

Window Programming

Visual C++ MFC Programming

Lecture 03

김예진

Dept. of Game Software

Announcement

- 03/16: HW #1 on ClassNet (Due: 03/24)

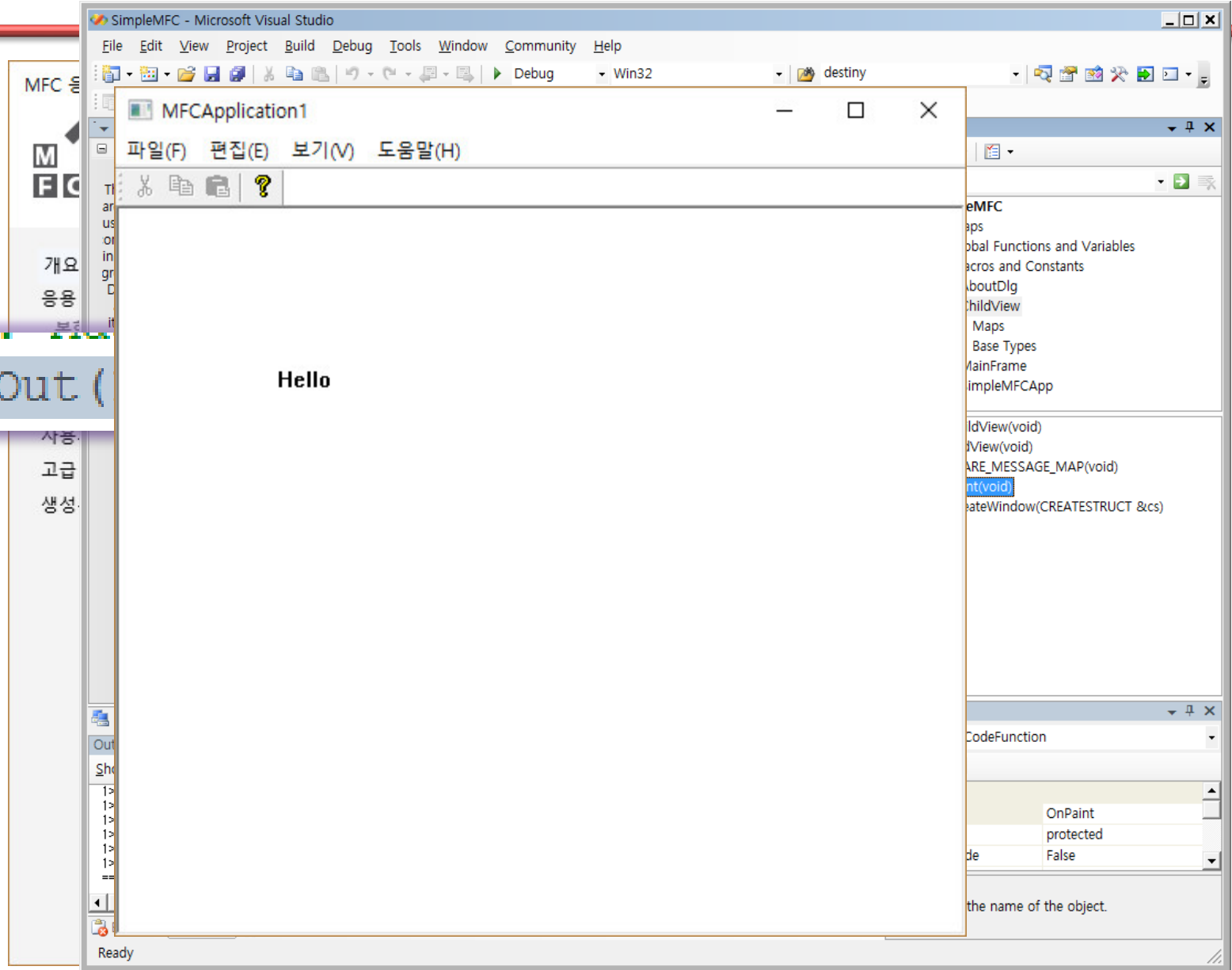
Plan

- MFC 주요 특징 리뷰
- 객체지향 (C++) 프로그래밍 기초
- HW #1 및 주의 사항

MFC 주요 특징

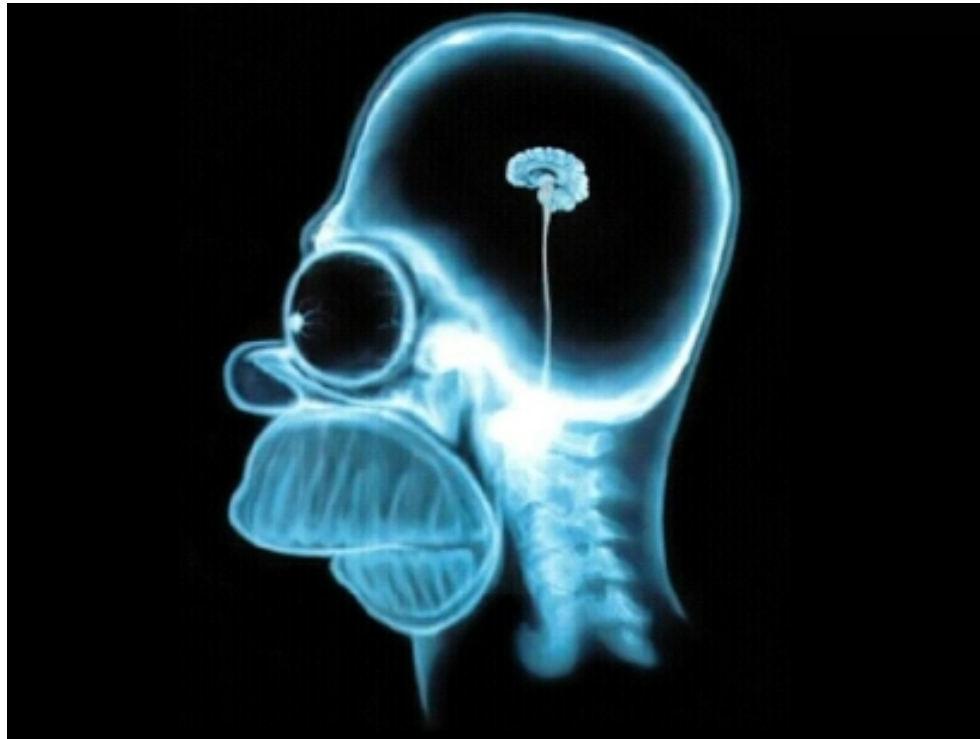
- 빠른 Window 프로그램 작성
 - Library 재사용, AppWizard, ClassWizard, 인쇄 기능 지원, 툴바와 상태바 처리, 데이터베이스(DB) 지원, OLE와 ActiveX 등
 - API를 직접 사용해서 구현할 경우 복잡도가 높은 부분을 MFC를 이용하면 쉽게 구현
- API 기반 program과 대등한 속도를 가짐
 - API 함수를 직접 호출: Ex) ReleaseCapture();
 - inline 함수의 활용: 자주 호출되어야 하는 함수의 경우
- Code 크기 증가를 최소화
 - 동적(Dynamic) library 활용
- C++ 언어를 이용하여 기존의 C 언어에 비해 API를 좀 더 편하게 사용할 수 있음
 - Ex) 오버로딩(Overloading) 및 디폴트 인자

MFC 응용 프로그램 작성



dc.TextOut (

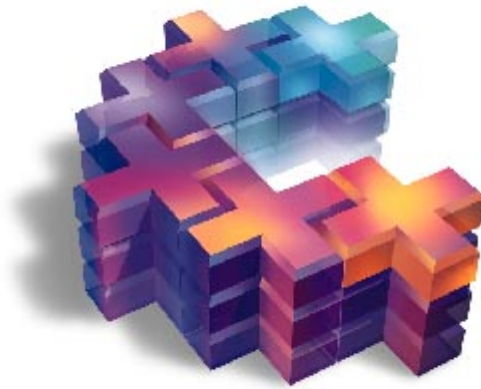
Look into the codes





C++?

