
    Circle(int radius = 0) { this->radius = radius; }
    void show() {
        cout << "radius = " << radius << " 인 원" << endl;
    }

    friend Circle& operator ++(Circle& c);
    friend Circle operator ++(Circle& c, int x);
};

Circle& operator ++(Circle& c) { // 전위 ++. ++a를 위함
    c.radius++;
    return c;
}

Circle operator ++(Circle& c, int x) { // 후위 ++. a++를 위함
    Circle tmp = c;
    c.radius++;
    return tmp;
}
```

# 함수 객체(Function Object)

- 객체를 함수처럼 사용
- 연산자 중복으로 선언함: operator ( )

```
2 #include <iostream>
3 using namespace std;
4
5 class Plus
6 {
7 public:
8     int operator( )(int a, int b)
9     {
10         return a + b;
11     }
12 };
13
14 int main()
15 {
16     Plus pls;
17
18     cout << "pls(10, 20): " << pls(10, 20) << endl;
19     return 0;
20 }
```

explicit call: pls.operator()(10,20)

implicit call

# 함수 객체

- 사용하는 이유
  - 객체 멤버 활용
  - 일반함수 보다 호출이 빠름

```
2 #include <iostream>
3 using namespace std;
4
5 class MoneyBox
6 {
7     int total;
8
9 public:
10     MoneyBox(int _init = 0) : total(_init) { }
