

3장. 클래스의 완성

박 종 혁 교수
(서울과기대 컴퓨터공학과)

Tel: 970-6702
Email: jhpark1@seoultech.ac.kr

목차

1. 정보은닉과 캡슐화
2. 생성자와 소멸자
3. 객체와 배열 / 포인터, 그리고 this 포인터

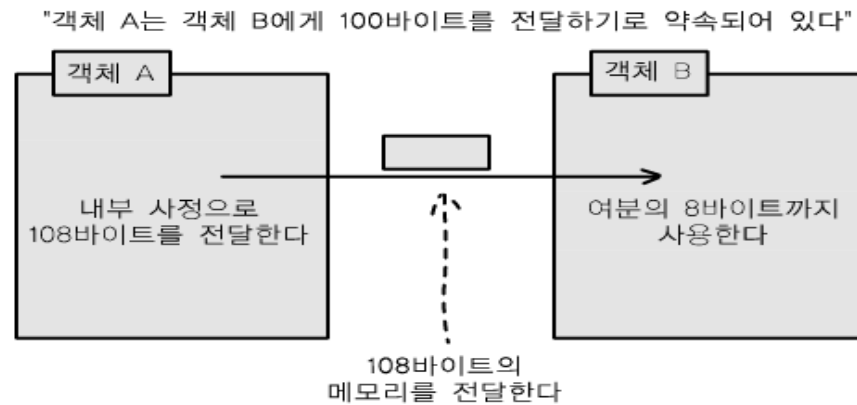
1. 정보은닉과 캡슐화

정보은닉과 캡슐화

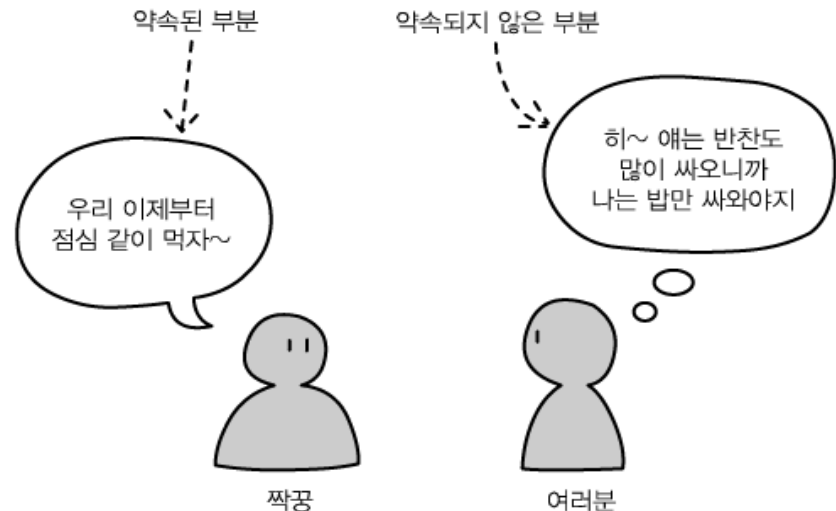
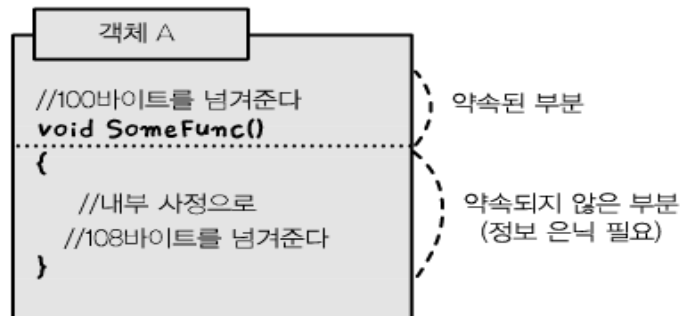
- 객체지향 프로그래밍의 객체 정보은닉, 캡슐화 지원
 - 오류 검출 등 유지보수 용이
- 정보은닉(자료은폐, information hiding)
 - 객체 내부의 자료를 객체 외부로부터 은폐하여 외부에서 접근금지
 - 객체 내부 자료를 private으로 선언하여 실현
 - 필요성 : 프로그램의 안정적 구현
- 캡슐화(Encapsulation)
 - 관련된 자료와 함수를 하나의 단위로 그룹화

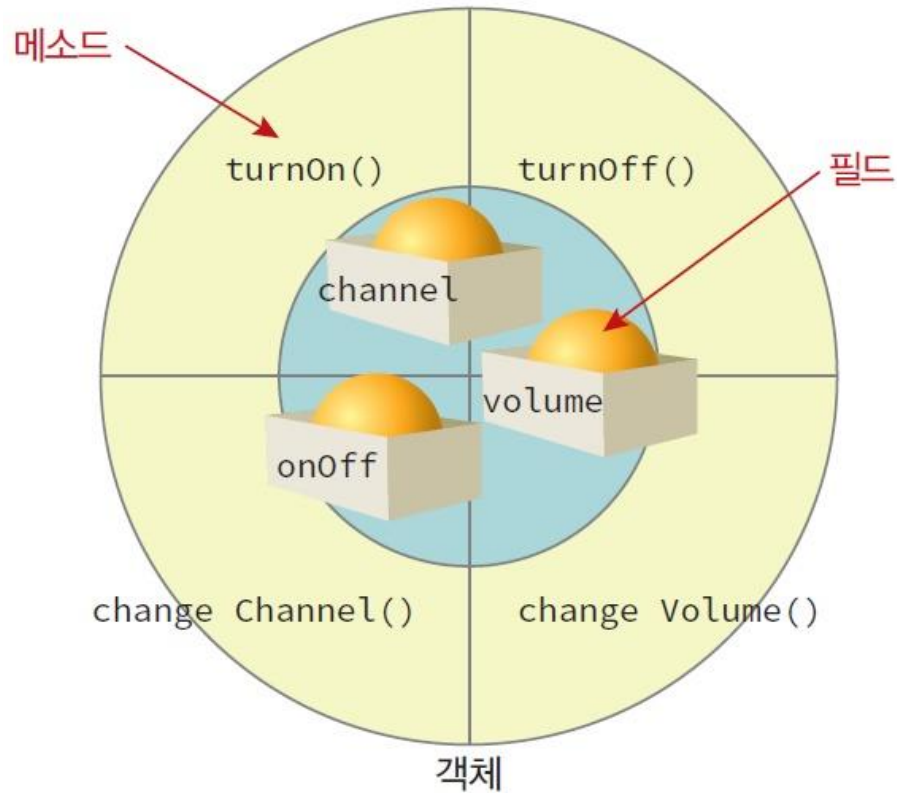
정보은닉(Data Hiding)

- 정보은닉-객체간에 약속되지 않은 부분을 숨기는 것
 - 예) 약속되지 않은 여분의 8바이트를 사용하는 것은 정보은닉에 어긋나는 일이다



- 약속된 부분과 약속되지 않은 부분





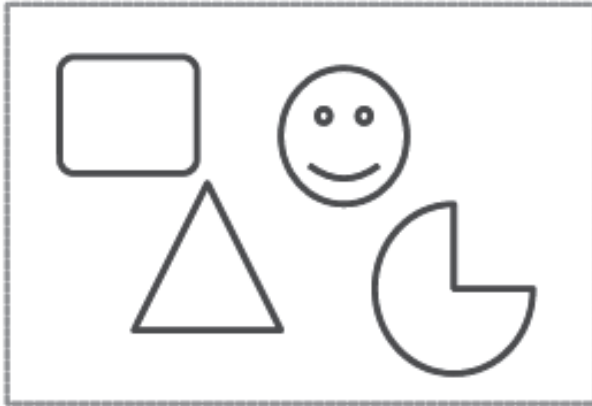
보통은 데이터들은 공개되지
않고 몇 개의 메소드 만이
외부로 공개됩니다.



정보은닉의 이해

좌 상단 [0, 0]

그림판



우 하단 [100, 100]

멤버변수의 외부접근을 허용하면, 잘못된 값이 저장되는 문제가 발생
→ 멤버변수의 외부접근을 차단: **정보은닉**

```
class Point
{
public:
    int x;    // x좌표의 범위는 0이상 100이하
    int y;    // y좌표의 범위는 0이상 100이하
};
```

정보은닉 실패

```
class Rectangle
{
public:
    Point upLeft;
    Point lowRight;
public:
    void ShowRecInfo()
    {
```

정보은닉 실패

```
        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;
}
```

Point의 멤버변수에는 0~100 이외의 값이 들어오는 것을 막는 장치가 없고,
Rectangle의 멤버변수에는 좌우 정보가 뒤바뀌어 저장되는 것을 막을 장치가 없다

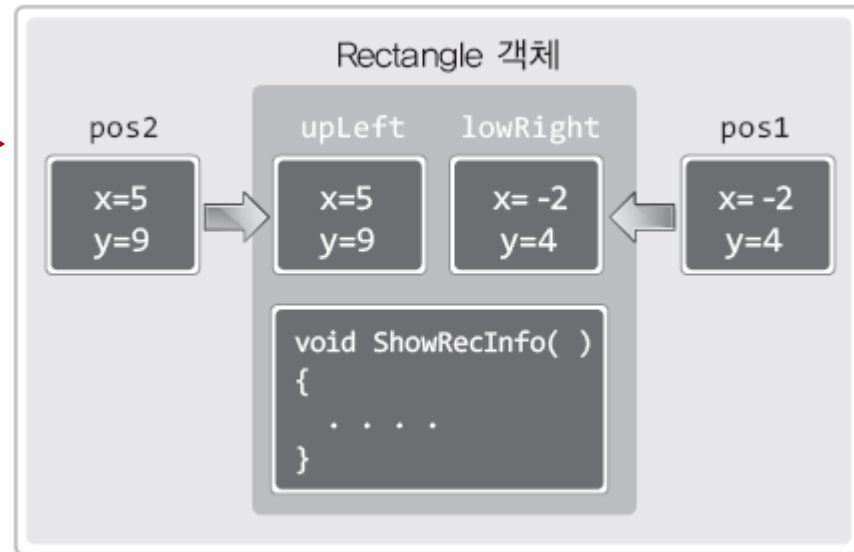
예제) 정보은닉 실패- Rectangle Fault

```
#include<iostream>
using namespace std;
class Point
{
public:
    int x; // x좌표의 범위는 0이상 100이하
    int y; // y좌표의 범위는 0이상 100이하
};
class Rectangle
{
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;
}
```

Rectangle 객체의 이해

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

객체 생성, 초기화



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 정보은닉 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!