CLASS?

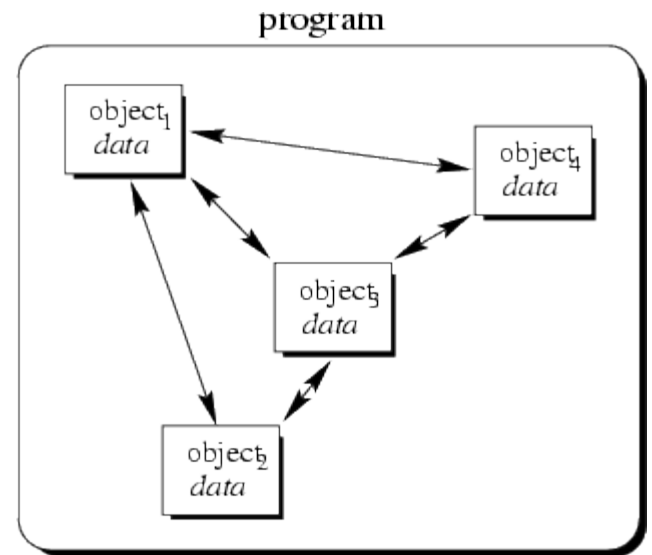
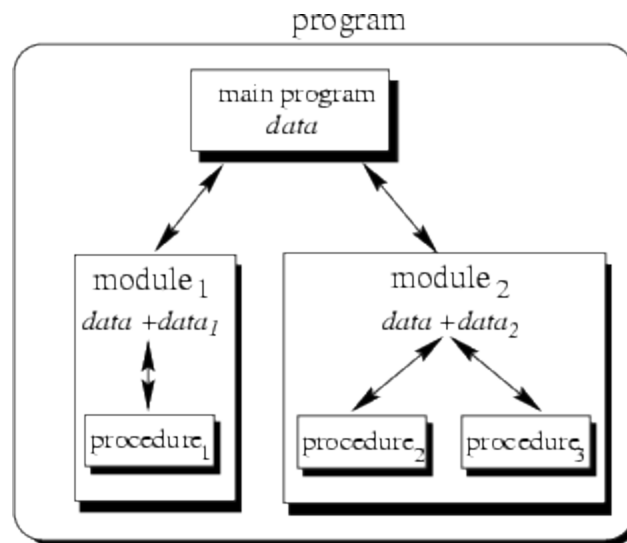
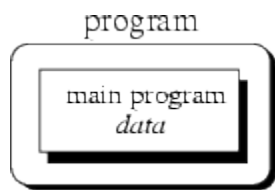
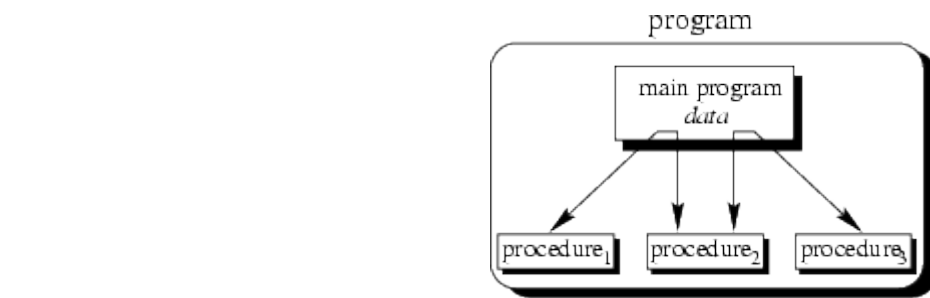


- C++
 - 향상된 C
 - C + Class
 - C의 효율성 + 객체지향개념(object-oriented)

절차지향과 객체지향

- 객체지향의 배경
 - 소프트웨어 모듈의 재사용과 독립성을 강조
- 객체지향과 절차지향의 비교
 - 절차지향 (Procedural-oriented) : 데이터 구조와 그 데이터를 변화시키는 procedure/function으로 구성
 - 객체지향 (Object-oriented) : 객체들이 **메시지(message)**를 통하여 통신함으로써 원하는 결과를 얻음. 각 객체는 고유의 **속성(attribute)**와 데이터를 처리할 수 있는 **메소드(method)**로 구성

절차지향과 객체지향



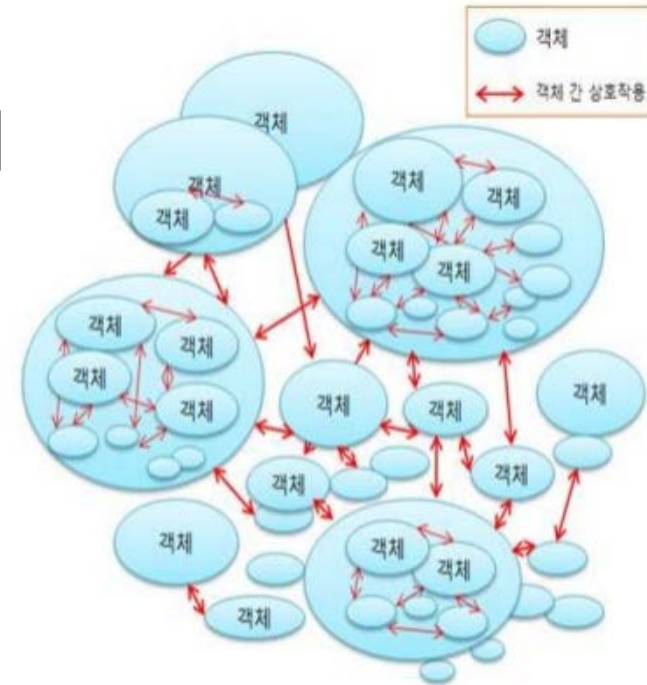
비구조적

구조적/절차지향

객체지향

객체

- 객체(Object)
 - 효율적으로 정보를 관리하기 위하여, 사람들이 의미를 부여하고 분류하는 논리적인(개념적인) 단위
 - 실 세계에 존재하는 하나의 단위에 대한 소프트웨어적 표현
 - 관련된 **변수**와 **함수**의 묶음
- 객체의 구성
 - 속성의 값을 나타내는 데이터(data): **attribute**
 - 데이터를 변경하거나 조작하는 기능(function): **method**



