정적(static) 변수

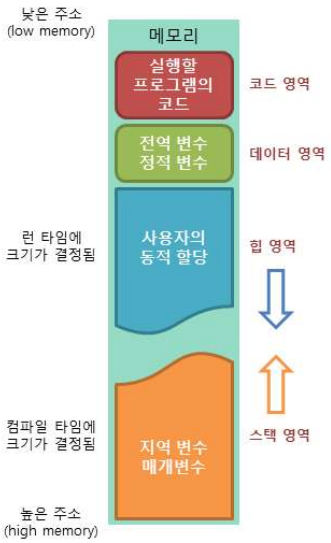
```
void Counter()
{
    static int cnt;
    cnt++;
    cout<<"Current cnt: "<<cnt<<endl;
}
int main(void)
{
    for(int i=0; i<10; i++)
        Counter();
    return 0;
}
```

static 변수는 초기화 하지 않으면 0 으로 초기화(한번만)

```
Current cnt: 1
Current cnt: 2
Current cnt: 3
Current cnt: 4
Current cnt: 5
Current cnt: 6
Current cnt: 7
Current cnt: 8
Current cnt: 9
Current cnt: 10
```

지역변수, 전역변수, 정적변수와 메모리 공간

- 지역변수, 매개변수
  - 함수 시작하면 메모리의 stack 영역에 변수 저장 공간 생성
  - 이들 변수 선언한 함수 종료되면 변수 저장 공간 없어짐
- 전역변수, 정적변수
  - 프로그램 시작하면 메모리의 data 영역에 이들 변수 저장 공간 생성
  - 프로그램 종료되면 변수 저장 공간 없어짐



정적(static) 변수 - 전역 변수

```
int count = 0;            // 전역변수
void func1( void ){
    printf( "%d ₩n", ++count );
}

void main( void ){
    func1();
    count = 9;    // 전역 변수에 접근이 가능
    func1();
}
```

<결과>
count = 1
count = 10

```
void func1( void ){
    static int count = 0;
    printf( "%d ₩n", ++count );
}

void main( void ) {
    func1();
   // count = 9;   ,  컴파일 에러,
    func1();
}
```

<결과>
count = 1
count = 2

=> 전역변수와 static 변수는 비슷하게 동작(함수 빠져 나와도 값이 남아 있음)

- 전역변수는 모든 함수가 사용할 수 있는 변수이다.
- static 변수는 전역변수이다. 단, 선언한 함수만 사용(접근)할 수 있는 전역변수이다.
  ➔ **"전역 변수를 선언하고, 특정 함수에서만 사용하고 싶을 때"' 사용**

참고) 정적(static) 변수

<참고> :

- static 변수 : 선언한 함수만 사용(접근)할 수 있는 전역변수이다.

- 전역변수 : 모든 파일, 모든 함수에서 사용 가능
- static 변수 : 여러 개의 파일을 컴파일 할 때 변수 이름의 충돌을 막아 준다. 즉 '그 파일에서만 그 변수를 사용하고 싶을 때' 사용. → 다른 파일에서는 접근 불가

- 일반 함수 : 모든 파일, 모든 함수에서 사용 가능
- static 함수 : 이 것도 마찬가지로 여러 개의 파일을 컴파일 할 때 함수 이름의 충돌을 막아 준다. 즉 ' 그 파일에서만 그 함수를 사용하고 싶을 때' 사용하면 된다. → 다른 파일에서는 호출 불가

- *gg*는 f1, f2, f3 모두 사용 가능

a.cpp	b.cpp
int gg=0; void main(){ ...} void f1(){...}	extern int gg; void f2(){ ...} void f3(){...}

- f2, f3 에서 *gg* 접근 불가

x.cpp	y.cpp

static int gg=0;
void main(){ ...}
void f1(){
...
	// gg 사용 못함
	void f2(){ ...}
	void f3(){...}