실행결과

벗어난 범위의 값 전달
초기화 실패
잘못된 위치정보 전달
직사각형 초기화 실패
좌 상단 : [2, 4]
우 하단 : [5, 9]

참고) 헤더 파일의 중복 include 문제

- 헤더 파일을 중복 include 할 때 생기는 문제

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

#include "Circle.h"
#include "Circle.h" // 컴파일 오류 발생
#include "Circle.h"

int main() {
    .....
}
```

circle.h(4): error C2011: 'Circle' : 'class' 형식 재정의

헤더 파일의 중복 include 문제를 조건 컴파일로 해결

조건 컴파일 문의 상수(CIRCLE_H)는 다른 조건 컴파일 상수와 충돌을 피하기 위해 클래스의 이름으로 하는 것이 좋음.

조건 컴파일 문.
Circle.h를 여러 번
include해도 문제
없게 하기 위함

```
#ifndef CIRCLE_H
#define CIRCLE_H
```

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

```
#endif
```

Circle.h

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

```
#include "Circle.h"
#include "Circle.h"
#include "Circle.h"
```

컴파일 오류 없음

```
int main() {
    .....
}
```

main.cpp

Point.h

```
#ifndef __POINT_H_  
#define __POINT_H_  
  
class Point  
{  
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);  
};  
  
#endif
```

Rectangle.h

```
#ifndef __RECTANGLE_H_  
#define __RECTANGLE_H_  
  
#include "Point.h"  
  
class Rectangle  
{  
private:  
    Point upLeft;  
    Point lowRight;  
  
public:  
    bool InitMembers(const Point &ul, const Point &lr);  
    void ShowRecInfo() const;  
};  
  
#endif
```

Rectangle.cpp

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

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;
}

void Rectangle::ShowRecInfo() const
{
    cout<<"좌 상단: "<< '['<<upLeft.GetX()<< ", ";
    cout<<upLeft.GetY()<< ']'<<endl;
    cout<<"우 하단: "<< '['<<lowRight.GetX()<< ", ";
    cout<<lowRight.GetY()<< ']'<<endl<<endl;
}
```

RectangleFaultFind.cpp

```
#include<iostream>
#include "Point.h"
#include "Rectangle.h"
using namespace std;

int main(void)
{
    Point pos1;
    if(!pos1.InitMembers(-2, 4))
        cout<<"초기화 실패"<<endl;
    if(!pos1.InitMembers(2, 4))
        cout<<"초기화 실패"<<endl;

    Point pos2;
    if(!pos2.InitMembers(5, 9))
        cout<<"초기화 실패"<<endl;

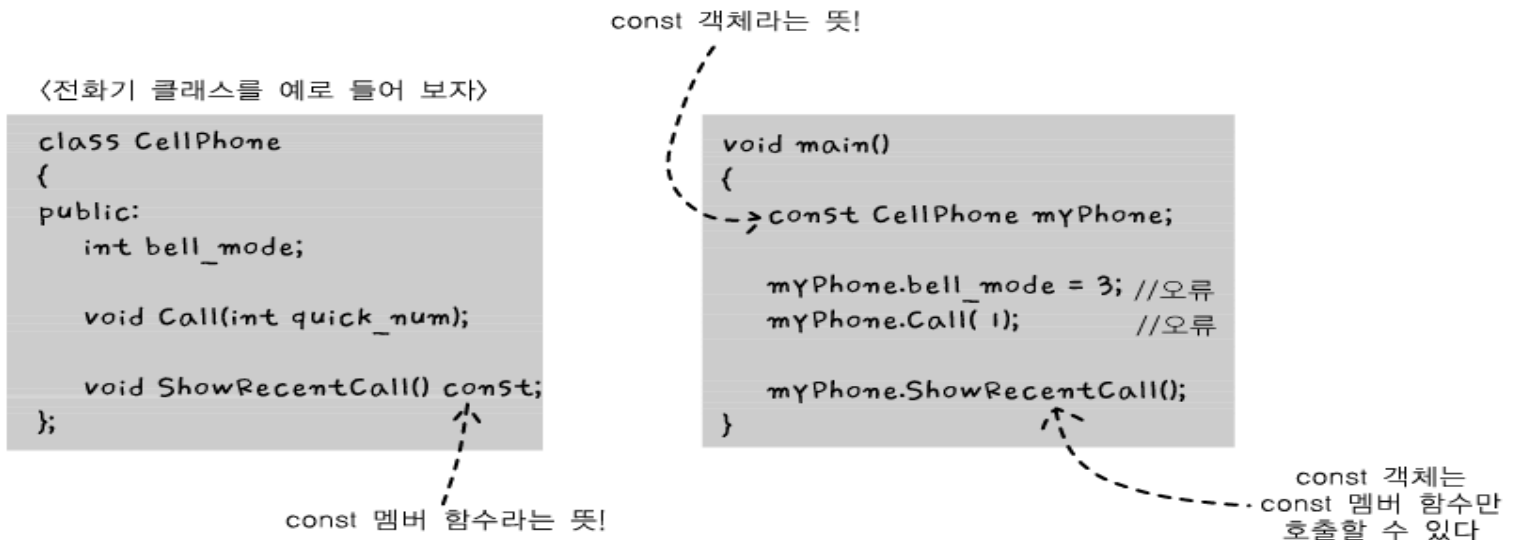
    Rectangle rec;
    if(rec.InitMembers(pos2, pos1))
        cout<<"직사각형 초기화 실패"<<endl;

    if(rec.InitMembers(pos1, pos2))
        cout<<"직사각형 초기화 실패"<<endl;

    rec.ShowRecInfo();
    return 0;
}
```

Const 함수

- 다음의 조건을 만족 하는 함수만 Const 함수로 만들 수 있다.
 - 멤버 변수의 값을 변경하지 않는 멤버 함수
- 멤버 함수를 Const 함수로 만들면 좋은 점
 - 다른 개발자가 “아, 이 함수는 멤버 변수의 값을 변경하지 않는구나”라고 생각하게 만든다.
 - 실수로 멤버 변수의 값을 바꾸려고 하면, 컴파일러가 오류 메시지를 통해서 알려준다.
 - 객체가 Const 속성을 가진 경우에는 Const 함수만 호출할 수 있다.



Const 함수의 사용

- Const 객체를 통해서 멤버 함수를 호출하는 예

```
class Point
{
public:
    // 멤버 함수
    void Print() const;

    void SetX(int value);
    void SetY(int value);
    int GetX() const {return x;}
    int GetY()    {return y;}
    // 일부러 const 함수로 만들지 않음

private:
    // 멤버 변수
    int x, y;
};
```

```
void Area( const Point& pt)
{
    // (o,o)과 pt 가 이루는 사각형의 면적을 구한다.
    int area = pt.GetX() * pt.GetY();
                                   // Error

    // 결과 출력
    cout << "(o, o)과 이 점이 이루는 사각형의 면적 = "
    << area << "\n";
}
```

- 매개변수 pt가 Const 객체이지만 CPoint::GetX()는 Const 함수이기 때문에 호출될 수 있다.
- CPoint::GetY() 는 Const 함수가 아니기 때문에 컴파일 오류를 발생!!

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()  
{  
    return num;  
}  
void ShowNum() const  
{  
    cout<<GetNum()<<endl;    // 컴파일 에러 발생  
}
```

이 둘은 멤버함수입니다.

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

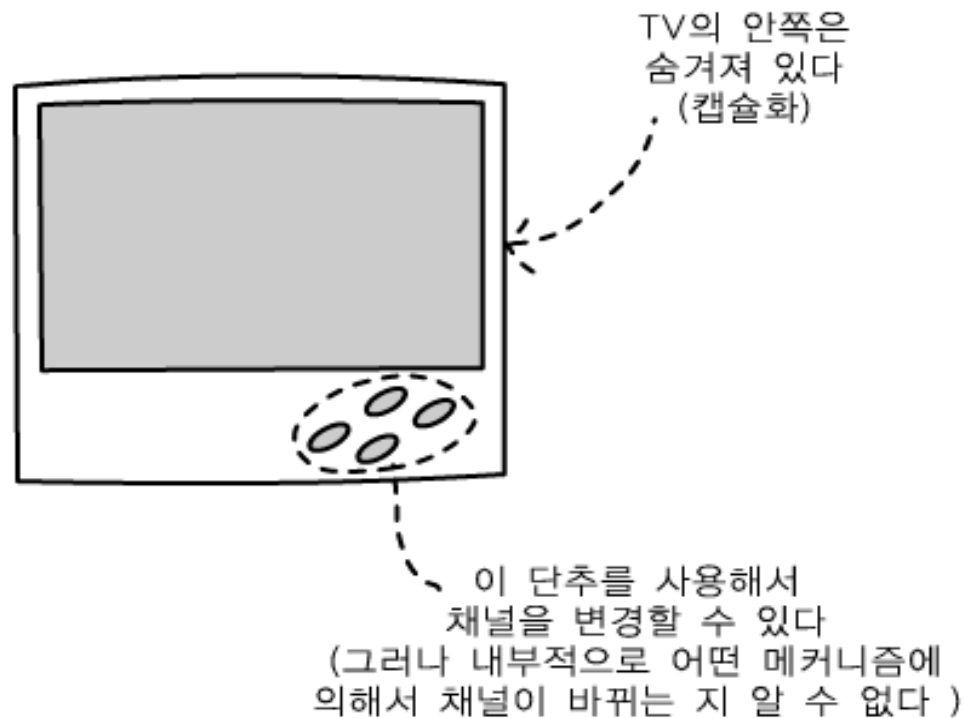
```
void InitNum(const EasyClass &easy)  
{  
    num=easy.GetNum();    // 컴파일 에러 발생  
}    GetNum이 const 선언되지 않았다고 가정!
```

const로 상수화 된 객체를 대상으로는 const 멤버함
수만 호출이 가능하다!

캡슐화 (Encapsulation)

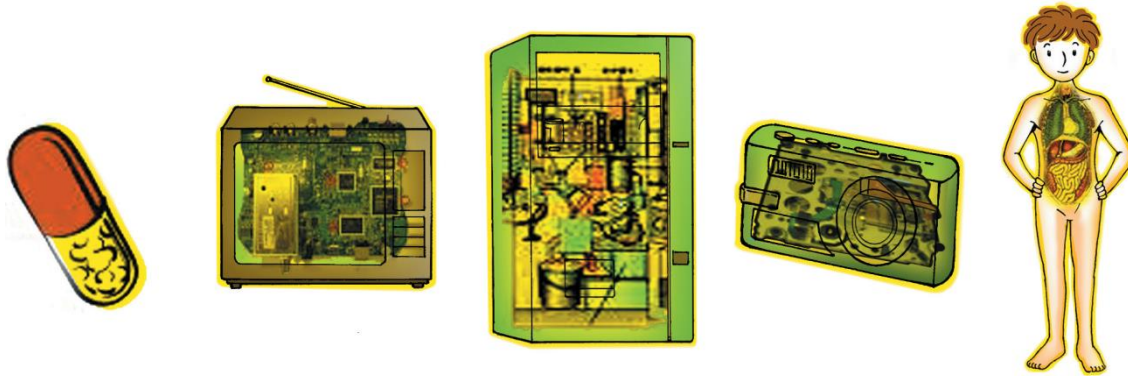
캡슐화(Encapsulation)

- 캡슐화-약속되지 않은 부분은 감싸서 숨겨버리는 것
- 캡슐화를 통해서 정보는닉을 달성할 수 있음

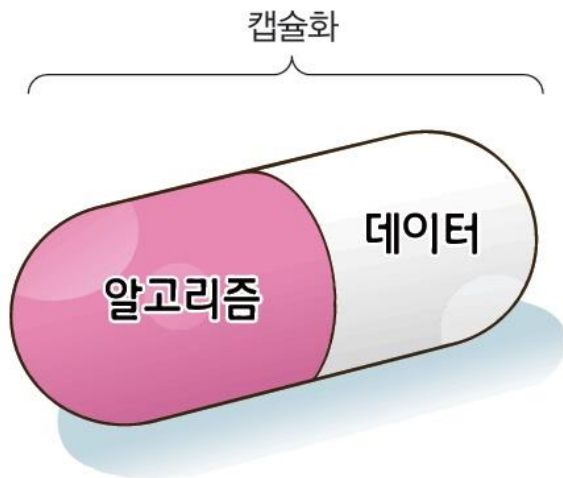


- 캡슐화(encapsulation)

- ▣ 객체의 본질적인 특성
- ▣ 객체를 캡슐로 싸서 그 내부를 보호하고 볼 수 없게 함
 - 캡슐에 든 약은 어떤 색인지 어떤 성분인지 보이지 않고, 외부로부터 안전
- ▣ 캡슐화 사례



- ▣ 캡슐화의 목적
 - 객체 내 데이터에 대한 보안, 보호, 외부 접근 제한



캡슐화는 데이터와 알고리즘을
하나로 묶는 것입니다.



캡슐화 되어 있지 않은 데이터
와 코드는 사용하기 어렵다.



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!

Encaps1.cpp

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

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

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

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

