08

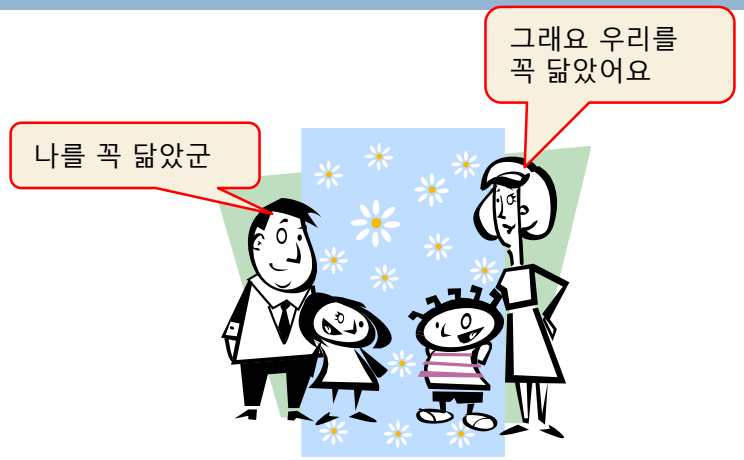
상속

학습 목표

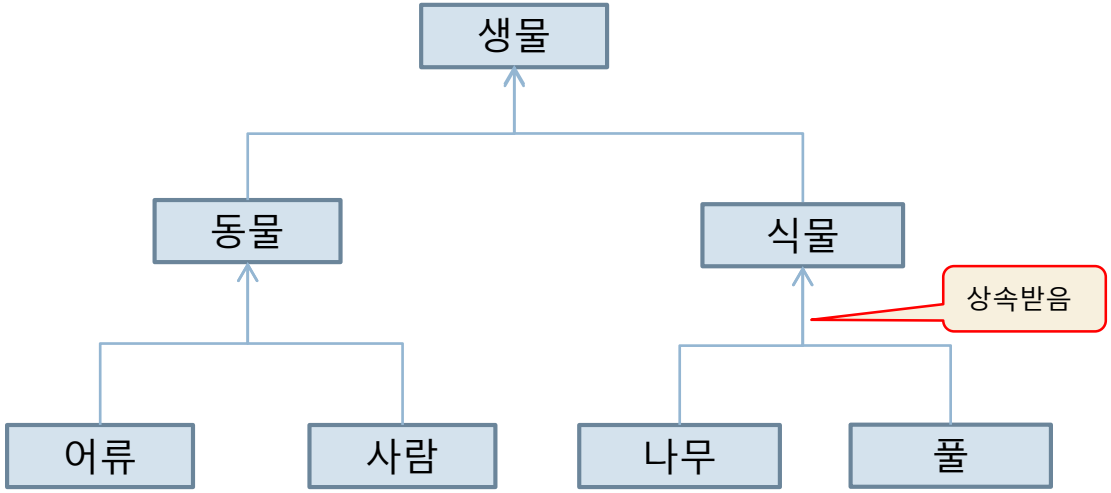
1. C++ 객체 지향 상속의 개념을 이해한다.
2. 상속을 선언하는 방법을 알고, 파생 클래스의 객체에 대해 이해한다.
3. 업 캐스팅과 다운 캐스팅 등 상속과 객체 포인터 사이의 관계를 이해한다.
4. protected 접근 지정에 대해 이해한다.
5. 상속 관계에 있는 파생 클래스의 생성 및 소멸 과정을 이해한다.
6. public, protected, private 상속의 차이점을 이해한다.
7. 다중 상속을 선언하고 활용할 수 있다.
8. 다중 상속을 문제점을 이해하고, 가상 상속으로 해결할 수 있다.