정적(static) 변수

- 함수의 static 변수 :
  - 선언한 함수만 사용(접근)할 수 있는 전역변수이다.
  - **"전역 변수를 선언하고, 특정 함수에서만 사용하고 싶을 때" 사용**
- 클래스의 static 변수 :
  - 선언한 클래스의 객체들만 사용(접근)할 수 있는 해당 클래스의 전역변수.
  - **"전역 변수를 선언하고, 특정 클래스의 객체들만이 사용하고 싶을 때" 사용**

정적 멤버

- 인스턴스 변수(instance variable), 일반 멤버변수: 객체마다 하나씩 있는 변수
- 정적 변수(static variable): 모든 객체를 통틀어서 하나만 존재하는 변수
  - 같은 클래스 객체들의 전역변수
  - 생성된 자동차(객체)들 개수 원함 → 각 객체(인스턴스) 멤버변수에 저장 못함
  - 정적 변수 사용으로 해결 → 초기값 0, 생성된 각 객체가 1 씩 증가
    - 객체 생성자에서 정적 변수 1씩 증가시킴

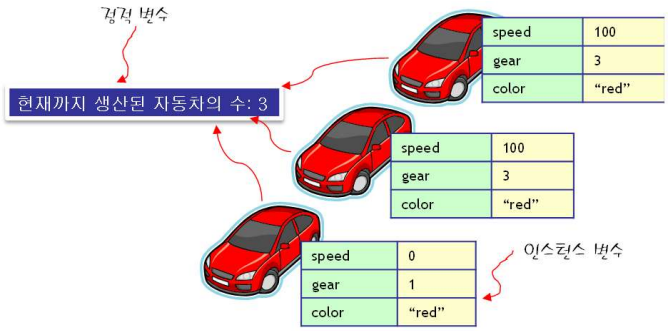


그림 7.6 정적 멤버

정적 멤버 변수

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

class Car {
        int speed;
        int gear;
        string color;
public:
        static int count;

        Car(int s=0, int g=1, string c="white"): speed(s), gear(g), color(c) {
                count++;  // 객체 생성시(생성자 호출)시 마다 1 증가
        }
        ~Car() {
                count--;
        }
};
int Car::count = 0;          // 정적 변수 초기화
```

정적 변수의 선언
Car 클래스 에서 만들어진
모든 객체에서 사용 가능
1개만 존재
초기화는 클래스 외부에서 수행

반드시 외부에서 정적 변수 초기화, 초기화 안하면 오류

정적 멤버 변수

반드시 외부에서 정적 변수 초기화, 초기화 안하면 오류

```
int Car::count = 0;          //①

int main(){
        cout <<"지금까지 생성된 자동차수= " << Car::count << endl;       // c1.count 불가

        Car c1(100, 0, "blue");              // count 1 증가 → 1
        Car c2(0, 0, "white");               // count 1 증가 → 2

        cout <<"지금까지 생성된 자동차수= " << Car::count << endl; //
        cout <<"지금까지 생성된 자동차수= " << c1.count << endl; // 가능
        cout <<"지금까지 생성된 자동차수= " << c2.count << endl; // 가능

        Car c3(0, 0, "red");                 // count 1 증가 → 3
        cout <<"지금까지 생성된 자동차수= " << c1.count << endl; //
        cout <<"지금까지 생성된 자동차수= " << c2.count << endl; //

        return 0;
}
```

정적변수 **count** 가 **private** 인 경우
아래와 같이 **main** 에서 접근 불가

```
지금까지 생성된 자동차 수 = 0
지금까지 생성된 자동차 수 = 2, 2, 2
지금까지 생성된 자동차 수 = 3, 3
```

count

c1
c2
c3

정적 멤버 변수 접근

- 앞의 예제, main 에서(정적변수를 public 으로 선언한 경우)