```
class ColdPatient
{
public:
    void TakeSinivelCap(const SinivelCap &cap) const
    {cap.Take();}
    void TakeSneezeCap(const SneezeCap &cap) const
    {cap.Take();}
    void TakeSnuffleCap(const SnuffleCap &cap) const
    {cap.Take();}
};

int main(void)
{
    SinivelCap scap;
    SneezeCap zcap;
    SnuffleCap ncap;

    ColdPatient sufferer;
    sufferer.TakeSinivelCap(scap);
    sufferer.TakeSneezeCap(zcap);
    sufferer.TakeSnuffleCap(ncap);
    return 0;
}
```

캡슐화 된 콘택600

```
class CONTAC600
```

```
{
```

```
private:
```

```
    SinivelCap sin;
```

```
    SneezeCap sne;
```

```
    SnuffleCap snu;
```

```
public:
```

```
    void Take() const
```

```
    {
```

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

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

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

```
    }
```

```
};
```

코감기와 관련 있는 것을 하나의 클래스로 묶었다

묶음으로 인해서 복잡한 복용의 방법을 약 복용자에게 노출시킬 필요가 없게 되었다

캡슐화의 이점

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

```
class ColdPatient
```

```
{
```

```
public:
```

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

```
};
```

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

Encaps2.cpp

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

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

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

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

class CONTAC600
{
private:
    SinivelCap sin;
    SneezeCap sne;
    SnuffleCap snu;
```

```
public:
    void Take() const
    {
        sin.Take();
        sne.Take();
        snu.Take();
    }
};

class ColdPatient
{
public:
    void TakeCONTAC600(const CONTAC600 &cap) const
    { cap.Take(); }
};

int main(void)
{
    CONTAC600 cap;
    ColdPatient sufferer;
    sufferer.TakeCONTAC600(cap);
    return 0;
}
```

정보 은닉과 캡슐화

- 정보 은닉은 쉬움, 캡슐화 어려움
 - 캡슐화는 일관되게 적용할 수 있는 단순한 개념이 아님
 - 구현하는 프로그램의 성격과 특성에 따라서 적용하는 범위가 상이함
 - 정답이란 것이 딱히 없는 개념

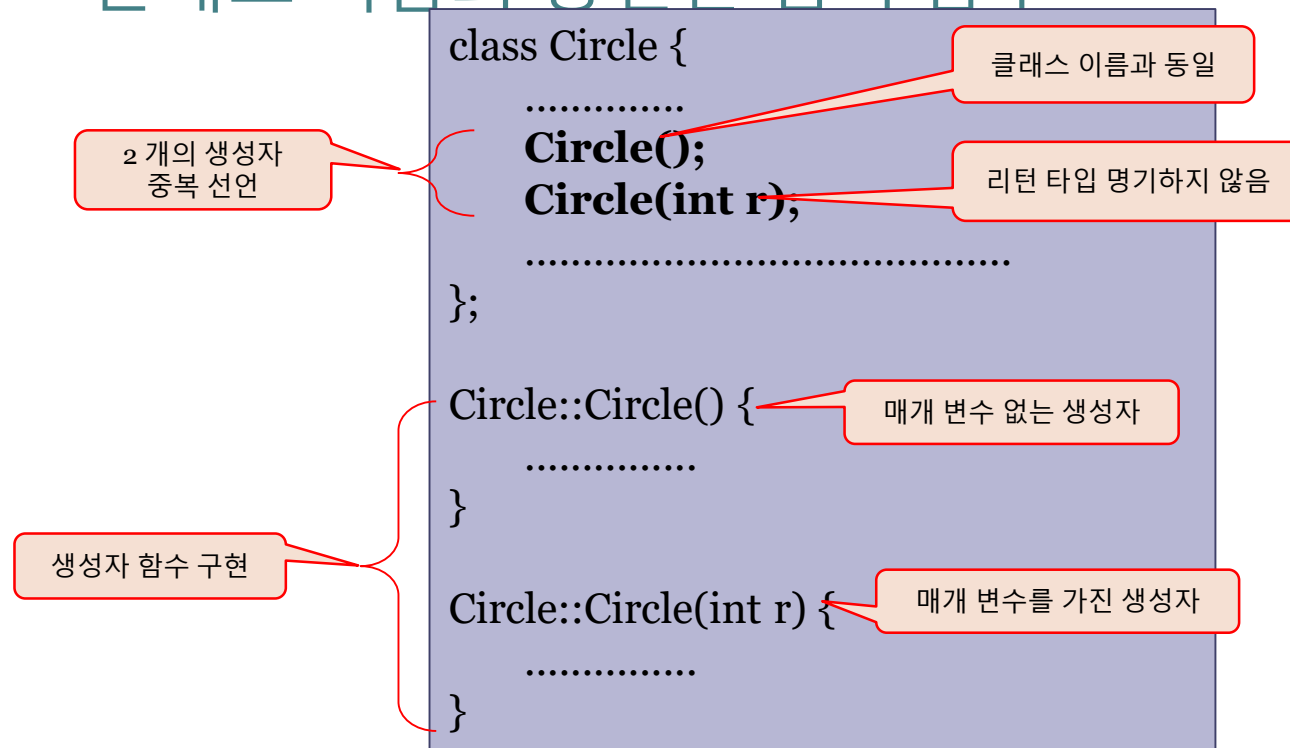
2. 생성자와 소멸자 (Constructor & Destructor)

생성자(constructor)

- 생성자의 필요성
 - 객체를 생성과 동시에 초기화하기 위해
 - 객체는 생성과 동시에 초기화되는 것이 좋은 구조
- 생성자란?
 - 객체 생성시 반드시 한번 호출되는 함수
 - 클래스와 같은 이름의 함수
 - 리턴형이 없으며 리턴하지도 않음

생성자

- 생성자(constructor)
 - 객체가 생성되는 시점에서 자동으로 호출되는 멤버 함수
 - 클래스 이름과 동일한 멤버 함수



예) 2 개의 생성자를 가진 Circle 클래스

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

```
class Circle {
public:
    int radius;
    Circle(); // 매개 변수 없는 생성자
    Circle(int r); // 매개 변수 있는 생성자
    double getArea();
};
```

```
Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

```
Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}
```

```
double Circle::getArea() {
    return 3.14*radius*radius;
}
```

Circle(); 자동 호출

Circle(30); 자동 호출

```
int main() {
```

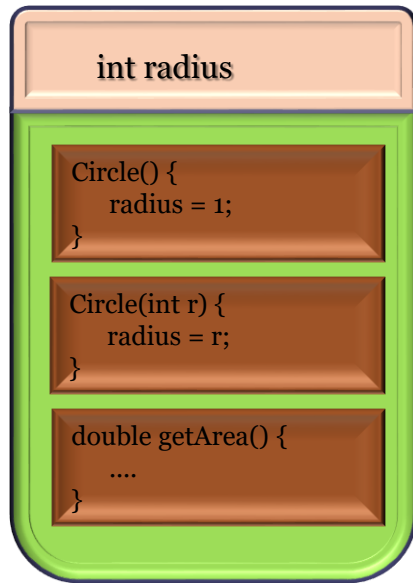
```
    Circle donut; // 매개 변수 없는 생성자 호출
    double area = donut.getArea();
    cout << "donut 면적은 " << area << endl;
```