객체지향 프로그래밍의 특징

- 상속 (Inheritance)
- 캡슐화 (Encapsulation)
- 다형성 (Polymorphism)

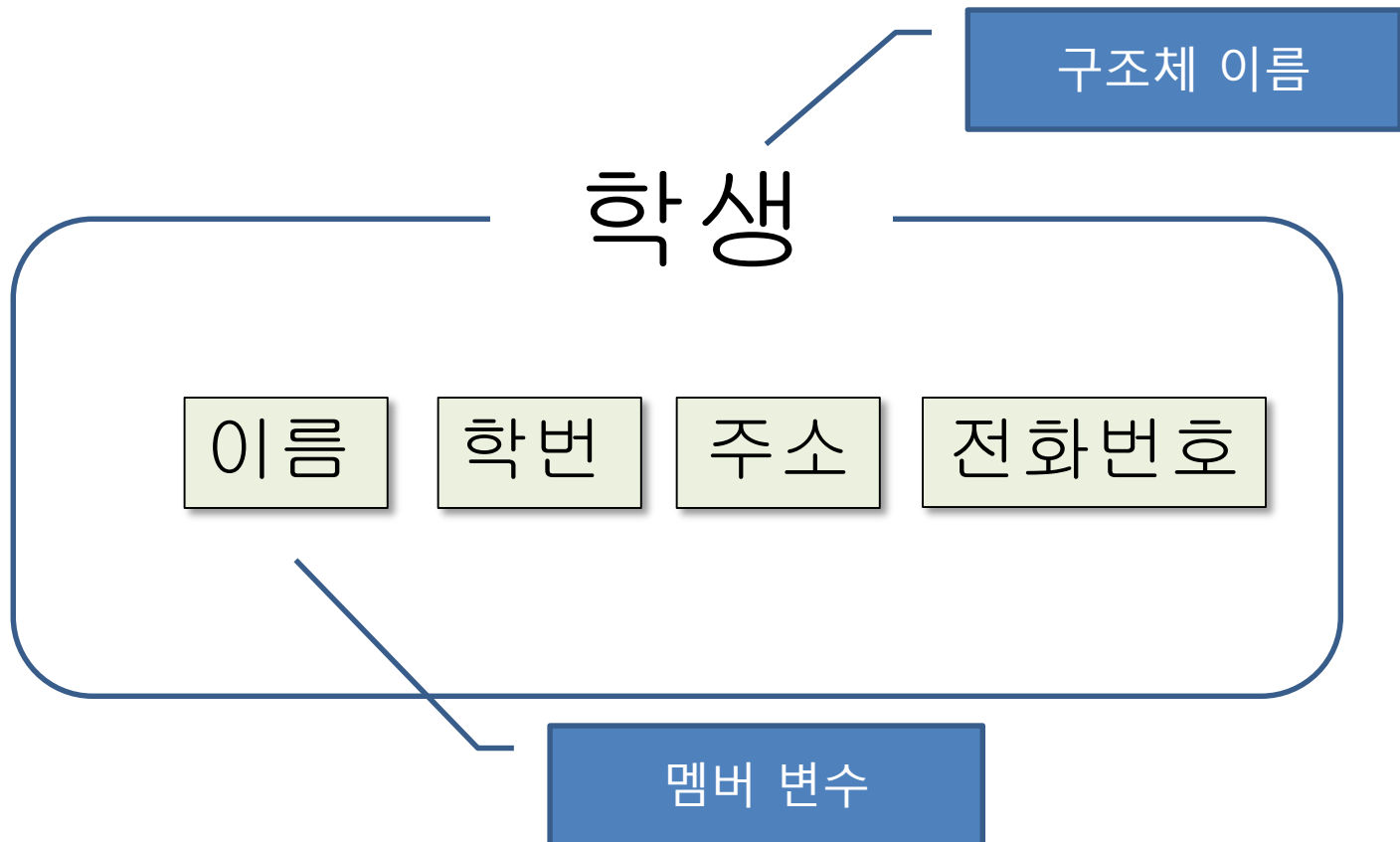
→ 각각의 의미에 대해 조사하기!

데이터를 묶는 단위: 구조체



구조체 (structure)

구조체 : 관련된 정보를 그룹화하여 표현



구조체의 정의

- 구조체의 정의

```
struct 구조체이름  
{  
    멤버변수타입1 변수이름1;  
    멤버변수타입2 변수이름2;  
    멤버변수타입3 변수이름3;  
    ...  
};
```

구조체의 정의

- Ex)

```
struct student
{
    char name[30];
    int number;
    float grade;
};
```


구조체 타입의 변수 선언

구조체이름 구조체변수이름;

```
struct student
{
    char name[30];
    int number;
    float grade;
};
```

```
student a, b;
```

멤버로의 접근

구조체변수이름.멤버변수이름

```
struct score
{
    int korean;
    int math;
};
```

```
score a;
a.korean = 30;
a.math = 80;
```

구조체를 가리키는 포인터

- 구조체를 변수형처럼 사용해서 포인터를 정의할 수 있다

```
struct rectangle
{
    int x,y;
    int width, height;
};
```

```
rectangle rc;
rectangle * p = &rc;
```

구조체의 멤버 변수 접근

- 구조체에서는:

구조체변수.멤버변수

- 구조체포인터는:

(*구조체포인터).멤버변수

구조체포인터->멤버변수

구조체의 멤버 변수 접근 예

```
struct rectangle
{
    int x,y;
    int width, height;
};
rectangle rc;
rectangle * p = &rc;
```

```
rc.x=10;
(*p).x = 10;
p->x = 10;
```

} 모두 같다

데이터와 그 기능을 묶는 단위
: Class

Class의 정의

- Point class를 정의하는 예

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

// Point 클래스를 정의한다.
class Point
{
public:
    // 멤버 변수들
    int x, y;

    // 멤버 함수
    void Print()
    {
        cout << "( " << x << ", " << y << ")\n";
    }
};
```

Class의 정의

- Point class를 정의하는 예

클래스의
정의를 의미

접근 제어와
관련된 키워드

```
// Point 클래스를 정의한다.
class Point
{
    public:
        // 멤버 변수들
        int x, y;

        // 멤버 함수
        void Print()
        {
            cout << "(" << x << ", " << y << ")\\n";
        }
};
```

세미 콜론도
잊지 말자

클래스의 정의 안에서
정의된 함수는
멤버 함수가 된다

객체의 생성과 사용(1)

- Class 객체를 생성하고 사용하는 예

```
int main()
{
    // 객체를 생성한다.
    Point pt1, pt2;