```
cout << c1.count << endl;          // ... (1)
cout << Car::count << endl;        // ... (2)
```

- (1) 과 같은 표현은 count 가 멤버변수인 것 같은 오해 발생
  - 클래스 내부에 있는 멤버변수 아니라 클래스 전역 변수
  - 각 객체가 가지고 있는 멤버변수 아님
  - 클래스 전체적으로 하나만 존재
- (2) 와 같은 표현이 좋음

정적 멤버 변수(private 로 선언한 경우)

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

class Car {
        int speed;
        int gear;
        string color;
        static int count;
public:
        Car(int s=0, int g=1, string c="white"): speed(s), gear(g), color(c) {
                count++;  // 객체 생성시(생성자 호출)시 마다 1 증가
        }
        ~Car() {
                count--;
        }
        int getCount() { return count; }
};
int Car::count = 0;             // 정적 변수 초기화
```

정적 멤버 변수(private 로 선언한 경우)

```
int main(){
        // Error, private
        //cout <<"지금까지 생성된 자동차수= " << Car::count << endl; // error

        Car c1(100, 0, "blue");          // count 1 증가 → 1
        Car c2(0, 0, "white");           // count 1 증가 → 2

        // 아래 두줄 코드 모두 오류, private
        //cout <<"지금까지 생성된 자동차수= " << c1.count << endl; // error
        //cout <<"지금까지 생성된 자동차수= " << c2.count << endl; // error

        cout <<"지금까지 생성된 자동차수= " << c1.getCount() << endl; // 가능
        cout <<"지금까지 생성된 자동차수= " << c2.getCount() << endl; // 가능

        return 0;
}
```

정적 멤버 함수

- 정적 멤버 함수는 static 수식자를 멤버 함수 선언에 붙인다.
- 클래스 이름을 통하여 호출 ← 클래스당 한 개 존재
- 정적 멤버 함수도 클래스의 모든 객체들이 공유

```
class Car {
        ...
public:
        static int count;    // 정적변수의 선언

        ...
        // 정적 멤버 함수
        static int getCount() {
                return count;
        }
};
```

정적 멤버 함수

```
class Car {
        ...
public:
        static int count;     // 정적변수의 선언

        ...
        // 정적 멤버 함수
        static int getCount() {
                return count;
        }
};
int Car::count=0;     // 정적 변수의 정의

int main()
{
        Car c1(100, 0, "blue");
        Car c2(0, 0, "white");
        int n = Car::getCount();
        cout <<"지금까지 생성된 자동차수= " << n << endl;
        return 0;
}
```

- 정적변수를 정적 함수에서 사용

```
지금까지 생성된 자동차 수 = 2
계속하려면 아무 키나 누르십시오 . . .
```

주의할 점

- 정적 멤버 함수 → 객체 생성 전에 호출 가능한 함수
  - 정적 멤버 함수에서 일반 멤버 변수들은 사용할 수 없다. ← static 변수만 사용가능
  - 정적 멤버 함수 내에서 일반 멤버 함수를 호출하면 역시 오류
  - this 포인터 사용 못함 → this 포인터는 객체의 주소

```
class Car {
        int speed;
        ...
public:
        int getSpeed() {
                return speed;
        }
        static int break() {
                int s = getSpeed();     // 오류: 일반 멤버 함수는 호출할 수 없음
                speed = 0;              // 오류: 일반 멤버 변수는 접근할 수 없음
                return s;
        }
};
```

정적 멤버 함수에서 일반 멤버는 사용할 수 없다.

정적 멤버 변수, 정적 멤버 함수 비교

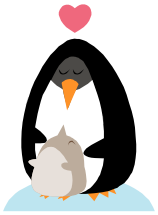
- static 멤버 변수
  - 객체 생성 전에 생성 -> 프로그램 실행과 동시에 클래스 밖에 있는 선언부에서 초기화되어 메모리 공간에 할당.(객체 생성 전에 메모리 할당 및 초기화)
    - static 멤버 변수는 객체 별로 존재하는 변수가 아님(객체 내부에 존재하지 않음),
  - 프로그램 전체 영역에서 하나만 존재하는 변수이다. → 멤버변수 아님(멤버변수는 각 객체마다 따로 존재)
  - 일반 멤버함수에서도 접근 가능, static 멤버 함수에서도 접근 가능