11
12     int operator( )(int money)
13     {
14         total += money;
15         return total;
16     }
17 };
18
19 int main()
20 {
21     MoneyBox mb;
22
23     cout << "mb(100): " << mb(100) << endl;
24     cout << "mb(500): " << mb(500) << endl;
25     cout << "mb(2000): " << mb(2000) << endl;
26
27     return 0;
28 }
```

# 상속(Inheritance)

- C++ 객체 지향 상속
- 업 캐스팅과 다운 캐스팅
- `protected` 접근지정
- 생성자 소멸자 실행 순서
- 상속지정
- 다중 상속

# C++에서의 상속(Inheritance)

## ■ C++에서의 상속이란?

- 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 통해 클래스의 재활용성 높임

# 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

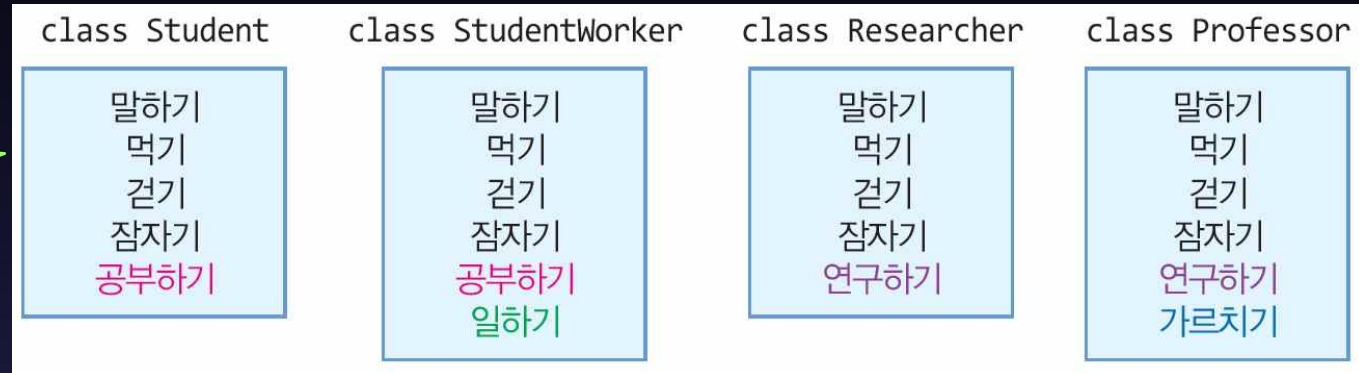
## 3. 클래스 재사용과 확장을 통한 소프트웨어 생산성 향상

- 빠른 소프트웨어 생산 필요
- 기존에 작성한 클래스의 재사용 - 상속
  - 상속받아 새로운 기능을 확장
- 앞으로 있을 상속에 대비한 클래스의 객체 지향적 설계 필요

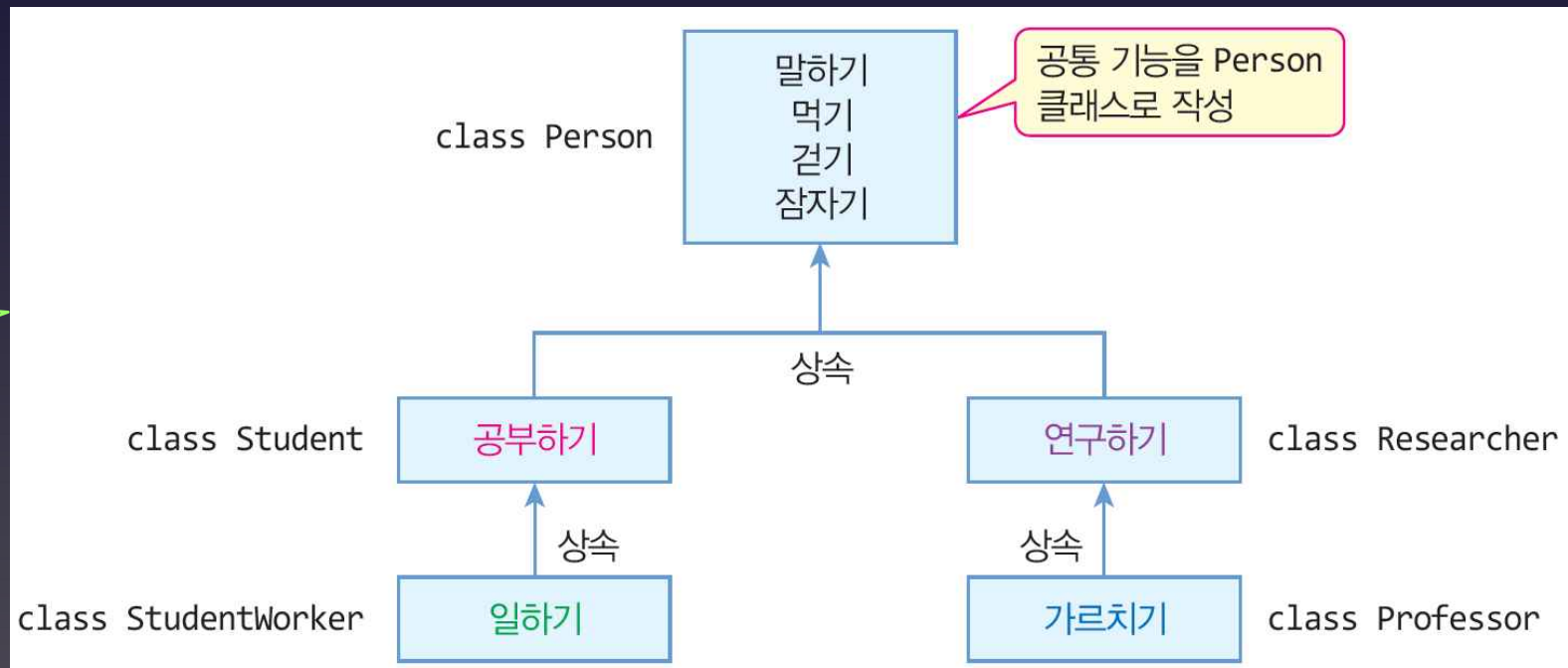


# 상속 관계로 클래스의 간결화 사례

기능이 중복된 4 개의 클래스

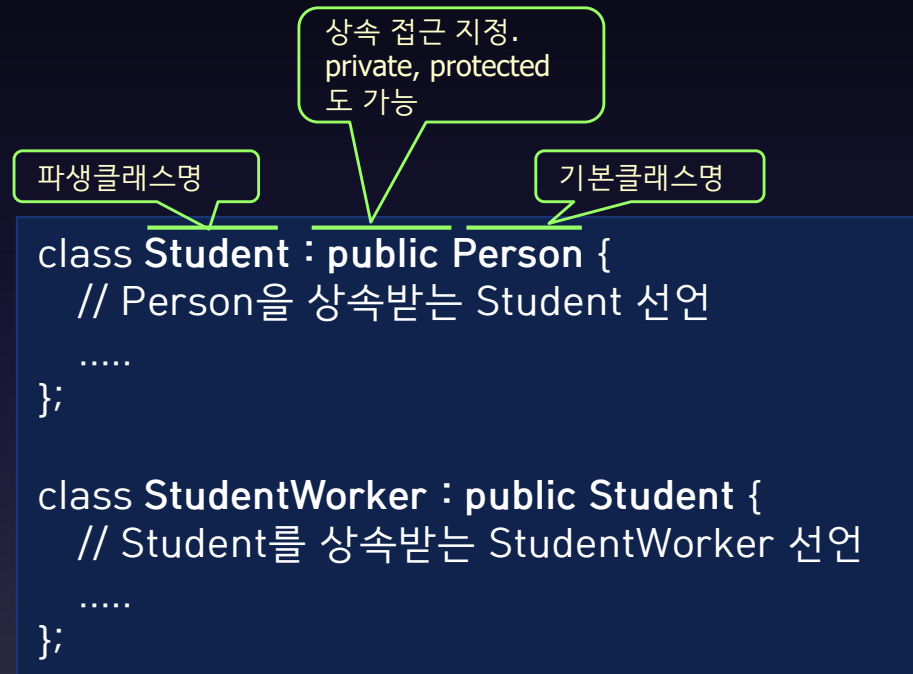


상속 관계로 클래스의 간결화



# 상속 선언

## ■ 상속 선언



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

# 예제 - ColorPoint

