9장 복사생성자와 정적 멤버

2020. 10. 15

순천향대학교 컴퓨터 공학과

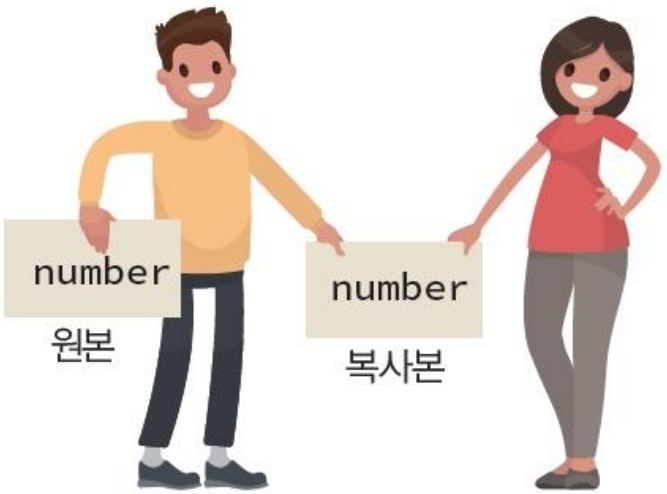
내용

- 함수 매개변수/반환 값으로서 객체
- 복사 생성자
- 정적 멤버

함수로 객체 전달하기

- 값에 의한 호출(call-by-value)
- 참조에 의한 호출(call-by-reference)

참조에 의한 호출



값에 의한 호출

함수에 객체 전달

- 함수를 호출하는 쪽에서 객체 전달
- 함수의 매개 변수 객체 생성
  - 매개 변수 객체의 공간이 스택에 할당
  - 호출하는 쪽의 객체가 매개 변수 객체에 그대로 복사
  - 객체 복사는 복사 생성자(copy constructor)가 호출되어서 수행
- 함수 종료시에 매개변수 객체 소멸
- 매개 변수 객체의 생성자가 호출될 경우에는?

```
class Circle {
private:
   int radius;
public:
   Circle();
   Circle(int r);
   ~Circle();
   double getArea()  { return 3.14*radius*radius; }
   int getRadius() { return radius; }
   void setRadius(int radius) { this->radius = radius
};
Circle::Circle() {
   radius = 1;
   cout << "생성자 실행 radius = " << radius << endl;
}
Circle::Circle(int radius) {
   this->radius = radius;
   cout << "생성자 실행 radius = "  << radius << endl;
}

Circle::~Circle() {
   cout << "소멸자 실행 radius = " << radius << endl;
}
```

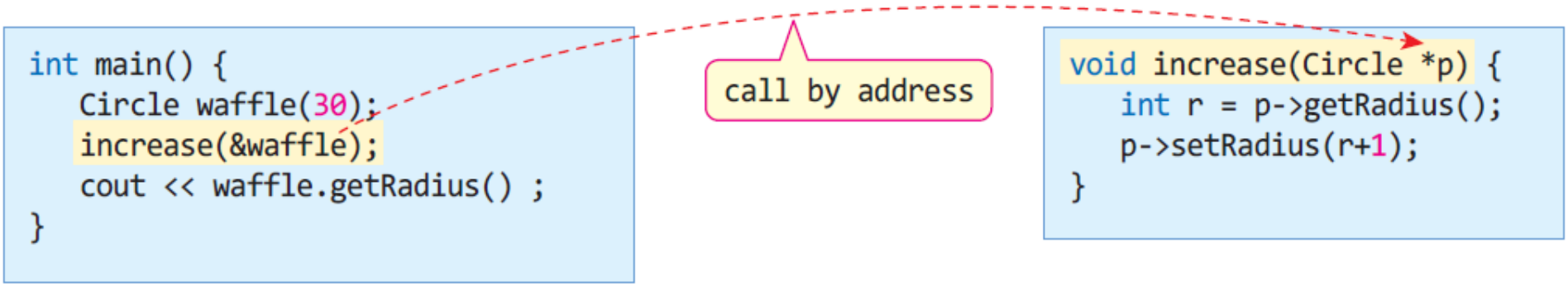
```
void increase(Circle c) {
   int r = c.getRadius();

   c.setRadius(r+1);
}

int main() { // 수행 결과는?
   Circle waffle(30);

   increase(waffle);
   cout << waffle.getRadius() <<
endl;
}
```

예제

```
class Pizza  {
            int radius;
public:
            Pizza(int r= 0) : radius{ r } {      }
            ~Pizza() {          }
            void setRadius(int r) { radius = r; }
            void print() { cout << "Pizza(" << radius << ")" << endl; }
};

void upgrade(Pizza p) {p.setRadius(20);   }

int main()  { // 수행 결과는?
            Pizza obj(10);

            upgrade(obj);
            obj.print();
            return 0;
}
```

함수에 객체 주소 전달

- 함수 호출시 객체의 주소만 전달
  - 함수의 매개 변수는 객체에 대한 포인터 변수로 선언
  - 함수 호출에 따른 생성자, 소멸자가 실행되지 않음



예제

- 다음 main()의 실행 결과는?

```
void upgrade(Pizza *p) {
        p->setRadius(20);
}

int main()
{
        Pizza obj(10);
        upgrade(&obj);

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

함수에 객체에 대한 참조 전달

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

class Circle {
   int radius;
public:
   Circle() { radius = 1; }
   Circle(int radius) { this->radius = radius; }
   void setRadius(int radius) { this->radius = radius; }
   double getArea() { return 3.14*radius*radius; }
};

int main() { // 수행 결과는?
   Circle circle;
   Circle &refc = circle;

   refc.setRadius(10);
   cout << refc.getArea() << " " << circle.getArea();
}
```

예제

```
void increaseCircle(Circle &c) {
   int r = c.getRadius();
   c.setRadius(r+1);
}

int main() {
   Circle waffle(30);

   increaseCircle(waffle);
   cout << waffle.getRadius() <<
endl;
}
```

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

Circle::Circle() {
   radius = 1;
   cout << "생성자 실행 radius = " << radius <<
endl;
}

Circle::Circle(int radius) {
   this->radius = radius;
   cout << "생성자 실행 radius = "  << radius <<
endl;
}

