

제9장 복사 생성자와 정적 멤버

1. 함수가 객체를 받아들이고 반환하는 과정을 자세히 살펴본다.
2. 복사 생성자의 개념을 이해한다.
3. 정적 멤버의 개념을 이해한다.

1

이번 장에서 만들어 볼 프로그램

1. 색상을 나타내는 **Color** 클래스를 정의하고 **Color** 객체를 **Circle** 클래스의 생성자로 전달해보자.
2. 싱글톤 디자인 패턴을 정적 멤버로 구현해보자.



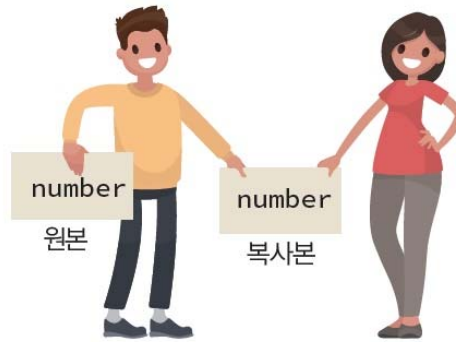
2

9.2 함수로 객체 전달하기

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



참조에 의한 호출



값에 의한 호출

3

함수로 객체를 전달하면?



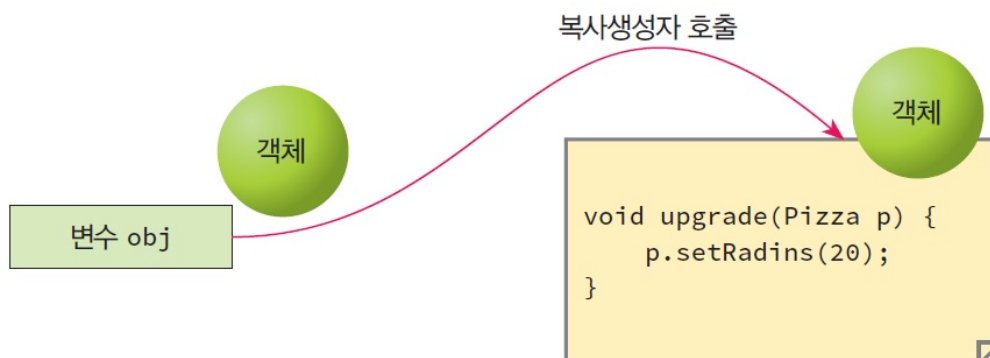
4

객체를 함수로 전달하기

```
#include <iostream>
using namespace std;
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() {
    Pizza obj(10);
    upgrade(obj);
    obj.print();
    return 0;
}
```

5


실행결과



6

객체의 주소를 함수로 전달하기

```
void upgrade(Pizza *p) {  
    p->setRadius(20);  
}  
  
int main()  
{  
    Pizza obj(10);  
    upgrade(&obj);  
  
    obj.print();  
    return 0;  
}
```

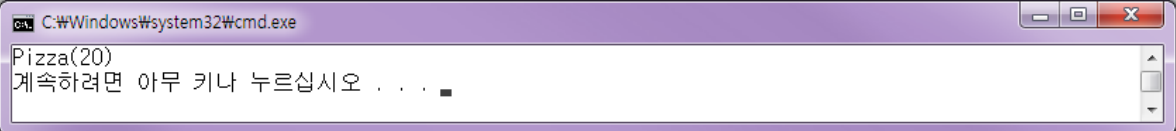


```
C:\Windows\system32\cmd.exe  
Pizza(20)  
계속하려면 아무 키나 누르십시오 . . .
```

7

참조자 매개변수 사용하기

```
void upgrade(Pizza& pizza) {  
    pizza.setRadius(20);  
}  
  
int main()  
{  
    Pizza obj(10);  
    upgrade(obj);  
  
    obj.print();  
    return 0;  
}
```

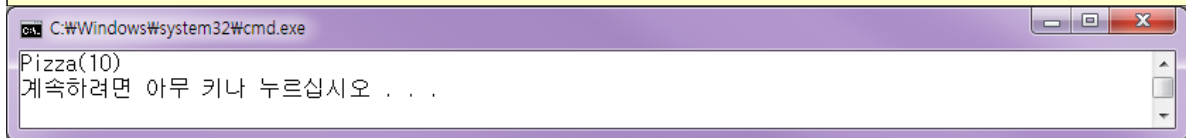


```
C:\Windows\system32\cmd.exe  
Pizza(20)  
계속하려면 아무 키나 누르십시오 . . .
```