```
    Circle pizza(30); // 매개 변수 있는 생성자 호출
    area = pizza.getArea();
    cout << "pizza 면적은 " << area << endl;
}
```

반지름 1 원 생성
donut 면적은 3.14
반지름 30 원 생성
pizza 면적은 2826

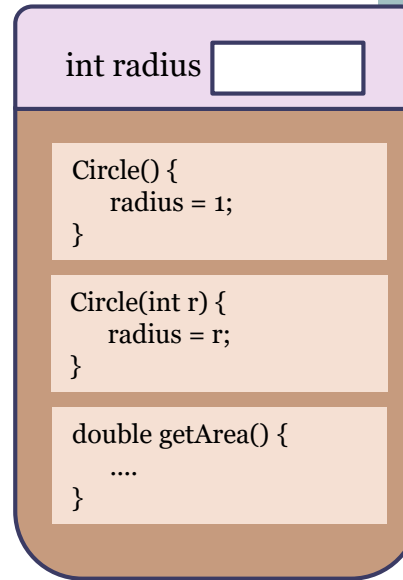
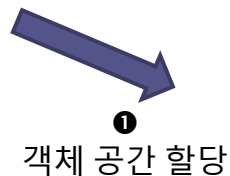
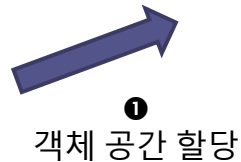
객체 생성 및 생성자 실행 과정

Circle donut;

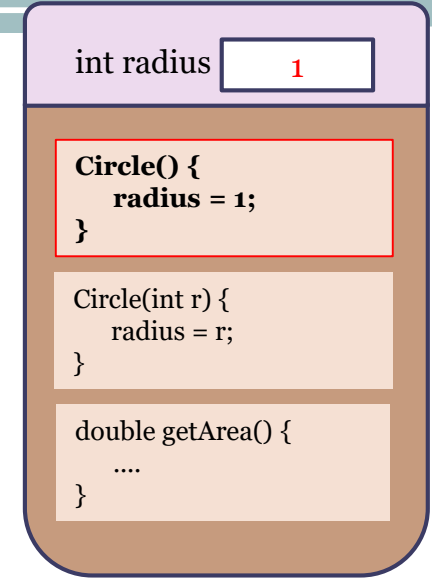
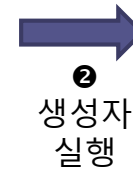


Circle 클래스

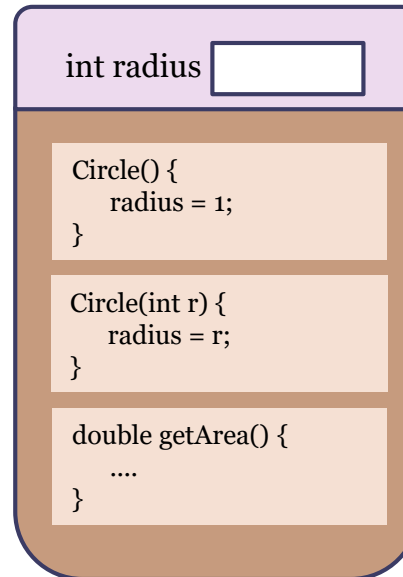
Circle pizza(30);



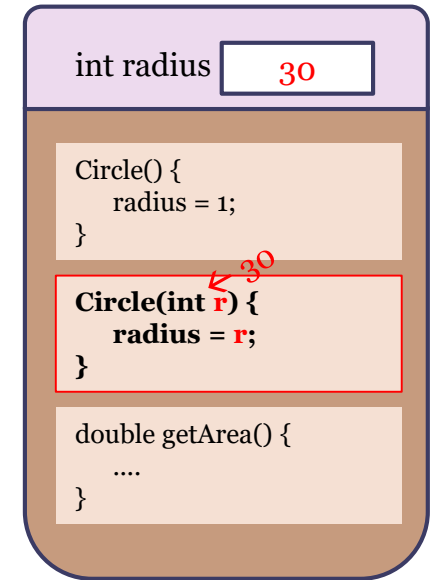
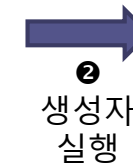
donut 객체



donut 객체



pizza 객체



pizza 객체

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!

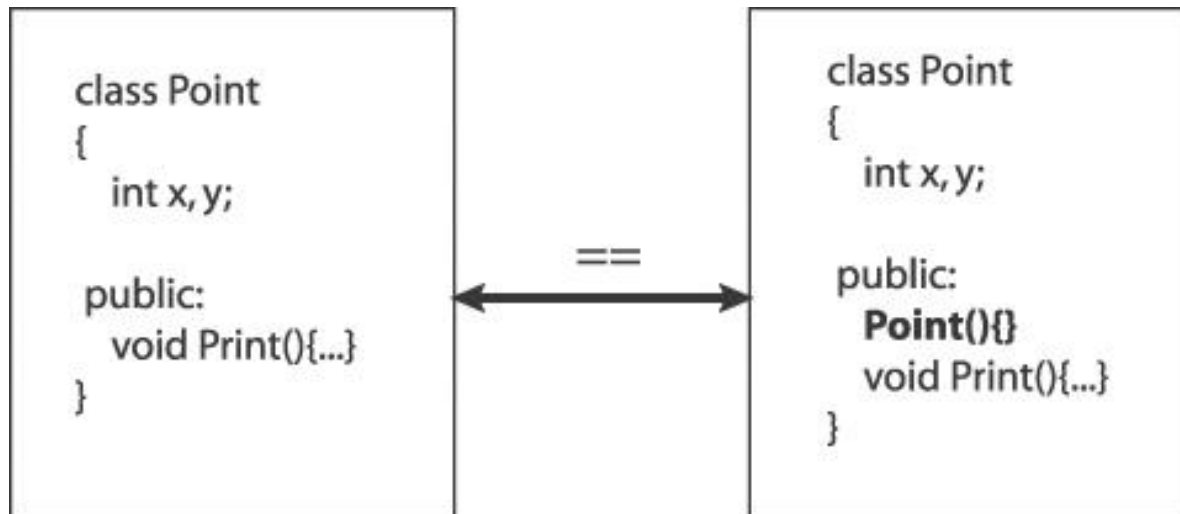
생성자는 객체 생성시 딱 한번 호출된다. 따라서 멤버변수의 초기화에 사용할 수 있다

생성자도 함수의 일종이므로, 오버로딩이 가능하고 디폴트 값 설정이 가능하다

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

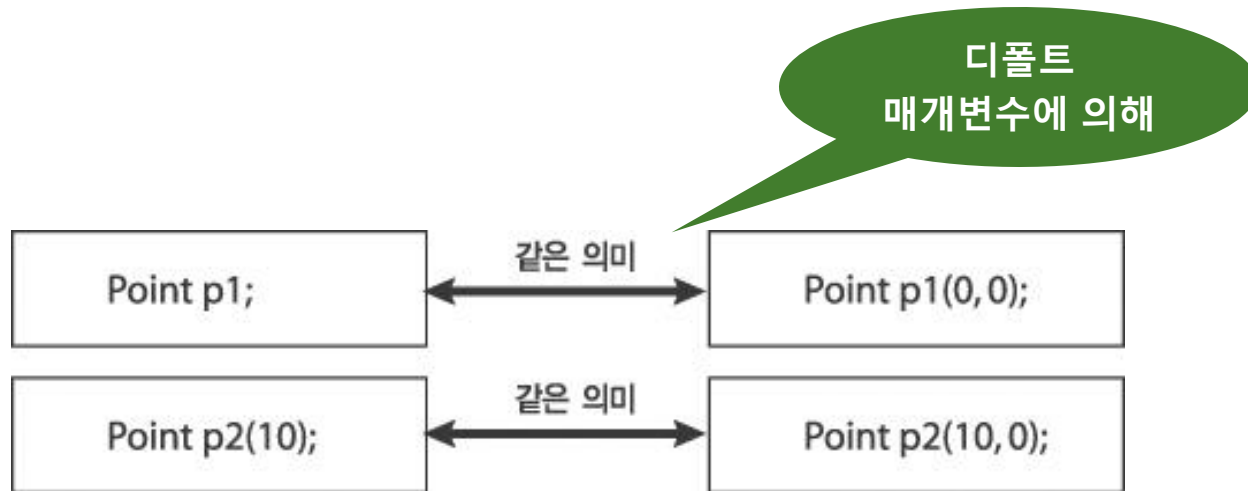
생성자

- 디폴트(default) 생성자
 - 생성자가 하나도 정의되어 있지 않은 경우
 - 자동으로 삽입이 되는 생성자
 - 디폴트 생성자가 하는 일? 아무것도 없다



생성자

- 생성자의 특징
 - 생성자도 함수다!
 - 따라서 함수 오버로딩이 가능하다
 - 디폴트 매개 변수의 설정도 가능하다



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

Constructor1.cpp

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

```
class SimpleClass
{
    int num1;
    int num2;
```

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

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

void ShowData() const
{
    cout<<num1<<' '<<num2<<endl;
}

};

int main(void)
{
    SimpleClass sc1;
    sc1.ShowData();

    SimpleClass sc2(100);
    sc2.ShowData();

    SimpleClass sc3(100, 200);
    sc3.ShowData();
    return 0;
}
```

멤버 이니셜라이저 기반의 멤버 초기화 사용

1. 클래스가 객체 포인터가 아닌 객체 멤버 변수를 가지고 있는 경우

→ 객체 멤버에 대한 생성자를 호출 할 수 없게 됨

2. 상수는 선언과 동시에 초기화 되어야함

→ 단 static 으로 선언한 멤버 변수는 초기화가 가능함

3. 래퍼런스 변수 또한 생성과 동시에 초기화 되어야함

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!

Q : Rectangle 객체를 생성하는 과정에서 Point 클래스의

생성자를 통해서 Point 객체를 초기화할 수 있을까?

A : 해결책으로 이니셜라이저 제시!

멤버 이니셜라이저 기반의 멤버 초기화 (1)

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.”

* 객체의 생성과정

- 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 (2)

```
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)
    {
    }
}
```

이니셜라이저는 멤버변수로 참조자 선언 가능 (3)

```
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!

ReferenceMember.cpp

```
#include <iostream>
using namespace std;
class AAA
{
public:
    AAA()
    {
        cout<<"empty object"<<endl;
    }
    void ShowYourName()
    {
        cout<<"I'm class AAA"<<endl;
    }
};
class BBB
{
private:
    AAA &ref;
    const int &num;

public:
    BBB(AAA &r, const int &n)
        : ref(r), num(n)
    {
    }
    void ShowYourName()
    {
        ref.ShowYourName();
        cout<<"and"<<endl;
        cout<<"I ref num "<<num<<endl;
    }
};
```

```
int main(void)
{
    AAA obj1;
    BBB obj2(obj1, 20);
    obj2.ShowYourName();
    return 0;
}
```

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 private이므로 클래스 외부에서는 이 생성자의 호출을 통해서 객체 생성이 불가능하다

AAA 클래스의 멤버함수 내에서도 AAA 클래스의 객체 생성이 가능하다!
생성자가 private이라는 것은 외부에서의 객체 생성을 허용하지 않겠다는 뜻이다!

예제) Rectangle 클래스 만들기

다음 main() 함수가 잘 작동하도록 Rectangle 클래스를 작성하고 프로그램을 완성하라. Rectangle 클래스는 width와 height의 두 멤버 변수와 3 개의 생성자, 그리고 isSquare() 함수를 가진다.

