

## 제5장 생성자와 접근제어

1. 객체 지향 기법을 이해한다.
2. 클래스를 작성할 수 있다.
3. 클래스에서 객체를 생성할 수 있다.
4. 생성자를 이용하여 객체를 초기화할 수 있다.
5. 접근자와 설정자를 사용할 수 있다.

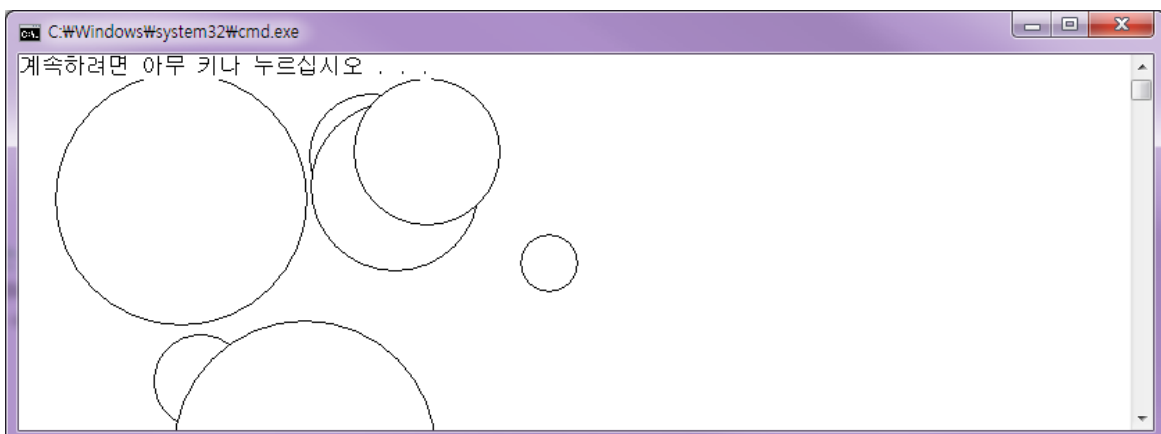
1

### 이번 장에서 만들어볼 프로그램

시간을 나타내는 클래스 **Time**의 객체를 생성자로 초기화



생성자를 이용하여 랜덤한 크기의 **Circle** 객체를 많이 생성



2

## 5.2 생성자

- 생성자(constructor)는 초기화를 담당하는 함수

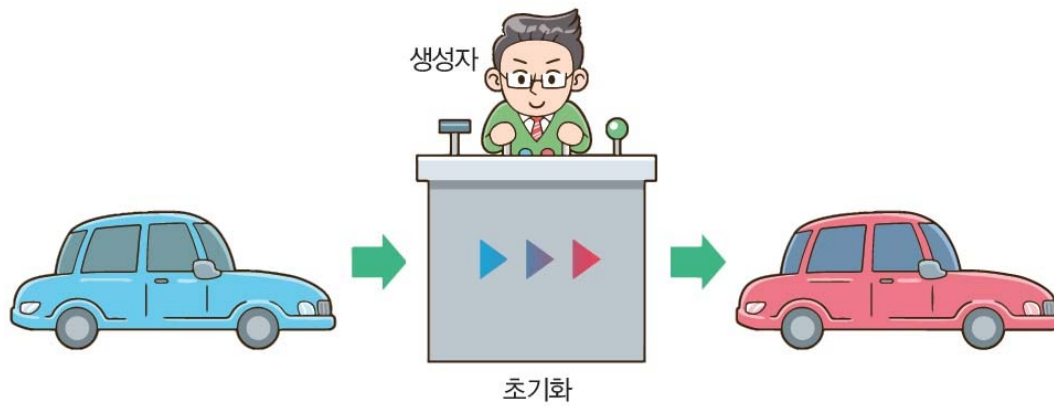


그림 5.1 생성자의 역할

3

## 생성자가 필요한 이유

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

```
class Time {
public:
    int hour;           // 시를 나타낸다. 0-23가 가능하다.
    int minute;         // 분을 나타낸다. 0-59가 가능하다.

    void print() {
        cout << hour << ":" << minute << endl;
    }
};
```

```
Time a;                // 객체 a를 생성한다.
a.hour = 26;
a.minute = 70;
```

4

## 생성자의 예

```
class Time {  
public:  
    int hour;           // 0-23  
    int minute;         // 0-59  
  
    Time(int h, int m) {  
        hour = h;  
        minute = m;  
    }  
    void print() {  
        cout << hour << ":" << minute << endl;  
    }  
};
```