8

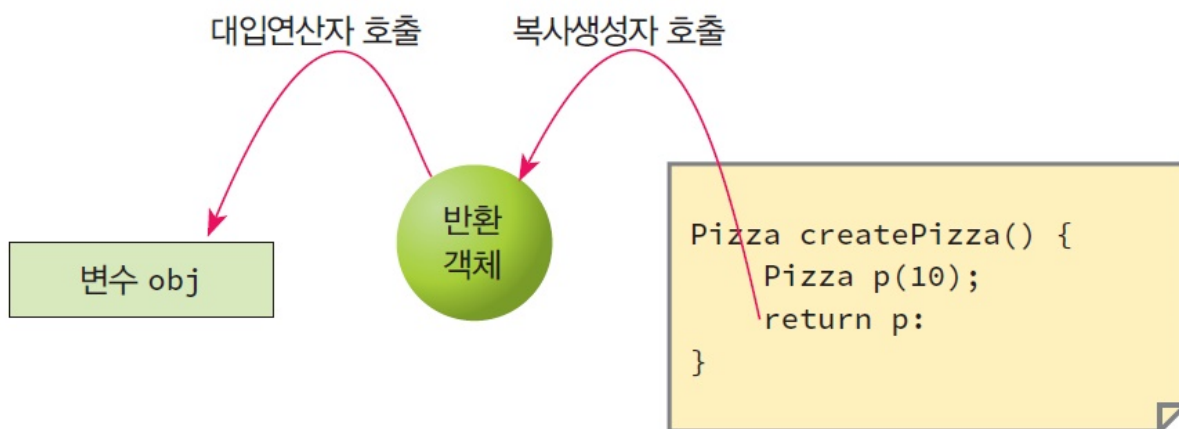
9.3 함수가 객체 반환하기

```
Pizza createPizza() {  
    Pizza p(10);  
    return p;  
}  
  
int main()  
{  
    Pizza obj;  
    obj = createPizza();  
  
    obj.print();  
    return 0;  
}
```



9

함수가 객체 반환하기



10

Lab: 객체를 함수로 전달하기

```
// 복소수를 클래스로 정의하고 복소수 덧셈 연산을 구현
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;
    }
};
```

11

Solution

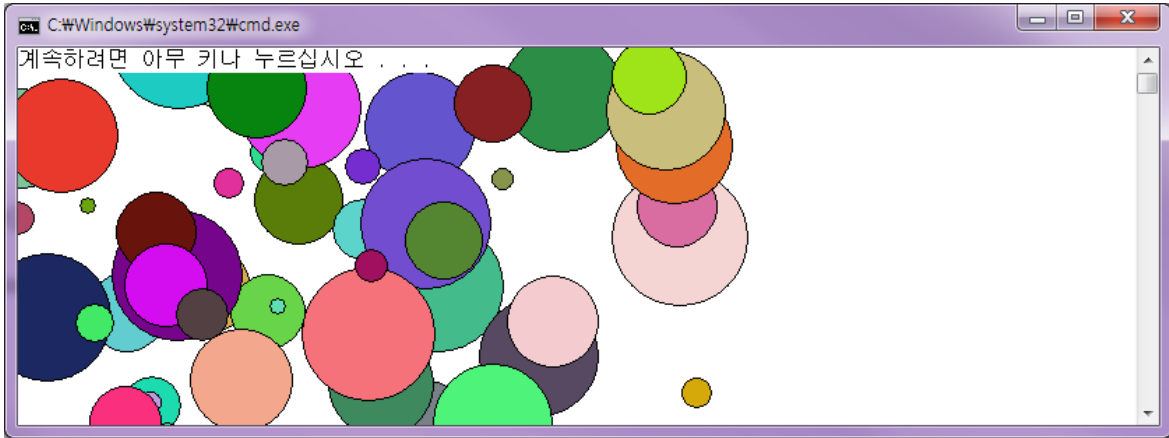
```
Complex add(Complex c1, Complex c2) {
    Complex temp;
    temp.real = c1.real + c2.real;
    temp.imag = c1.imag + c2.imag;
    return temp;
}

int main()
{
    Complex c1{ 1,2 }, c2{ 3,4 };
    Complex t;
    t = add(c1, c2);
    t.print();
    return 0;
}
```

12

복사생성자 이용

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



13

예제

```
#include <windows.h>           // 그리기를 위하여 필요하다.
#include <iostream>
#include <vector>
using namespace std;

class Color
{
public:
    int red, green, blue;
    Color() {
        red = rand() % 256;
        green = rand() % 256;
        blue = rand() % 256;
    }
};
```

14

예제

```
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();
};
// 원을 화면에 그리는 코드이다. 이해하지 않아도 된다.
void Circle::draw()
{
    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);
}
```