```
int main() {  
    Rectangle rect1;  
    Rectangle rect2(3, 5);  
    Rectangle rect3(3);  
  
    if(rect1.isSquare()) cout << "rect1은 정사각형이다." << endl;  
    if(rect2.isSquare()) cout << "rect2는 정사각형이다." << endl;  
    if(rect3.isSquare()) cout << "rect3는 정사각형이다." << endl;  
}
```

```
rect1은 정사각형이다.  
rect3는 정사각형이다.
```

Solution)

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

class Rectangle {
public:
    int width, height;

    Rectangle();
    Rectangle(int w, int h);
    Rectangle(int length);
    bool isSquare();
};

Rectangle::Rectangle() {
    width = height = 1;
}

Rectangle::Rectangle(int w, int h) {
    width = w; height = h;
}

Rectangle::Rectangle(int length) {
    width = height = length;
}

// 정사각형이면 true를 리턴하는 멤버 함수
bool Rectangle::isSquare() {
    if(width == height) return true;
    else return false;
}
```

```
int main() {
    Rectangle rect1;
    Rectangle rect2(3, 5);
    Rectangle rect3(3);

    if(rect1.isSquare()) cout << "rect1은 정사각형이다." << endl;
    if(rect2.isSquare()) cout << "rect2는 정사각형이다." << endl;
    if(rect3.isSquare()) cout << "rect3는 정사각형이다." << endl;
}
```

3 개의 생성자가 필요함

rect1은 정사각형이다.

소멸자

- 객체가 소멸되는 시점은?
 - 기본 자료형 변수, 구조체 변수가 소멸되는 시점과 동일하다
- 함수 내에 선언된 객체
 - 함수 호출이 끝나면 소멸된다
- 전역적으로 선언된 객체
 - 프로그램이 종료될 때 소멸된다
 - 이렇게 객체 생성할 일 (거의) 없다!

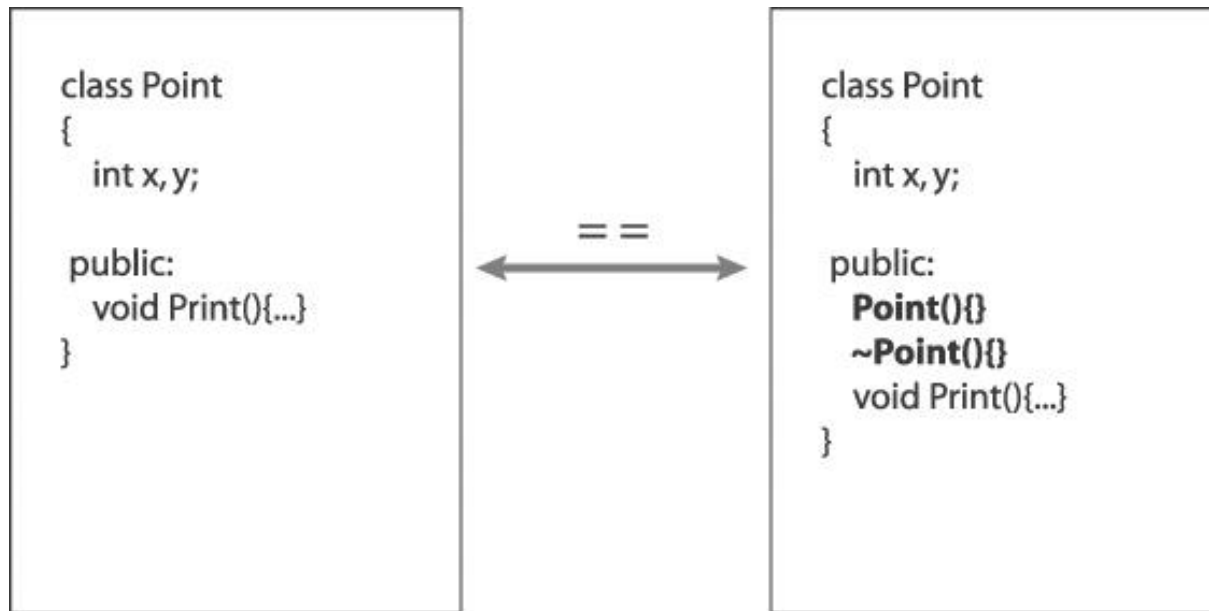
소멸자

- 소멸자의 특성과 필요성

- 소멸자는 생성자와 반대로 객체가 선언되어 사용되는 블록이 끝날 때(객체가 소멸될 때) 자동적으로 호출되는 함수
- 소멸자는 주로 객체가 생성될 때 동적으로 할당한 메모리를 객체의 소멸과 더불어 해제하고자 할 때 사용
- 소멸자는 클래스 이름 앞에 ~(tilde) 기호를 붙여서 정의
- 리턴하지 않으며, 리턴타입 없음
- 전달인자 항상 void
 - 오버로딩, 디폴트매개변수의 선언 불가능
- 객체의 소멸 순서
 - 첫째 : 소멸자 호출
 - 둘째 : 메모리 반환(해제)

소멸자

- 디폴트(Default) 소멸자
 - 객체의 소멸 순서를 만족시키기 위한 소멸자
 - 명시적으로 소멸자 제공되지 않을 경우 자동 삽입
 - 디폴트 생성자와 마찬가지로 하는 일 없다!



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

소멸자(1)

- 소멸자를 사용해서 할당한 메모리를 해제하는 예

```
class DynamicArray
{
public:
    int* arr;
    DynamicArray(int arraySize);
    ~DynamicArray();
};

DynamicArray::DynamicArray(int arraySize)
{
    // 동적으로 메모리를 할당한다.
    arr = new int [arraySize];
}

DynamicArray::~DynamicArray()
{
    // 메모리를 해제한다.
    delete[] arr;
    arr = NULL;
}
```

```
int main()
{
    // 몇 개의 정수를 입력할지 물어본다.
    int size;
    cout << "몇 개의 정수를 입력하시겠습니까? ";
    cin >> size;

    // 필요한 만큼의 메모리를 준비한다.
    DynamicArray da(size);

    // 정수를 입력 받는다.
    for (int i = 0; i < size; ++i)
        cin >> da.arr[i];

    // 역순으로 정수를 출력한다.
    for (i = size - 1; i >= 0; --i)
        cout << da.arr[i] << " ";

    cout << "\n";

    // 따로 메모리를 해제해 줄 필요가 없다.

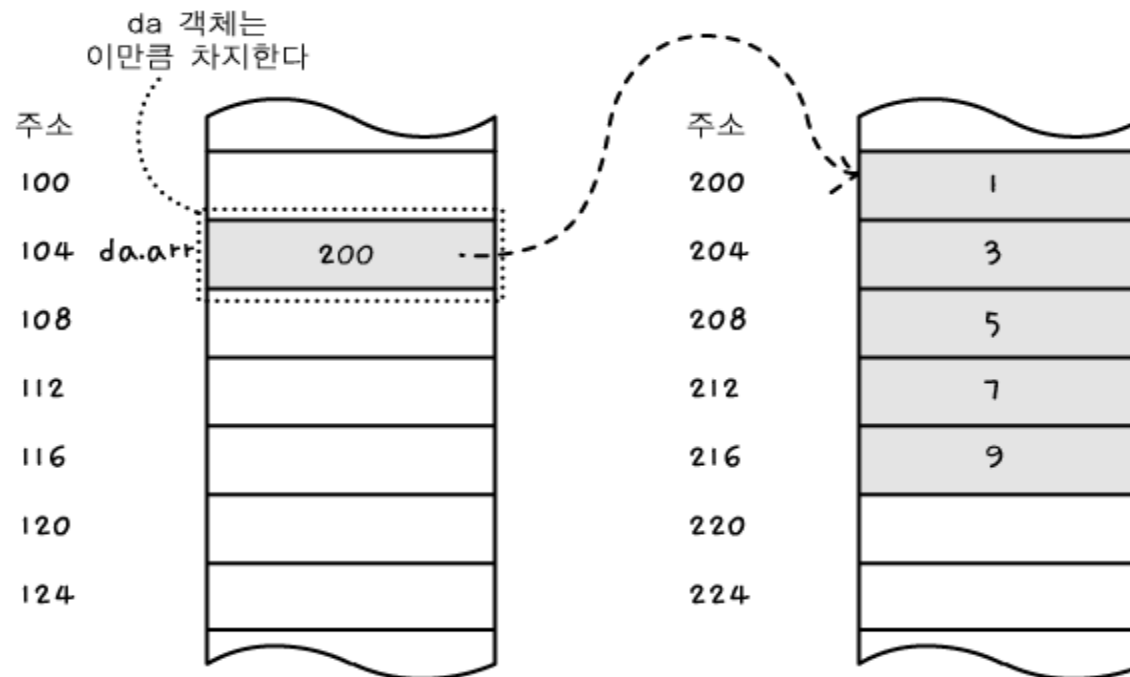
    return 0;
}
```

- 실행 결과

```

C:\ "d:\한빛\source\W21_classes\objects\W15\debug\W15.exe"
몇 개의 정수를 입력하시겠습니까? 5
1 3 5 7 9
9 7 5 3 1
Press any key to continue
  
```

- DynamicArray 객체를 생성한 모습



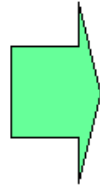
생성자 함수의 특징

- ▣ 생성자의 목적
 - 객체가 생성될 때 객체가 필요한 초기화를 위해
 - 멤버 변수 값 초기화, 메모리 할당, 파일 열기, 네트워크 연결 등
- ▣ 생성자 이름
 - 반드시 클래스 이름과 동일
- ▣ 생성자는 리턴 타입을 선언하지 않는다
 - 리턴 타입 없음 void 타입도 안됨
- ▣ 객체 생성 시 오직 한 번만 호출
 - 자동으로 호출됨 임의로 호출할 수 없음. 각 객체마다 생성자 실행
- ▣ 생성자는 중복 가능
 - 생성자는 한 클래스 내에 여러 개 가능
 - 중복된 생성자 중 하나만 실행
- ▣ 생성자가 선언되어 있지 않으면 기본 생성자 자동으로 생성
 - 기본 생성자 – 매개 변수 없는 생성자
 - 컴파일러에 의해 자동 생성

