정적(static) 변수

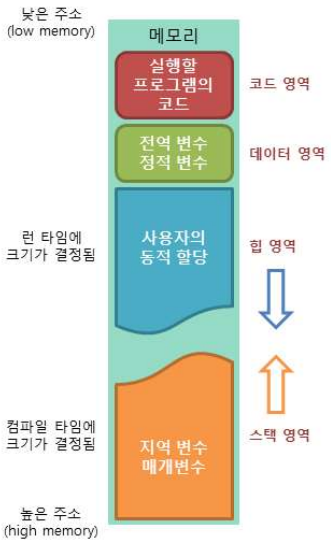
```
void Counter()
{
    static int cnt;
    cnt++;
    cout<<"Current cnt: "<<cnt<<endl;
}
int main(void)
{
    for(int i=0; i<10; i++)
        Counter();
    return 0;
}
```

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

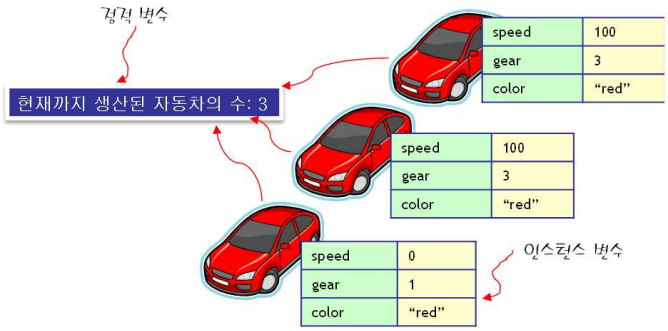


그림 7.6 정적 멤버

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(정적) 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;
}
```

클래스 정적 변수를 객체를 통하여 접근 → 다음 쪽 방법이 더 호율적

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
- **주로 private인 static 멤버 변수에 접근하려 할 때 많이 사용**

```
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

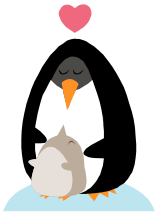
- static 멤버 변수
  - 객체 생성 전에 생성 -> 프로그램 실행과 동시에 클래스 밖에 있는 선언부에서 초기화되어 메모리 공간에 할당.(객체 생성 전에 메모리 할당 및 초기화)
    - static 멤버 변수는 객체 별로 존재하는 변수가 아님(객체 내부에 존재하지 않음),
  - 프로그램 전체 영역에서 하나만 존재하는 변수이다. → 멤버변수 아님(멤버변수는 각 객체마다 따로 존재)
  - 일반 멤버함수에서도 접근 가능, static 멤버 함수에서도 접근 가능

- 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



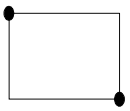
사용 관계

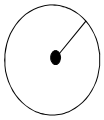
- 클래스 A의 멤버 함수에서 클래스 B의 멤버 함수들을 호출

```
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.

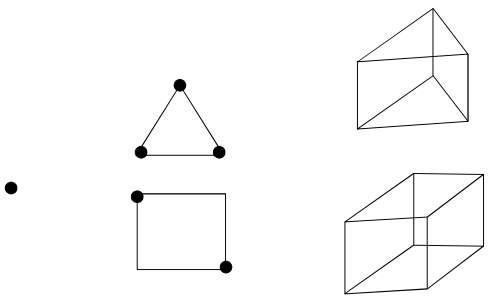
- 원(반지름, 중심점) → 클래스, 반지름 → 단순 하나의 값(int)
  중심점(x 좌표, y 좌표) → 다른 클래스 객체,

- 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

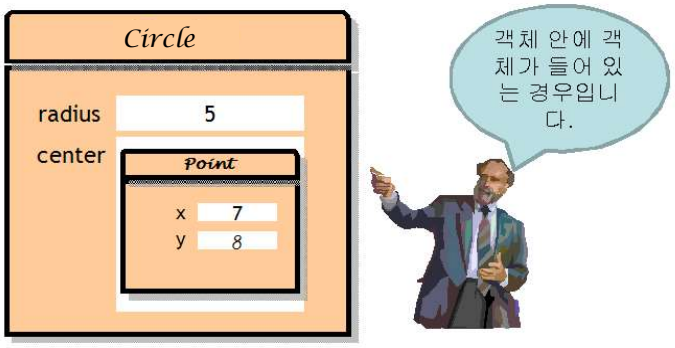
- 실생활에서 포함 관계 많음
- 포함 관계 1
  - 점 → 삼각형, 사각형 … → 사각 기둥, 삼각 기둥

- 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;
   …
};
```