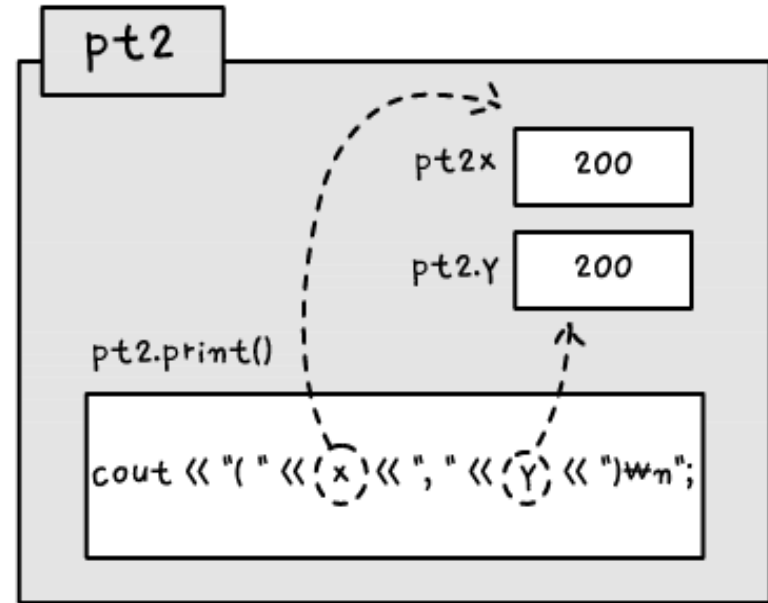
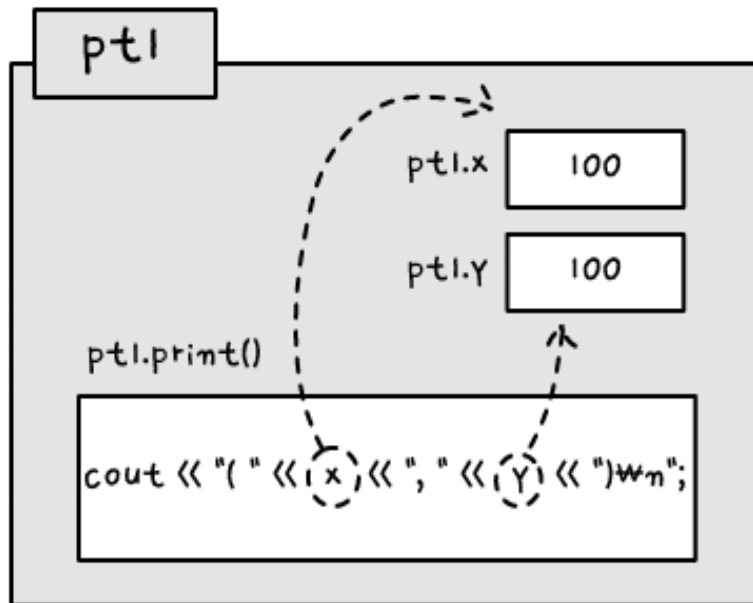
    // pt1, pt2를 초기화 한다.
    pt1.x = 100;
    pt1.y = 100;
    pt2.x = 200;
    pt2.y = 200;

    // pt1, p2의 내용을 출력한다.
    pt1.Print();
    pt2.Print();

    return 0;
}
```

객체의 생성과 사용(2)

- Print() 함수에서 사용하는 x, y의 의미



멤버 함수의 위치

- 클래스의 정의 바깥쪽에 멤버 함수를 정의한 예

```
class Point
{
public:
    // 멤버 변수
    int x, y;

    // 멤버 함수
    void Print();
};

void Point::Print()
{
    cout << "( " << x << ", " << y << ")\n";
}
```

생성자와 소멸자

- **생성자(Constructor)**는 객체를 생성할 때 자동으로 호출되는 함수
 - 그러므로 생성자는 객체를 사용할 수 있도록 초기화 하는 코드를 넣기에 알맞은 장소
- **소멸자(Destructor)**는 객체를 소멸할 때 자동으로 호출되는 함수
 - 그러므로 소멸자는 객체가 사용한 리소스를 정리하는 코드를 넣기에 알맞은 장소

디폴트 생성자(Default Constructor)

- 디폴트 생성자의 추가

```
class Point
{
public:
    int x, y;
    void Print();
    Point();
};

Point::Point()
{
    x = 0;
    y = 0;
}

// 실제 실행 시..

Point pt;           // 생성자가 호출된다.
pt.Print();
```

인자가 있는 생성자

- 인자(Arguments)가 있는 생성자의 추가

```
class Point
{
public:
    int x, y;
    void Print();
    Point();
    Point(int initX, int initY);
    Point(int initX2, int initY2); → O.k.?
    Point(int initX, float initY); → O.k.?
};

Point::Point(int initX, int initY)
{
    x = initX;
    y = initY;
}

// 중간 생략

Point pt(3, 5);
pt.Print();
```

function
overloading {

소멸자

- 소멸자를 사용해서 할당한 메모리를 해제한 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;
}
```

객체의 동적인 생성

- 동적 메모리 할당을 사용해서 객체를 생성한 예

```
Point pt(50, 50);

Point* p1 = new Point();           // 디폴트 생성자 사용
Point* p2 = new Point(100, 100);  // 인자있는 생성자 사용
Point* p3 = new Point( pt);       // 복사 생성자 사용

p1->Print();
p2->Print();
p3->Print();

delete p1;
delete p2;
delete p3;
```


접근 권한 설정하기(1)

- 멤버의 접근 권한을 설정하는 예

```
class AccessControl
{
public:
    char publicData;
    void publicFunc() {};

protected:
    int protectedData;
    void protectedFunc() {};

private:
    float privateData;
    void privateFunc() {};
};

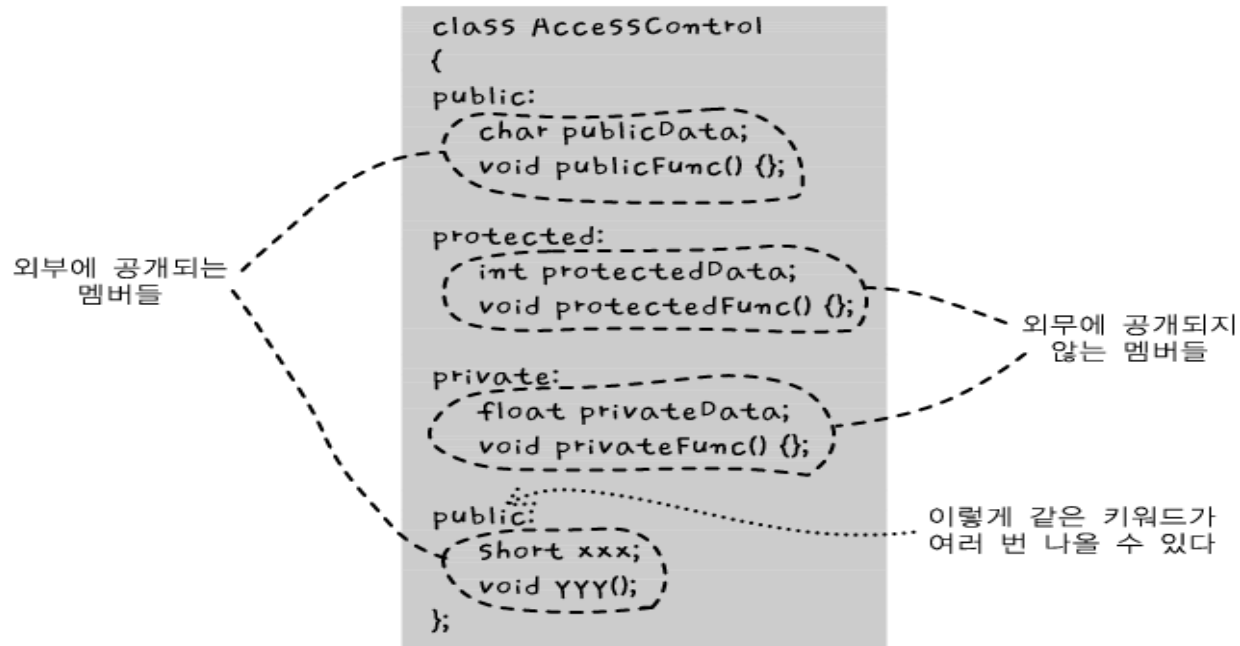
int main()
{
    // 객체를 생성하고, 각 멤버에 접근해보자
    AccessControl ac;

    ac.publicData = 'A'; // 성공
    ac.publicFunc();     // 성공
    ac.protectedData = 100; // 실패
    ac.protectedFunc();   // 실패
    ac.privateData = 4.5f; // 실패
    ac.privateFunc();     // 실패

    return 0;
}
```