소멸자 특징

- ▣ 소멸자의 목적
 - 객체가 사라질 때 마무리 작업을 위함
 - 실행 도중 동적으로 할당 받은 메모리 해제, 파일 저장 및 닫기, 네트워크 닫기 등
- ▣ 소멸자 함수의 이름은 클래스 이름 앞에 ~를 붙인다
 - 예) `Circle::~~Circle() { ... }`
- ▣ 소멸자는 리턴 타입이 없고, 어떤 값도 리턴하면 안됨
 - 리턴 타입 선언 불가
- ▣ 중복 불가능
 - 소멸자는 한 클래스 내에 오직 한 개만 작성 가능
 - 소멸자는 매개 변수 없는 함수
- ▣ 소멸자가 선언되어 있지 않으면 기본 소멸자가 자동 생성
 - 컴파일러에 의해 기본 소멸자 코드 생성
 - 컴파일러가 생성한 기본 소멸자 : 아무 것도 하지 않고 단순 리턴

객체의 일생



객체의 생성

생성자()

객체의 사용

소멸자()

객체의 파괴

객체의 일생

생성자/소멸자 실행 순서

- 객체가 선언된 위치에 따른 분류
 - 지역 객체
 - 함수 내에 선언된 객체로서, 함수가 종료하면 소멸된다
 - 전역 객체
 - 함수의 바깥에 선언된 객체로서, 프로그램이 종료할 때 소멸된다
- 객체 생성 순서
 - 전역 객체는 프로그램에 선언된 순서로 생성
 - 지역 객체는 함수가 호출되는 순간에 순서대로 생성
- 객체 소멸 순서
 - 함수가 종료하면, 지역 객체가 생성된 순서의 역순으로 소멸
 - 프로그램이 종료하면, 전역 객체가 생성된 순서의 역순으로 소멸
- new를 이용하여 동적으로 생성된 객체의 경우
 - new를 실행하는 순간 객체 생성
 - delete 연산자를 실행할 때 객체 소멸

예제) 지역 객체와 전역 객체의 생성 및 소멸 순서

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

class Circle {
public:
    int radius;
    Circle();
    Circle(int r);
    ~Circle();
    double getArea();
};

Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::~~Circle() {
    cout << "반지름 " << radius << " 원 소멸" << endl;
}

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

다음 프로그램의 실행 결과는 무엇인가?

```
Circle globalDonut(1000);
Circle globalPizza(2000);

void f() {
    Circle fDonut(100);
    Circle fPizza(200);
}

int main() {
    Circle mainDonut;
    Circle mainPizza(30);
    f();
}
```

예제) 지역 객체와 전역 객체의 생성 및 소멸 순서

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

class Circle {
public:
    int radius;
    Circle();
    Circle(int r);
    ~Circle();
    double getArea();
};

Circle::Circle() {
    radius = 1;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::Circle(int r) {
    radius = r;
    cout << "반지름 " << radius << " 원 생성" << endl;
}

Circle::~~Circle() {
    cout << "반지름 " << radius << " 원 소멸" << endl;
}

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

다음 프로그램의 실행 결과는 무엇인가?

```
Circle globalDonut(1000);
Circle globalPizza(2000);
```

전역 객체 생성

```
void f() {
    Circle fDonut(100);
    Circle fPizza(200);
}
```

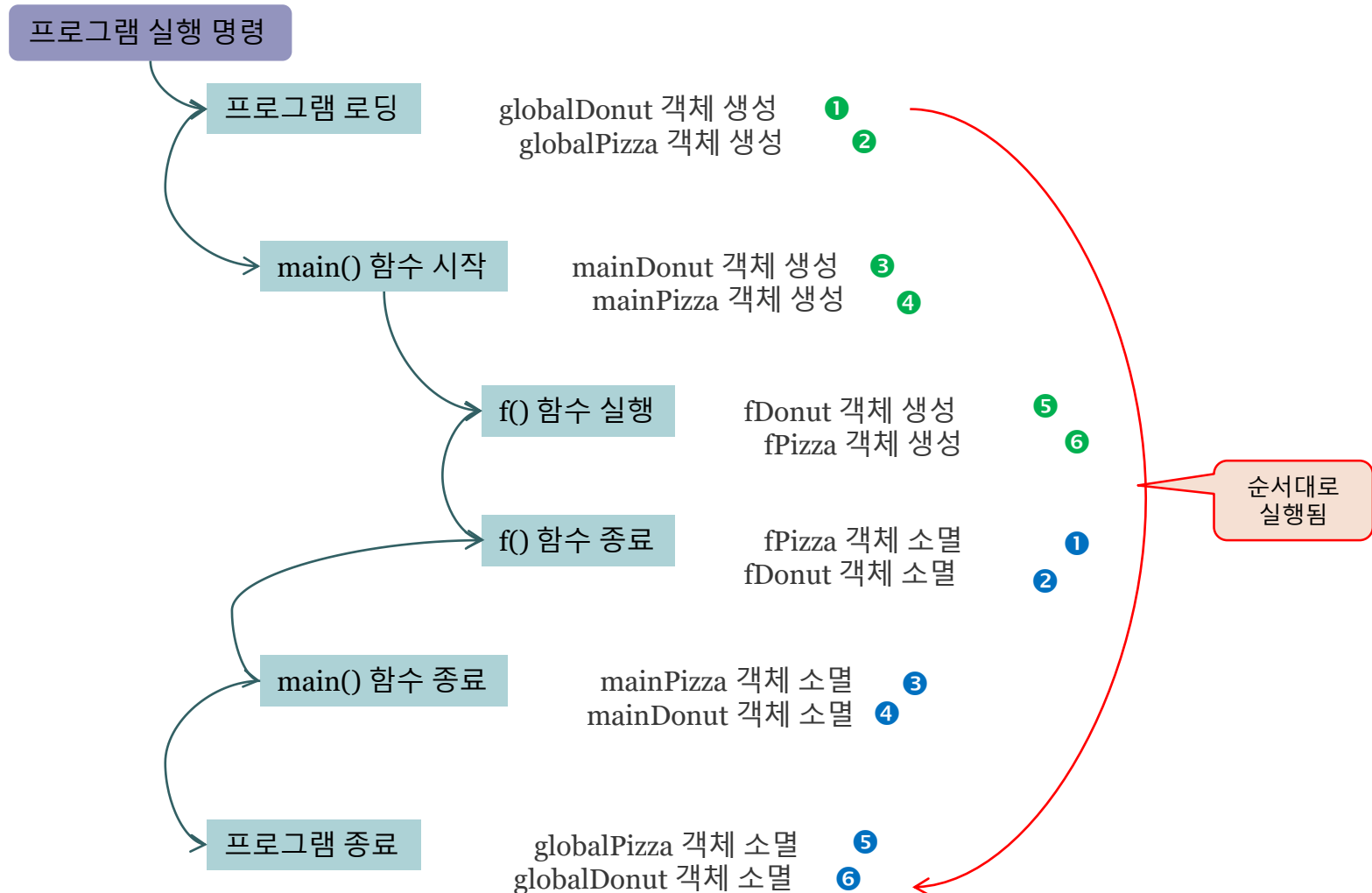
지역 객체 생성

```
int main() {
    Circle mainDonut;
    Circle mainPizza(30);
    f();
}
```

지역 객체 생성

```
반지름 1000 원 생성
반지름 2000 원 생성
반지름 1 원 생성
반지름 30 원 생성
반지름 100 원 생성
반지름 200 원 생성
반지름 200 원 소멸
반지름 100 원 소멸
반지름 30 원 소멸
반지름 1 원 소멸
반지름 2000 원 소멸
반지름 1000 원 소멸
```

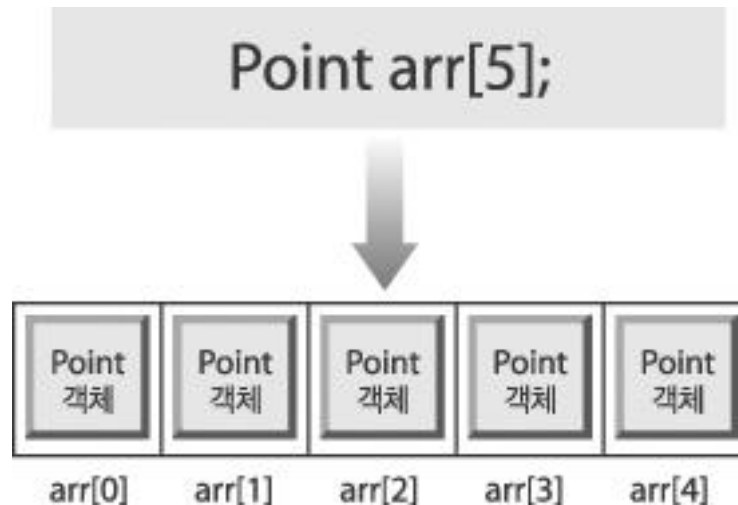
지역 객체와 전역 객체의 생성과 소멸 과정



3. 객체와 배열 / 포인터, 그리고 this 포인터

클래스와 배열

- 객체 배열과 생성자
 - 객체 배열은 객체를 멤버로 지니는 배열
 - 객체 배열은 기본적으로 void 생성자의 호출 요구



클래스와 배열

```
#include<iostream>
using std::cout;
using std::endl;

class Point {
    int x;
    int y;
public:
    Point(){
        cout<<"Point() call!"<<endl;
        x=y=0;
    }
    Point(int _x, int _y){
        x=_x;
        y=_y;
    }

    int GetX()    { return x; }
    int GetY()    { return y; }

    void SetX(int _x){ x=_x; }
    void SetY(int _y){ y=_y; }
};
```

```
int main()
{
    Point arr[5];

    for(int i=0; i<5; i++)
    {
        arr[i].SetX(i*2);
        arr[i].SetY(i*3);
    }

    for(int j=0; j<5; j++)
    {
        cout<<"x: "<<arr[j].GetX()<<' ';
        cout<<"y: "<<arr[j].GetY()<<endl;
    }

    return 0;
}
```

객체 포인터

- 객체에 대한 포인터
 - C 언어의 포인터와 동일
 - 객체의 주소 값을 가지는 변수
- 포인터로 멤버를 접근할 때
 - 객체포인터->멤버

```
Circle donut;
double d = donut.getArea();
```

객체에 대한 포인터 선언

```
Circle *p; // (1)
p = &donut; // (2)
d = p->getArea(); // (3)
```

포인터에 객체 주소 저장

멤버 함수 호출

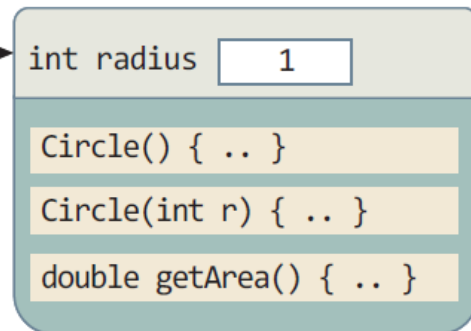
(1) Circle *p;