- static 멤버 함수(참고)
  - 객체 생성 전에 static 함수 존재 → 생성 되지 않은 객체의 멤버변수/함수 호출 못함.
    - this 포인터 사용 금지 ← this 포인터는 객체 자신을 지칭
    - static 멤버 함수는 객체 내에 존재하는 함수가 아님
  - static 멤버 함수는 static 변수에만 접근 가능하고, static 멤버 함수만 호출 가능하다.
  - 용도 : 주로 private인 static 멤버 변수에 접근하려 할 때 많이 사용

클래스와 클래스 간의 관계

- 사용(use): 하나의 클래스가 다른 클래스를 사용한다.
- 포함(has-a): 하나의 클래스가 다른 클래스를 포함한다.
- 상속(is-a): 하나의 클래스가 다른 클래스를 상속한다. → 8장

- 클래스 설계시 위의 관계가 있으면 규칙에 따라 설계



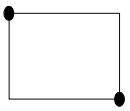
사용 관계

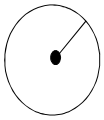
- 클래스 A의 멤버 함수에서 클래스 B의 멤버 함수들을 호출

```
ClassA::func()
{
        ClassB   obj;        // 사용 관계
        obj.func();          // public 이면 호출 가능
        ...                  // private 이면 호출 불가
}
```

포함(has-a) 관계

- has-a 관계(7장)
  - 한 객체가 다른 객체를 포함하는 관계 →" a 는 b 를 포함한다.(가지고 있다.) " 성립
    - 자동차는 바퀴를 포함한다.
    - 사각형은 두점(좌상단점, 하단점)을 가지고 있다.
    - 원은 중심점과 반지름을 가지고 있다.
    - 도서관은 책을 가지고 있다.
    - 학생은 볼펜을 가지고 있다. …





- is-a 관계(8장)
  - 한 객체가 다른 객체의 특수한 경우 → "a 는 b 이다 " 성립
    - 승용차는 자동차 이다. 트럭은 자동차이다.
    - 사자, 개, 고양이 는 동물이다.
- → 위의 2 경우 모두 a, b 를 클래스로 만들고 관계를 만들어줌

포함(has-a) 관계

- has-a 관계(7장)
  - 한 객체가 다른 객체를 포함하는 관계 →" a 는 b 를 포함한다.(가지고 있다.) " 성립
    - 원은 중심점과 반지름을 가지고 있다.

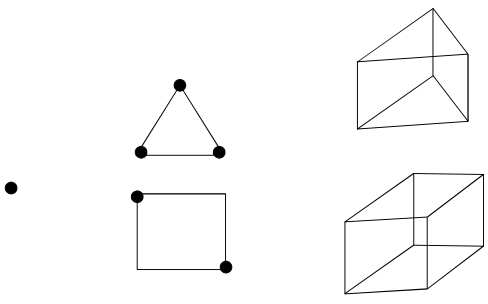
- 원(반지름, 중심점) → 클래스, 반지름 → 단순 하나의 값(int)
  중심점(x 좌표, y 좌표) → 다른 클래스 객체,

- 객체가 다른 객체에서도 사용 가능하면 클래스로 만든다
  - 점 클래스 → 원의 중심점, 사각형 좌상단점/우하단점, 직선 시작점/끝점 에서 사용 가능
  - 클래스는 "재사용 가능" 이 중요 개념

포함(has-a) 관계

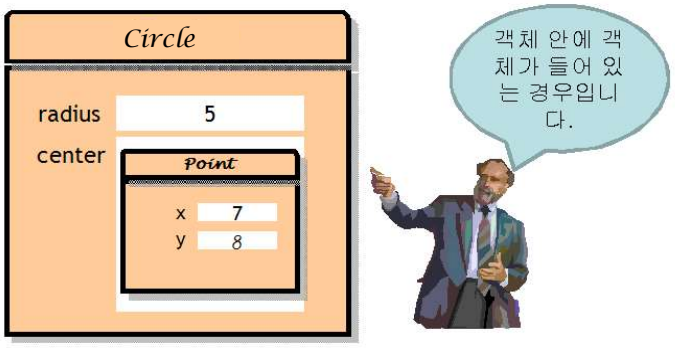
- 실생활에서 포함 관계 많음
- 포함 관계 1
  - 점 → 삼각형, 사각형 … → 사각 기둥, 삼각 기둥