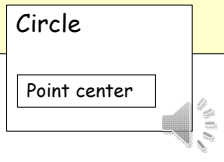
```
// 방법 2
class Point {
private:
   int x;
   int y;
   …
};

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)
      {    }
```

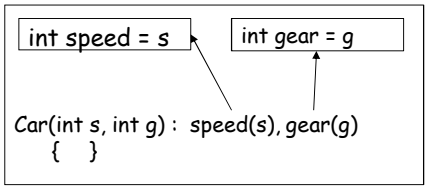
생성자에서 초기화는 (1) or (2) 가능

(3) 은 지역변수 → 생성자 안에서만 사용

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 사용하는 것이 좋음.

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

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


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

```
class Circle {
private:
     int radius;
     Point center;
public:
     Circle();     // default 생성자
     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 멤버 값 지정 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 멤버변수 값으로 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), center(0, 0){ }  대신해서 아래도 가능하지만 ...

→ Circle::Circle(): radius(0) { }          // Point 의 default 생성자(컴파일러가 생성) 자동 호출
→ Circle::Circle(): radius(0), center( ) { }
// Circle::Circle(): radius(0), center  { }  은 error


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

→ Circle::Circle(int r) : radius(r) { }
→ Circle::Circle(int r) : radius(r) , center( ) { }
// Circle::Circle(int r) : radius(r) , center { } 은 error
```

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 ?

p1( ), p2( )

```
10 20
10 20
```

7.9 포함 (has-a) 관계

```
// 시각을 나타내는클래스
class Time {
private:
   int time;
   int minute;
   int second;

public:
   Time();
   Time(int t, int m, int s);
   void print();
};

Time::Time() {
   time = 0;    minute = 0;
   second = 0;
}

Time::Time(int t, int m, int s) {
   time = t;     minute = m;
   second = s;
}
```

```
void Time::print()  {
  cout << time << "시" << minute << "분" <<
            second << "초₩n";
}

// Time class → 현재시각만 알려주는 시계
//                        (시각은 시, 분, 초)
// AlarmClock class → 현재시각, 알람시각을
//                        알려주는 시계

int main() {
            Time alarm(6, 0, 0);
            Time current(12, 56, 34);
            AlarmClock  c(alarm, current);

            c.print();
            return 0;
}
```

```
현재 시각: 12시 56분 34초
알람 시각: 6시 0분 0초
```

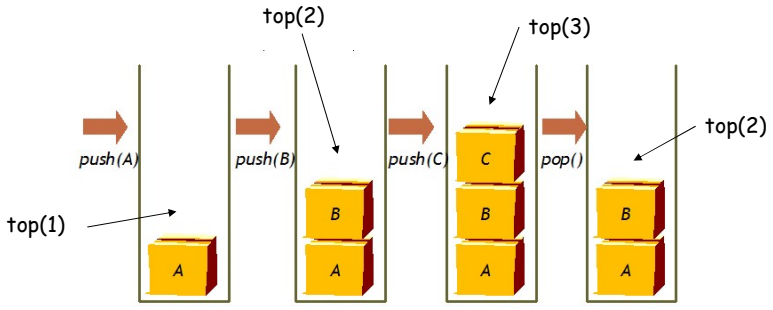
AlarmClock 이 Time 을 두 개 포함
: 현재 시각, 알람시각을 AlarmClock 이 포함