15

예제

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

16

9.4 복사생성자

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

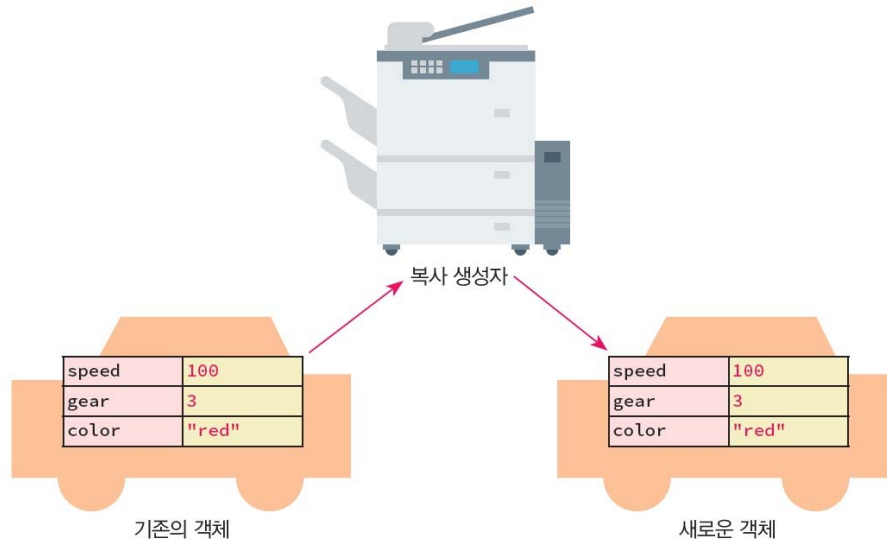


그림 9.1 복사 생성자는 다른 객체의 내용을 복사하여서 새로운 객체를 생성한다.

17

복사 생성자의 선언

문법 9.1

복사 생성자

```
MyClass( const MyClass& other )
{
    ....// other로 현재 객체를 초기화한다.
}
```


18

복사 생성자가 필요없는 경우

```
#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;
}
```

19

실행결과



```
C:\Windows\system32\cmd.exe
kim의 나이: 21 clone의 나이: 21
kim의 나이: 23 clone의 나이: 21
계속하려면 아무 키나 누르십시오 . . .
```

```
Person(const Person& other) : age{other.age}
{
}
}
```

기본 복사생성자:
자동으로 생성

20

복사 생성자가 필요한 경우

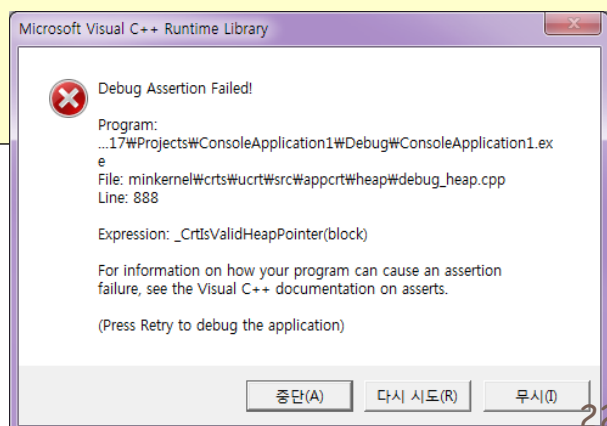
```
#include <iostream>
using namespace std;
class MyArray {
public:
    int size;
    int* data;
    MyArray(int size)
    {
        this->size = size;
        data = new int[size];
    }
    ~MyArray()
    {
        if (data != NULL) delete[] this->data;
    }
};
```

21

복사 생성자가 필요한 경우

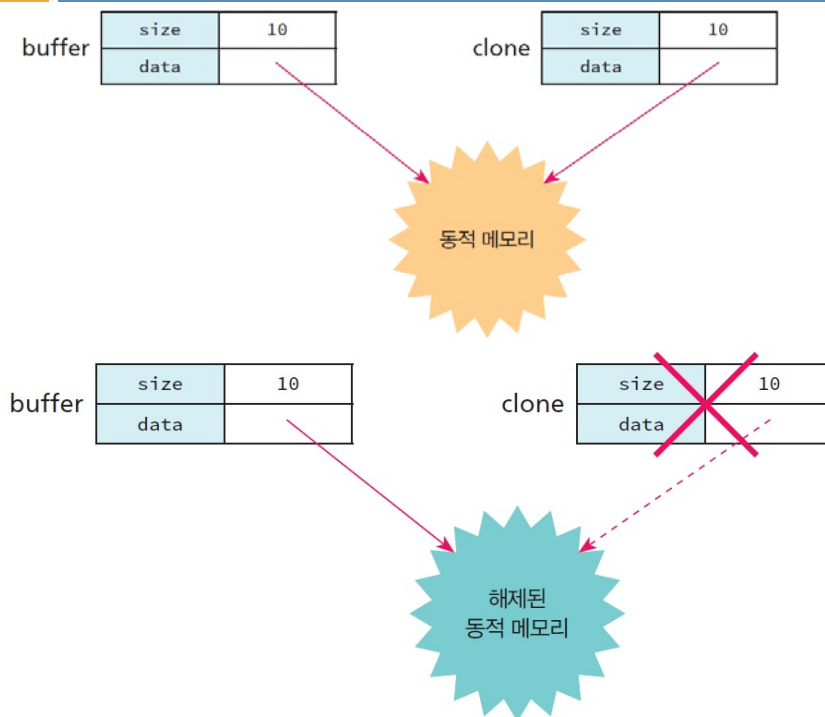
```
int main()
{
    MyArray buffer(10);
    buffer.data[0] = 1;
    {
        MyArray clone = buffer;
    }
    buffer.data[0] = 2;

    return 0;
}
```