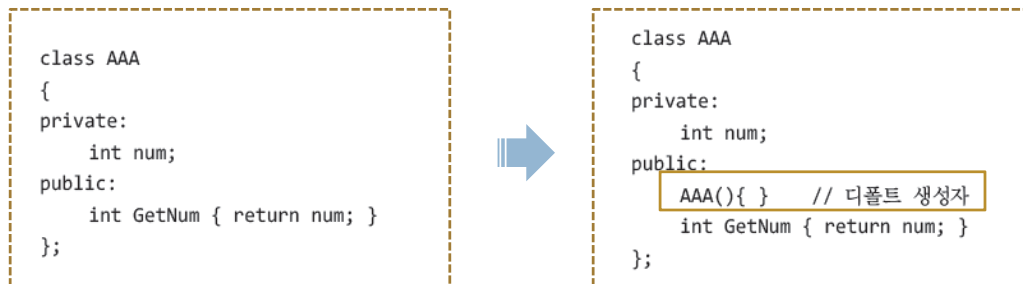
5

## 생성자를 호출하는 방법

```
Time a;           // ① 오류이다! 초기화값이 없다!  
Time b(10, 25);   // ② OK 하지만 예전의 방법이다.  
Time c { 10, 25 }; // ③ OK 최신의 방법이다.  
Time d = { 10, 25 }; // ④ OK 하지만 간결하지 않다.
```

6

## 디폴트 생성자



생성자를 정의하지 않으면 인자를 받지 않고, 하는 일이 없는 디폴트 생성자라는 것이 컴파일러에 의해서 추가된다.  
따라서 모든 객체는 무조건 생성자의 호출 과정을 거쳐서 완성된다.

7

## 생성자의 중복 정의

```
#include <iostream>
using namespace std;
class Time {
public:
    int hour;           // 0-23
    int minute;         // 0-59
    Time() {
        hour = 0;
        minute = 0;
    }
    Time(int h, int m) {
        hour = h;
        minute = m;
    }
    void print() {
        cout << hour << ":" << minute << endl;
    }
};
```

8

## 디폴트 인수를 사용하는 생성자

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

class Time {
public:
    int hour;           // 0-23
    int minute;         // 0-59

    Time(int h=0, int m=0) {
        hour = h;
        minute = m;
    }
    void print() {
        cout << hour << ":" << minute << endl;
    }
};
```

9

## 디폴트 인수를 사용하는 생성자

```
int main()
{
    Time a;           // OK
    Time b{ 10, 25 }; // OK

    a.print();
    b.print();
    return 0;
}
```



10

## 멤버 초기화 리스트

```
Time(int h, int m) : hour{h}, minute{m}
{
}

Time(int h, int m) : hour(h), minute(m)
{
}

Time(int h=0, int m=0) : hour{h}, minute{m} {
}
```

11


## 이니셜라이저를 이용한 변수 및 상수의 초기화

```
class SoSimple
{
private:
    int num1;
    int num2;
public:
    SoSimple(int n1, int n2) : num1(n1)
    {
        num2=n2;
    }
    . . . . .
};
```

왼쪽에서 보이듯이 이니셜라이저를 통해서 멤버변수의 초기화도 가능하며, 이렇게 초기화 하는 경우 선언과 동시에 초기화되는 형태로 바이너리가 구성된다. 즉, 다음의 형태로 멤버변수가 **선언과 동시에 초기화**된다고 볼 수 있다.

```
int num1 = n1;
```

따라서 **const**로 선언된 멤버변수도 초기화가 가능하다. 선언과 동시에 초기화 되는 형태이므로...



```
class FruitSeller
{
private:
    const int APPLE_PRICE;
    int numOfApples;
    int myMoney;
public:
    FruitSeller(int price, int num, int money)
    : APPLE_PRICE(price), numOfApples(num), myMoney(money)
    {
    }
}
```

12

## 멤버변수로 참조자 선언하기

```
class BBB
{
private:
    AAA &ref;
    const int &num;
public:
    BBB(AAA &r, const int &n)
        : ref(r), num(n)
    { // empty constructor body
    }
```

이니셜라이저의 초기화는 선언과 동시에 초기화 되는  
형태이므로, 참조자의 초기화도 가능하다!

13

## 생성자 불일치