포함 (has-a) 관계

```
// 알람시계를 나타낸다.
class AlarmClock {
private:
        Time alarmTime;       // 알람시각, Time 클래스를 포함
        Time currentTime;     // 현재시각
public:
        AlarmClock(Time a, Time c);       // 생성자
        void print();                     // 객체의정보 출력
};

//AlarmClock::AlarmClock(Time a, Time c) {   // 생성자
//        alarmTime = a;
//        currentTime = c;
//}

AlarmClock::AlarmClock(Time a, Time c)       // 생성자(위 내용보다 권장)
        : alarmTime(a), currentTime(c)
{ }

void AlarmClock::print(){
   cout << "현재시각: ";          currentTime.print();
   cout << "알람시각: ";          alarmTime.print();
}
```

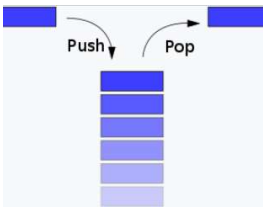
스택

- 스택 정의 → LIFO(Last In First Out , 최근에 저장된 것이 먼저 나옴)
  - 스택은 한 쪽 끝에서만 자료를 넣거나(push) 뺄(pop) 수 있는 선형 구조,
  - top : 스택에서 저장할 공간의 인덱스
  - push(n) : 인자 n을 top 위치에 저장하고 top++, true 리턴,
    - 스택이 꽉 차 있으면 false return
  - pop(n) : top--, top에 있는 값을 인자 n으로 반환하고 true return,
    - 스택이 비어 있으면 false return



스택

- 정수를 저장하는 스택, 스택에 저장할 수 있는 정수의 최대 갯수는 생성자에서 매개변수로 받음,
  - default 생성자에서는 10개 저장하는 스택 생성
  - 생성자에서는 매개변수로 전달된 갯수의 정수 저장하는 저장공간 생성
  - 스택 복사 생성 가능
- 멤버함수
  - default 생성자, 생성자, 소멸자, 복사 생성자
  - push, pop : 앞쪽 설명
- 멤버변수
  - 스택에서 저장 할 위치 인덱스 저장 변수(top)
  - 스택 저장 공간 갯수
  - 저장공간 → 정수형 pointer 변수, 동적 생성



스택

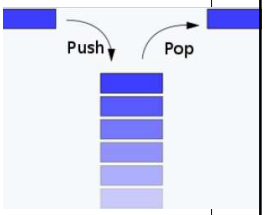
- **정수를 저장하는 스택, 스택에 저장할 수 있는 정수의 최대 갯수는 생성자에서 지정,**
- **default 생성자에서는 10개만 저장하는 스택 생성**

```
int main() {
	MyIntStack a(10);  // 생성자
	a.push(10);		// 정수 10 을 스택 a에 push(저장)
	a.push(20);

	MyIntStack b = a; // 복사 생성
	b.push(30);		// 정수 30 을 스택 b에 push (저장)

	int n;
	a.pop(n);// 스택 a pop,	"n = a.pop();" 구현도 가능
	cout <<  " 스택 a에서 pop 한 값 " << n << endl;

	b.pop(n); // 스택 b pop
	cout << "스택 b에서 pop 한 값 " << n << endl;
}
```

스택

- **정수를 저장하는 스택, 스택에 저장할 수 있는 정수의 최대 갯수는 생성자에서 지정,**
- **default 생성자에서는 10개만 저장하는 스택 생성**

```
class MyIntStack {
    int* p;    // 스택 메모리로 사용할 포인터
    int size;  // 스택의 최대 크기
    int top;   // 스택의 탑을 가리키는 인덱스, 스택에서 저장/pop 할 위치 index
public:
    MyIntStack();
    MyIntStack(int size);
    MyIntStack(const MyIntStack& s);// 복사생성자
    ~MyIntStack();
    bool push(int n); // 정수 n을 스택에 푸시한다. 스택이 꽉 차 있으면 false를, 아니면 true 리턴

    bool pop(int& n); // 스택의 탑에 있는 값을 n에 팝. 만일 스택이 비어 있으면 false를, 아니면 true 리턴
};
```