Circle::~Circle() {
   cout << "소멸자 실행 radius = " << radius <<
endl;
}
```

예제

```
void upgrade(Pizza& pizza) {
        pizza.setRadius(20);
}

int main() // 수행결과는?
{
        Pizza obj(10);

        upgrade(obj);
        obj.print();
        return 0;
}
```

함수로부터 객체 반환

```
Pizza createPizza() {
        Pizza p(10);
        return p;
}

int main() // 수행 결과는?
{
        Pizza obj;

        obj = createPizza();
        obj.print();
        return 0;
}
```

Lab1

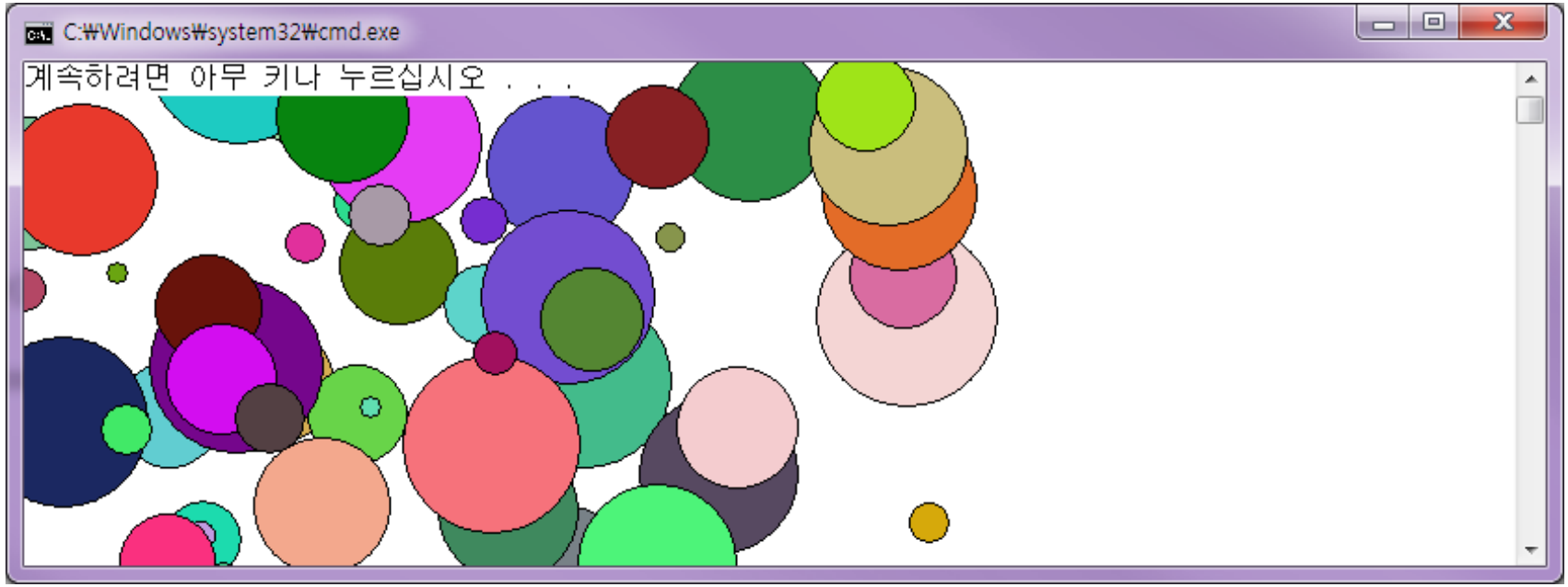
```
class Complex {
public:
        double real, imag;
        Complex(double r = 0.0, double i = 0.0) : real{ r }, imag{ i } {
                cout << "생성자 호출";
                print();
        }
        ~Complex() { cout << "소멸자 호출"; print();      }
        void print() {
                cout << real << "+" << imag << "i" << endl;
        }
};
```

Lab1 (계속)

```
Complex add(Complex c1, Complex c2) {
        Complex temp;

        temp.real = c1.real + c2.real;
        temp.imag = c1.imag + c2.imag;

        return temp;
}

int main()
{
        Complex c1{ 1,2 }, c2{ 3,4 };
        Complex t;

        t = add(c1, c2);
        t.print();

        return 0;
}
```

Lab2

- 랜덤한 색상을 생성하고 이것을 원을 나타내는 Circle 객체로 전달하여 컬러풀한 원들이 그려지도록 하자.



```
class Color {
public:
        int red, green, blue;
        Color() {
                red = rand() % 256;
                green = rand() % 256;
                blue = rand() % 256;
        }
};
class Circle {
        int x, y;
        int radius;
        Color color;
public:
        Circle(int x, int y, int r, Color c) : x(x), y(y),
radius(r), color(c) { }
        void draw();
};
```

```
void Circle::draw() { //  원을 화면에 그린다
  int r = radius / 2;
  HDC hdc = GetWindowDC(GetForegroundWindow());
  SelectObject(hdc, GetStockObject(DC_BRUSH));
  SetDCBrushColor(hdc, RGB(color.red, color.green, color.blue));
  Ellipse(hdc, x - r, y - r, x + r, y + r);
}
```

```
class Color{
public:
        int red, green, blue;
        Color() {
                red = rand() % 256;
                green = rand() % 256;
                blue = rand() % 256;
        }
};
class Circle {
        int x, y, radius;
        Color color;
  public:
        Circle(int x, int y, int r, Color c) : x(x), y(y), radius(r), color(c) { }
        void draw();
};
```