```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n) { }
};
```

SoSimple simObj1(10); (○)

SoSimple \* simPtr1=new SoSimple(2); (○)

SoSimple simObj2; (×)

SoSimple \* simPtr2=new SoSimple; (×)

이 형태로 객체 생성이 가능하기 위해서는 다음  
형태의 생성자를 별도로 추가해야 한다.

**SoSimple() : num(0) { }**

생성자가 삽입되었으므로, 디폴트 생성자는 추가되지 않는다. 따라서 인자를 받지  
않는 void형 생성자의 호출은 불가능하다.

14

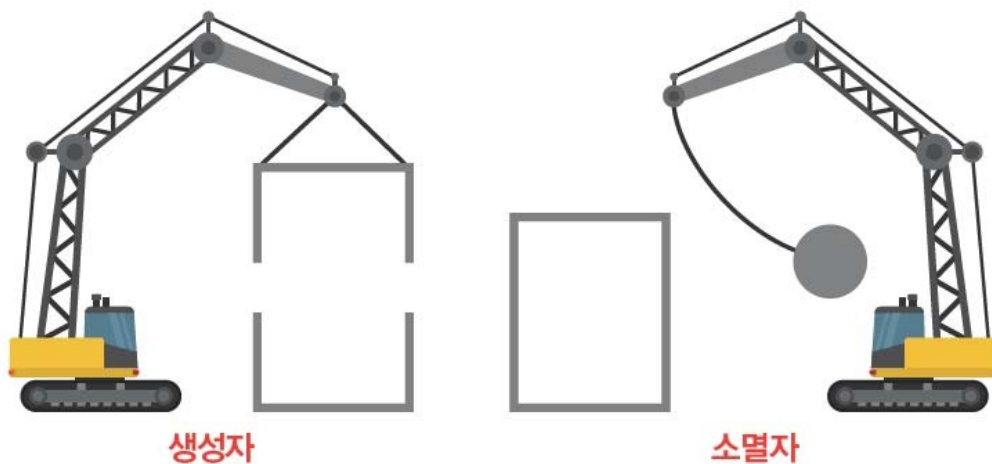
## 문제

- 계산기 기능의 **Calculator** 클래스를 정의하라. 덧셈, 뺄셈, 곱셈, 나눗셈 기능이 있어야 하고, 연산을 할 때마다 어떠한 연산을 몇 번 수행했는지 기록해야 한다.

```
int main(void)
{
    Calculator cal;
    cout << "3.2 + 2.4 = " << cal.Add(3.2, 2.4) << endl;
    cout << "3.5 / 1.7 = " << cal.Div(3.5, 1.7) << endl;
    cout << "2.2 - 1.5 = " << cal.Min(2.2, 1.5) << endl;
    cout << "4.9 / 1.2 = " << cal.Div(4.9, 1.2) << endl;
    cal.ShowOpCount();
    return 0;
}
```

15

## 5.3 소멸자



16



## 소멸자의 이해

```
class AAA
{
    // empty class
};
```

```
~AAA() { . . . }
```

AAA 클래스의 소멸자! 객체 소멸 시  
자동으로 호출된다.



```
class AAA
{
public:
    AAA() { }
    ~AAA() { }
};
```

생성자와 마찬가지로 소멸자도 정의하지  
않으면 디폴트 소멸자가 삽입된다.

17

## 소멸자의 활용

```
class Person
{
private:
    char * name;
    int age;
public:
    Person(char * myname, int myage)
    {
        int len=strlen(myname)+1;
        name=new char[len];
        strcpy(name, myname);
        age=myage;
    }
    void ShowPersonInfo() const
    {
        cout<<"이름: "<<name<<endl;
        cout<<"나이: "<<age<<endl;
    }
    ~Person()
    {
        delete []name;
        cout<<"called destructor!"<<endl;
    }
};
```

생성자에서 할당한 메모리 공간을 소멸시키기  
좋은 위치가 소멸자이다.

18

## 소멸자

```
#include <string.h>
class MyString {
private:
    char *s;
    int size;
public:
    MyString(char *c) {
        size = strlen(c)+1;
        s = new char[size];
        strcpy(s, c);
    }
    ~MyString() {
        delete[] s;
    }
};
int main() {
    MyString str("abcdefghijk");
}
```

19

## 생성자와 소멸자

### 주의사항

- 만일 클래스 객체를 전역변수로 선언한다면 그 클래스의 생성자가 **main()** 함수보다 먼저 호출된다.
- 생성자는 다중 정의할 수 있다.
- 소멸자는 다중 정의할 수 없다.
- **main()** 함수가 끝난 후에 소멸자가 호출될 수 있다. (전역변수)
- 생성자와 소멸자는 생략할 수 있으나 이 경우 컴파일러가 기본 생성자/소멸자를 만들어 넣는다.
- 만일 생성자를 다중 정의했다면 기본 생성자를 아예 생략할 수도 있다.
- **배열로 객체를 동적 생성했다면 반드시 배열로 삭제해야 한다.**

20

## Lab: Rect 클래스

- 사각형을 나타내는 **Rectangle** 클래스에 생성자를 추가해 보자. 객체를 생성하고 사각형의 넓이를 출력하라. 멤버 함수는 클래스 외부에서 정의한다.



21

## solution