```
#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 {
    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
}
```

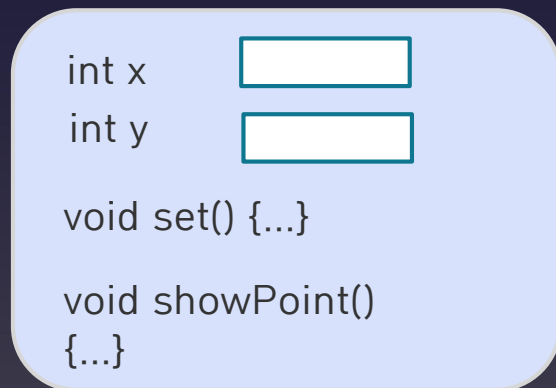
Red:(3,4)

# 파생 클래스의 객체 구성

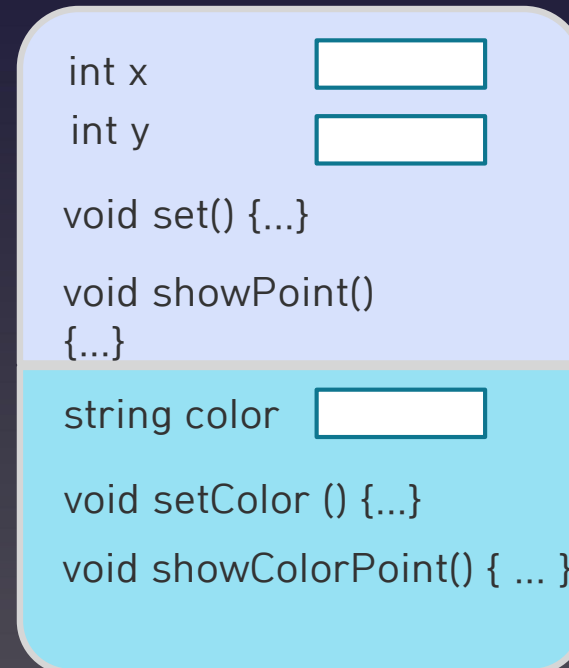
```
class Point {  
    int x, y; // 한 점 (x,y) 좌표 값  
public:  
    void set(int x, int y);  
    void showPoint();  
};
```

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

Point p;



ColorPoint cp;

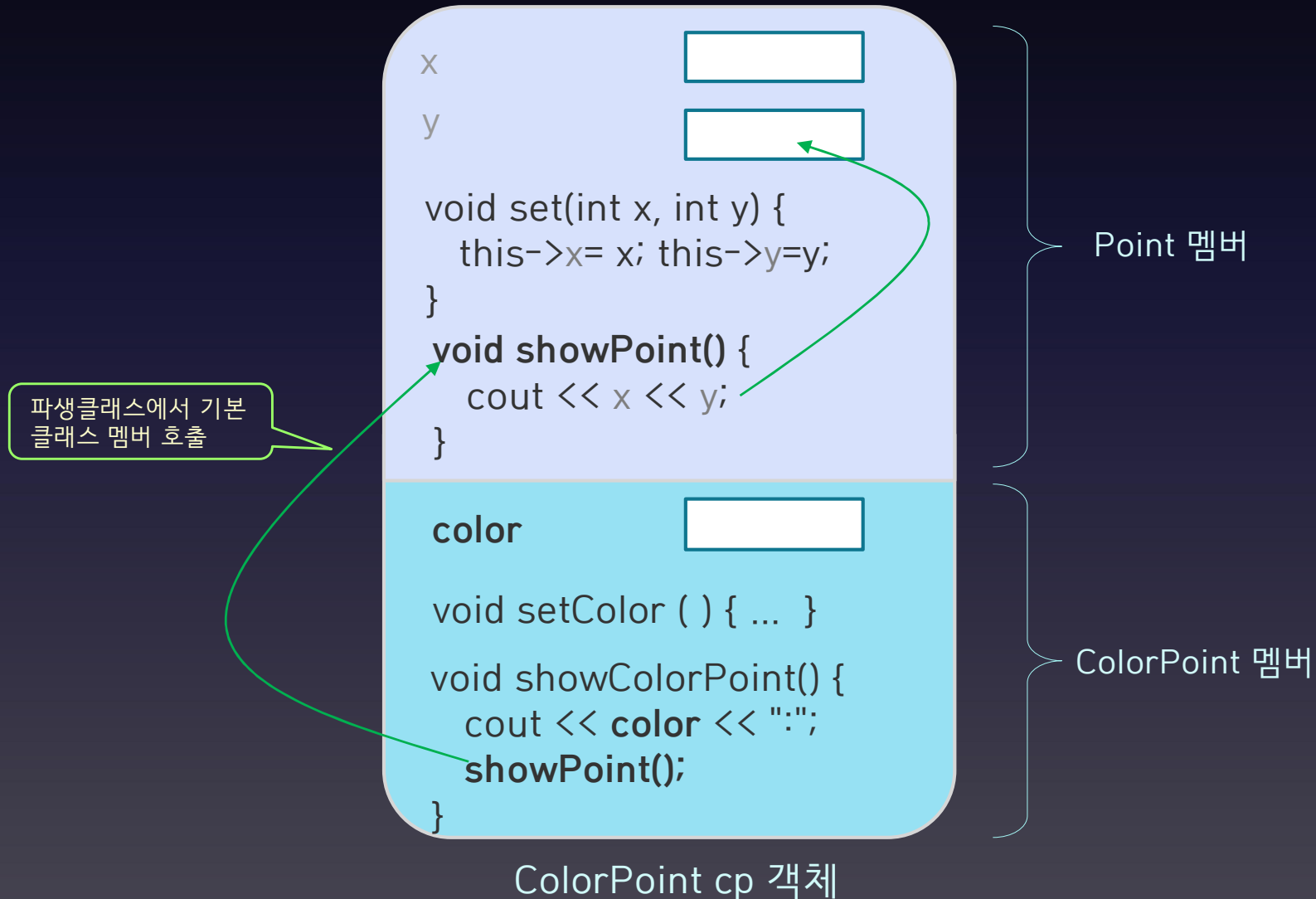


파생 클래스의 객체는 기본 클래스의 멤버 포함

기본클래스 멤버

파생클래스 멤버

# 파생 클래스에서 멤버 접근



# 외부에서 파생 클래스 접근

x, y는 Point 클래스에 private이므로  
set(), showPoint()에서만 접근 가능

기본클래스  
멤버 호출

파생클래스  
멤버 호출

파생클래스  
멤버 호출