유전적 상속과 객체 지향 상속

유산 상속



유전적 상속 : 객체 지향 상속



유전적 상속과 관계된
생물 분류

C++에서의 상속(Inheritance)

- 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

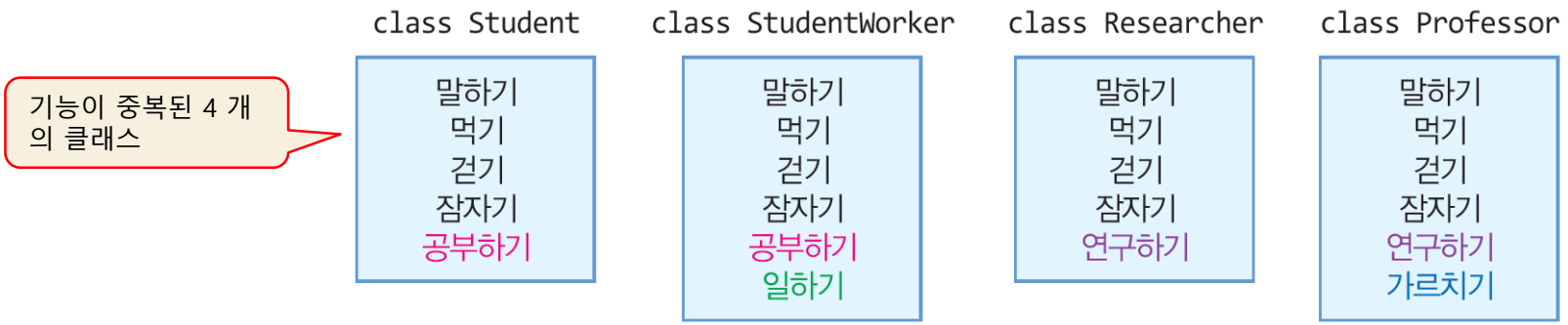
상속의 목적 및 장점

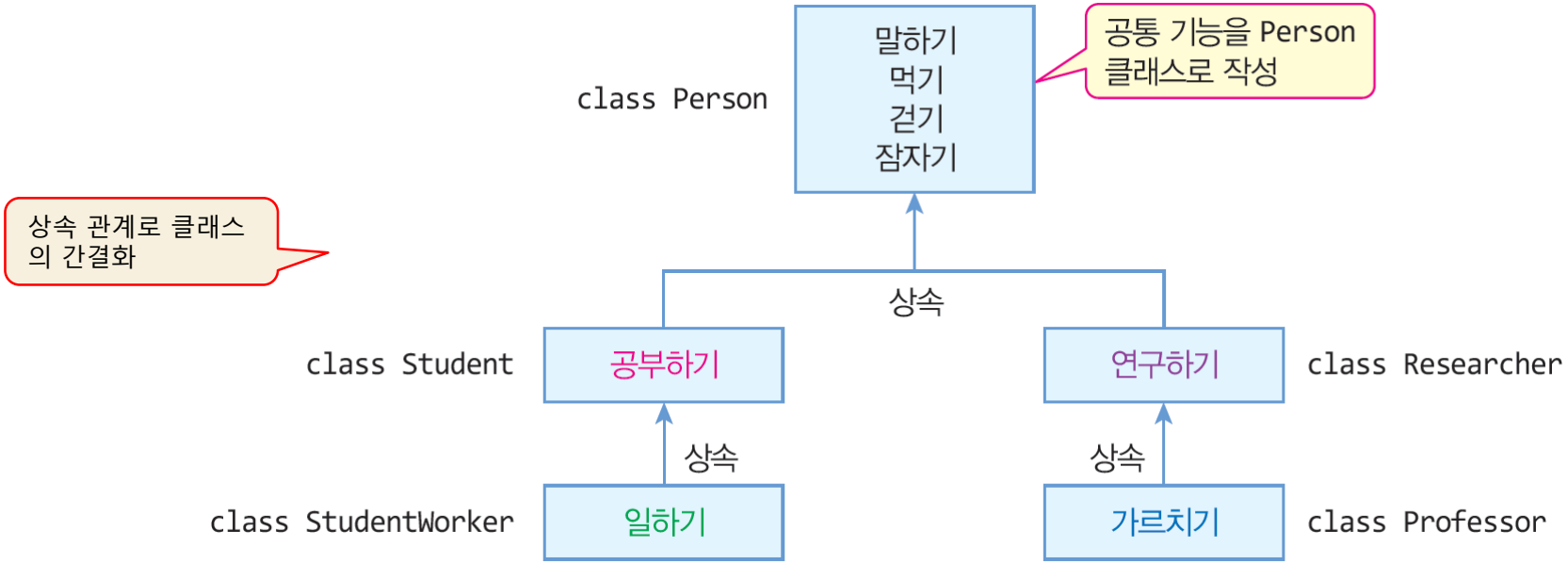
1. 간결한 클래스 작성
   - 기본 클래스의 기능을 물려받아 파생 클래스를 간결하게 작성