```
#include <iostream>
using namespace std;
class Rectangle {
    int width, height;
public:
    Rectangle(int w, int h);
    int calcArea();
};
Rectangle::Rectangle(int w, int h)
{
    width = w;
    height = h;
}
int Rectangle::calcArea()
{
    return width*height;
}
```

22

## solution

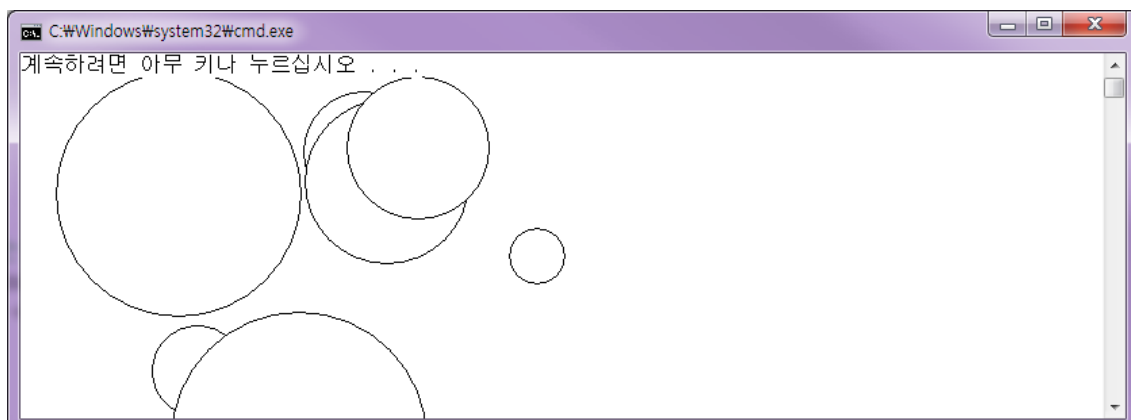
```
int main()
{
    Rectangle r{ 3, 4 };

    cout << "사각형의 넓이 : " << r.calcArea() << '\n';
    return 0;
}
```

23

## Lab: Circle 클래스

- 원을 나타내는 **Circle** 클래스를 작성하였다. 원은 중심점 및 반지름과 색상으로 표현된다. 약 10개의 **Circle** 객체를 생성하면서 랜덤한 위치와 랜덤한 반지름으로 화면에 원을 그려보자. 원의 중심점과 반지름은 생성자를 호출하여 설정하라. 멤버 함수는 클래스 외부에 정의한다.



24

## solution

```
#include <windows.h>
#include <iostream>
using namespace std;
class Circle
{
    int x, y, radius;
    string color;
public:
    Circle(int xval = 0, int yval = 0, int r = 0, string c = "");
    double calcArea() { return radius*radius*3.14;}
    void draw();
};
Circle::Circle(int xval, int yval, int r, string c): x{xval}, y{yval}, radius{r},
color{c}
{}
void Circle::draw() {
    HDC hdc = GetWindowDC(GetForegroundWindow());
    Ellipse(hdc, x-radius, y-radius, x+radius, y+radius);
}
```

25

## 실습

- 앞에서 작성한 **Rect** 클래스에 생성자를 추가하여 화면에 사각형을 랜덤하게 그리는 프로그램을 작성하고 실행 예를 보이시오.

26

# 생성자 다중정의 및 위임

생성자 위임은 C++11 표준부터 지원한다.

```
class CMyPoint
{
public:
    CMyPoint(int x)
    {
        cout << "CMyPoint(int)" << endl;
        ...
    }
    CMyPoint(int x, int y)
    // x 값을 검사하는 코드는 이미 존재하므로 재사용한다.
    : CMyPoint(x)
    {
        ...
    }
    ...
private:
    int m_x = 0;
    int m_y = 0;
};
```

생성자에서 다른 생성자가 호출되도록 '위임'했다.

27

## □ 생성자 위임 두 번 시키기

```
class CMyPoint
{
public:
    CMyPoint(int x)
    { cout << "CMyPoint(int)" << endl; ... }
    CMyPoint(int x, int y): CMyPoint(x)
    { ... }
    CMyPoint(int x, int y, int z): CMyPoint(x, y) {... }
private:
    int m_x = 0;
    int m_y = 0;
};
```

28

# 명시적 디폴트 생성자

---

## □ default

- ▣ 별도로 정의를 기술하지 않고도 선언과 정의를 분리

