

윤성우의 열혈 C++ 프로그래밍



윤성우 저 열혈강의 C++ 프로그래밍 개정판

Chapter 04. 클래스의 완성

윤성우의 열혈 C++ 프로그래밍



Chapter 04-1. 정보은닉

윤성우 저 열혈강의 C++ 프로그래밍 개정판

정보은닉의 이해

```
class Point
```

정보은닉 실패

```
{
public:
```

```
    int x;    // x좌표의 범위는 0이상 100이하
    int y;    // y좌표의 범위는 0이상 100이하
};
```

```
class Rectangle
```

정보은닉 실패

```
{
public:
```

```
    Point upLeft;
    Point lowRight;
```

```
public:
```

```
    void ShowRecInfo()
```

```
{
```

```
    cout<<"좌 상단: "<<'['<<upLeft.x<<" ";
    cout<<upLeft.y<<'>]<<endl;
    cout<<"우 하단: "<<'['<<lowRight.x<<" ";
    cout<<lowRight.y<<'>]<<endl<<endl;
```

```
}
```

```
};
```

```
int main(void)
```

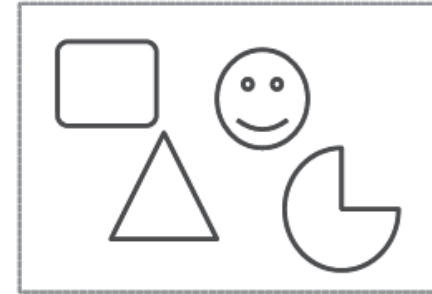
```
{
```

```
    Point pos1={-2, 4};
    Point pos2={5, 9};
    Rectangle rec={pos2, pos1};
    rec.ShowRecInfo();
    return 0;
```

```
}
```

좌 상단 [0, 0]

그림판



우 하단 [100, 100]

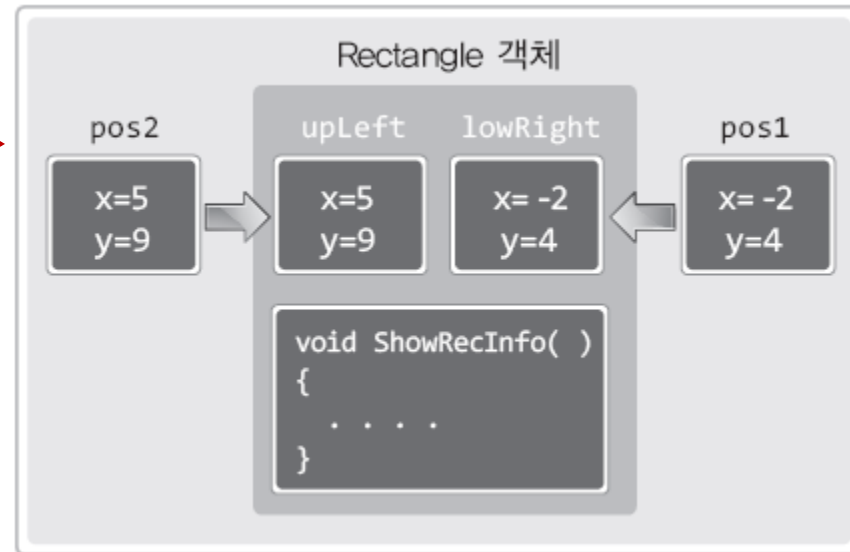
예제에서 보이듯이 멤버변수의 외부접근을 허용하면, 잘못된 값이 저장되는 문제가 발생할 수 있다. 따라서 멤버변수의 외부접근을 막게 되는데, 이를 가리켜 **정보인닉**이라 한다.

Point의 멤버변수에는 0~100 이외의 값이 들어오는 것을 막는 장치가 없고,

Rectangle의 멤버변수에는 좌우 정보가 뒤바뀌어 저장되는 것을 막을 장치가 없다.

Rectangle 객체의 이해

```
int main(void)
{
    Point pos1={-2, 4};
    Point pos2={5, 9};
    Rectangle rec={pos2, pos1};
    rec.ShowRecInfo();
    return 0;
}
```



클래스의 객체도 다른 객체의 멤버가 될 수 있다.