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

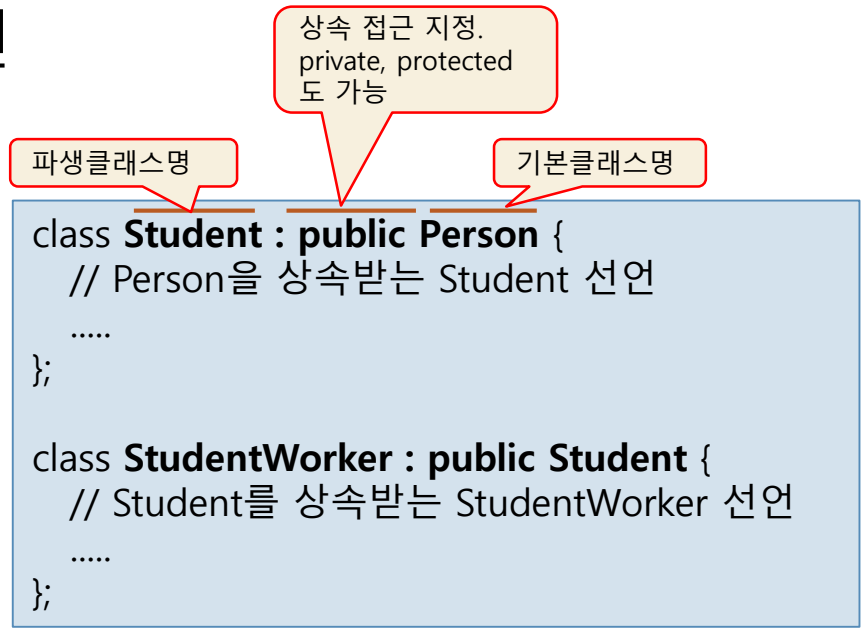
상속 관계로 클래스의 간결화 사례





상속 선언

상속 선언



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

예제 8-1 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
}
```

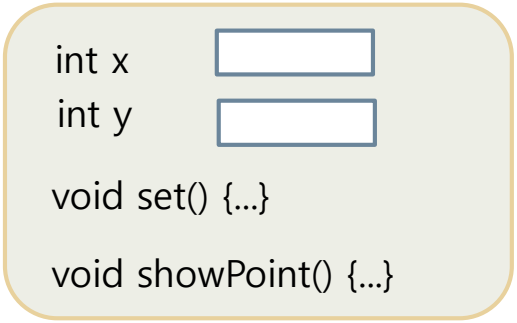
```
Red:(3,4)
```

파생 클래스의 객체 구성

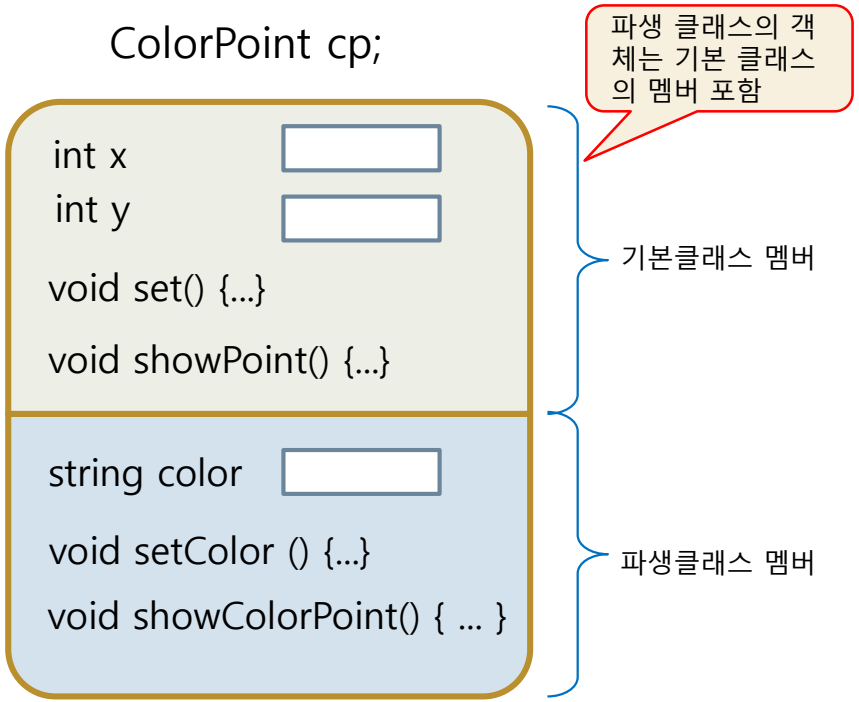