(2) p=&donut;



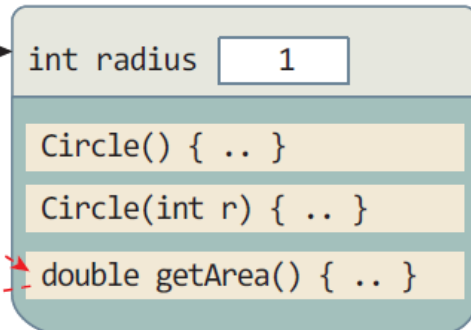
donut 객체



(3) d=p->getArea();



donut 객체



호출



예제) 객체 포인터 선언 및 활용

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

class Circle {
    int radius;
public:
    Circle() {radius = 1; }
    Circle(int r) { radius = r; }
    double getArea();
};

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

```
int main() {
    Circle donut;
    Circle pizza(30);

    // 객체 이름으로 멤버 접근
    cout << donut.getArea() << endl;

    // 객체 포인터로 멤버 접근
    Circle *p;
    p = &donut;
    cout << p->getArea() << endl; // donut의 getArea() 호출
    cout << (*p).getArea() << endl; // donut의 getArea() 호출

    p = &pizza;
    cout << p->getArea() << endl; // pizza의 getArea() 호출
    cout << (*p).getArea() << endl; // pizza의 getArea() 호출
}
```

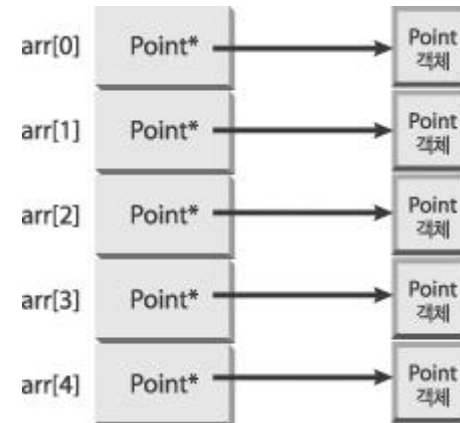
```
3.14
3.14
3.14
2826
2826
```

포인터와 배열

- 객체 포인터 배열

- 객체를 가리킬 수 있는 포인터를 멤버로 지니는 배열
- 객체의 동적 생성 방법

```
Point* arr[5];
```



포인터와 배열

```
#include<iostream>
using std::cout;
using std::endl;
class Point {
    int x;
    int y;
public:
    Point(){
        cout<<"Point() call!"<<endl;
        x=y=0;
    }
    Point(int _x, int _y){
        x=_x;
        y=_y;
    }
    int GetX() { return x; }
    int GetY() { return y; }
    void SetX(int _x){ x=_x; }
    void SetY(int _y){ y=_y; }
};
```

```
int main()
{
    Point* arr[5];

    for(int i=0; i<5; i++)
    {
        arr[i]=new Point(i*2, i*3);
        // new에 의한 객체 동적 생성.
    }

    for(int j=0; j<5; j++)
    {
        cout<<"x: "<<arr[j]->GetX()<<' ';
        cout<<"y: "<<arr[j]->GetY()<<endl;
    }

    for(int k=0; k<5; k++)
    {
        delete arr[k];//힙에 저장된 객체 소멸.
    }

    return 0;
}
```

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

예제) Circle 클래스의 배열 선언 및 활용

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

```
class Circle {
    int radius;
public:
    Circle() { radius = 1; }
    Circle(int r) { radius = r; }
    void setRadius(int r) { radius = r; }
    double getArea();
};

double Circle::getArea() {
    return 3.14*radius*radius;
}
```

```
int main() {
    Circle circleArray[3]; // (1) Circle 객체 배열 생성

    // 배열의 각 원소 객체의 멤버 접근
    circleArray[0].setRadius(10); // (2)
    circleArray[1].setRadius(20);
    circleArray[2].setRadius(30);

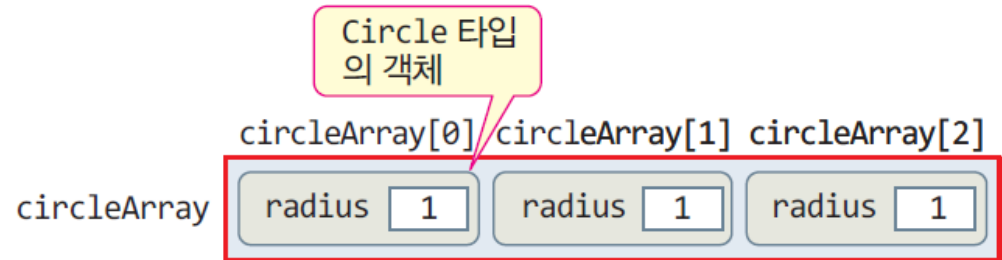
    for(int i=0; i<3; i++) // 배열의 각 원소 객체의 멤버 접근
        cout << "Circle " << i << "의 면적은 " << circleArray[i].getArea() << endl;

    Circle *p; // (3)
    p = circleArray; // (4)
    for(int i=0; i<3; i++) { // 객체 포인터로 배열 접근
        cout << "Circle " << i << "의 면적은 " << p->getArea() << endl;
        p++; // (5)
    }
}
```

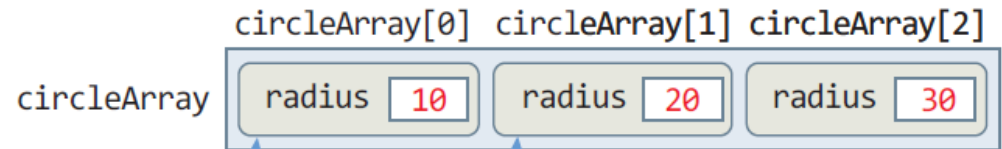
```
Circle 0의 면적은 314
Circle 1의 면적은 1256
Circle 2의 면적은 2826
Circle 0의 면적은 314
Circle 1의 면적은 1256
Circle 2의 면적은 2826
```

배열 생성과 활용(실행 과정)

(1) `Circle circleArray[3];`



(2) `circleArray[0].setRadius(10);`
`circleArray[1].setRadius(20);`
`circleArray[2].setRadius(30);`



(3) `Circle *p;`



(4) `p = circleArray;`



(5) `p++;`



객체 배열 초기화

- 객체 배열 초기화 방법
 - 배열의 각 원소 객체당 생성자 지정하는 방법

```
Circle circleArray[3] = { Circle(10), Circle(20), Circle() };
```

- circleArray[0] 객체가 생성될 때, 생성자 Circle(10) 호출
- circleArray[1] 객체가 생성될 때, 생성자 Circle(20) 호출
- circleArray[2] 객체가 생성될 때, 생성자 Circle() 호출

예제) 객체 배열 초기화

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

class Circle {
    int radius;
public:
    Circle() {radius = 1; }
    Circle(int r) { radius = r; }
    void setRadius(int r) { radius = r; }
    double getArea();
};

double Circle::getArea() {
    return 3.14*radius*radius;
}

int main() {
    Circle circleArray[3] = { Circle(10), Circle(20), Circle() }; // Circle 배열 초기화

    for(int i=0; i<3; i++)
        cout << "Circle " << i << "의 면적은 " << circleArray[i].getArea() << endl;
}
```

circleArray[0] 객체가 생성될 때, 생성자 Circle(10),
circleArray[1] 객체가 생성될 때, 생성자 Circle(20),
circleArray[2] 객체가 생성될 때, 기본 생성자 Circle()
이 호출된다.

Circle 0의 면적은 314
Circle 1의 면적은 1256
Circle 2의 면적은 3.14

객체 배열 생성시 기본 생성자 호출

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

class Circle {
    int radius;
public:
    double getArea() {
        return 3.14*radius*radius;
    }
};

int main() {
    Circle circleArray[3];
}
```

컴파일러가 자동으로 디폴트 생성자
Circle() {} 삽입.
컴파일 오류가 발생하지 않음

디폴트 생성자 Circle() 호출

(a) 생성자가 선언되어
있지 않은 Circle 클래스

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

class Circle {
    int radius;
public:
    Circle(int r) { radius = r; }
    double getArea() {
        return 3.14*radius*radius;
    }
};

int main() {
    Circle waffle(15);
    Circle circleArray[3];
}
```

Circle(int r)
호출

Circle() {}를 추가해줘야 함 ←

기본 생성자 Circle() 호출.
기본 생성자가 없으므로 컴
파일 오류

error.cpp(15): error C2512: 'Circle' : 사용할
수 있는 적절한 기본 생성자가 없습니다

(b) 디폴트 생성자가 없으므로 컴파일 오류

자기참조(self-reference)

- 자기참조

- 클래스의 멤버함수는 자신을 호출한 객체를 가리키는 포인터를 명시
- this라는 키워드가 자기 자신 객체의 포인터를 가리킴

자기 참조 예 (1)

```
#include <iostream>
using std::cout;
using std::endl;

class Person {
public:
    Person* GetThis(){
        return this; //this 포인터를 리턴
    }
};