Point 클래스의 정보은닉 결과

```
class Point
{
    정보은닉!
private:
    int x;
    int y;
public:
    bool InitMembers(int xpos, int ypos);
    int GetX() const;
    int GetY() const;
    bool SetX(int xpos);
    bool SetY(int ypos);
};
```

정보은닉으로 인해서 추가되는 액세스 함수들!

클래스의 멤버변수를 **private**으로 선언하고, 해당 변수에 접근하는 함수를 별도로 정의해서, 안전한 형태로 멤버변수의 접근을 유도하는 것이 바로 '정보은닉'이며, 이는 좋은 클래스가 되기 위한 기본조건이 된다!

```
bool Point::SetX(int xpos)
{
    if(0>xpos || xpos>100)
    {
        cout<<"벗어난 범위의 값 전달"<<endl;
        return false;
    }
    x=xpos;
    return true;
}
```

벗어난 범위의 값 저장을
원천적으로 막고 있다!

함수만 한번 잘 정의되면 잘못된 접근은 원천적으로 차단된다! 하지만 정보은닉을 하지 않는다면, 접근할 때마다 주의해야 한다!

Rectangle 클래스의 정보는닉 결과

```
class Rectangle
{
private:
    Point upLeft;
    Point lowRight;
public:
    bool InitMembers(const Point &ul, const Point &lr);
    void ShowRecInfo() const;
};
```

```
bool Rectangle::InitMembers(const Point &ul, const Point &lr)
{
    if(ul.GetX() > lr.GetX() || ul.GetY() > lr.GetY())
    {
        cout<<"잘못된 위치정보 전달"<<endl;
        return false;
    }
    upLeft=ul;
    lowRight=lr;
    return true;
}
```

좌 상단과 우 하단이 바뀌는
것을 근본적으로 차단!



const 함수

멤버함수의 const 선언

```
int GetX() const;
int GetY() const;
void ShowRecInfo() const;
```

const 함수 내에서는 동일 클래스에 선언된
멤버변수의 값을 변경하지 못한다!

```
int GetNum()           이 둘은 멤버함수입니다.
{
    return num;
}
void ShowNum() const
{
    cout<<GetNum()<<endl;    // 컴파일 에러 발생
}
```

const 함수는 const가 아닌 함수를 호출하지 못한다!
간접적인 멤버의 변경 가능성까지 완전히 차단!

```
void InitNum(const EasyClass &easy)
{
    num=easy.GetNum();    // 컴파일 에러 발생
}    GetNum()이 const 선언되지 않았다고 가정!
```

const로 상수화 된 객체를 대상으로는 const 멤버함
수만 호출이 가능하다!



윤성우의 열혈 C++ 프로그래밍



Chapter 04-2. 캡슐화

윤성우 저 열혈강의 C++ 프로그래밍 개정판

콘택600과 캡슐화

```
class SinivelCap    // 콧물 치료용 캡슐
{
public:
    void Take() const {cout<<"콧물이 싹~ 납니다."<<endl;}
};

class SneezeCap     // 재채기 치료용 캡슐
{
public:
    void Take() const {cout<<"재채기가 멎습니다."<<endl;}
};

class SnuffleCap    // 코막힘 치료용 캡슐
{
public:
    void Take() const {cout<<"코가 뻥 뚫립니다."<<endl;}
};
```



약의 복용순서가 정해져 있다고 한다면,
캡슐화가 매우 필요한 상황이 된다!

콘택 600을 표현한 클래스들...

코감기는 항상 콧물, 재채기, 코막힘을 동반한다고 가정하면 캡슐화 실패!

캡슐화란! 관련 있는 모든 것을 하나의 클래스 안에 묶어 두는 것!