```
int main()
{
    for (int i = 0; i < 100; i++) {
        Circle obj(rand() % 500, rand() % 500, rand() % 100, Color());
        obj.draw();
    }
    return 0;
}
```

복사 생성자

- 복사 생성자(copy constructor)는 동일한 클래스의 객체를 복사하여 객체를 생성할 때, 사용하는 생성자이다.

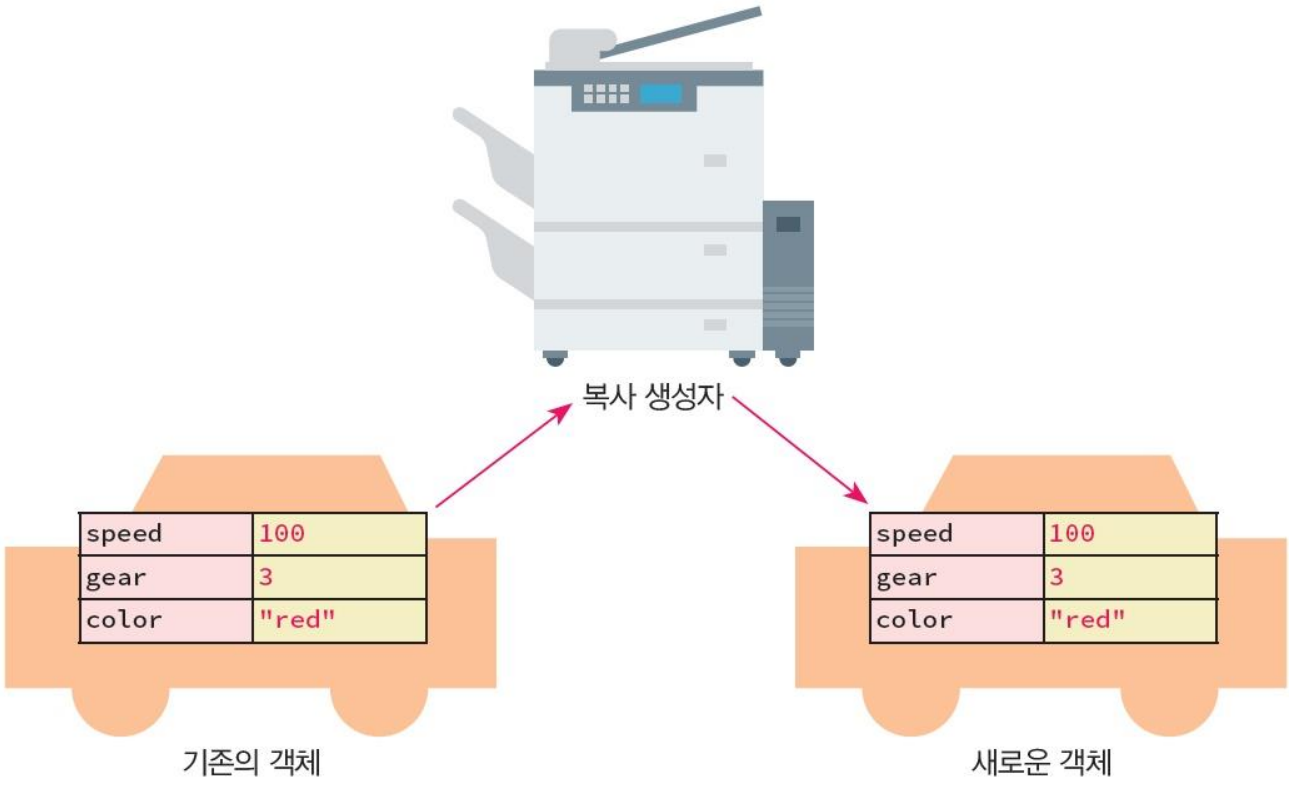


그림 9.1 복사 생성자는 다른 객체의 내용을 복사하여서 새로운 객체를 생성한다.

복사 생성자 (2)

- 복사 생성자(copy constructor)란?
  - 객체의 복사 생성시 호출되는 특별한 생성자
- 특징
  - 한 클래스에 오직 한 개만 선언 가능
  - 복사 생성자는 생성자와 클래스 내에 중복 정의
  - 자신 클래스에 대한 참조 매개 변수를 가짐
- 복사 생성자 선언

```
class Circle {
   .....
   Circle(Circle& c); // 복사 생성자 선언
   .....
};
```

자기 클래스에 대한 참조 매개 변수

예제

```
class Circle {
private:
  int radius;
public:
  Circle(Circle& c); // 복사 생성자 선언
  Circle() { radius = 1; }
  Circle(int radius) { this->radius = radius; }
  double getArea() { return 3.14*radius*radius; }
};
Circle::Circle(Circle& c) { // 복사 생성자 구현
  this->radius = c.radius;
  cout << "복사 생성자 실행 radius = " << radius <<
endl;
}
int main() {
  Circle src(30);
  Circle dest(src); // dest 객체의 복사 생성자 호출

  cout << "원본의 면적 = " << src.getArea() << endl;
  cout << "사본의 면적 = " << dest.getArea() << endl;
}
```

디폴트 복사 생성자

- 복사 생성자가 선언되어 있지 않으면, 컴파일러는 자동으로 디폴트 복사 생성자 삽입

```
class Circle {
   int radius;
pubic:
   Circle(int r);
   double getArea();
};
```

디폴트 복사 생성자 삽입

```
Circle::Circle(Circle& c) {
   this->radius = c.radius;
}
```

```
Circle dest(src); // dest 객체가 생성되고,
                  // 이 객체의 디폴트 복사 생성자 호출
```

예제

- 복사 생성자가 없는 **Book** 클래스

```
class Book {
  double price; // 가격
  int pages;      // 페이지수
  char *title;       // 제목
  char *author; // 저자이름
public:
  Book(double pr, int pa, char* t, char* a;);
  ~Book()
};
```

디폴트 복사 생성자 삽입

```
Book(Book& book) {
  this->price = book.price;
  this->pages = book.pages;
  this->title = book.title;
  this->author = book.author;
}
```

예제

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

class Person {
public:
        int age;
        Person(int a) : age{a} {    }
};

int main() {
  Person kim(21);
  Person clone{ kim };

  cout << "kim의 나이: " << kim.age << " clone의 나이: " << clone.age << endl;
  kim.age = 23;
  cout << "kim의 나이: " << kim.age << " clone의 나이: " << clone.age << endl;

  return 0;
}
```

복사 생성자가 자동 호출되는 경우


```
void f(Person person) {
  person.changeName("dummy");
}

Person g() {
  Person mother(2, "Jane");
  return mother;
}

int main() {
  Person father(1, "Kitae");
  Person son = father;
  f(father);
  g();
}
```

'값에 의한 호출'로 객체가 전달될 때.
person 객체의 복사 생성자 호출: 실 매개변수 객체 복사

함수에서 객체를 리턴할 때.
mother 객체의 복사본 생성.
복사본의 복사 생성자 호출: mother 객체 복사

객체로 초기화하여 객체가 생성될 때.
son 객체의 복사 생성자 호출: father 객체 복사

```
class Person {
public:
 int age;
 Person(int a) : age{a} {
  cout << "constructor is called" << endl;
 }
 Person() { age = 1;}
 Person(Person& p) {
  this->age = p.age;
  cout << "copy constructor is called" << endl;
 }
}; // class Person

void f(Person p) {
 cout << "f() is called" << endl;
}
Person g() {
 Person p(50);
 return p;
}
```

```
int main() {
 Person obj1(20);
 Person obj2(30);

 Person obj3 = obj1;
 f(obj3);
 cout << g().age << endl;
 cout << "--------------" << endl;
 cout << obj2.age << endl;
 obj2 = obj1;
 cout << obj2.age << endl;
 return 0;
}
```

생각?