- 포함 관계 2
  - 학생 → 과 동아리 → 학과



6장 예제 3(포함, has-a 관계) - 235 쪽

- Circle 객체 안에 Point 객체가 들어 있는 경우



포함(has-a) 관계, 6장 예제 3

```
// 방법 1
class Circle_1 {
private:
   int radius;
   int x, y;
   …
};
```

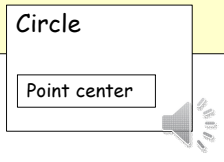
```
// 방법 2
class Point {
private:
   int x;
   int y;
   …
};

class Circle_2 {
private:
   int radius;
   Point center;
   …
};
```

등장 객체 파악(2 가지 방법 표현 가능) : 원 속성은 반지름, 중심점

1. 원 → 중심점을 (x, y) 로 표현
2. 원 → 중심점을 Point 클래스로 표현

Circle_1 클래스 보다는 Circle_2 클래스가 더 좋음.

- 결과는 두 방법 모두 동일
- 최소단위로 작성하면 다른 곳에 재활용 가능
- Point 클래스는 선분 양 끝점, 다각형 꼭지점들 등 다양하게 사용 가능.
- 실제 도형 표현시 방법 2 좋음(점은 원, 사각형, 직선들 표현시 이들 모두에서 사용)



has-a 관계 사용시

```
class Tri {
   int px1, py1, px2, py2;
   int px3, py3;
   ...
};

class Rec {
   int px1, py1, px2, py2;
   int px3, py3, px4, py4;
   ...
};
```

```
class Point {
   int px, py
   ...
};

class Tri {
   Point p1, p2, p3;
   ...
};

class Rec {
   Point p1, p2, p3, p4;
   ...
};
```

생성자에서 멤버 초기화(6장 내용)

```
class Car {
      int speed;     // 속도
      int gear;      // 기어

public:
      Car(int s, int g) {        … (1)
            speed = s;
            gear = g;
            int x = 0;           … (3)
      }

      Car(int s, int g)          … (2)
        : speed(s), gear(g)
      {    }

      Car( )                     … (4)
      {    }
```

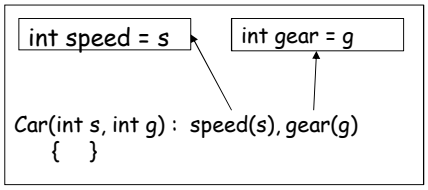
생성자에서 초기화는 (1) or (2) 가능

(3) 은 지역변수 → 생성자 안에서 생성

speed, gear 는 멤버변수 → 생성자 함수 들어가기 바로 전에 멤버변수 미리 생성

(4) 경우 생성자 수행 전에 멤버변수는 생성됨(쓰레기 값 저장)

아래 그림은 (2) 에서 객체/멤버 생성 과정 설명



Has-a 관계 생성자에서 멤버 초기화 1 (6장 내용)

```
class Point {
   int x, y;
public:
   Point(int xx, int yy)
      : x(xx), y(yy)
   { }
      …
};

class Circle {
   int radius;
   Point center;

public:
  Circle(int xx, int yy, int r)
      : radius(r), center(xx, yy) … (1)
   { }
      …
};
```

원 → (중심점, 반지름) → class 로 구성
점 → (x, y) → class 로 구성

(1) 에서
- radius(r) → int radius(r); 로 해석
- center(x, y) → Point center(x, y); 로 해석

생성자 수행 전에 멤버 변수/객체 생성 → center 객체 생성시 Point class 생성자 호출.