```
class CTest
{
public:
    // 디폴트 생성자 선언 및 정의!
    CTest(void) = default;
    int m_nData = 5;
};
```

29

---

## □ delete

- ▣ 생성자 다중 정의를 통해 새로운 생성자를 기술하고 디폴트 생성자를 기술하지 않는 경우
  - ▣ 명시적으로 디폴트 생성자가 사라졌음을 나타냄
- ```
CTest(void) = delete;
```

30

## 5.4 접근제어

### □ access control

- 외부에서 특정한 멤버 변수나 멤버 함수에 접근하는 것을 제어하는 것
- private member
- public member

31

## 접근제어

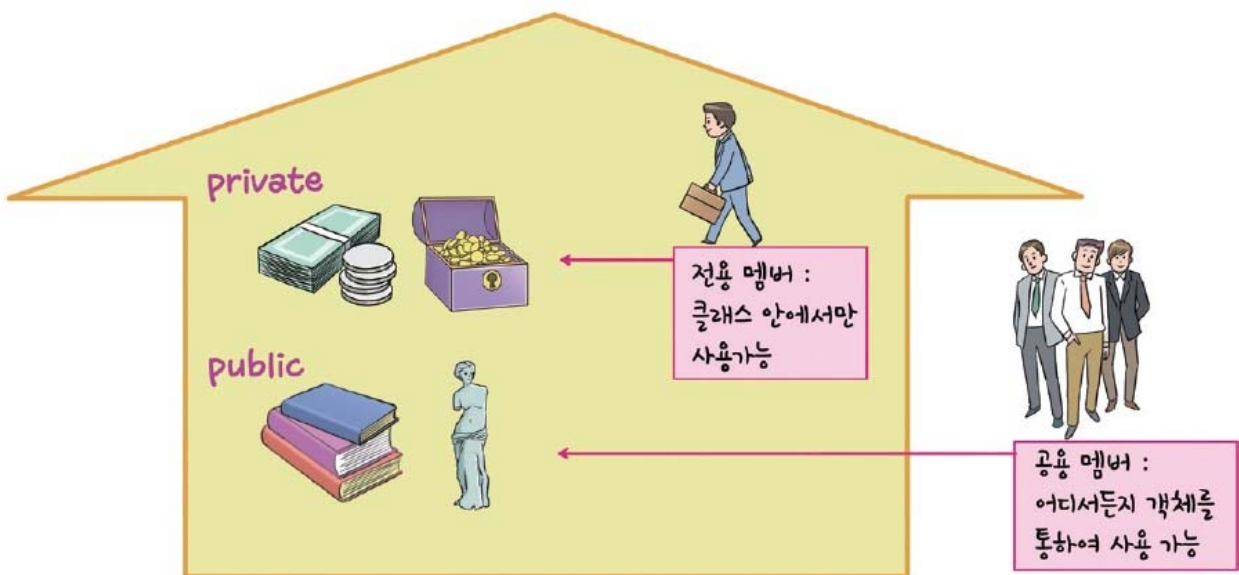


그림 5.2 접근 지정자

32



## 접근제어 지시자

### 접근제어 지시자

- ▶ **public** 어디서든 접근허용
- ▶ **protected** 상속관계에 놓여있을 때, 유도 클래스에서의 접근허용
- ▶ **private** 클래스 내(클래스 내에 정의된 함수)에서만 접근허용

```
class Car
{
private:
    char gamerID[CAR_CONST::ID_LEN];
    int fuelGauge;
    int curSpeed;
public:
    void InitMembers(char * ID, int fuel);
    void ShowCarState();
    void Accel();
    void Break();
};
```

```
int main(void)
{
    Car run99;
    run99.InitMembers("run99", 100);
    run99.Accel();
    run99.Accel();
    run99.Accel();
    run99.ShowCarState();
    run99.Break();
    run99.ShowCarState();
    return 0;
}
```

Car의 멤버함수는 모두 **public**이므로  
클래스의 외부에 해당하는 **main** 함수에서  
접근가능!

33

### □ private

- 같은 타입 객체의 **private** 멤버에는 접근가능

```
class First {
private:
    int num1, num2;
public:
    First(int n1=0, int n2=0) : num1(n1), num2(n2) { }
    void ShowData() { cout<<num1<<" ", "<<num2<<endl; }
    First& operator=(const First& ref)
    {
        cout<<"First& operator=( )" <<endl;
        num1=ref.num1;
        num2=ref.num2;
        return *this;
    }
};
```

34

## 예제

```
class Time {  
private:                                // 이후에 선언되는 멤버는 모두 전용 멤버가 된다.  
    int hour;                          // 0-23  
    int minute;                        // 0-59  
public:  
    Time(int h, int m);  
    void inc_hour();  
    void print();  
};  
Time::Time(int h, int m) {  
    hour = h;  
    minute = m;  
}  
void Time::inc_hour() {  
    ++hour;  
    if (hour > 23)  
        hour = 0;  
}
```

35

## 예제