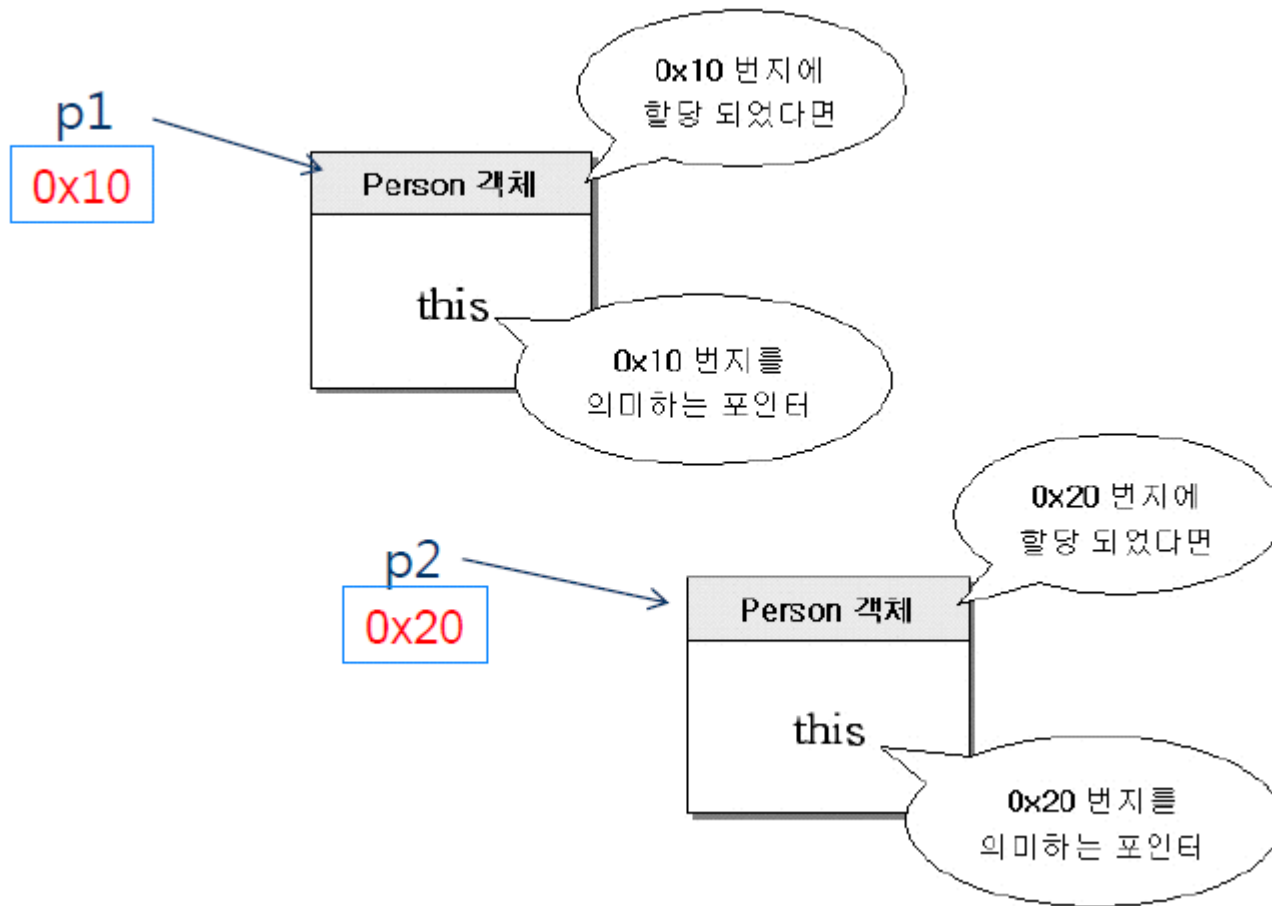
int main()
{
    cout<<"***** p1의 정보 *****"<<endl;
    Person *p1 = new Person();
    cout<<"포인터 p1: "<<p1<<endl;
    cout<<"p1의 this: "<<p1->GetThis()<<endl;

    cout<<"***** p2의 정보 *****"<<endl;
    Person *p2 = new Person();
    cout<<"포인터 p2: "<<p2<<endl;
    cout<<"p2의 this: "<<p2->GetThis()<<endl;

    return 0;
}
```

자기참조(self-reference)

- this 포인터의 의미



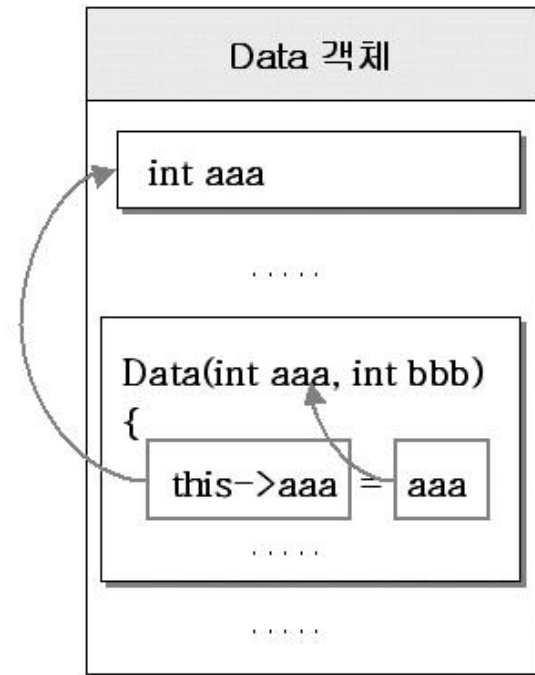
자기 참조 예 (2)

```
#include <iostream>
using std::cout;
using std::endl;
```

```
class Data {
    int aaa;
    int bbb;
public :
    Data(int aaa, int bbb) {
        //aaa=aaa;
        this->aaa=aaa;

        //bbb=bbb;
        this->bbb=bbb;
    }
    void printAll() {
        cout<<aaa<<" "<<bbb<<endl;
    }
};
```

```
int main(void)
{
    Data d(100, 200);
    d.printAll();
    return 0;
}
```



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=0x7fff48456328
0x7fff48456328, 100
num=200, address=0x7fff4845632c
0x7fff4845632c, 200
```

```
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 num1을 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;
    }
};
```

* Self-reference란

객체 자신을 참조할 수 있는 참조자

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;
}
```

참고문헌

- 뇌를 자극하는 C++ 프로그래밍, 이현창, 한빛미디어, 2011
- 열혈 C++ 프로그래밍(개정판), 윤성우, 오렌지미디어, 2012
- C++ ESPRESSO, 천인국 저, 인피니티북스, 2011
- 명품 C++ Programming, 황기태, 생능출판사, 2018
- 어서와 C++는 처음이지, 천인국, 인피니티북스, 2018



Q&A

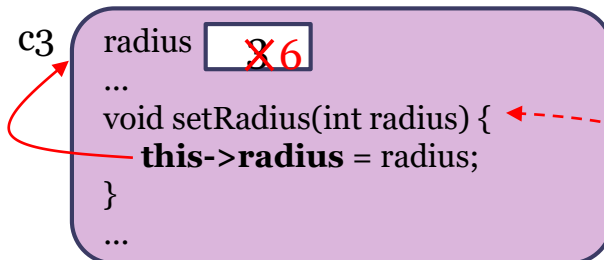
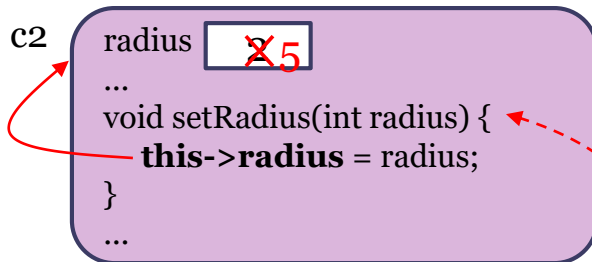
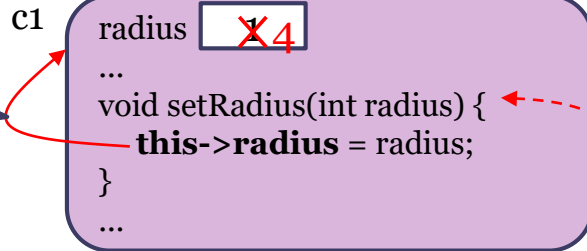
추가 자료

-This 포인터

this와 객체

* 각 객체 속의 this는 다른 객체의 this와 다름

this는 객체 자신
에 대한 포인터

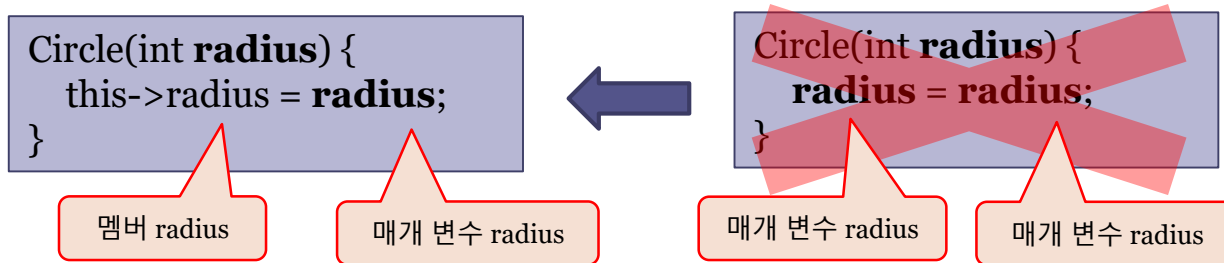


```
class Circle {  
    int radius;  
public:  
    Circle() {  
        this->radius=1;  
    }  
    Circle(int radius) {  
        this->radius = radius;  
    }  
    void setRadius(int radius) {  
        this->radius = radius;  
    }  
};
```

```
int main() {  
    Circle c1;  
    Circle c2(2);  
    Circle c3(3);  
  
    c1.setRadius(4);  
    c2.setRadius(5);  
    c3.setRadius(6);  
}
```

this가 필요한 경우

- 매개변수의 이름과 멤버 변수의 이름이 같은 경우



- 멤버 함수가 객체 자신의 주소를 리턴할 때
 - 연산자 중복 시에 매우 필요

```
class Sample {
public:
    Sample* f() {
        ....
        return this;
    }
};
```

this의 제약 사항

- 멤버 함수가 아닌 함수에서 this 사용 불가
 - 객체와의 관련성이 없기 때문
- static 멤버 함수에서 this 사용 불가
 - 객체가 생기기 전에 static 함수 호출이 있을 수 있기 때문에

this 포인터의 실체 - 컴파일러에서 처리

```
class Sample {
    int a;
public:
    void setA(int x) {
        this->a = x;
    }
};
```

(a) 개발자가 작성한 클래스

컴파일러에 의해
변환

```
class Sample {
    ...
public:
    void setA(Sample* this, int x) {
        this->a = x;
    }
};
```

this는 컴파일러에 의해 묵시적으로 삽입된 매개 변수

(b) 컴파일러에 의해 변환된 클래스

Sample ob;

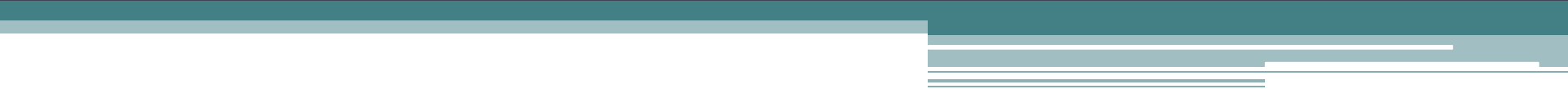
ob.setA(5);

컴파일러에 의해 변환

ob.setA(**&ob**, 5);

ob의 주소가 this 매개 변수에 전달됨

(c) 객체의 멤버 함수를 호출하는 코드의 변환



Q&A