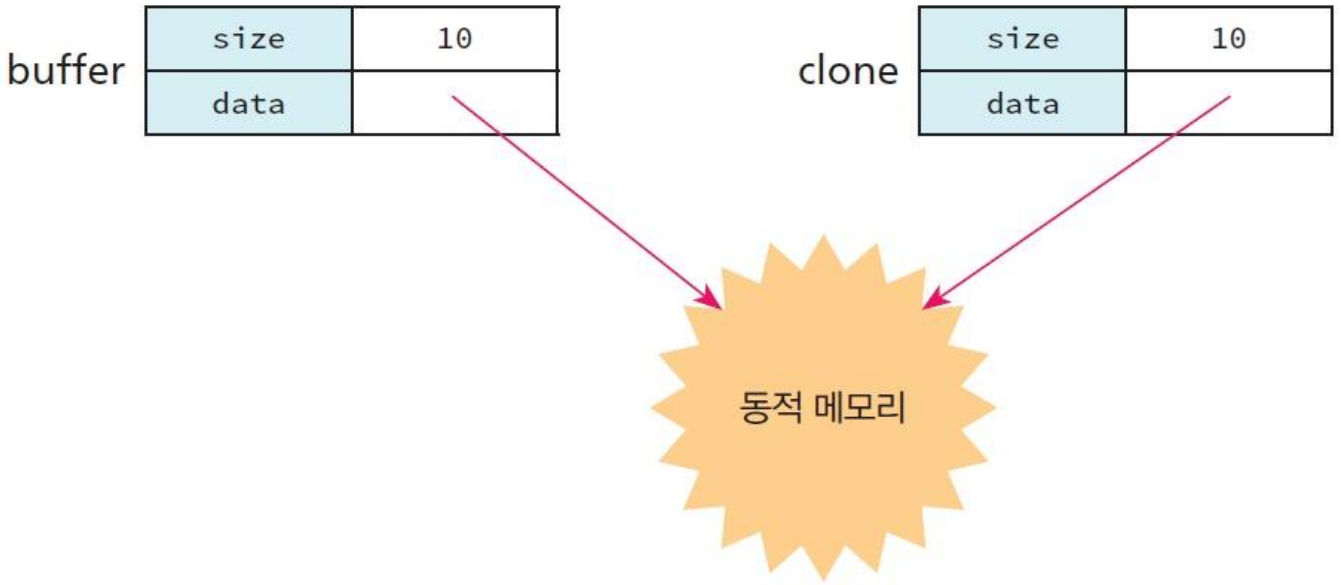
- 다음 main() 실행 결과는?

```
class MyArray {
public:
        int size;
        int* data;

        MyArray(int size)
        {
          this->size = size;
          data = new int[size];
        }
        ~MyArray()
        {
          if (data != NULL) delete[] this->data;
        }
};
```

```
int main()
{
        MyArray buffer(10);
        buffer.data[0] = 1;
        {
          MyArray clone = buffer;
        }
        buffer.data[0] = 2;

        return 0;
}
```

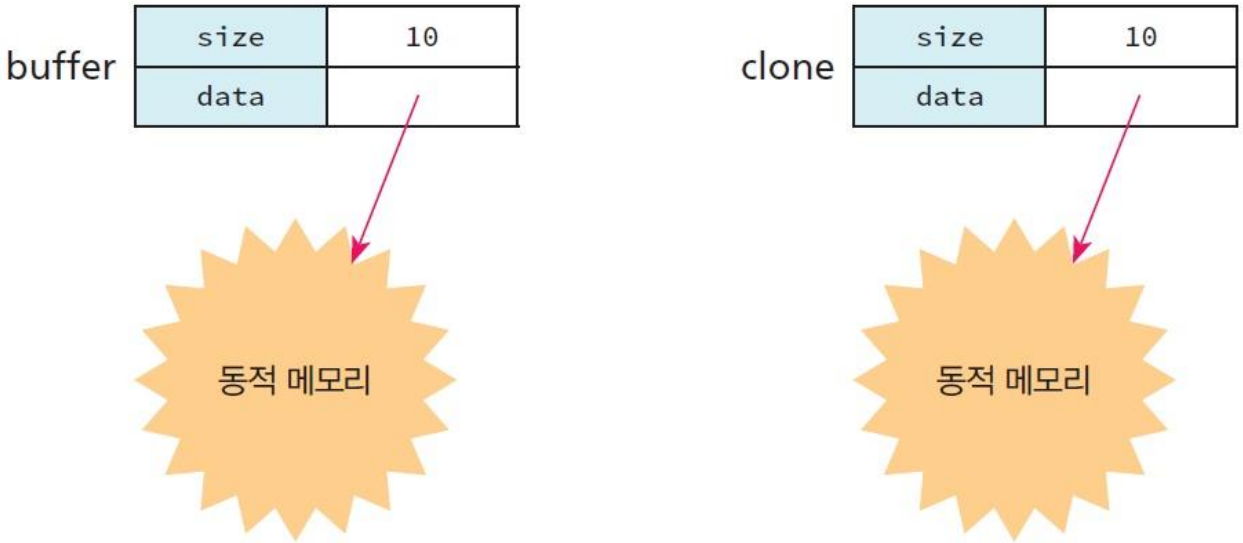
why?