접근 권한 설정하기(2)

- 멤버의 접근 권한 설정하기



- 접근 권한 키워드에 대한 요약 (뒤에서 더욱 자세히 분류)
 - `public` : 외부에서의 접근을 허용한다.
 - `protected`, `private` : 외부에서 접근할 수 없다.

상속과 포함

(inheritance & containment)

상속

- 앞으로의 예제 설명:

문서저장클래스

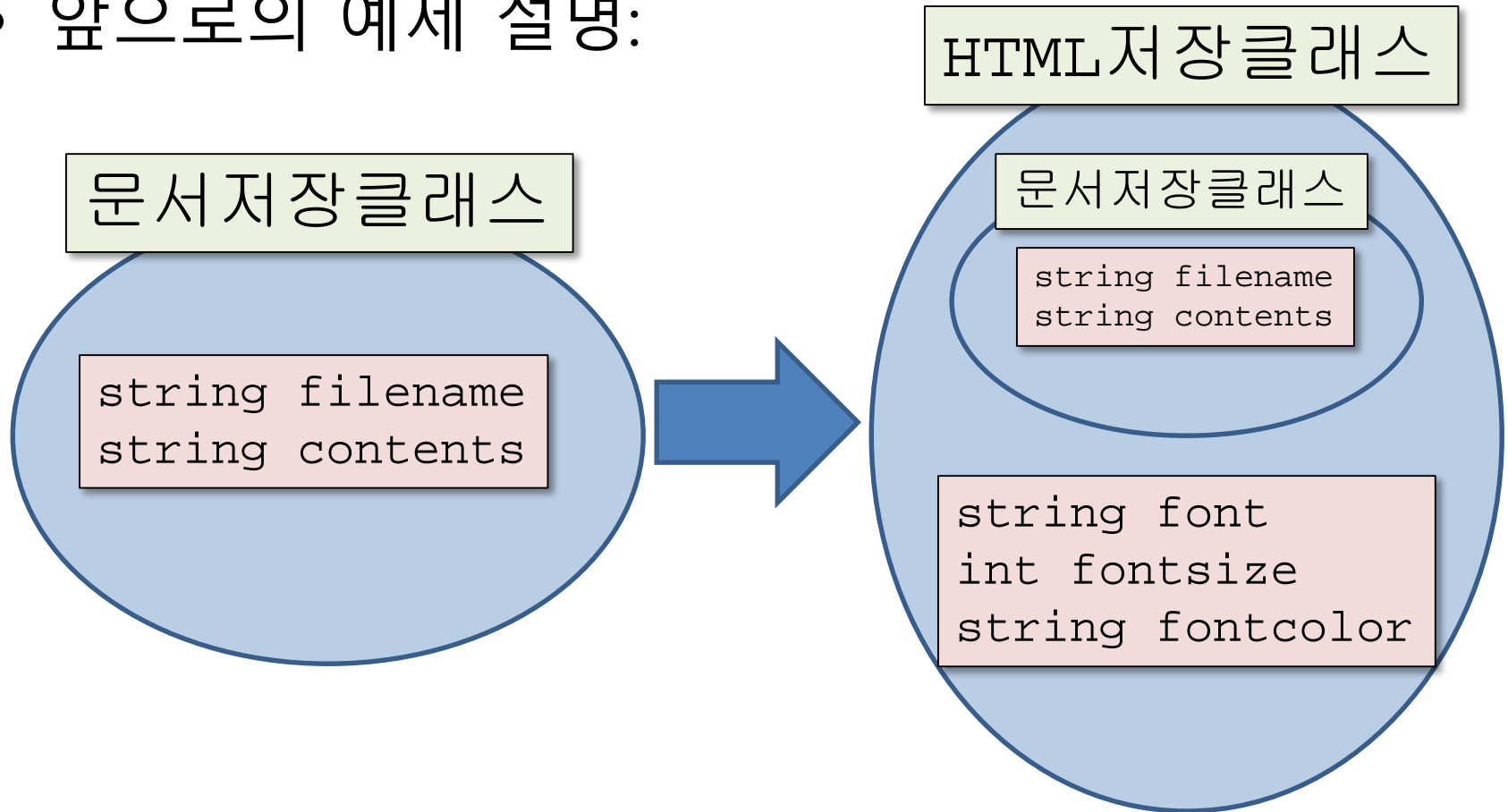
```
string filename  
string contents
```

HTML저장클래스

```
string filename  
string contents  
string font  
int fontsize  
string fontcolor
```

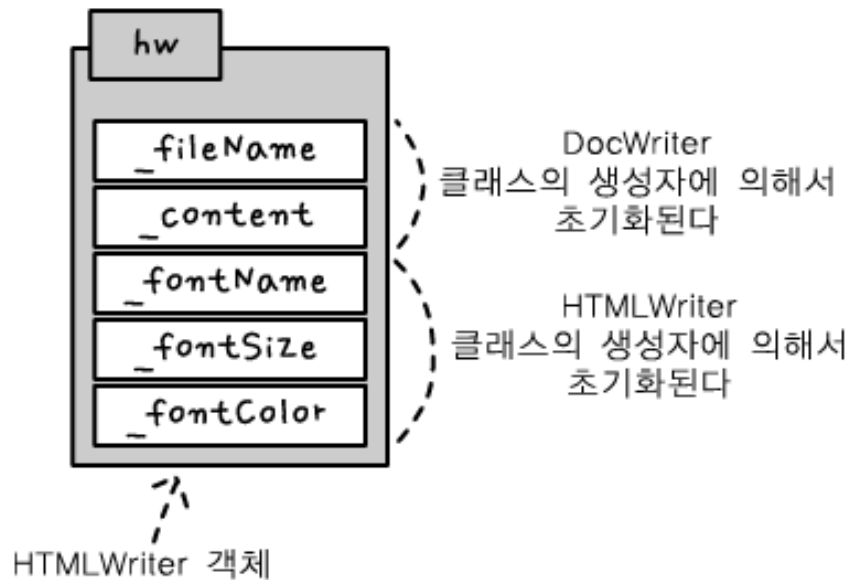
상속

- 앞으로의 예제 설명:



생성자와 소멸자

- 자식 객체가 생성될 때, 자식 클래스의 생성자 뿐만 아니라 부모 클래스의 생성자도 호출된다.