```
class Point {
   int x, y; // 한 점 (x,y) 좌표 값
public:
   void set(int x, int y);
   void showPoint();
};
```

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

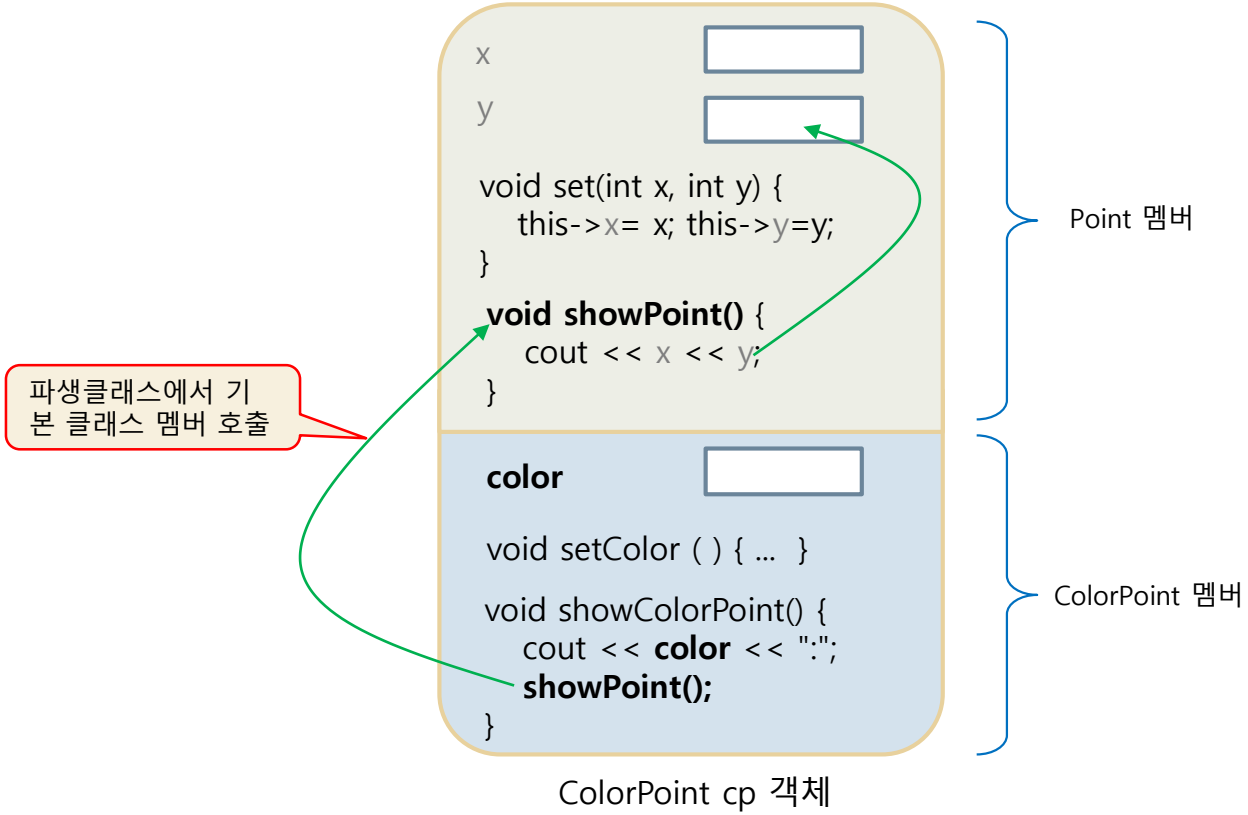
Point p;



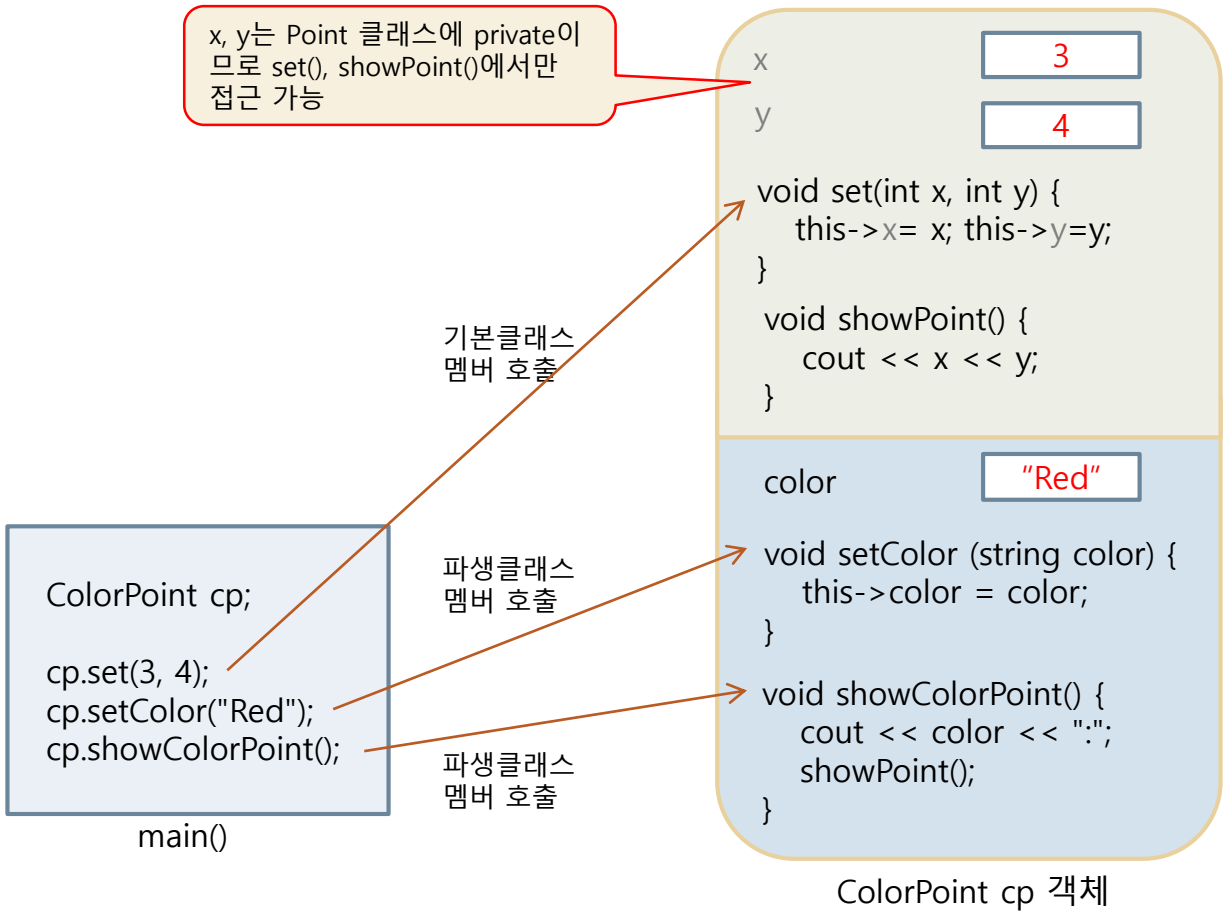
ColorPoint cp;



파생 클래스에서 기본 클래스 멤버 접근