캡슐화 된 콘택600

캡슐화의 이점

A 클래스가 캡슐화가 잘 되어있다면, A 클래스가 변경되더라도, A와 연관된 B, C, D 클래스는 변경되지 않거나 변경되더라도 그 범위가 매우 최소화된다.

```
class CONTAC600
```

```
{
```

```
private:
```

```
SinivelCap sin;
```

```
SneezeCap sne;
```

```
SnuffleCap snu;
```

코감기와 관련 있는 것을 하나의 클래스로 묶었다.

```
public:
```

```
void Take() const
```

```
{
```

```
    sin.Take();
```

```
    sne.Take();
```

```
    snu.Take();
```

```
}
```

묶음으로 인해서 복잡한 복용의 방법을 약 복용자에게 노출시킬 필요가 없게 되었다.

```
};
```

```
class ColdPatient
```

```
{
```

```
public:
```

```
    void TakeCONTAC600(const CONTAC600 &cap) const { cap.Take(); }
```

```
};
```

아무리 CONTAC600 클래스가 바뀌어도, 약의 복용순서가 바뀌더라도, 이와 관련있는 ColdPatient 함수는 바뀌지 않는다.

윤성우의 열혈 C++ 프로그래밍



Chapter 04-3. 생성자와 소멸자

윤성우 저 열혈강의 C++ 프로그래밍 개정판

생성자의 이해

```
class SimpleClass
{
private:
    int num;
public:
    SimpleClass(int n) // 생성자(constructor)
    {
        num=n;
    }
    int GetNum() const
    {
        return num;
    }
};
```

클래스의 이름과 동일한 이름의 함수이면서 반환형이 선언되지 않았고 실제로 반환하지 않는 함수를 가리켜 생성자라 한다!

생성자는 객체 생성시 딱 한번 호출된다. 따라서 멤버변수의 초기화에 사용할 수 있다.

생성자도 함수의 일종이므로, 오버로딩이 가능하고 디폴트 값 설정이 가능하다.

```
SimpleClass sc(20); // 생성자에 20을 전달
SimpleClass * ptr = new SimpleClass(30); // 생성자에 30을 전달
```



생성자의 함수적 특성

생성자도 함수의 일종이므로 오버로딩이 가능하다.

```
SimpleClass()
{
    num1=0;
    num2=0;
}
SimpleClass(int n)
{
    num1=n;
    num2=0;
}
SimpleClass(int n1, int n2)
{
    num1=n1;
    num2=n2;
}
```

```
SimpleClass sc1();           (×)
SimpleClass sc1;             (○)
SimpleClass * ptr1=new SimpleClass;  (○)
SimpleClass * ptr1=new SimpleClass(); (○)
```

```
SimpleClass sc2(100);
SimpleClass * ptr2=new SimpleClass(100);
```

```
SimpleClass sc3(100, 200);
SimpleClass * ptr3=new SimpleClass(100, 200);
```

```
SimpleClass(int n1=0, int n2=0)
{
    num1=n1;
    num2=n2;
}
```

생성자도 함수의 디폴트 값 설정이 가능하다.

Point, Rectangle 클래스에 생성자 적용

```
class Point
{
private:
    int x;
    int y;
public:
    Point(const int &xpos, const int &ypos);    // 생성자
    int GetX() const;
    int GetY() const;
    bool SetX(int xpos);
    bool SetY(int ypos);
};
```

```
Point::Point(const int &xpos, const int &ypos)
{
    x=xpos;
    y=ypos;
}
```

```
class Rectangle
{
private:
    Point upLeft;
    Point lowRight;
public:
    Rectangle(const int &x1, const int &y1, const int &x2, const int &y2);
    void ShowRecInfo() const;
};
```

이 위치에서 호출할 생성자를 명시할 수 없다!
이에 대한 해결책으로 이니셜라이저 제시!



멤버 이니셜라이저 기반의 멤버 초기화

멤버 이니셜라이저는 함수의 선언 부가 아닌, 정의 부에 명시한다.

```
Rectangle::Rectangle(const int &x1, const int &y1, const int &x2, const int &y2)
    :upLeft(x1, y1), lowRight(x2, y2)
{
    // empty
}
```

“객체 upLeft의 생성과정에서 x1과 y1을 인자로 전달받는 생성자를 호출하라.”

“객체 lowRight의 생성과정에서 x2와 y2을 인자로 전달받는 생성자를 호출하라.”