- 부모 클래스에 오버로딩된 여러 생성자가 있다면 그 중에서 어떤 생성자가 호출될 지 결정할 수 있다. (다음 페이지 참조)

부모 클래스의 생성자

- 자식 클래스의 생성자에서 부모 클래스의 생성자를 지정한 예

```
// HTMLWriter.h
class HTMLWriter : public DocWriter
{
public:
    HTMLWriter(void);
    HTMLWriter(const string& fileName, const string& content);
    ~HTMLWriter(void);
    ...

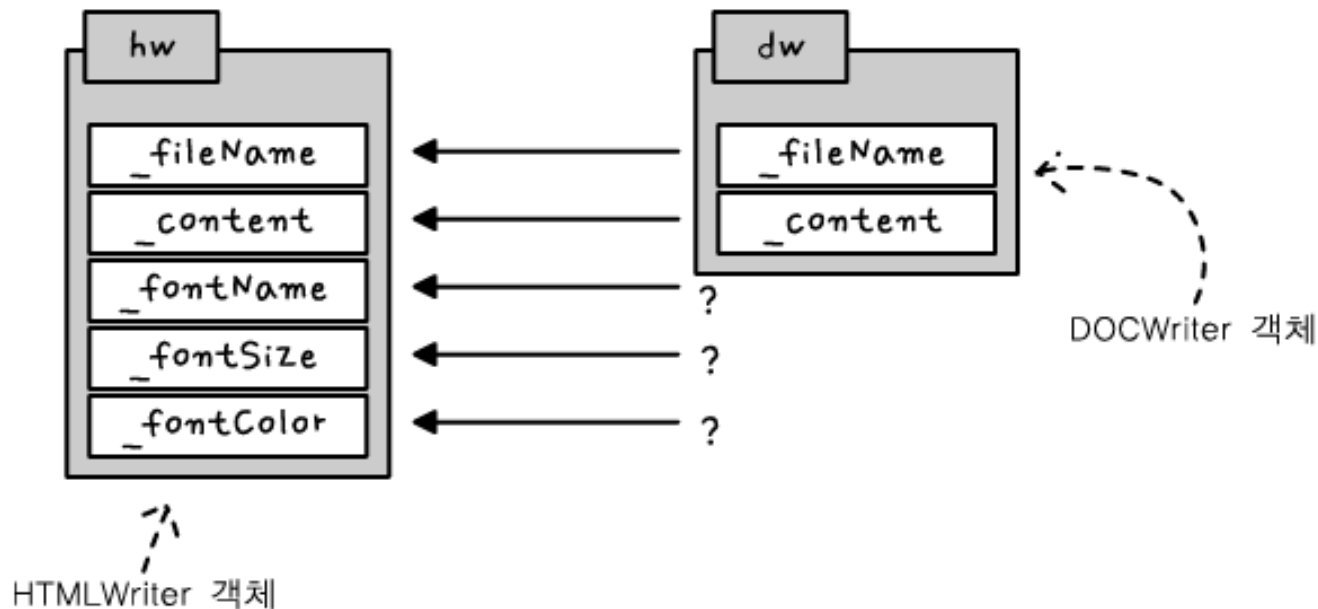
// HTMLWriter.cpp
HTMLWriter::HTMLWriter(const string& fileName, const string& content)
: DocWriter( fileName, content)    // 부모의 생성자 지정
{
    // 디폴트 폰트를 지정한다.
    _fontName = "굴림";
    _fontSize = 3;
    _fontColor = "black";
}
...
```

부모와 자식 객체 간의 대입(1)

- 부모 클래스의 객체를 자식 클래스의 객체에 대입할 수 **없음**

```
HTMLWriter hw;           // 자식 클래스의 객체 생성
DocWriter dw;            // 부모 클래스의 객체 생성

// 부모 클래스의 객체를 자식 클래스의 객체로 대입
hw = dw;                  // Error!!
```

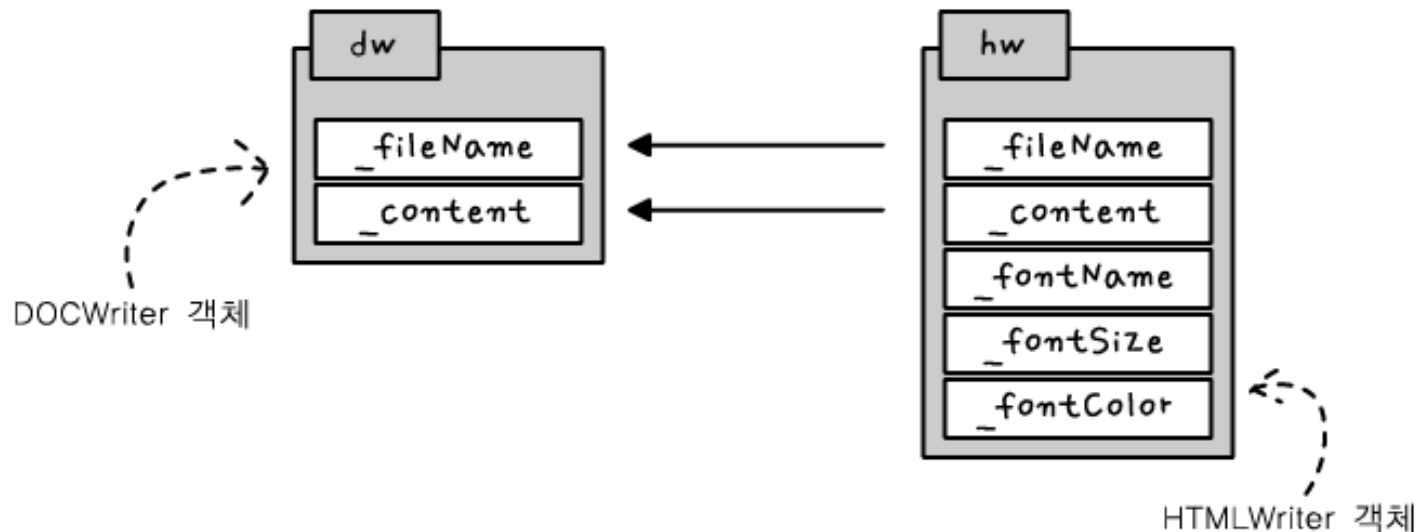


부모와 자식 객체 간의 대입(2)

- 자식 클래스의 객체를 부모 클래스의 객체에 대입할 수 **있음**

```
HTMLWriter hw;           // 자식 클래스의 객체 생성
DocWriter dw;            // 부모 클래스의 객체 생성

// 자식 클래스의 객체를 부모 클래스의 객체로 대입
dw = hw;                  // OK
```