```
ColorPoint cp;
```

```
cp.set(3, 4);  
cp.setColor("Red");  
cp.showColorPoint();
```

```
main()
```

x

3

y

4

```
void set(int x, int y) {  
    this->x= x; this->y=y;  
}  
void showPoint() {  
    cout << x << y;  
}
```

color

"Red"

```
void setColor (string color)  
{  
    this->color = color;  
}  
void showColorPoint() {  
    cout << color << ":";  
    showPoint();  
}
```

ColorPoint cp 객체

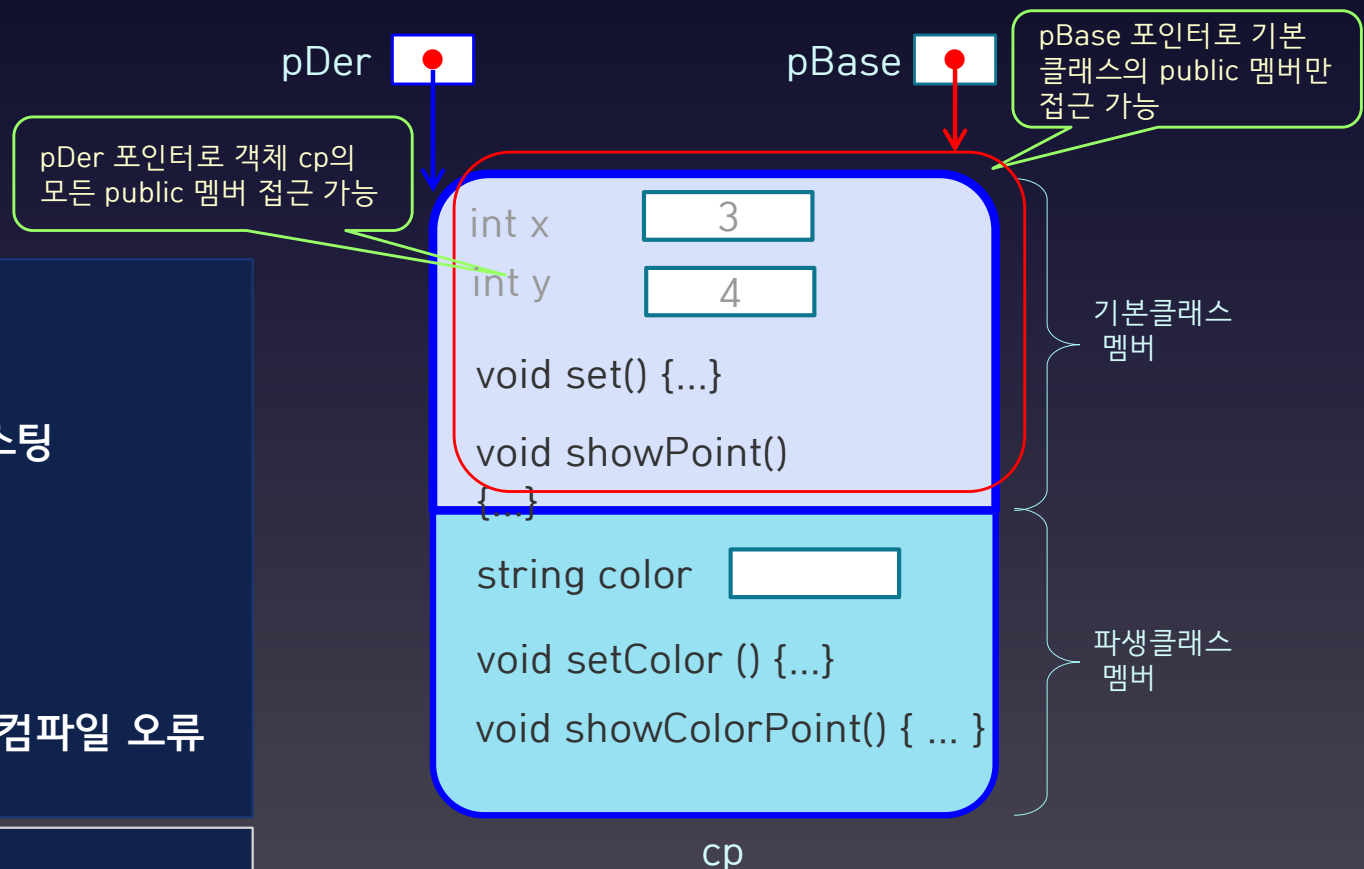
# 상속과 객체 포인터 - 업 캐스팅

## ■ 업 캐스팅(up-casting)

- 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  
}
```

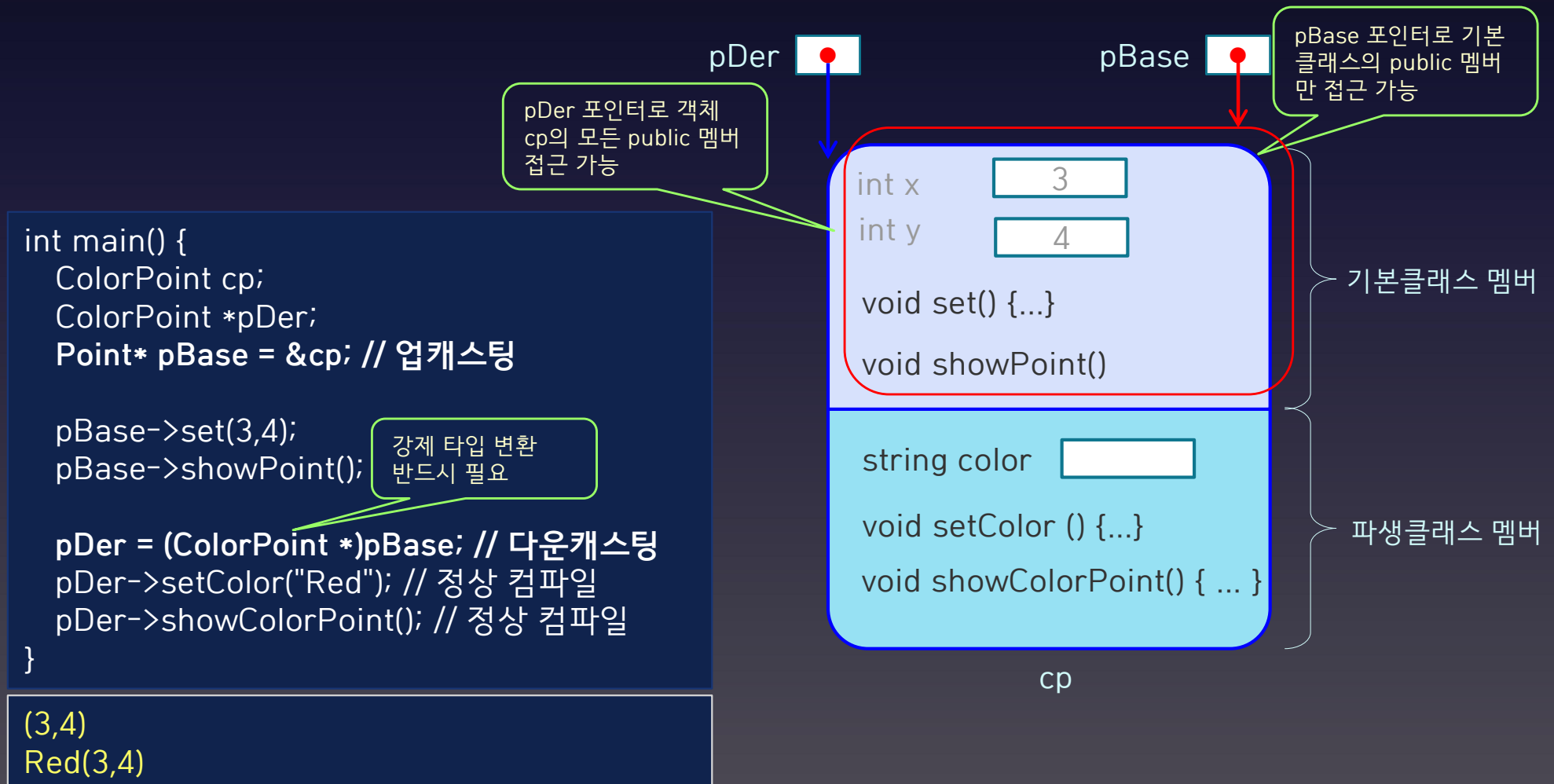
(3,4)  
Red(3,4)