- 디폴트 복사 생성자는 객체 복사시 멤버 대 멤버로 복사.
  - 원본에 포인터 멤버가 존재할 경우에는 그 멤버의 주소만 복사본에 복사됨
  - 이러한 복사를 얕은 복사(shallow copy)라 함

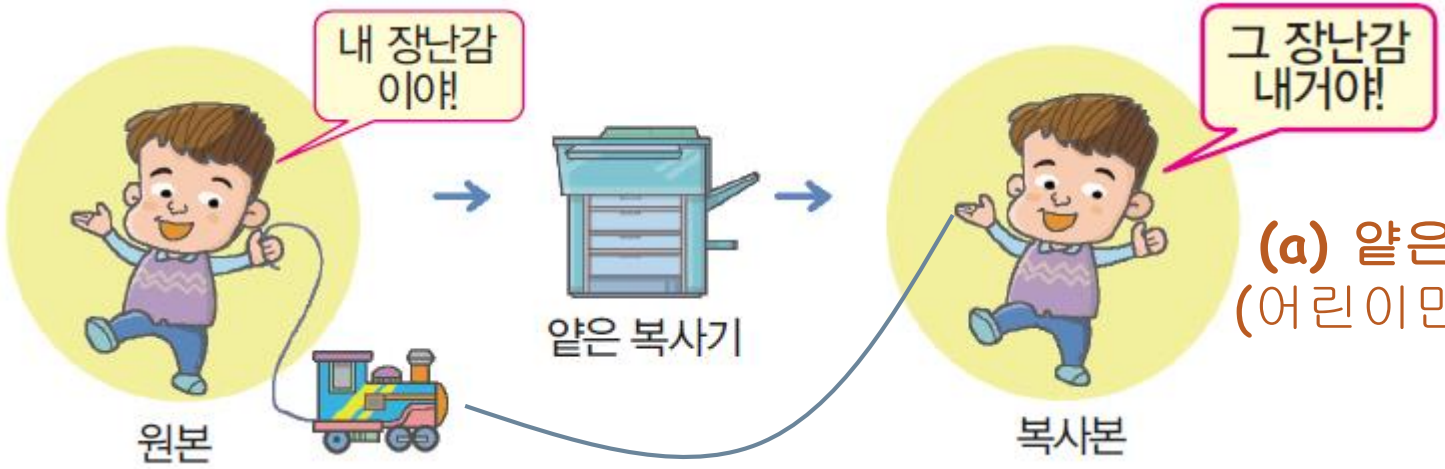


깊은 복사 (deep copy)

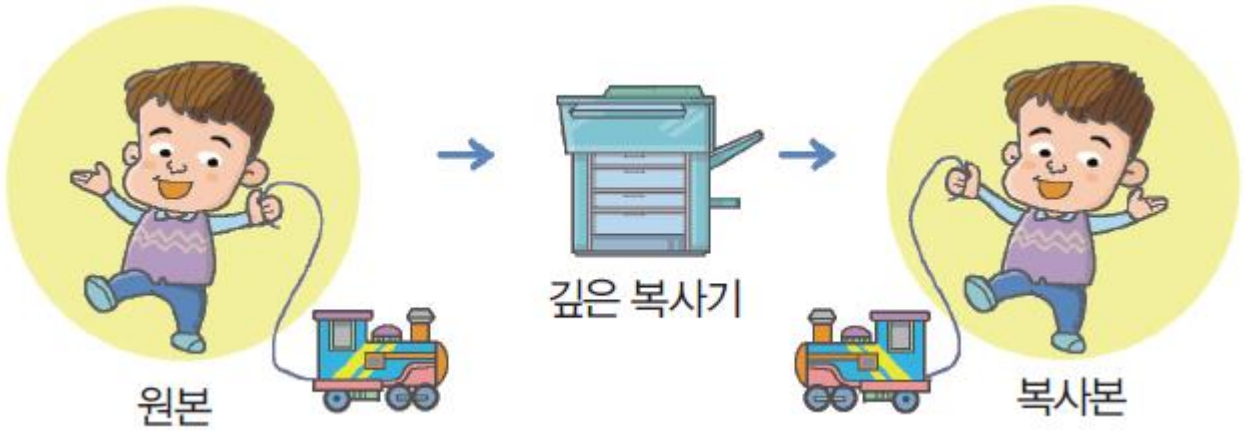
- 원본 객체에 동적 할당된 메모리를 가리키는 포인터 멤버 존재시에, 복사본 객체에 원본의 포인터뿐만 아니라 포인터가 가리키는 메모리까지 복사하는 것을 깊은 복사(deep copy)라고 함
- 깊은 복사를 위해서 복사 생성자를 별도로 정의하는 것이 필요함



얕은 복사와 깊은 복사



(a) 얕은 복사
(어린이만 복사)



(b) 깊은 복사
(어린이가 소유한 장난감도 복사)