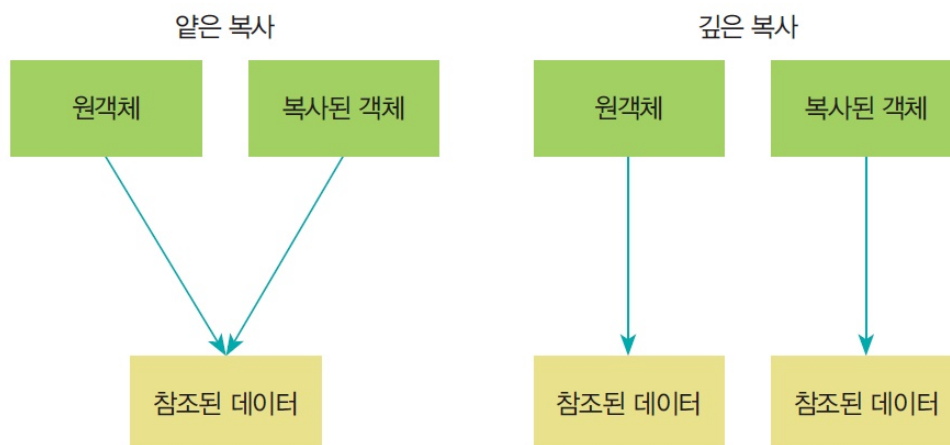
22

why?



23

깊은 복사와 얕은 복사



24

복사 생성자가 필요한 경우

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

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

25

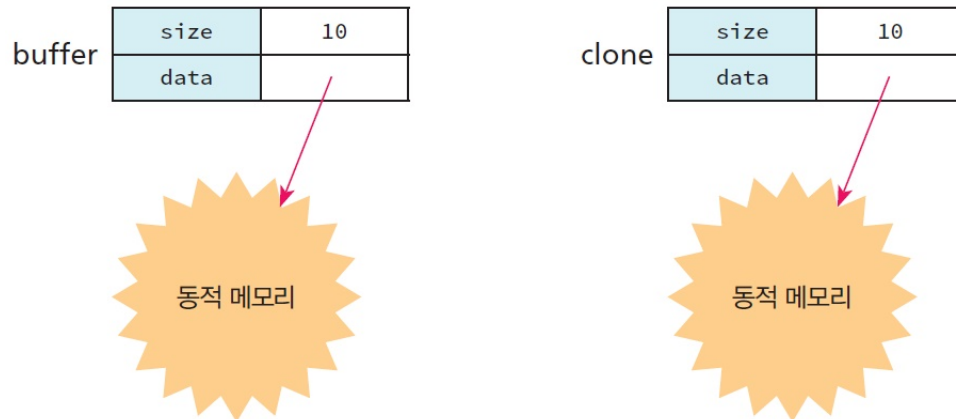
복사 생성자가 필요한 경우

```
MyArray::MyArray(const MyArray& other)
{
    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;
}
```

26

복사 생성자를 정의해주면



27

복사 생성자의 호출시점

복사 생성자가 호출되는 시점

- case 1: 기존에 생성된 객체를 이용해서 새로운 객체를 초기화하는 경우(앞서 보인 경우)
- case 2: Call-by-value 방식의 함수호출 과정에서 객체를 인자로 전달하는 경우
- case 3: 객체를 반환하되, 참조형으로 반환하지 않는 경우

메모리 공간의 할당과 초기화가 동시에 일어나는 상황

case 1

```
Person man1("Lee dong woo", 29);
Person man2=man1;    // 복사 생성자 호출
```

case 2 & case 3

```
SoSimple SimpleFuncObj(SoSimple ob)
{
    . . . . .
    return ob;
}

int main(void)
{
    SoSimple obj;
    SimpleFuncObj(obj);
    . . . . .
}
```

```
int SimpleFunc(int n)
{
    . . . . .
    return n;
}

int main(void)
{
    int num=10;
    cout<<SimpleFunc(num)<<endl;
    . . . . .
}
```