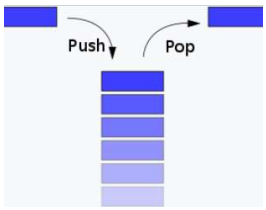


스택

```
MyIntStack::MyIntStack() {
    size = 10;      // default 크기
    top = 0;        // push/pop 위치
    p = new int[size]; // 스택 메모리 할당
}
MyIntStack::MyIntStack(int size) {
    this->size = size;
    top = 0;
    p = new int[size]; // 스택 메모리 할당
}
MyIntStack::MyIntStack(const MyIntStack& src) {
    size = src.size;
    top = src.top;
    p = new int[size];
    for (int i = 0; i < size; i++)
        p[i] = src.p[i]; // 원본 객체의 메모리 복사
}
```

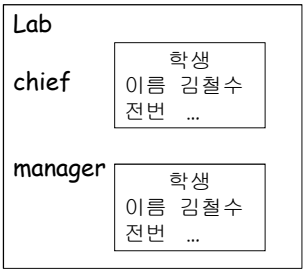
스택

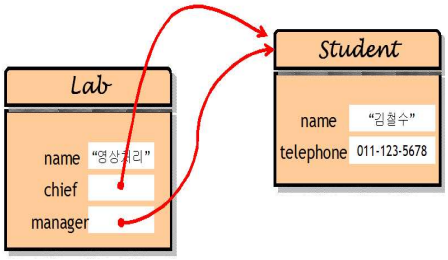
```
bool MyIntStack::push(int n) {
      if (top == size)
            return false;  // stack full
      p[top] = n;          top++;
      return true;
}
bool MyIntStack::pop(int& n) {
      if (top == 0)
            return false;  // stack empty
      top--;               n = p[top];
      return true;
}
MyIntStack::~MyIntStack() {
      if (p)  delete[] p; //스택 메모리 반환
}
```



예제 #2 객체 포인터(has-a 관계?? , 생략)

- 한 연구실에 여러 학생이 있고 그 중에 실장과 총무를 한 명씩 맡는 상황이 일반적
  - 많은 학생이 한 연구실에 포함하게 하려면 포인터 배열이나 linked-list 를 사용해야 함.
  - 코드 복잡
- 여기서는 연구실에 학생 한 명 있고 그 학생이 실험실의 실장과 총무를 겸하는 경우로 설정
  - Lab 클래스 안에 실장, 총무 객체 포함하게 작성 → 코드 복잡
  - But 객체 포인터를 사용하여서 중복을 줄인다. → Lab/학생 클래스를 분리, 연결만
  - 객체 포인터 사용하는 예제.





전체적인 두 클래스 요약

```
int main(){
   Lab lab("영상처리");
   Student *p= new Student("김철수", "1234");

   lab.setChief(p);        // 실장 임명
   lab.setManager(p);   // 총무 임명
   lab.print();              // 실장 총무 출력

  delete p;
  return 0;
}
```

연구실은 학생들을 포함

영상 처리연구실
실장은 김철수
총무는 김철수

참고) 포인터 멤버변수 포함하기에 소멸자 있는 것이 좋음, but ??