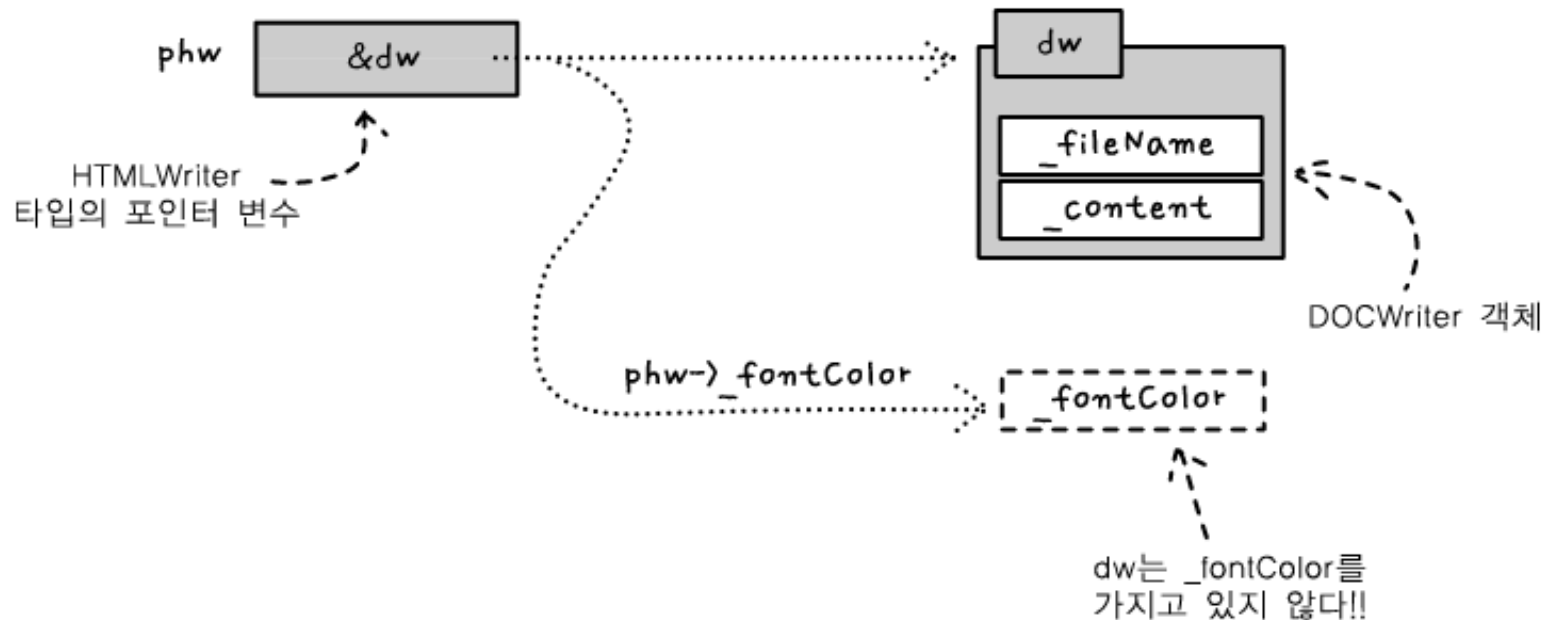


포인터, 레퍼런스의 형변환(1)

- 자식 클래스의 포인터로 부모 객체를 가리킬 수 **없음**

```
DocWriter dw; // 부모 클래스의 객체 생성

// 자식 클래스의 포인터 변수로 부모 객체를 가리킨다.
HTMLWriter* phw = &dw; // Error!!
```

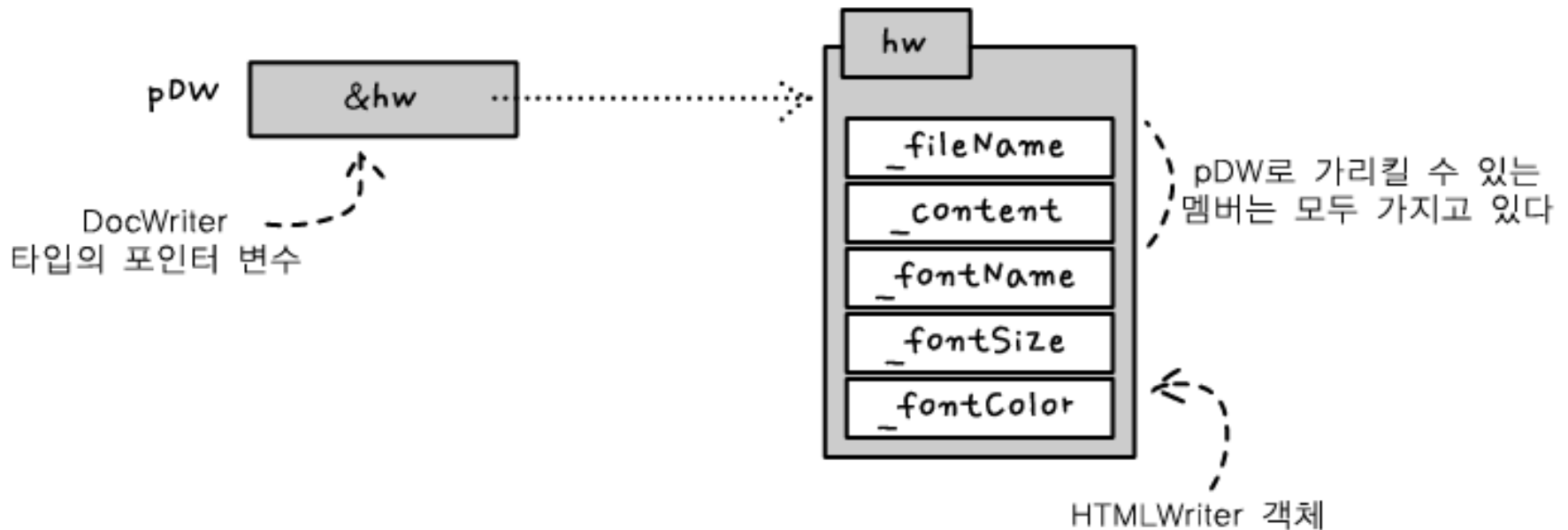


포인터, 레퍼런스의 형변환(2)

- 부모 클래스의 포인터로 자식 객체를 가리킬 수 있음

```
HTMLWriter hw; // 자식 클래스의 객체 생성

// 부모 클래스의 포인터 변수로 자식 객체를 가리킨다.
DocWriter* pDW = &hw; // OK
```



포인터, 레퍼런스의 형변환(3)

- 레퍼런스의 경우도 포인터와 동일한 규칙
 - 자식 클래스의 레퍼런스로 부모 객체를 참조할 수 **없음**

```
DocWriter dw;           // 부모 클래스의 객체 생성

// 자식 클래스의 레퍼런스 변수로 부모 객체를 참조한다.
HTMLWriter& hw = dw; // Error!!
```

- 부모 클래스의 레퍼런스로 자식 객체를 참조할 수 **있음**

```
HTMLWriter hw;           // 자식 클래스의 객체 생성

// 부모 클래스의 레퍼런스 변수로 자식 객체를 참조한다.
DocWriter& dw = hw;      // OK
```

접근 제어(1)

- 접근 제어 키워드

	자신의 멤버 함수에서 접근	자식 클래스의 멤버 함수에서 접근	외부에서 접근
private 멤버	O	X	X
protected 멤버	O	O	X
public 멤버	O	O	O

(O: 접근 가능, X: 접근 불가)

- 접근 제어 가이드라인

- 외부로부터 숨겨야 하는 멤버는 protected로 지정한다.
- 그 밖의 경우는 public으로 지정한다.
- 반드시 자식 클래스에 숨기고 싶다면 private로 지정한다.