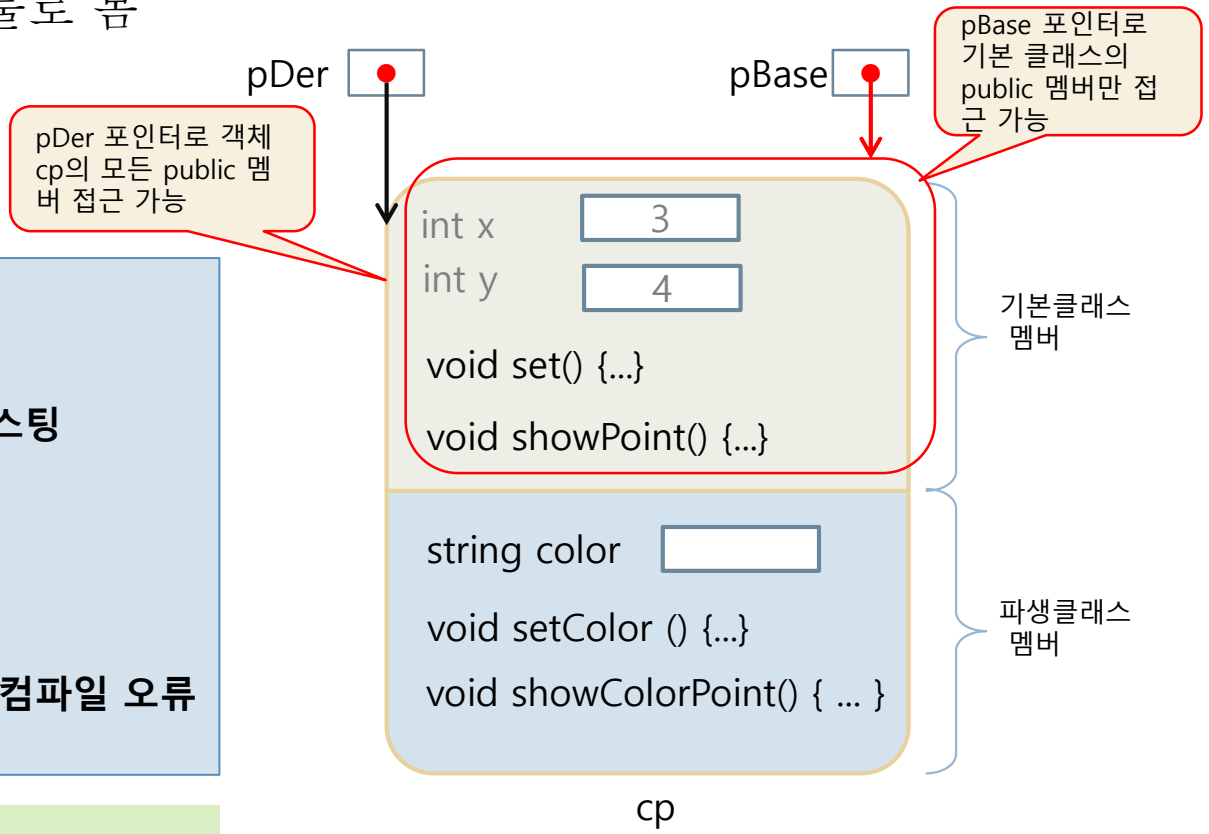


외부에서 파생 클래스 객체에 대한 접근



상속과 객체 포인터 – 업 캐스팅

- 업 캐스팅(up-casting)
  - 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)
```