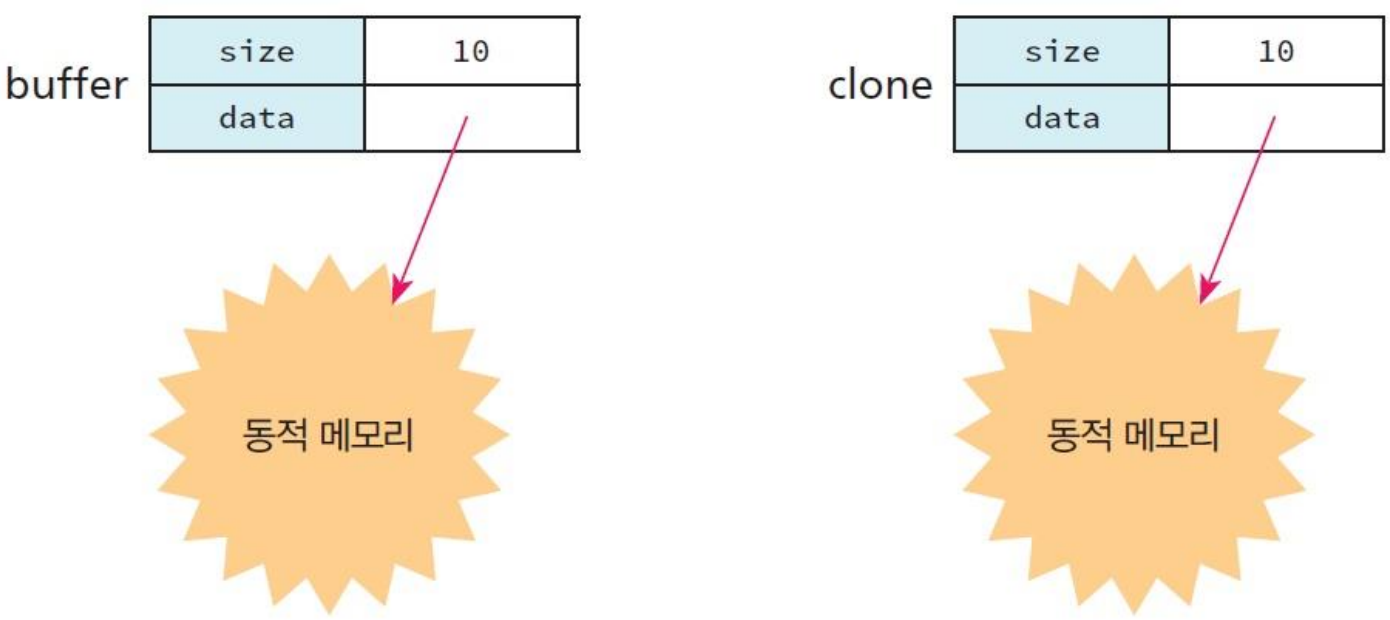
예제

```
class MyArray {
public:
        int size;
        int* data;
        MyArray(int size);
        MyArray(const MyArray& other); // 복사 생성자 선언
        ~MyArray();
};

MyArray::MyArray(int size)
{
        this->size = size;
        data = new int[size];
}
```

```
MyArray::MyArray(const MyArray& other)
{ // 깊은 복사 수행하는 복사 생성자 정의
        this->size = other.size;
        this->data = new int[other.size];
        for (int i = 0; i < size; i++)
                this->data[i] = other.data[i];
}

MyArray::~MyArray() {
        if (data != nullptr) delete[] this->data;
        data = nullptr;
}
```

예제 (계속)

```
int main()
{
        MyArray buffer(10);
        buffer.data[0] = 1;
        {
            MyArray clone = buffer; // 복사 생성자 호출
        }
        buffer.data[0] = 2;

        return 0;
}
```



객체 할당

- 이미 생성된 객체를 다른 객체에 할당할 때는 복사 생성자가 호출되지 않는다.

```
class Person {
public:
        int age;
        Person(int a) : age(a) {    }
};

int main()
{
        Person obj1(20);
        Person obj2(20);

        obj2 = obj1;        // obj1의 모든 데이터 멤버가 obj2에 할당
        return 0;
}
```

객체 비교

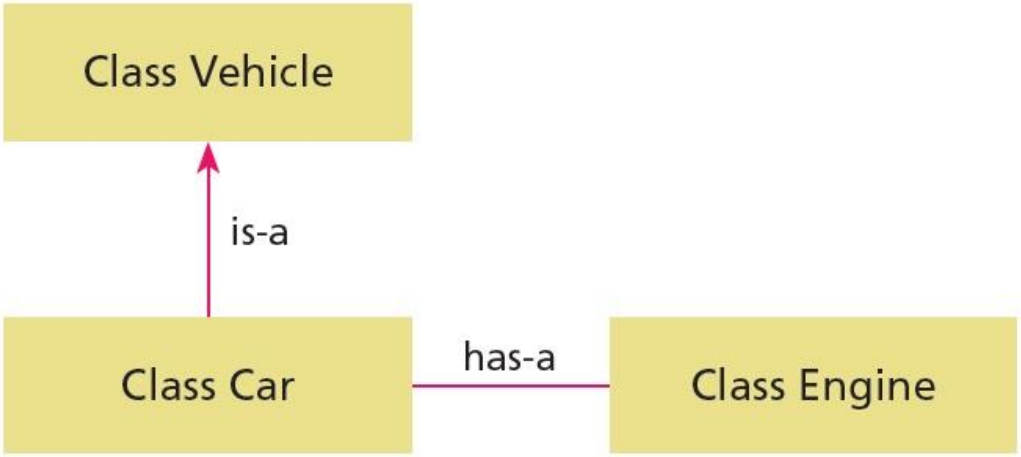
```
class Person {
public:
        int age;
        Person(int a) : age(a) {    }
};

int main() {
        Person obj1(20);
        Person obj2(20);

        if (obj1 == obj2) { // 오류: Person 클래스에 ==가 정의되어 있지 않음
                cout << "같습니다" << endl;
        }
        else {
                cout << "같지 않습니다" << endl;
        }
        return 0;
}
```

객체간의 관계

- is-a 관계: 객체 지향 프로그래밍에서 is-a의 개념은 상속을 기반으로 한다. 우리는 아직 상속은 학습하지 않았다. "A는 B 유형의 물건"이라고 말하는 것과 같다. 예를 들어, Apple은 과일의 일종이고, Car는 자동차의 일종이다.

- has-a 관계: has-a는 하나의 객체가 다른 객체를 포함하는 관계이다. 예를 들어서 Car에는 Engine이 있고 House에는 Bathroom이 있다.

객체간의 관계



예제

```
class Date {
        int year, month, day;
public:
        Date(int y, int m, int d) : year{ y }, month{ m }, day{ d } {       }
        void print() {
                cout << year << "." << month << "." << day <<endl;
        }
};
```

```
class Person {
        string name;
        Date birth;
public:
        Person(string n, Date d) : name{ n }, birth{ d } {       }
        void print() {
                cout << name << ":";
                birth.print();
                cout << endl;
        }
};
```

예제 (계속)

```
int main()
{
        Date d{ 1998, 3, 1 };
        Person p{ "김철수", d };
        p.print();
        return 0;
}
```

클래스의 static vs non-static 멤버

- static 멤버
  - 프로그램이 시작할 때 생성
  - 클래스 당 하나만 생성, 클래스 멤버라고 불림
  - 클래스의 모든 인스턴스(객체)들이 공유하는 멤버