- 1단계: 메모리 공간의 할당
- 2단계: 이니셜라이저를 이용한 멤버변수(객체)의 초기화
- 3단계: 생성자의 몸체부분 실행

이니셜라이저의 실행을 포함한 객체 생성의 과정



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 초기화된다고 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)
    {
    }
}
```


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!



디폴트 생성자

```
class AAA
{
private:
    int num;
public:
    int GetNum { return num; }
};
```



```
class AAA
{
private:
    int num;
public:
    AAA(){ }    // 디폴트 생성자
    int GetNum { return num; }
};
```

생성자를 정의하지 않으면 인자를 받지 않고, 하는 일이 없는 디폴트 생성자라는 것이 컴파일러에 의해서 추가된다.

따라서 모든 객체는 무조건 생성자의 호출 과정을 거쳐서 완성된다.

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.

private 생성자

```
class AAA
{
private:
    int num;
public:
    AAA() : num(0) {}
    AAA& CreateInitObj(int n) const
    {
        AAA * ptr=new AAA(n);
        return *ptr;
    }
    void ShowNum() const { cout<<num<<endl; }
private:
    AAA(int n) : num(n) {}
};
```

그러나 이렇듯 클래스 내부에서는 **private** 생성자의 호출이 가능하다.

생성자가 **private**이므로 클래스 외부에서는 이 생성자의 호출을 통해서 객체 생성이 불가능하다.

AAA 클래스의 멤버함수 내에서도 AAA 클래스의 객체 생성이 가능하다!
생성자가 **private**이라는 것은 외부에서의 객체 생성을 허용하지 않겠다는 뜻이다!



소멸자의 이해

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



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

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

AAA 클래스의 소멸자! 객체 소멸 시 자동으로 호출된다.

생성자와 마찬가지로 소멸자도 정의하지 않으면 디폴트 소멸자가 삽입된다.

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.

윤성우의 열혈 C++ 프로그래밍



Chapter 04-4. 클래스와 배열 그리고
this 포인터

윤성우 저 열혈강의 C++ 프로그래밍 개정판

객체 배열과 객체 포인터 배열

```
Person arr[3];  
Person * parr=new Person[3];
```

객체 배열! 객체로 이뤄진 배열, 따라서 배열
생성시 객체가 함께 생성된다.
이 경우 호출되는 생성자는 void 생성자

```
Person * arr[3];  
arr[0]=new Person(name, age );  
arr[1]=new Person(name, age );  
arr[2]=new Person(name, age );
```

객체 포인터 배열! 객체를 저장할 수 있는 포
인터 변수로 이뤄진 배열! 따라서 별도의 객체
생성 과정을 거쳐야 한다.

객체 관련 배열을 선언할 때에는 객체 배열을 선언할지, 아니면 객체 포인터 배열
을 선언할지를 먼저 결정해야 한다.



this 포인터의 이해

실행결과

```
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;
    }
};
```

```
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.

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 **num1**을 의미한다. 객체의 주소 값으로 접근할 수 있는 대상은 멤버변수이지 지역변수가 아니기 때문이다!

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;
    }
};
```

실행결과

객체생성

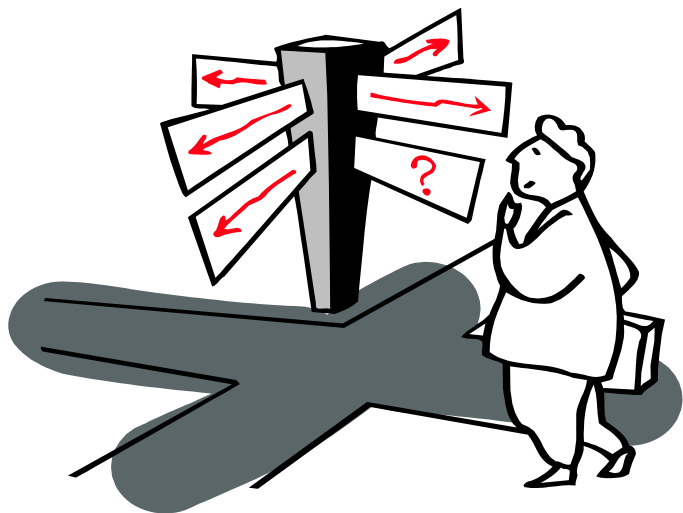
5

5

6

8

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



Chapter 이가 끝났습니다. 질문 있으신지요?