인자 전달 시 선언과 동시에 초기화

반환 시 메모리

공간 할당과 동시에 초기화

29

복사 생성자의 호출 case 2의 확인 PassObjCopycon.cpp

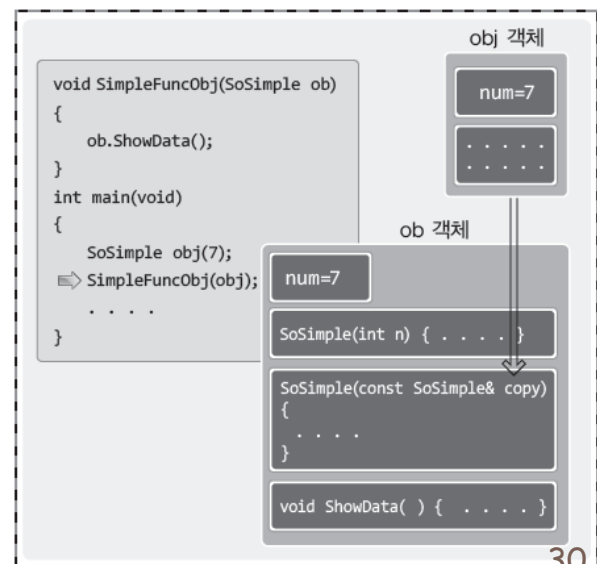
```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    { }
    SoSimple(const SoSimple& copy) : num(copy.num)
    {
        cout<<"Called SoSimple(const SoSimple& copy)"<<endl;
    }
    void ShowData()
    {
        cout<<"num: "<<num<<endl;
    }
};

void SimpleFuncObj(SoSimple ob)
{
    ob.ShowData();
}
```

```
int main(void)
{
    SoSimple obj(7);
    cout<<"함수호출 전"<<endl;
    SimpleFuncObj(obj);
    cout<<"함수호출 후"<<endl;
    return 0;
}
```

함수호출 전
Called SoSimple(const SoSimple& copy)
num: 7
함수호출 후

실행결과



30

복사 생성자의 호출 case3의 확인, ReturnObjCopycon.cpp

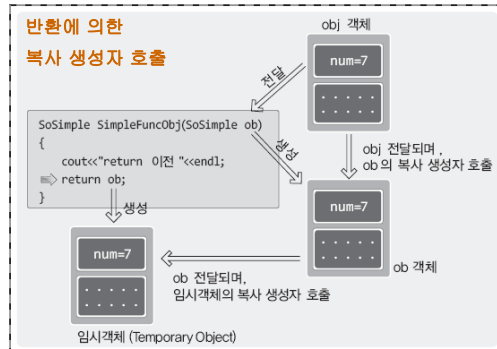
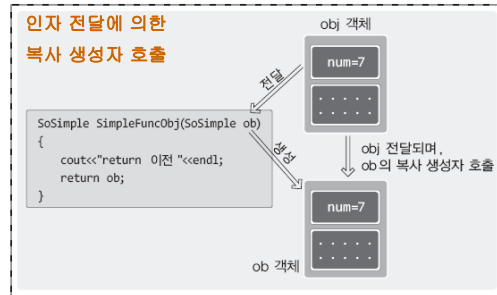
```

class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    { }
    SoSimple(const SoSimple& copy) : num(copy.num)
    {
        cout<<"Called SoSimple(const SoSimple& copy)"<<endl;
    }
    SoSimple& AddNum(int n)
    {
        num+=n;
        return *this;
    }
    void ShowData()
    {
        cout<<"num: "<<num<<endl;
    }
};

SoSimple SimpleFuncObj(SoSimple ob)
{
    cout<<"return 이전 "<<endl;
    return ob;
}

int main(void)
{
    SoSimple obj(7);
    SimpleFuncObj(obj).AddNum(30).ShowData();
    obj.ShowData();
    return 0;
}
    
```

Called SoSimple(const SoSimple& copy)
return 이전
Called SoSimple(const SoSimple& copy)
num: 37
num: 7 **실행결과**



31

반환할 때 만들어진 객체의 소멸 시점, ReturnObjDeadTime.cpp

```

class Temporary
{
private:
    int num;
public:
    Temporary(int n) : num(n)
    {
        cout<<"create obj: "<<num<<endl;
    }
    ~Temporary()
    {
        cout<<"destroy obj: "<<num<<endl;
    }
    void ShowTempInfo()
    {
        cout<<"My num is "<<num<<endl;
    }
};
    
```

```

int main(void)
{
    Temporary(100);
    cout<<"***** after make!"<<endl<<endl;

    Temporary(200).ShowTempInfo();
    cout<<"***** after make!"<<endl<<endl;
    const Temporary &ref=Temporary(300);
    cout<<"***** end of main!"<<endl<<endl;
    return 0;
}

create obj: 100
destroy obj: 100
***** after make!

create obj: 200
My num is 200
destroy obj: 200
***** after make!

create obj: 300
***** end of main!
destroy obj: 300
    
```

실행결과

참조값이 반환되므로
참조자로 참조 가능!

Temporary(200).ShowTempInfo(); → (임시객체의 참조 값).ShowTempInfo();

32

실습: ReturnObjDeadTime.cpp

```
class SoSimple
{
private:
int num;
public:
SoSimple(int n) : num(n)
{
cout<<"New Object: "<<this<<endl;
}
SoSimple(const SoSimple& copy) :
num(copy.num)
{
cout<<"New Copy obj: "<<this<<endl;
}
~SoSimple()
{
cout<<"Destroy obj: "<<this<<endl;
}
};

SoSimple SimpleFuncObj(SoSimple
ob)
{
cout<<"Parm ADR: "<<&ob<<endl;
return ob;
}

int main(void)
{
SoSimple obj(7);
SimpleFuncObj(obj);
cout<<endl;
SoSimple
tempRef=SimpleFuncObj(obj);
cout<<"Return Obj
"<<&tempRef<<endl;
return 0;
}
```

33

복사 생성자가 사용되는 3가지 경우는?