- non-static 멤버
  - 객체가 생성될 때 함께 생성
  - 객체마다 객체 내에 생성
  - 인스턴스 멤버라고 불림

static 멤버 선언

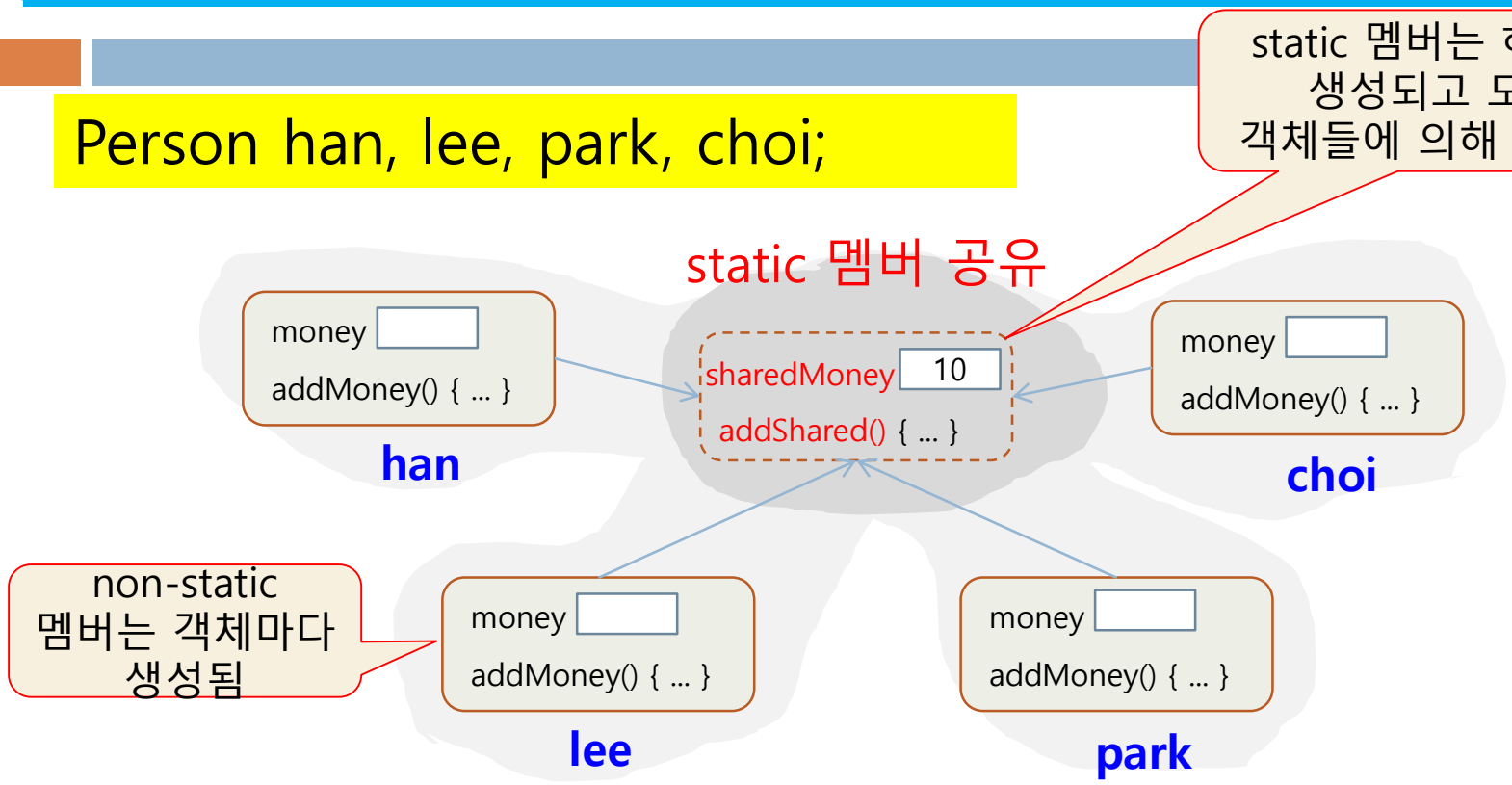
- 멤버 앞에 static 키워드를 붙여서 static 멤버로 선언
- static 멤버 변수(정적 변수)는 외부에서 전역 변수로 선언되어야 하고, 전체 프로그램 내에 한 번만 생성 가능

예제

```
class Person {
public:
  double money; // 개인 소유의 돈
  void addMoney(int money) {
    this->money += money;
  }
  static int sharedMoney; // 공금
  static void addShared(int n) {
    sharedMoney += n;
  }
};

int Person::sharedMoney = 10; // 정적 변수 선언
```

static 멤버와 non-static 멤버의 관계

Person han, lee, park, choi;



- han, lee, park, choi 등 4 개의 Person 객체 생성
- sharedMoney와 addShared() 함수는 하나만 생성되고 4 개의 객체들의 의해 공유됨

static 멤버와 non-static 멤버 비교

항목	non-static 멤버	static 멤버

선언 사례	class Sample {
	  int n;
	  void f();
	};
		class Sample {
		  static int n;
		  static void f();
		};

공간 특성	멤버는 객체마다 별도 생성 • 인스턴스 멤버라고 부름	멤버는 클래스 당 하나 생성 • 멤버는 객체 내부가 아닌 별도의 공간에 생성 • 클래스 멤버라고 부름
시간적 특성	객체와 생명을 같이 함 • 객체 생성 시에 멤버 생성 • 객체 소멸 시 함께 소멸 • 객체 생성 후 객체 사용 가능	프로그램과 생명을 같이 함 • 프로그램 시작 시 멤버 생성 • 객체가 생기기 전에 이미 존재 • 객체가 사라져도 여전히 존재 • 프로그램이 종료될 때 함께 소멸
공유의 특성	공유되지 않음 • 멤버는 객체 별로 따로 공간 유지	동일한 클래스의 모든 객체들에 의해 공유됨

static 멤버 사용 : 객체 멤버로 접근

- static 멤버는 객체 이름이나 객체 포인터로 접근

```
객체.static멤버
객체포인터->static멤버
```

- 예제

```
Person lee;
lee.sharedMoney = 500; // 객체.static멤버 방식

Person *p;
p = &lee;
p->addShared(200); // 객체포인터->static멤버 방식
```