Has-a 관계 생성자에서 멤버 초기화 2

```
class Point {
   int x, y;
public:
   Point(int xx=0, int yy=0)
      : x(xx), y(yy)
   { }
      …
};
```

- **생성자 함수 수행 전에 멤버 변수/멤버객체 는 미리 생성됨**
- (1)은 객체 생성시 인수 사용 Point 생성자 호출
- (2) 는 객체 명시 없어도 Point 객체 생성 → Point 생성자 호출 ← center 생략된 것.
- (3)은 멤버 둘다 생략된 것. ← radius 는 쓰레기값 저장. Point 생성자 호출
- (4)는 Point 복사 생성자 호출
- (5)는 생성자 안에서 값/객체 할당

```
class Circle {
   int radius;
   Point center;
public:
 Circle(int xx, int yy, int r)      … (1)
      : radius(r), center(xx, yy)
  { }

  Circle(int xx, int yy, int r)      … (2)
      : radius(r)
  { }

  Circle(int xx, int yy, int r)   { }   … (3)

  Circle(Point p, int r) : radius(r),  center(p)  … (4)
  { }

 Circle(Point a, int r){                  … (5)
      radius = r;
      center = a;          // Point center=a 는
                           // 지역객체 생성하는 것
 }
      …
};
```

Has-a 관계 생성자에서 멤버 초기화 2

```
class Point {
  int x, y;
public:
  Point(int xx=0, int yy=0)
     : x(xx), y(yy)
  { }
     …
};

class Circle {
  int radius;
  Point center;

public:
 Circle(int xx, int yy, int r)      … (1)
     : radius(r), center(xx, yy)
  { }

  Circle(int xx, int yy, int r)
     : radius(r)                    … (2)
  { }
     …
};
```

생성자 작성 → (1) 혹은 (2) 가능

생성자 함수 들어가기 전에 멤버 변수/객체는 미리 생성됨

(1)은 객체 생성시 인수 사용 생성자 호출

(2) 는 객체 명시 없지만 멤버 Point 객체 생성 → 생성자 호출

has-a 관계에서 다른 클래스의 생성자 호출

객체 생성시 생성자 호출된다. → 객체를 만들면 생성자 호출됨

```
class Point {
private:
  int x, y;
public:
  Point(int xx, int yy) : x(xx), y(yy) { }
     …
};

class Circle {
private:
  int radius;
  Point center;

public:

  Circle(int xx, int yy, int r)
     : radius(r), center(xx, yy)
  { }
     …
};
```

```
main(){
     Circle a(5, 1, 2); …
```

```
        a
radius=2
center=(5, 1)
```

- radius(r) → int radius(r); 로 해석
변수 생성

- center(x, y) → Point center(x, y); 로 해석
객체 생성 → Point class 생성자 호출.

has-a 관계에서는
생성자에서 다른(포함되는) 객체의 생성자 호출을 위하여 초기화 목록을 사용하여야 한다.

예제 3 - 생성자 구성

```
// 클래스 구성하는 문제.

int main() {
   Point p(5, 3);

   Circle c1, c2(3), c3(p, 4), c4(9, 7, 5);

   c1.print();
   c2.print();
   c3.print();
   c4.print();
   return 0;
}
```

```
중심: ( 0, 0 )
반지름: 0
중심: ( 0, 0 )
반지름: 3
중심: ( 5, 3 )
반지름: 4
중심: ( 9, 7 )
반지름: 5
```

// Circle 클래스 생성자를 아래와 같은 결과 나오도록
// 구성하라. 변수 순서는 x, y, radius

1. 인수 없는 것→ c1 → (0, 0, 0) 으로 초기화
2. (인수 1개) → c2(r) → (0, 0, r) 으로 초기화
3. (인수 2개) → c3(p(x, y), r) → (x, y, r) 으로 초기화
4. (인수 3개) → c4(x, y, r) → (x, y, r) 으로 초기화