-
- 어떤 경우에 별도의 복사 생성자를 정의해야 하는가?

객체와 연산자

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

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

    obj2 = obj1;    // obj1의 멤버 변수가 obj2으로 복사된다.
    return 0;
}
```

대입 연산자

깊은 복사는 포인터가 가리키는 메모리에 대한 별도의 사본을 만드는 것이다.

```
class CMyData
{
public:
    CMyData(int nParam)
    {
        m_pnData = new int;
        *m_pnData = nParam;
    }
    // 복사 생성자 선언 및 정의 //
    CMyData(const CMyData &rhs)
    {
        cout << "CMyData(const CMyData &)" << endl;
        // 메모리를 할당한다.
        m_pnData = new int;
        // 포인터가 가리키는 위치에 값을 복사한다.
        *m_pnData = *rhs.m_pnData;
    }
    int *m_pnData;
    ...
}
```

37

대입 연산자

단순 대입 연산자 '함수'에서도 동일한 문제가 발생한다.

따라서 깊은 복사가 필요하다면 복사 생성자 외에 대입 연산자도 정의해야 한다.

```
int main()
{
    CMyData a(10);
    CMyData b(20);

    // 단순 대입을 시도하면 모든 멤버의 값을 그대로 복사한다.
    a = b; // -> 문제 해결하기

    cout << a.GetData() << endl;
    return 0;
}
```

38

복사 생성자와 단순 대입 연산자의 코드는 비슷한 구조를 갖는다.

```
// 단순 대입 연산자 함수를 정의한다.
CMyData& operator=(const CMyData &rhs)
{
    *m_pnData = *rhs.m_pnData;
    // 객체 자신에 대한 참조를 반환한다.
    return *this;
}
```

39

묵시적 변환, 변환 생성자

매개변수가 한개인 생성자이다.

이 코드는 형식인수와 실인수 형식이 다름에도 컴파일 오류가 발생하지 않는다.

```
class CTestData {
public:
    CTestData(int nParam) : m_nData(nParam)
    {
        cout << "CTestData(int)" << endl;
    }
    ...
};
void TestFunc(CTestData param)
{
    cout << "TestFunc(): " << param.GetData() << endl;
}
int main( )
{
    TestFunc(5);
    return 0;
}
```

40

변환 생성자

매개변수가 한 개인 생성자이다.
이 코드에 등장하는 CTestData 클래스의 인스턴스 수는?

```
class CTestData {
public:
    CTestData(int nParam) : m_nData(nParam)
    {
        cout << "CTestData(int)" << endl;
    }
    ...
};
void TestFunc(const CTestData & param)
{
    cout << "TestFunc(): " << param.GetData() << endl;
}
int main( )
{
    TestFunc(5);
    return 0;
}
```

41

변환 생성자

변환 생성자를 선언할 때는 반드시 explicit로 선언한다.

```
class CTestData {
public:
    explicit CTestData(int nParam) : m_nData(nParam)
    {
        cout << "CTestData(int)" << endl;
    }
    ...
};
void TestFunc(const CTestData & param)
{
    cout << "TestFunc(): " << param.GetData() << endl;
}
int main( )
{
    TestFunc(5);
    return 0;
}
```

42

허용되는 변환 (형변환 연산자, 형변환자)

허용되는 변환 형식을 규정하면 형식간의 호환성이 생긴다.

```
class CTestData
{
public:
    explicit CTestData(int nParam) : m_nData(nParam) { }
    // CTestData 클래스는 int 자료형식으로 변환될 수 있다!
    operator int(void) { return m_nData; }

    int GetData() const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }
private:
    int m_nData = 0;
};
int main( )
{
    CTestData a(10);
    cout << a << endl;
    return 0;
}
```

43


비교연산자

```
class Person {
public:
    int age;
    Person(int a) : age(a) { }
};
int main() {
    Person obj1(20);
    Person obj2(20);

    if (obj1 == obj2) {
        cout << "같습니다" << endl;
    }
    else {
        cout << "같지 않습니다" << endl;
    }
    return 0;
}
```

44

실행결과

문	코드	설명	프로젝트	파일	줄	Suppression State
▶	 E0349	이러한 피연산자와 일치하는 "==" 연산자가 없습니다.	ConsoleApplication1	소스.cpp	12	

연산자 중복을 사용하여야 함

45

임시객체와 이동 시멘틱 ,이름 없는 임시 객체

함수 반환이나 연산 과정에서 코드에 보이지 않는 인스턴스가 생겼다 사라진다.