Lab, student 클래스

```
// 학생을 나타낸다.
class Student {
private:
   string name;  // 학생 이름
   string telephone;

public:
   Student(const string n="", const string t="");
   string getName() const;

// 아래 함수들 사용 안함
   string getTelephone() const;
   void setTelephone(const string t);
   void setName(const string n);
};
```

```
// 연구실을 나타낸다.
class Lab {
   string name;  // 연구실 이름
   Student *chief;       // 실장
   Student *manager;  // 총무

public:
   Lab(string n="");
   void setChief(Student *p);
   void setManager(Student *p);
   void print() const;
};
```

- 일반적으로 멤버에 포인터 변수가 있으면 소멸자 있어야 함
- 이 프로그램은 소멸자 있으면 얕은 복사 문제 발생 가능
  - chief 제거, manager 제거 → 충돌
  - 그래서 main에서 제거하는 코드 작성
- 불완전 프로그램
  - 실제는 학생이 여러 명 존재 → linked list 등으로 구현
  - 학생을 멤버로 갖는 객체를 만들어 관리하는 것이 좋음
  - 그 객체에서 각 학생들을 생성, 관리, 소멸(소멸자)
- Lab 멤버에서는 단순히 포인팅만 하고 소멸자 불필요
- 포인터 변수 존재한다고 항상 소멸자 필요는 아님(프로그램 특성에 따라 판단)

참고) 포인터 멤버변수 포함하기에 소멸자 있는 것이 좋음, but ??

```
int main(){
   Lab lab("영상처리");
   Student *p= new Student("김철수", "1234");

   lab.setChief(p);        // 실장 임명
   lab.setManager(p);   // 총무 임명
   lab.print();              // 실장 총무 출력

  delete p;
  return 0;
}
```

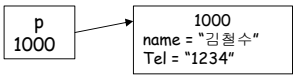
```
// 연구실을 나타낸다.
class Lab {
   string name;  // 연구실 이름
   Student *chief;       // 실장
   Student *manager;  // 총무

public:
   Lab(string n="");
   void setChief(Student *p);
   void setManager(Student *p);
   void print() const;
};
```

예제 #2 예제(Student class)

```
// 학생을 나타낸다.
class Student {
private:
   string name;
   string telephone;

public:
   Student(const string n="", const string t="");
   string getName() const;

            // 아래 함수들 사용 안함
   string getTelephone() const;
   void setTelephone(const string t);
   void setName(const string n);
};

Student::Student(const string n, const string t) {
   name = n;
   telephone = t;
}

string Student::getName() const {
      return name;
}
```

```
      // 아래 함수들 사용 안함
string Student::getTelephone() const
{
      return telephone;
}
void Student::setTelephone(const string t)
{
      telephone = t;
}

void Student::setName(const string n)
{
      name = n;
}
```

```
int main(){
  Student *p= new Student("김철수", "1234");
              ...
}
```

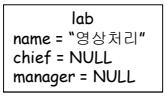


예제 #2 Lab 클래스

```
// 연구실을나타낸다.
class Lab {
   string name;
   Student *chief;
   Student *manager;
public:
   Lab(string n="");
   void setChief(Student *p);
   void setManager(Student *p);
   void print() const;
};

Lab::Lab(const string n){
      name = n;
      chief = NULL;
      manager = NULL;
}

void Lab::setChief(Student *p) {
      chief = p;
}
```

```
void Lab::setManager(Student *p){
   manager = p;
}

void Lab::print() const{
      ...
}
```

```
int main(){
   Lab lab("영상처리");
   Student *p= new Student("김철수", "1234");
              ...
}
```



영상 처리연구실
실장은 김철수
총무는 김철수

예제 #2 Lab 멤버함수

```
void Lab::setChief(Student *p) {
      chief = p;
}

void Lab::setManager(Student *p){
   manager = p;
}
```

```
void Lab::print() const{
   cout << name << "연구실" << endl;
   if(chief  != NULL)
      cout << "실장은" << chief->getName() << endl;
   else
      cout << "실장은 현재 없음" << endl;

   if( manager != NULL)
      cout << "총무는" << manager->getName() << endl;
   else
      cout << "총무는 현재 없습니다 << endl";
}
```

lab
name = "영상처리"
chief = 1000
manager = 1000

p
1000

1000
name = "김철수"
Tel = 1234

```
int main(){
  Lab lab("영상처리");
  Student *p= new Student("김철수", "1234");

  lab.setChief(p);
  lab.setManager(p);
  lab.print();

  delete p;
  return 0;
}
```