예제 3 - 235 쪽

```
// default 생성자는 멤버 값 지정 없이 객체 생성시 필요

class Point {
private:
     int x;
     int y;
public:
     Point();
     Point(int a, int b);
     void print();
};


void Point::print() {
   cout << "( " << x << ", " << y << " )\n";
}
```

```
class Circle {
private:
     int radius;
     Point center;
public:
     Circle();
     Circle(int r);
     Circle(Point p, int r);
     Circle(int x, int y, int r);
     void print();
};

void Circle::print() {
   cout << "중심: ";
   center.print();
   cout << "반지름: " << radius << endl << endl;
}
```

- main 에서 다음과 같이 객체 생성

```
Point p(5, 3);
Circle c1, c2(3), c3(p, 4), c4(9, 7, 5);
```

예제 3 - 235 쪽, 각 클래스 생성자들

```
Point p(5, 3);
Circle c1, c2(3), c3(p, 4), c4(9, 7, 5);
```

```
Point::Point() : x(0), y(0) { }  // 디폴트 생성자

Point::Point(int a, int b) : x(a), y(b) { }
```

```
Circle::Circle(): radius(0), center(0, 0){ }

Circle::Circle(int r) : radius(r), center(0, 0) { }

Circle::Circle(Point p, int r) : radius(r), center(p) { }

Circle::Circle(int x, int y, int r) : radius(r), center(x, y)  { }
```

```
main 에서
// x, y, r → 값 없는 것 0으로 지정
Circle c1, c2(3), c3(p, 4), c4(9, 7, 5);
```

center(x, y) → Point center(x, y); 로 해석
앞의 Point class 생성자 호출.

center(p) → Point center(p); 로 해석

→ 복사 생성자 호출(p의 멤버변수값을 center 의 멤버변수값으로 복사)

예제 3 - 235 쪽(다음도 가능)

```
Point p(5, 3);
Circle c1, c2(3), c3(p, 4), c4(9, 7, 5);
```

```
Point::Point() : x(0), y(0) { }  // 디폴트 생성자

Point::Point(int a, int b) : x(a), y(b) { }
```

```
Circle::Circle(): radius(0) { }    // Point 의 default 생성자(컴파일러가 생성) 자동 호출
//  Circle::Circle(): radius(0), center  { }       // Point center; 로 해석
                                                   // → Point class default 생성자 호출

Circle::Circle(int r) : radius(r) { }
// Circle::Circle(int r) : radius(r) , center { }


Circle::Circle(Point p, int r) : radius(r), center(p) { }

Circle::Circle(int x, int y, int r) : radius(r), center(x, y)  { }
```

멤버가 다른 객체인 경우(Has a 관계) 초기화

```
class Point {
    int x, y;
public:
    Point() : x(10), y(20) { }
    void prn() {
        cout << x << "  " << y << endl;
    }
};

class Rectangle {
    Point p1, p2;
public:
    Rectangle(int x1, int y1, int x2, int y2)  { }
    void prn() {
        p1.prn();
        p2.prn();
    }
};

int main() {
    Rectangle r1(10, 10, 100, 100);
    r1.prn();
    return 0;
}
```

Rectangle 생성자에
Point 객체 생성 코드 없는 경우
출력은 **?**

```
10 20
10 20
```

멤버가 다른 객체인 경우(Has a 관계) 초기화

```
class Point {
    int x, y;
public:
    Point() : x(10), y(20) { }
    Point(int x1, int y1) : x(x1), y(y1) { }
    void prn() {
        cout << x << "  " << y << endl;
    }
};

class Rectangle {
    Point p1, p2;
public:
    Rectangle(int x1, int y1, int x2, int y2)
        : p1(x1, y1), p2(x2, y2) { }
    void prn() {
        p1.prn();
        p2.prn();
    }
};

int main() {
    Rectangle r1(10, 10, 100, 100);
    r1.prn();
    return 0;
}
```

결과가 다음과 같으려면 **?**

```
10 10
100 100
```