업 캐스팅

생물을 가리키는 손가락으로 어류, 포유류, 사람, 식물 등 생물의 속성을 상속받은 객체들을 가리키는 것은 자연스럽습니다. 이것이 업 캐스팅의 개념입니다.

생물을 가리키는 손가락으로 컵을 가리키면 오류

상속과 객체 포인터 – 다운 캐스팅

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

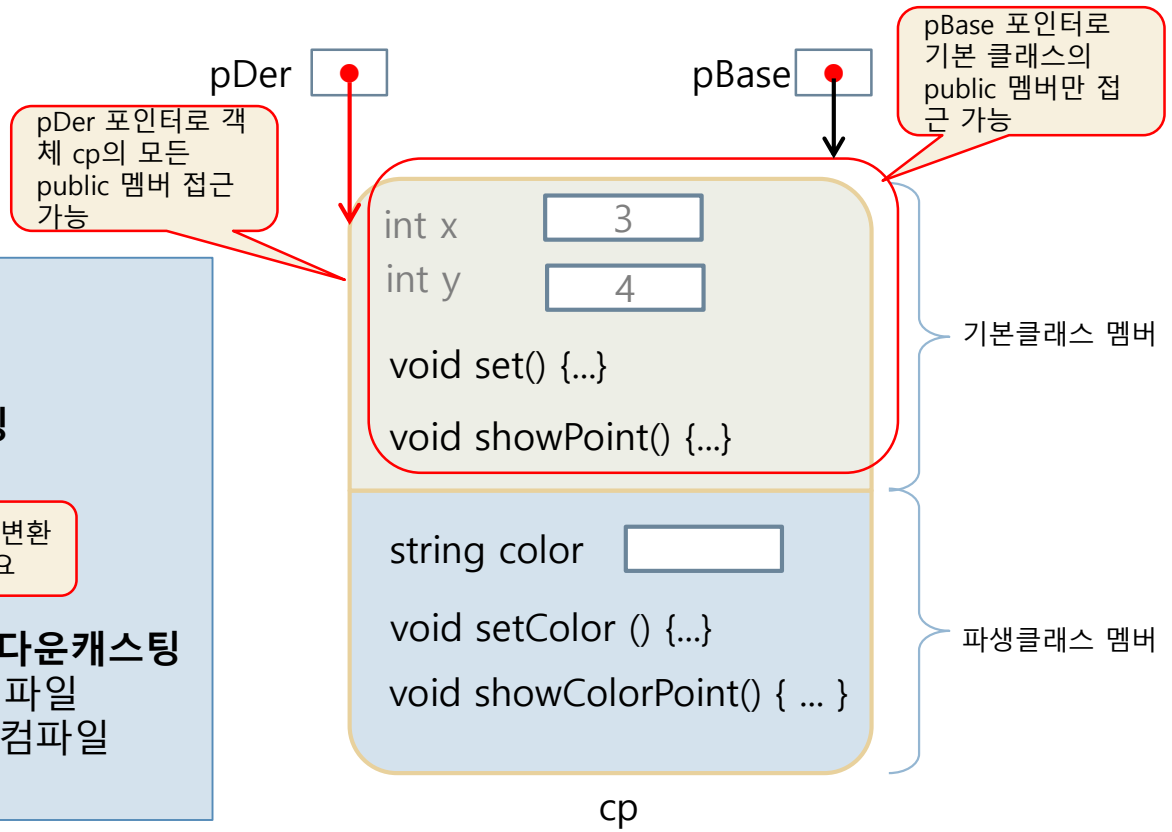
```
int main() {
   ColorPoint cp;
   ColorPoint *pDer;
   Point* pBase = &cp; // 업캐스팅