# 상속과 객체 포인터 - 다운 캐스팅

## ■ 다운 캐스팅(down-casting)

- 기본 클래스의 포인터가 파생 클래스의 포인터에 치환되는 것





# protected 접근 지정

## ■ 접근 지정자

### - private 멤버

- 선언된 클래스 내에서만 접근 가능
- 파생 클래스에서도 기본 클래스의 `private` 멤버 직접 접근 불가

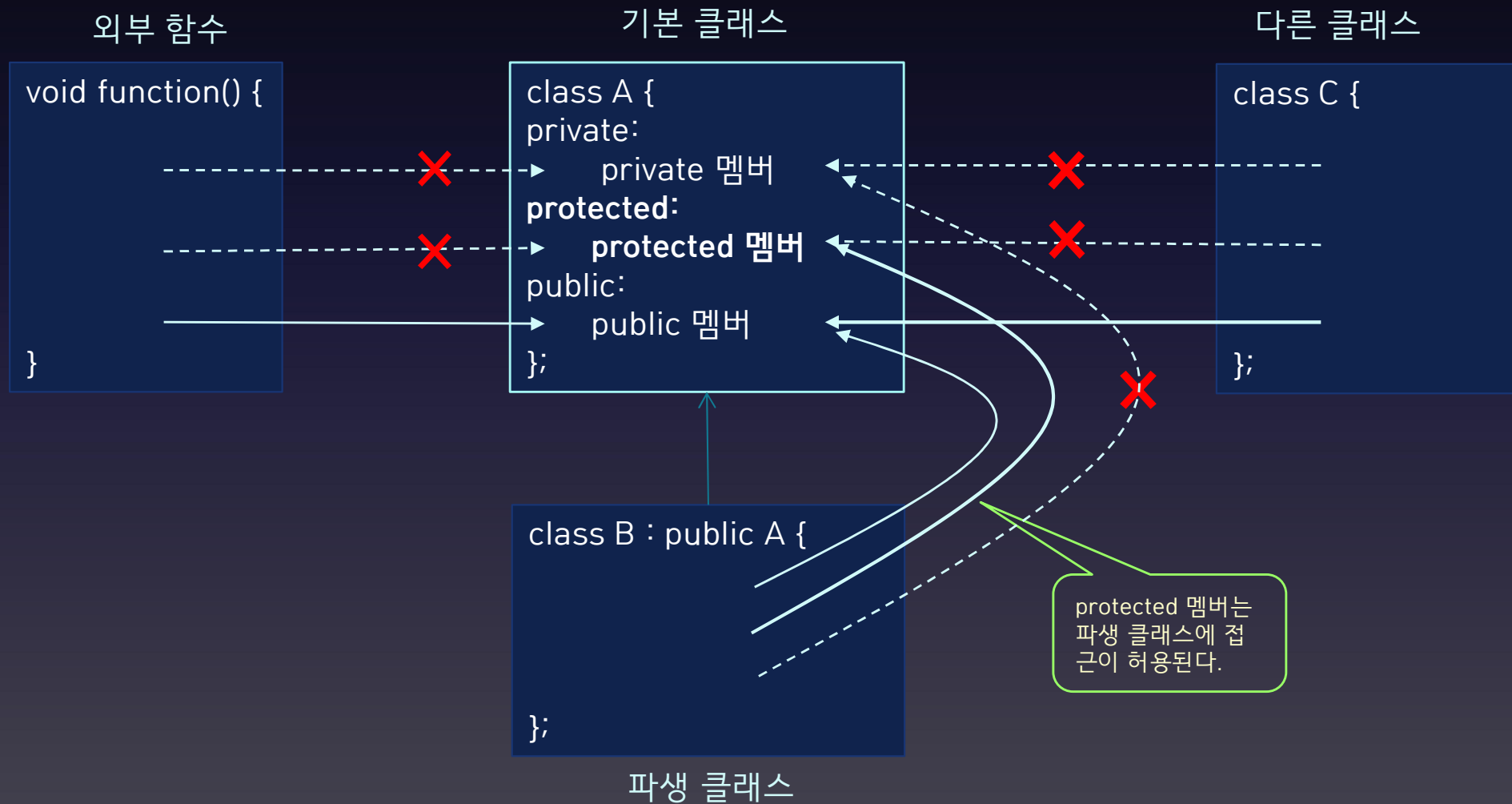
### - public 멤버

- 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



# 예제 - protected 멤버에 대한 접근

```
#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
}
```

오류

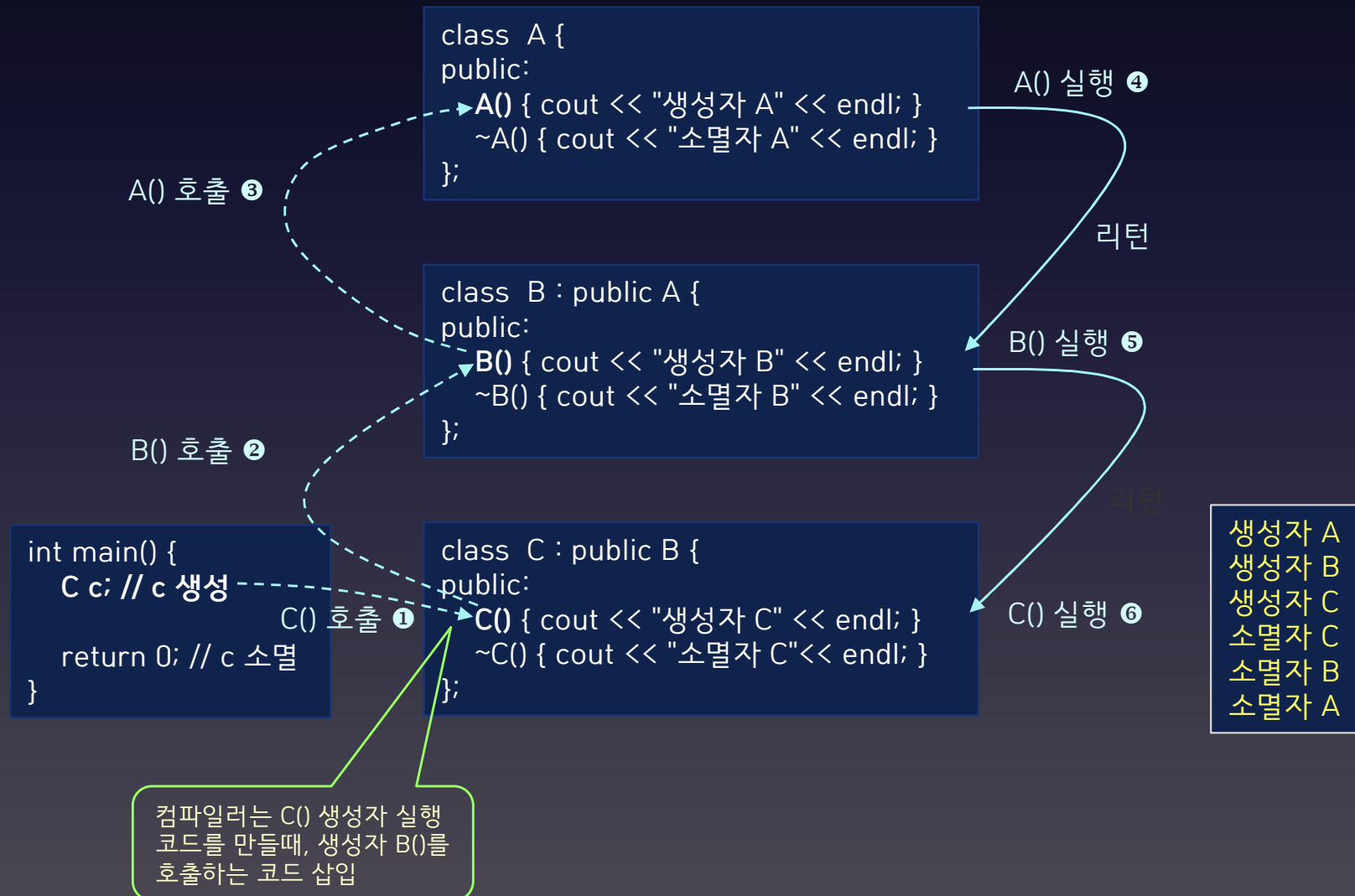
오류

오류

오류

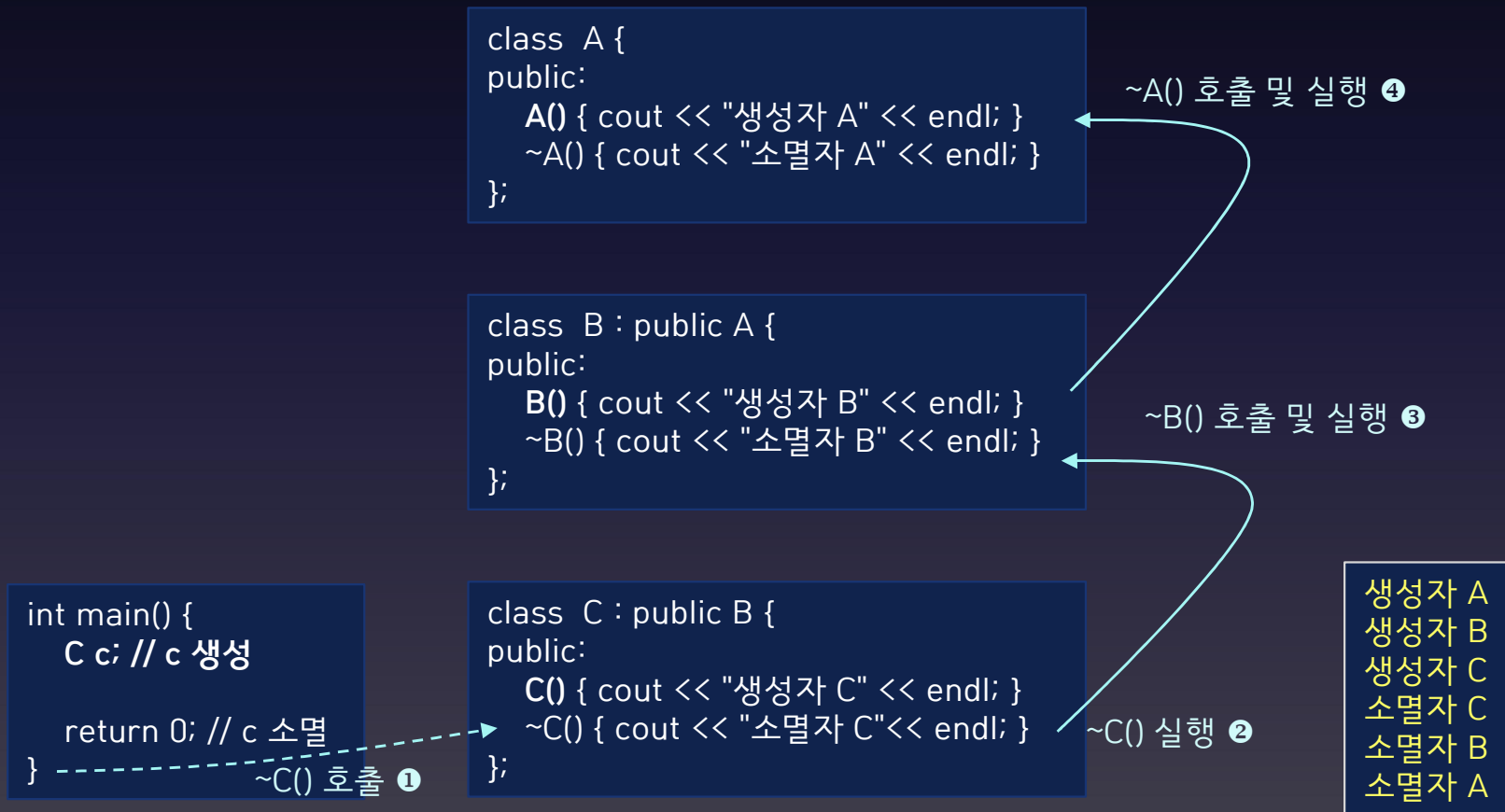
# 생성자 소멸자 실행 순서

## ■ 생성자 호출 관계 및 실행 순서



# 생성자 소멸자 실행 순서

## ■ 소멸자 호출관계 및 실행순서



# 묵시적 기본 클래스의 생성자 선택

파생 클래스의 생성자에서 기본 클래스의 기본 생성자 호출

컴파일러는 묵시적으로 기본 클래스의 기본 생성자를 호출하도록 컴파일함

```
class A {  
public:  
    A() { cout << "생성자 A" << endl; }  
    A(int x) {  
        cout << " 매개변수생성자 A" << x << endl;  
    }  
};
```

```
int main() {  
    B b;  
}
```

```
class B : public A {  