```
// CTestData 객체를 반환하는 함수다.
CTestData TestFunc(int nParam)
{
    // CTestData 클래스 인스턴스인 a는 지역변수다.
    // 따라서 함수가 반환되면 소멸한다.
    CTestData a(nParam, "a");
    return a;
}

int main( )
{
    CTestData b(5, "b");
    cout << "*****Before*****" << endl;
    // 함수가 반환하면서 임시 객체가 생성됐다가 대입 연산 후 즉시 소멸한다!
    b = TestFunc(10);
    cout << "*****After*****" << endl;
    cout << b.GetData() << endl;
    return 0;
}
```

46

이름 없는 임시 객체

임시 객체의 생성과 소멸 규칙

- 임시 객체는 함수 반환이나 연산 과정에서 생겨난다.
- 임시 객체는 모두 **r-value**이다.
- 임시 객체는 이어지는 연산에 참여직후 자동으로 소멸한다.
- 만일 이름 없는 임시 객체에 대해 참조자를 선언할 경우 참조자가 속한 scope가 닫힐 때까지 임시 객체도 살아 남는다.

```
CTestData &rData = TestFunc(10);
```

47

r-value 참조

`int &&nData = 3 + 7;` 처럼 (연산의) 임시 결과에 대한 참조자 선언이다.

```
void TestFunc(int &&rParam)
{
    cout << "TestFunc(int &&)" << endl;
}

int main( )
{
    // 3 + 4 연산 결과는 r-value이다. 절대로 l-value가 될 수 없다.
    TestFunc(3 + 4);
    return 0;
}
```

48

이동 시맨틱

복사 생성자와 대입 연산자에 r-value 참조를 조합해서 새로운 생성(이동 생성자) 및 대입(이동 대입 연산자)의 경우를 만들어 낸 것이다.

```
class CTestData {
public: CTestData()          { cout << "CTestData()" << endl; }
      ~CTestData() { cout << "~CTestData()" << endl; }
      CTestData(const CTestData &rhs) : m_nData(rhs.m_nData)
      { cout << "CTestData(const CTestData &)" << endl; }
      // 이동 생성자
      CTestData(CTestData &&rhs) : m_nData(rhs.m_nData)
      { cout << "CTestData(const CTestData &&)" << endl; }
      int GetData() const { return m_nData; }
      void SetData(int nParam) { m_nData = nParam; }
private:
      int m_nData = 0;
};
//이동생성자 정의시 이동 대입연산자나 복사생성자 반드시 같이 정의해야 함
```

49

move semantics

TestFunc() 함수 반환시 임시 객체의 이동 생성자가 호출된다.
곧 사라질 임시 객체(a)에 대해 얇은 복사를 수행하여 성능을 높이는 것이 핵심이다.

```
CTestData TestFunc(int nParam)
{
    cout << "***TestFunc(): Begin***" << endl;
    CTestData a;
    a.SetData(nParam);
    cout << "***TestFunc(): End*****" << endl;
    return a;
}

int main( )
{
    CTestData b;
    cout << "**Before*****" << endl;
    b = TestFunc(20);
    cout << "**After*****" << endl;
    CTestData c(b);
    return 0;
}
```

50

함수의 매개변수가 기본형식이 아니라 클래스라면 매개변수 형식은 어떻게 정하는 것이 바람직한지 이유를 답하시오.

복사 생성자 및 단순 대입연산자를 반드시 정의해야 하는 클래스는 어떤 클래스인가?

이동 시맨틱이 등장한 가장 큰 원인은 무엇인가?

51

만일 다음과 같은 코드에서 컴파일 오류가 없었다면 CTestData 클래스는 잠재적인 문제를 가진 것으로 볼 수 있다. 그 문제는 무엇이고 어떻게 막을 수 있는가?

```
void TestFunc(const CTestData &param)
{
    ...
}
int main()
{
    TestFunc(5);
    return 0;
}
```

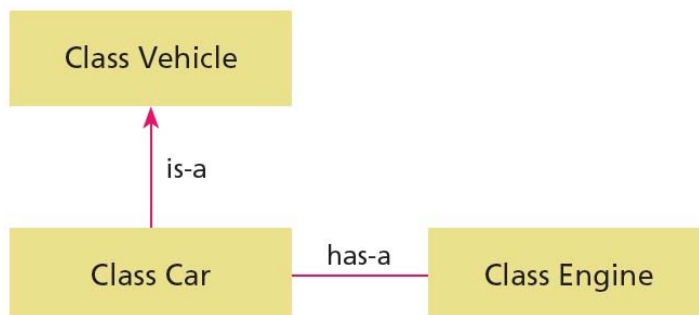
52

9.5 is-a와 has-a

- **is-a** 관계: 객체 지향 프로그래밍에서 **is-a**의 개념은 상속을 기반으로 한다. 우리는 아직 상속은 학습하지 않았다. "A는 B 유형의 물건"이라고 말하는 것과 같다. 예를 들어, **Apple**은 과일의 일종이고, **Car**는 자동차의 일종이다.
- **has-a** 관계: **has-a**는 하나의 객체가 다른 객체를 가지고 있는 관계이다. 예를 들어서 **Car**에는 **Engine**이 있고 **House**에는 **Bathroom**이 있다.

53

is-a와 has-a



54

has-a 관계

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

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;
    }
};
```