영상 처리연구실
실장은 김철수
총무는 김철수

예제#3 복소수→ has-a 관계 아님

```
void main(void) {
        Complex x(2, 3), y(4, -6), z;  // 책과 다른 값

        cout << "첫번째 복소수 x: ";
        x.print();

        cout << "두번째 복소수 y: ";
        y.print();

        z = x.add(y);  // z = x + y, 10장 에서 자세히

        cout << " z = x + y = ";
        z.print();

}
```

첫번째 복소수 x: 2 + 3i
두번째 복소수 y: 4 - 6i
z = x + y = 6 - 3i

Complex class

- 멤버변수

```
double real;        // 실수부
double imag;        // 허수부
```

- 멤버함수

```
Complex();                          // 생성자(0, 0) 초기화
Complex(double a, double b); // 생성자

double getReal();   // 실수부 반환. → 불필요
double getImag();  // 허수부 반환. → 불필요

// 복소수의 덧셈연산을 구현한다.
Complex add(const Complex& c);  // 합 결과 반환
void print();            // 복소수를 출력한다.
```

예제#3 복소수

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

class Complex
{
private:
  double real;          // 실수부
  double imag;          // 허수부

public:
  Complex();                        // 생성자
  Complex(double a, double b);      // 생성자

  //double getReal();   // 실수부 반환
  //double getImag();  //허수부 반환

  // 복소수의 덧셈 연산을 구현한다.
  Complex add(const Complex& c);

  void print();         // 복소수를 출력한다.
};
```

```
Complex::Complex(){ real = 0;  imag = 0; }

Complex::Complex(double a, double b){
            real = a;     imag = b;
}

// 복소수의 덧셈 연산 구현
// z = x.add(y);  // z = x + y
Complex Complex::add(const Complex& c){
  Complex temp;
  temp.real = this->real + c.real;
  temp.imag = this->imag + c.imag;

  return temp;          // 객체 반환.
}

void Complex::print() {
  char s;               // 책에 없는 내용
  if (imag > 0)         s = '+'; // 6 은 + 부호 필요
  else                  s = ' ';  // -6 경우 부호 불필요
                                  // -6에 "-" 표시 있음
  cout << real << s << imag << "i" << endl;
}
```

임시객체 생성, 소멸

- 임시 객체와 복사 생성자 호출 생략

```
Car c1(0, 1, "blue");          // c1 객체 생성 ← 임시객체 생성 없음
Car c4 = Car(0, 1, "blue");    // 먼저 이름이 없는 임시 객체를 만들어
                               // 멤버변수 초기화후 c4 라는 이름으로 사용(실제 복사 생성자 호출 안함)
```

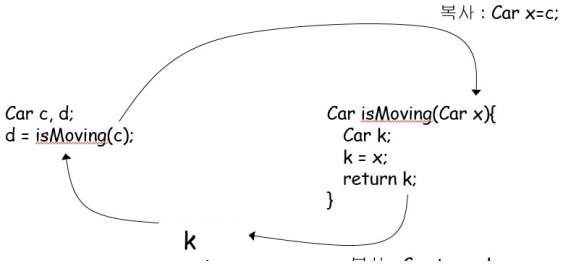
➔ 바운딩되지 않은(이름이 없는) 객체 생성시 임시 객체 생성

- 위 경우, 이론 상으로 복사 생성자가 호출되는 상황인데 실제는 호출이 생략된다.
  - 이론상으로 임시객체를 만들고 c4 로 복사해야 하지만
- 즉, 다음 경우에 복사생성자 호출 생략됨
  - 바운딩되지 않은(다른 변수와 연결되지 않은) 이름이 없는 임시 객체가 같은 타입의 객체 생성하며 복사될 때 복사생성자를 호출하지 않음
- 이것은 해석을 다음과 같이 하면 됨
  - 임시객체를 소멸시키지 않고 이름을 c4로 하여 계속 사용한다(임시객체 소멸 안함)
  - 새로운 객체 생성을 하지 않고, 있는 객체(임시객체)를 재사용하여 효율성을 높임.

81

함수 반환시 임시객체

1. **함수에서 객체 반환시** (컴파일러 version up 되며 변경)
   1) 이전에는 임시객체 생성 복사하여 전달/소멸 → 여러 가지 복잡 문제 발생 가능
   2) 현재는 함수 객체 반환시 임시 객체 사용 안 한다고 생각하면 됨 → 코딩 간단해짐



- k 전달되고 → d = k; 수행 후 k 소멸 (→ 임시객체 없다고 생각하면 됨 → 실제는 임시객체 사용하지만 자세한 내용 복잡함, 불필요)
- 만약 Car e = isMoving(c); 인 경우 → Car e = 임시객체; (객체 선언과 동시에 객체 복사) → 앞 페이지와 같은 형태 → 임시 객체 소멸 안하고 e 로 사용

정적 변수와 객체 반환 함수