   pBase->set(3,4);
   pBase->showPoint();

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

강제 타입 변환 반드시 필요

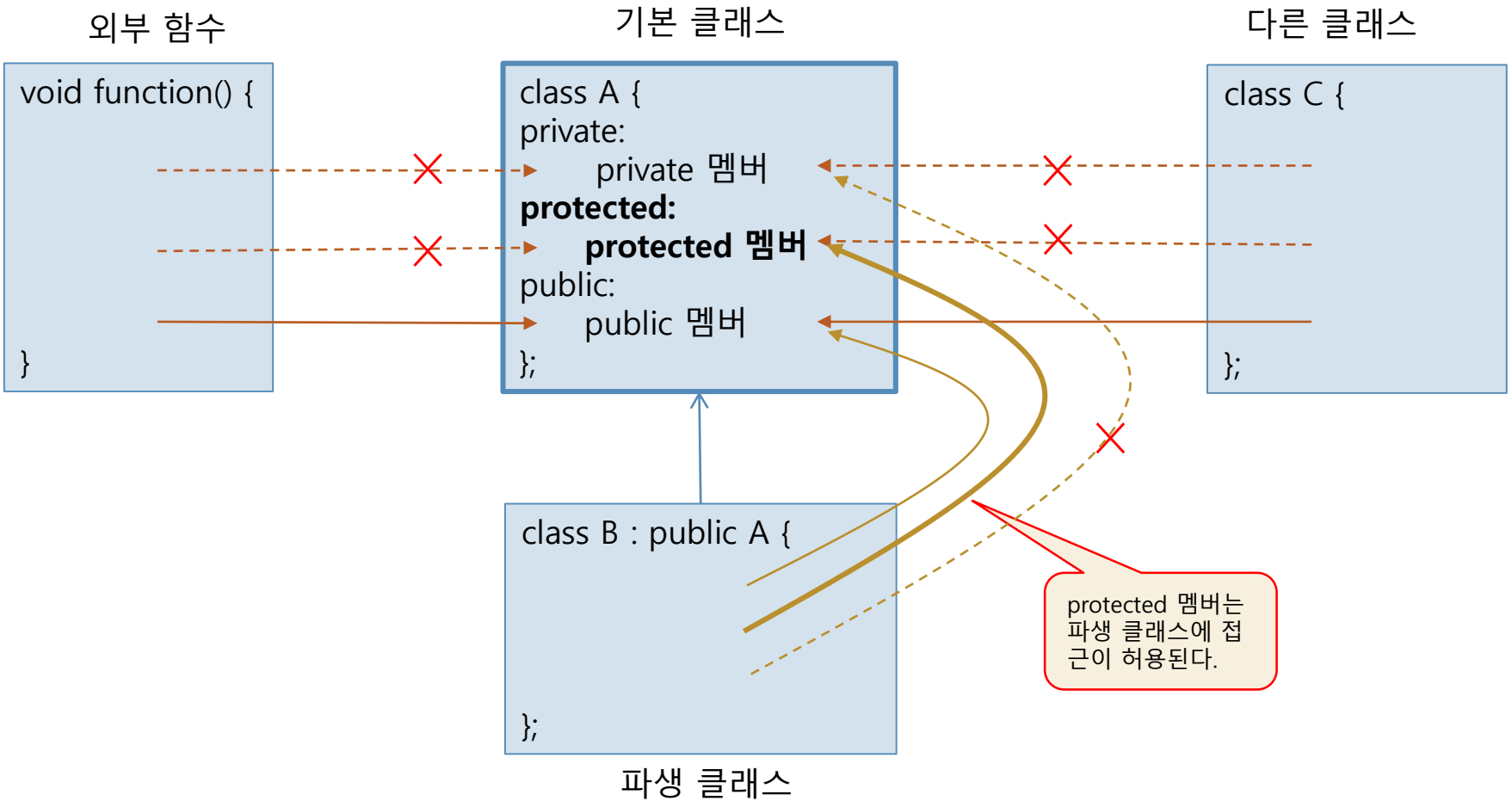
```
(3,4)
Red(3,4)
```



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



예제 8-2 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
}
```