55

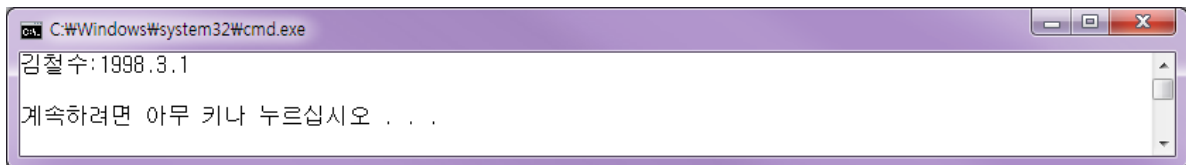
has-a 관계

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

56

has-a 관계

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



57

9.6 정적 변수

- 정적 변수는 **static**를 붙여서 선언하는 변수로서 클래스마다 하나만 생성된다.

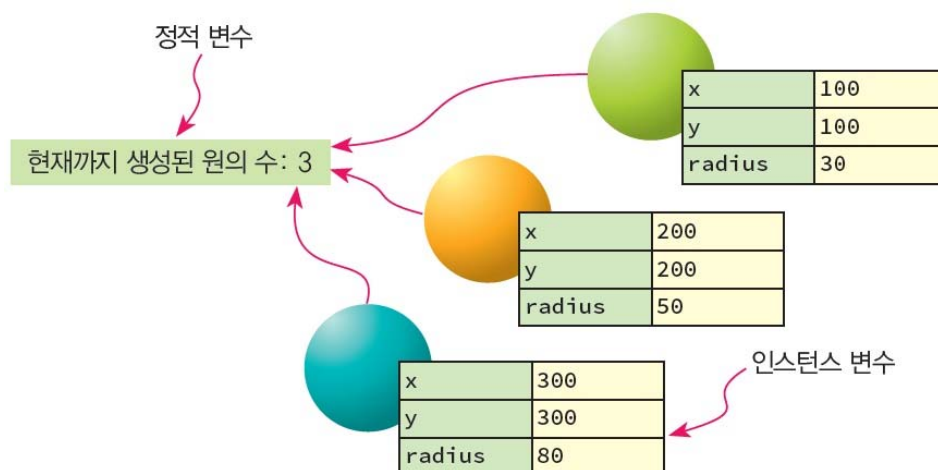


그림 9.2 정적 멤버

58

정적 변수 예제

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

int main()
{
    Circle c1;
    Circle c2;
    ...
}
```

59

전체 소스

```
#include <iostream>
#include <string>
using namespace std;
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++;
    }
    ~Circle() { --count; }
};
```

60

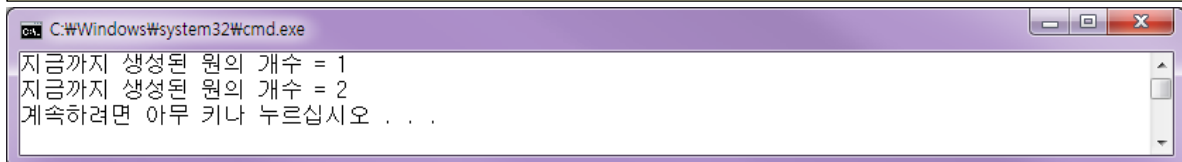
전체 소스

```
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;

    return 0;
}
```



C:\Windows\system32\cmd.exe

```
지금까지 생성된 원의 개수 = 1
지금까지 생성된 원의 개수 = 2
계속하려면 아무 키나 누르십시오 . . .
```

61

정적 멤버 함수

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

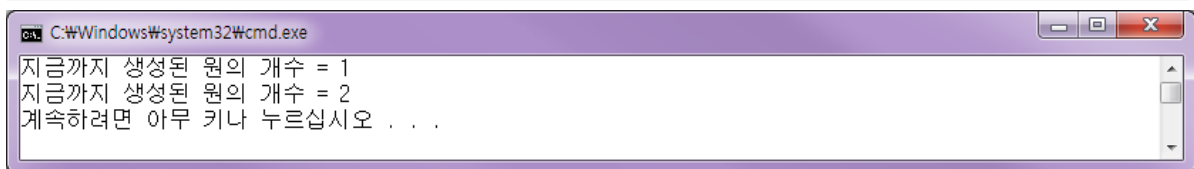
int Circle::count = 0; // ①
```

62

정적 멤버 함수

```
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; //

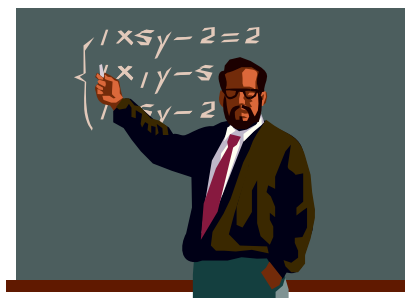
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
지금까지 생성된 원의 개수 = 1
지금까지 생성된 원의 개수 = 2
계속하려면 아무 키나 누르십시오 . . .
```

63

Q & A



64