```
Time a { 24, 59 };  
++a.hour;                // 이제는 오류가 발생한다!!
```

36

## 접근자(getter)와 설정자(setter)

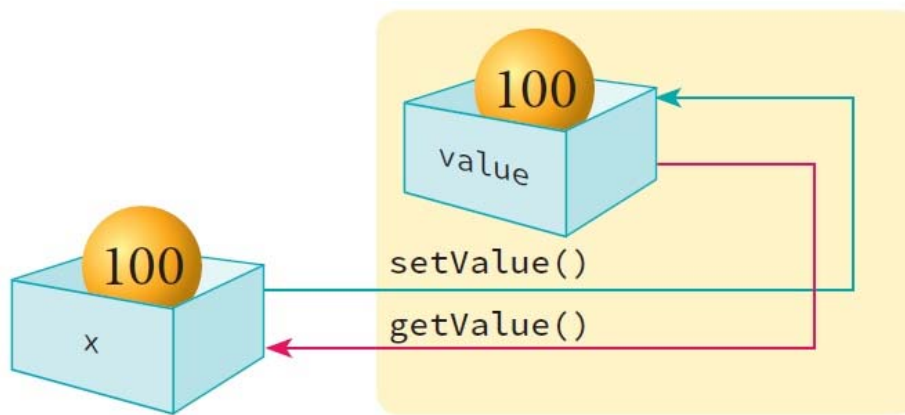


그림 5.3 접근자와 설정자는 멤버 변수의 접근을 제한한다.

37

## 예제

```
#include <iostream>
using namespace std;
class Time {
public:
    Time(int h, int m);
    void inc_hour();
    void print();
    int getHour() { return hour; }
    int getMinute() { return minute; }
    void setHour(int h) { hour = h; }
    void setMinute(int m) { minute = m; }
private:
    int hour;           // 0-23
    int minute;         // 0-59
};
```

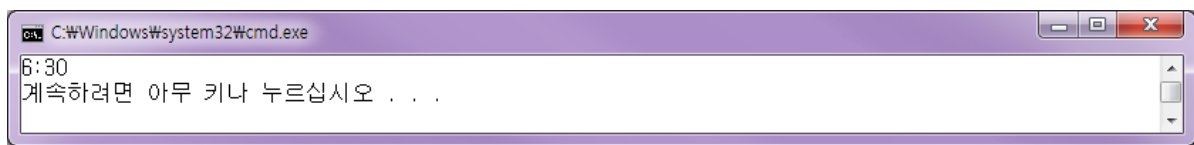
38

## 예제

```
int main()
{
    Time a{ 0, 0 };

    a.setHour(6);
    a.setMinute(30);

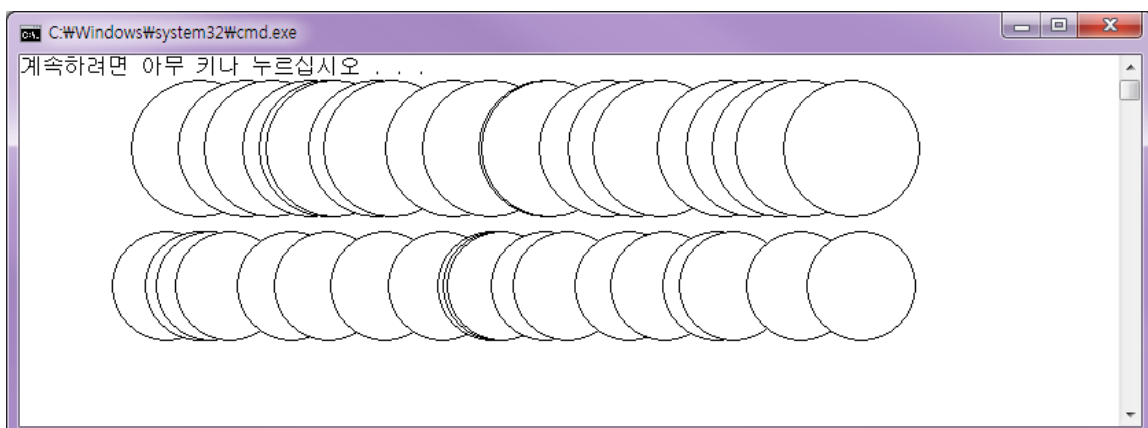
    a.print();
    return 0;
}
```



39

## Lab: 원들의 경주

- 지금까지 학습한 내용을 바탕으로 “원들의 경주” 게임을 다시 작성하여 보자. 두 개의 원을 생성한 후에 난수를 발생하여 원들을 움직인다.



40

## Solution