```
class Num {
   int nn;
   static int c;// 객체 생성시마다 1 증가
public:
   Num(int r = 0) : nn(r) { c++; }
   ~Num() { c--; }

   Num Add(const Num& x) const {
      Num T;
      T.nn = nn + x.nn;
      return T;
   }
   static int GetCount() { return c; }
   void Print() const {
      cout << nn << endl;
   }
};
```

```
int Num::c = 0;

int main() {
   Num com1(1), com2(3);
   Num com3;
   cout << "# num = " << Num::GetCount() << endl;

   com3 = com1.Add(com2);
   cout << "# num = " << Num::GetCount() << endl;

   Num com4 = com1.Add(com3);
   cout << "# num = " << Num::GetCount() << endl;

   return 0;
}
```

VS 2019 에서는 출력 3 2 2
→ 작성 시 유의 사항(??) 지키면 해결 가능

컴파일러 변경으로
VS 2022 에서는 출력 3 3 4 → 간단, 상식적

함수 반환시 임시객체

```
생성 1
..11.....
복사 1
생성 0
소멸 1
소멸 2
..33.....
소멸 2
소멸 1
```

함수에서 반환값 객체 k가 전달된다고 생각하면 됨

```
class Num {
public:
   int nn;
   Num(int r = 0) : nn(r) {    cout << "생성 " << nn << this << endl;   }
   Num(const Num& a) {
      nn = a.nn;       cout << "복사 " << nn << this << endl;   }
   ~Num() {       cout << "소멸 " << nn << this << endl;   }
};
Num add(Num a) {      // 복사생성자
   Num k;      k.nn = a.nn + 1;
   return k;   // k 가 전달된다고 생각.
}
int main() {
   Num x(1);    cout << ".11.." << endl;

   z = add(x);    cout << "..33.." << endl;
   return 0;
}
```

```
생성 1 0000004DC552F854 -> x 생성
..11.....
복사 1 0000004DC552F994  -> Num a=x 복사
생성 0 0000004DC552F9D4 -> k 생성
소멸 1 0000004DC552F994 -> a 소멸, k는 전달
소멸 2 0000004DC552F9D4 -> z = k; 대입하고 k 소멸
..33.....
소멸 2 0000004DC552F874 → x, z 소멸
소멸 1 0000004DC552F854
```


과제

- DOOR 과제에 제출(수업활동 일지 아님)
- 텍스트 파일 제출(OOP_7_학번.txt) → 한글 파일 제출 아님
- 293쪽 연습문제 6번 → 소멸자 불필요

```
int main()
{
        Date bir(1987,4,27);
        Date hir(2011,2,05);
        Employee emp("홍길동", bir, hir);
        emp.print();
        return 0;
}
```

```
직원의 이름 : 홍길동
직원의 생일 : 1987년4월27일
직원의 입사일 : 2011년2월5일
Press any key to continue
```