예제

```
class Person {
public:
   double money; // 개인 소유의 돈
   void addMoney(int money) {
      this->money += money;
   }
   static int sharedMoney; // 공금
   static void addShared(int n) {
      sharedMoney += n;
   }
};

int Person::sharedMoney=10;  // 10으로 초기화

int main() {
   Person han;
   han.money = 100; // han의 개인 돈
   han.sharedMoney = 200; // static 멤버 접근

   Person lee;
   lee.money = 150; // lee의 개인 돈
   lee.addMoney(200); // lee의 개인 돈
   lee.addShared(200); // static 멤버 접근

   cout << han.money << ' ' << lee.money << endl;
   cout << han.sharedMoney << ' ' << lee.sharedMoney << endl;
}
```

static 멤버 사용 : 클래스명과 범위 지정 연산자(::)로 접근

- 클래스 이름과 범위 지정 연산자(::)로 접근 가능
  - static 멤버는 클래스마다 오직 한 개만 생성되기 때문

클래스명::static멤버

```
han.sharedMoney = 200; <->   Person::sharedMoney = 200;
lee.addShared(200);      <->   Person::addShared(200);
```

- non-static 멤버는 클래스 이름으로 접근 불가

```
Person::money = 100; // 컴파일 오류. money는 non-static 멤버
Person::addMoney(200); // 컴파일 오류. addMoney()는 non-static 멤버
```

예제

```
class Person {
public:
   double money;                    // 개인 소유의 돈
   void addMoney(int money) {
      this->money += money;
   }
   static int sharedMoney;          // 공금
   static void addShared(int n) {
      sharedMoney += n;
   }
};

int Person::sharedMoney=10;

int main() {
   Person::addShared(50); // static 멤버 접근
   cout << Person::sharedMoney << endl;

   Person han;
   han.money = 100;
   han.sharedMoney = 200;           // static 멤버 접근
   Person::sharedMoney = 300;       // static 멤버 접근
   Person::addShared(100);          // static 멤버 접근

   cout << han.money << ' ' << Person::sharedMoney << endl;
}
```

```
class Circle {
        int x, y;
        int radius;

public:
        static int count;   // 정적 변수
        Circle() : x{0}, y{0}, radius{0} {
                count++;
        }
        Circle(int x, int y, int r) : x{x}, y{y}, radius{r} {
                count++;
        }
};

int Circle::count = 0;

int main()
{
        Circle c1;
        cout << "지금까지 생성된 원의 개수 = " << Circle::count << endl;

        Circle c2(100, 100, 30);
        cout << "지금까지 생성된 원의 개수 = " << Circle::count << endl;
}
```

```
class Circle {
        int x, y;
        int radius;
public:
        static int count;               // 정적 변수
        Circle() : x{0}, y{0}, radius{0} {
                count++;
        }
        Circle(int x, int y, int r) : x{x}, y{y}, radius{r} {
                count++;
        }
        static int getCount() {       // 정적 멤버 함수
                return count;
        }
};
int Circle::count = 0;

int main() {
        Circle c1;
        cout << "지금까지 생성된 원의 개수 = " << Circle::getCount() << endl;

        Circle c2(100, 100, 30);
        cout << "지금까지 생성된 원의 개수 = " << Circle::getCount() << endl;
}
```

static 용도

- 전역 변수나 전역 함수를 클래스에 캡슐화
  - 전역 변수나 전역 함수를 static으로 선언하여 클래스 멤버로 선언
  - 전역 변수나 전역 함수를 가능한 사용하지 않도록
- 객체 사이에 공유 변수를 만들고자 할 때
  - static 멤버를 선언하여 모든 객체들이 공유

static 멤버를 가진 Math 클래스

전역 함수들을 가진 좋지 않은 예

```
int abs(int a) { return a>0?a:-a; }
int max(int a, int b) { return  a>b)?a:b; }
int min(int a, int b) { return (a>b)?b:a; }

int main() {
   cout << abs(-5) << endl;
   cout << max(10, 8) << endl;
   cout << min(-3, -8) << endl;
}
```

Math 클래스를 만들고 전역 함수들을 static 멤버로 캡슐화

```
class Math {
public:
   static int abs(int a) { return a>0?a:-a; }
   static int max(int a, int b) { return (a>b)?a:b; }
   static int min(int a, int b) { return (a>b)?b:a; }
};

int main() {
   cout << Math::abs(-5) << endl;
   cout << Math::max(10, 8) << endl;
   cout << Math::min(-3, -8) << endl;
}
```

static 멤버를 공유의 목적으로 사용하는 예

```
class Circle {
private:
   static int numOfCircles;
   int radius;
public:
   Circle(int r=1);
   ~Circle() { numOfCircles--; }
   double getArea()  { return 3.14*radius*radius;}
   static int getNumOfCircles() { return numOfCircles; }
};
Circle::Circle(int r) {
   radius = r;
   numOfCircles++;
}
int Circle::numOfCircles = 0;

int main() {
   Circle *p = new Circle[10];
   cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;
   delete [] p;
   cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;
   Circle a;
   cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;

   Circle b;
   cout << "생존하고 있는 원의 개수 = " << Circle::getNumOfCircles() << endl;
}
```

static 멤버 함수는 static 멤버만 접근 가능

- static 멤버 함수가 접근할 수 있는 것
  - static 멤버 함수
  - static 멤버 변수
  - 함수 내의 지역 변수

- static 멤버 함수는 non-static 멤버에 접근 불가
  - 객체가 생성되지 않은 시점에서 static 멤버 함수가 호출될 수 있기 때문

static 멤버 함수가 non-static 멤버 변수 접근 시

- Static 멤버 함수는 객체 생성 전에도 접근 가능

```
class PersonError {
  int money;
public:
  static int getMoney() { return money; } // 오류!!
  void setMoney(int money) {
    this->money = money;
  }
};
int main(){
  int n = PersonError::getMoney();

  PersonError errorKim;
  errorKim.setMoney(100);
}
```

non-static 멤버 함수는 static에 접근 가능

```
class Person {
  public: double money;
  static int sharedMoney;
   ....
  int total() {
    return money + sharedMoney;
  }
};
```

static 멤버 함수는 this 사용 불가

- static 멤버 함수는 객체 생성 전부터 호출 가능
- static 멤버 함수에서 this 사용 불가

```
class Person {
public:
   double money;
   static int sharedMoney;
      ....
   static void addShared(int n) {
     this->sharedMoney + = n; // 컴파일 오류
    }
};
```