```
#include <iostream>
#include <windows.h>
using namespace std;

class Circle {
public:
    Circle(int xval, int yval, int r);
    void draw();
    void move();
private:
    int x, y, radius;
};

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

41

## Solution

```
void Circle::draw() {
    HDC hdc = GetWindowDC(GetForegroundWindow());
    Ellipse(hdc, x - radius, y - radius, x + radius, y + radius);
}

void Circle::move() {
    x += rand() % 50;
}

int main()
{
    Circle c1{ 100, 100, 50 };
    Circle c2{ 100, 200, 40 };
    for (int i = 0; i < 20; i++) {
        c1.move();
        c1.draw();
        c2.move();
        c2.draw();
        Sleep(1000);
    }
    return 0;
}
```

42

## this 포인터의 이해

```
class SoSimple
{
private:
    int num;
public:
    SoSimple(int n) : num(n)
    {
        cout<<"num="<<num<<" ";
        cout<<"address="<<this<<endl;
    }
    void ShowSimpleData()
    {
        cout<<num<<endl;
    }
    SoSimple * GetThisPointer()
    {
        return this;
    }
};
```

### 실행결과

```
num=100, address=0012FF60
0012FF60, 100
num=200, address=0012FF48
0012FF48, 100
```

```
int main(void)
{
    SoSimple sim1(100);
    SoSimple * ptr1=sim1.GetThisPointer();    // sim1 객체의 주소 값 저장
    cout<<ptr1<<" ";
    ptr1->ShowSimpleData();
    SoSimple sim2(200);
    SoSimple * ptr2=sim2.GetThisPointer();    // sim2 객체의 주소 값 저장
    cout<<ptr2<<" ";
    ptr2->ShowSimpleData();
    return 0;
}
```

this 포인터는 그 값이 결정되어 있지 않은 포인터이다. 왜냐하면 this 포인터는 this가 사용된 객체 자신의 주소값을 정보로 담고 있는 포인터이기 때문이다.

43

## this 포인터의 활용

```
class TwoNumber
{
private:
    int num1;
    int num2;
public:
    TwoNumber(int num1, int num2)
    {
        this->num1=num1;
        this->num2=num2;
    }
};
```



```
TwoNumber(int num1, int num2)
: num1(num1), num2(num2)
{
    // empty
}
```

this->num1은 멤버변수 num1을 의미한다. 객체의 주소 값으로 접근할 수 있는 대상은 멤버변수이지 지역변수가 아니기 때문이다!

44

## Self-reference의 반환

```
class SelfRef
{
private:
    int num;
public:
    SelfRef(int n) : num(n)
    {
        cout<<"객체생성"<<endl;
    }
    SelfRef& Adder(int n)
    {
        num+=n;
        return *this;
    }
    SelfRef& ShowTwoNumber()
    {
        cout<<num<<endl;
        return *this;
    }
};
```

```
int main(void)
{
    SelfRef obj(3);
    SelfRef &ref=obj.Adder(2);
    obj.ShowTwoNumber();
    ref.ShowTwoNumber();
    ref.Adder(1).ShowTwoNumber().Adder(2).ShowTwoNumber();
    return 0;
}
```

실행결과

객체생성

5  
5  
6  
8

45

## 5.5 객체와 함수

- 객체가 함수의 매개 변수로 전달되는 경우
- 객체의 참조자가 함수의 매개 변수로 전달되는 경우
- 함수가 객체를 반환하는 경우

46

## 객체가 함수의 매개 변수로 전달되는 경우

- 값을 전달한다.
- 어떤 피자 체인점에서 미디엄 크기의 피자를 주문하면 무조건 라지 피자로 변경해준다고 하자. 다음과 같이 프로그램을 작성하면 피자의 크기가 커질까?

47

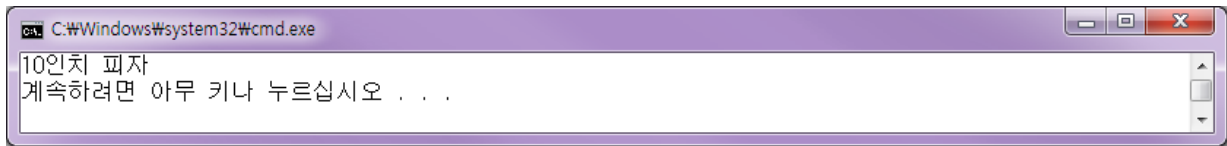
```
#include <iostream>
using namespace std;
class Pizza {
public:
    Pizza(int s) : size(s) {}
    int size;                // 단위: 인치
};
void makeDouble(Pizza p){
    p.size *= 2;
}

int main(){
    Pizza pizza(10);
    makeDouble(pizza);
    cout << pizza.size << "인치 피자" << endl;
    return 0;
}
```