public:  
    B() { // A() 호출하도록 컴파일됨  
        cout << "생성자 B" << endl;  
    }  
};
```

생성자 A  
생성자 B

# 기본 생성자가 없는 경우

컴파일러가 B()에 대한 짝으로 A()를 찾을 수 없음

✗

컴파일 오류 발생 !!!

```
int main() {  
    B b;  
}
```

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

```
class B : public A {  
public:  
    B() { // A() 호출하도록 컴파일됨  
        cout << "생성자 B" << endl;  
    }  
};
```

error C2512: 'A' : 사용할 수 있는 적절한 기본 생성자가 없습니다.

# 파생 클래스의 매개변수 생성자

파생 클래스의 매개 변수를 가진 생성자가 기본 클래스의 기본 생성자 호출

컴파일러는 묵시적으로 기본 클래스의 기본 생성자를 호출하도록 컴파일함

```
class A {  
public:  
    A() { cout << "생성자 A" << endl; }  
    A(int x) {  
        cout << "매개변수생성자 A" << x << endl;  
    }  
};
```

```
class B : public A {  
public:  
    B() { // A() 호출하도록 컴파일됨  
        cout << "생성자 B" << endl;  
    }  
    B(int x) { // A() 호출하도록 컴파일됨  
        cout << "매개변수생성자 B" << x << endl;  
    }  
};
```

```
int main() {  
    B b(5);  
}
```

생성자 A  
매개변수생성자 B5



# 명시적 기본 클래스의 생성자 선택

파생 클래스의 생성자가 명시적으로 기본 클래스의 생성자를 선택 호출함

```
class A {  
public:  
    A() { cout << "생성자 A" << endl; }  
    A(int x) {  
        cout << "매개변수생성자 A" << x << endl;  
    }  
};
```

A(8) 호출

```
class B : public A {  
public:  
    B() { // A() 호출하도록 컴파일됨  
        cout << "생성자 B" << endl;  
    }  
    B(int x) : A(x+3) {  
        cout << "매개변수생성자 B" << x << endl;  
    }  
};
```

B(5) 호출

```
int main() {  
    B b(5);  
}
```

매개변수생성자 A8  
매개변수생성자 B5

# 기본 생성자 호출 코드 삽입

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

컴파일러가 묵시적으로  
삽입한 코드

컴파일러가 묵시적으로  
삽입한 코드

# 예제 - 생성자 매개 변수 전달

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

```
class TV {
    int size; // 스크린 크기
public:
    TV() { size = 20; }
    TV(int size) { this->size = size; }
    int getSize() { return size; }
};
```