48



## 실행결과



49

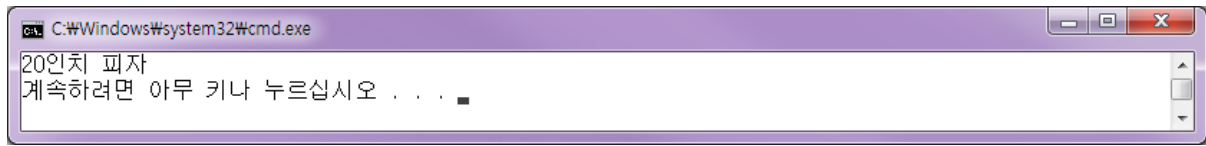
## 객체의 참조자가 함수의 매개 변수로 전달되는 경우

```
#include <iostream>
using namespace std;
class Pizza {
public:
    Pizza(int s) : size(s) {}
    int size;           // 단위: 인치
};
void makeDouble(Pizza& p) {
    p.size *= 2;
}
int main() {
    Pizza pizza(10);
    makeDouble(pizza);
    cout << pizza.size << "인치 피자" << endl;
    return 0;
}
```

50

## 실행결과

---



51

## 함수가 객체를 반환하는 경우

---

- 함수가 객체를 반환할 때도 객체의 내용이 복사될 뿐 원본이 전달되지 않는다.

52

## 예제

```
#include <iostream>
using namespace std;
class Pizza {
public:
    Pizza(int s) : size(s) {}
    int size;           // 단위: 인치
};
Pizza createPizza() {
    Pizza p(10);
    return p;
}
int main() {
    Pizza pizza = createPizza();
    cout << pizza.size << "인치 피자" << endl;
    return 0;
}
```

53

## 실행결과



54

# 상수형 메서드

멤버 변수에 읽기 접근은 가능하지만 쓰기는 허용되지 않는다.  
그러나 mutable로 선언한 멤버는 쓰기 허용된다. (예외)

```
class Ctest {
public: ...
    // 상수형 메서드로 선언 및 정의했다.
    int GetData() const
    {
        // 멤버 변수의 값을 읽을 수는 있지만 쓸 수는 없다.
        m_nData = 20;
        SetData(20);
        return m_nData;
    }
    int SetData(int nParam) { m_nData = nParam; }
private:
    int m_nData = 0;
};
```

55

# 정적 멤버

정적 멤버는 사실상 전역 변수나 함수로 생각하는 것이 좋다.

- 정적 멤버는 인스턴스 선언 없이 호출할 수 있다.  
: 예) CTest::PrintData();
- 정적 메서드는 this 포인터가 없다.
  - ▣ 정적 메소드에서 일반 멤버 변수를 사용할 수 없다.
- 정적 변수는 반드시 선언과 정의를 분리한다.
- 정적 변수는 '동시성'(예를 들어 멀티스레드 기반 프로그램)을 지원하지 못해 문제가 발생하므로 꼭 필요한 경우에만 제한적으로 사용한다.

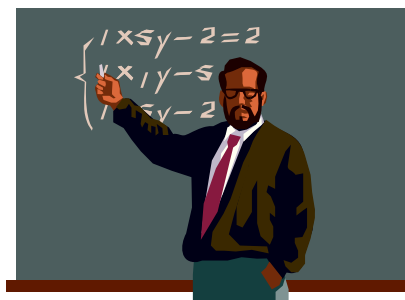
56

## 실습

- 비행기를 나타내는 **Airplane**이라는 클래스를 설계하라. **Airplane** 클래스는 이름(**name**), 승객수(**capacity**), 속도(**speed**)를 멤버 변수로 가지고 있다.
  - ▣ 멤버 변수를 정의하라. 모든 멤버 변수는 전용 멤버로 하라.
  - ▣ 모든 멤버 변수에 대한 접근자와 설정자 멤버 함수를 작성한다.
  - ▣ **Airplane** 클래스의 생성자 몇 개를 중복 정의하라. 생성자는 모든 데이터를 받을수도 있고 아니면 하나도 받지 않을 수 있다.
  - ▣ **Airplane** 객체의 현재 상태를 콘솔에 출력하는 **print()** 함수도 포함시켜라.
  - ▣ **main()**에서 **Airplane** 객체 2개를 생성하고 접근자와 설정자를 호출하라.

57

## Q & A



58