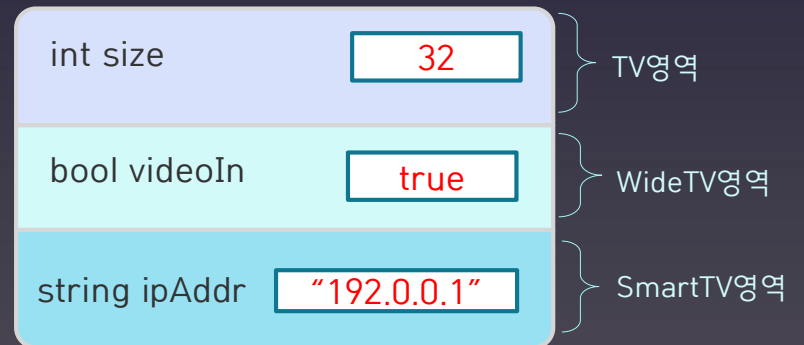
```
class WideTV : public TV { // TV를 상속받는 WideTV
    bool videoIn;
public:
    WideTV(int size, bool videoIn) : TV(size) {
        this->videoIn = videoIn;
    }
    bool getVideoIn() { return videoIn; }
};
```

```
class SmartTV : public WideTV { // WideTV를 상속받는 SmartTV
    string ipAddr; // 인터넷 주소
public:
    SmartTV(string ipAddr, int size) : WideTV(size, true) {
        this->ipAddr = ipAddr;
    }
    string getIpAddr() { return ipAddr; }
};
```

```
int main() {
    // 32 인치 크기에 "192.0.0.1"의 인터넷 주소를 가지는 스마트 TV 객체 생성
    SmartTV htv("192.0.0.1", 32);
    cout << "size=" << htv.getSize() << endl;
    cout << "videoIn=" << boolalpha << htv.getVideoIn() << endl;
    cout << "IP=" << htv.getIpAddr() << endl;
}
```

boolalpha는 불린 값을 true, false로 출력되게 하는 조작자

size=32  
videoIn=true  
IP=192.0.0.1

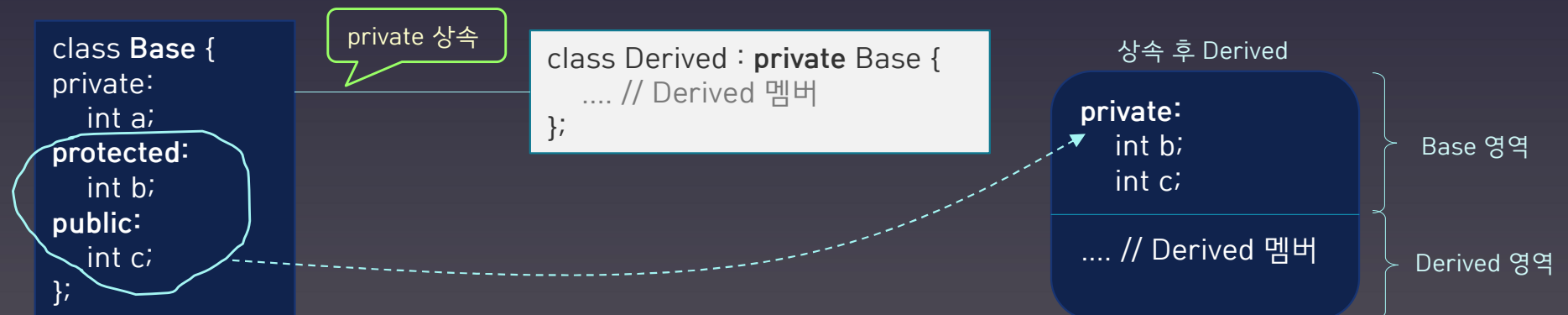
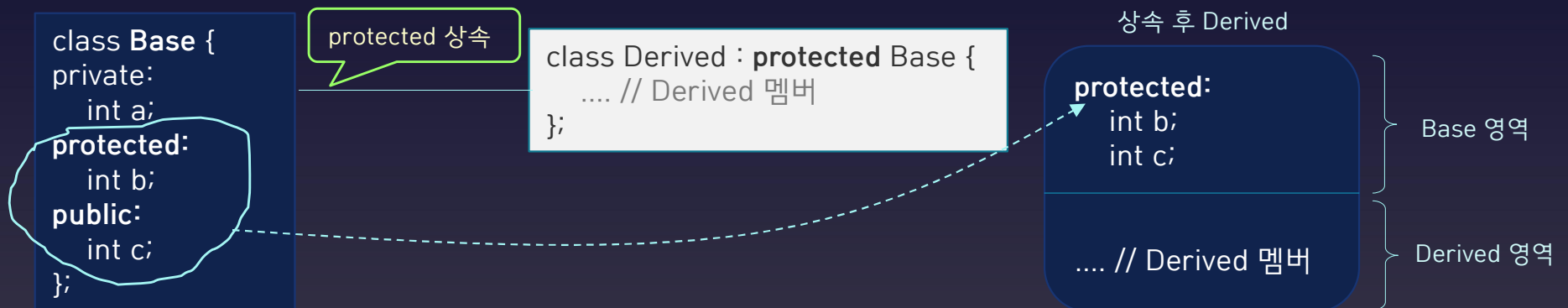


htv

# 상속 지정

- 상속 선언 시 `public`, `private`, `protected`의 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

# 접근 지정 속성 변화



# 예제 - protected 상속 사례

- 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
}
```

# 예제 - protected 상속 사례

- 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, ⑥

# 예제 - 상속이 중첩될 때 접근 지정

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

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



# 예제 - 상속이 중첩될 때 접근 지정

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

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

컴파일 오류

③, ④

실습

- 아래와 같은 코드에서 `Circle`을 상속받은 `NamedCircle`클래스를 작성하시오

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

class Circle {
    int radius;
public:
    Circle(int radius=0) {
        this->radius = radius;
    }
    int    getRadius() { return radius; }
    void   setRadius(int radius) { this->radius = radius; }
    double getArea() { return 3.14*radius*radius; }
};

int main() {
    NamedCircle waffle(3, "waffle"); // 반지름 3, 이름 waffle
    waffle.show();
}
```

실수 - 다

```
class NamedCircle : public Circle {
    string name;
public:
    NamedCircle(int radius, string name);
    void show();
};

NamedCircle::NamedCircle(int radius, string name)
    : Circle(radius) {
    this->name = name;
}

void NamedCircle::show() {
    cout << "Radius: " << getRadius() << "Name : " << name << endl;
}
```