접근 제어(2)

- 상속과 관련해서 접근 제어 키워드 실험

```
class Parent
{
    private:
        int priv;
    protected:
        int prot;
    public:
        int pub;
};

class Child : public Parent
{
    public:
        void AccessParents()
        {
            int n;
            // 부모의 멤버에 접근을 시도
            n = priv; // 실패
            n = prot; // 성공
            n = pub;      // 성공
        }
};
```

포함과 상속의 구분

- Has-a 관계와 Is-a 관계
 - Has-a 관계 : A 가 B 를 가지고(포함하고) 있는 관계
 - 예) 자동차는 타이어를 가지고 있다.
 - Is-a 관계 : A 가 B 인 관계
 - 예) 사과는 과일이다.
- 포함과 상속을 구분해서 사용하기 위한 가이드라인
 - Has-a 관계의 경우에는 포함을 사용한다.
 - Is-a 관계의 경우에는 상속을 사용한다.

상속의 연습 - 도형 class 만들기

- Shape Class 정의하기

- 멤버변수: `float _x, _y`
- 생성자: `Shape(float x, float y)` ← `_x`와 `_y`의 값을 설정
- 멤버함수: `void Draw() const`
 - 아래와 같은 내용 출력
[SHAPE] position = ('_x값', '_y값')

- Rectangle Class 정의하기

- Shape Class로부터 상속
- 멤버변수: `float _width, _height`
- 생성자: `Rectangle(float x, float y, float w, float h)`
- 멤버함수: `void Draw() const`
 - 아래와 같은 내용 출력
[RECTANGLE] position = ('_x값', '_y값'), size = ('_width', '_height')

상속의 연습 - 도형 class 만들기

- Circle Class 정의하기

- Shape Class로부터 상속
- 멤버변수: `float _radius`
- 생성자: `Circle(float x, float y, float radius)`
- 멤버함수: `void Draw() const`
 - 아래와 같은 내용 출력
`[CIRCLE] position = ('_x값', '_y값'), radius = '_radius'`

상속의 연습 - 도형 class 만들기

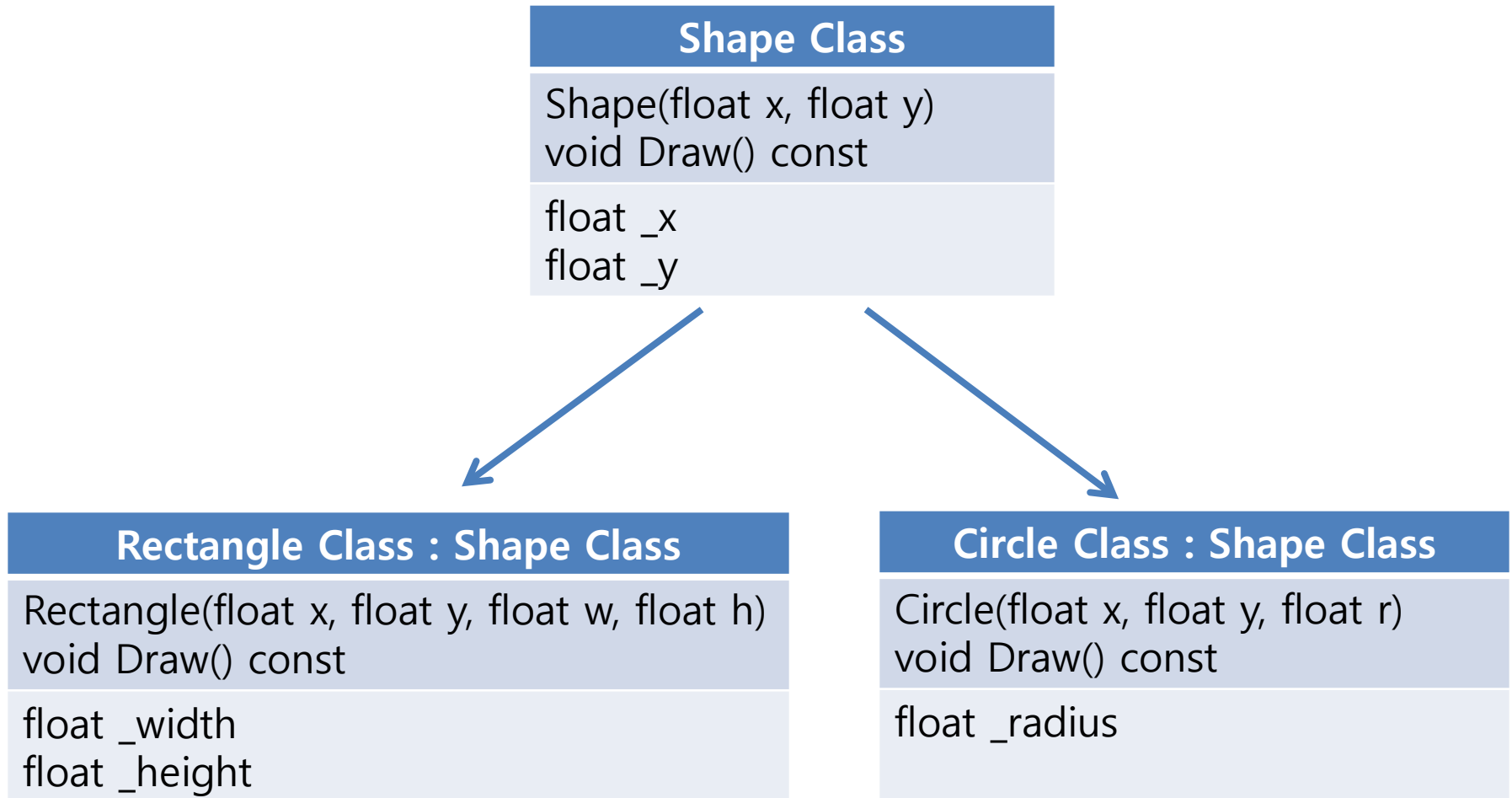
- 도형 Class 테스트

```
int main()
{
    Shape a(100,40);
    Rectangle b(120,40,50,20);
    Circle c(200,100,50);

    a.Draw();
    b.Draw();
    c.Draw();

    return 0;
}
```

도형 Class의 계층도

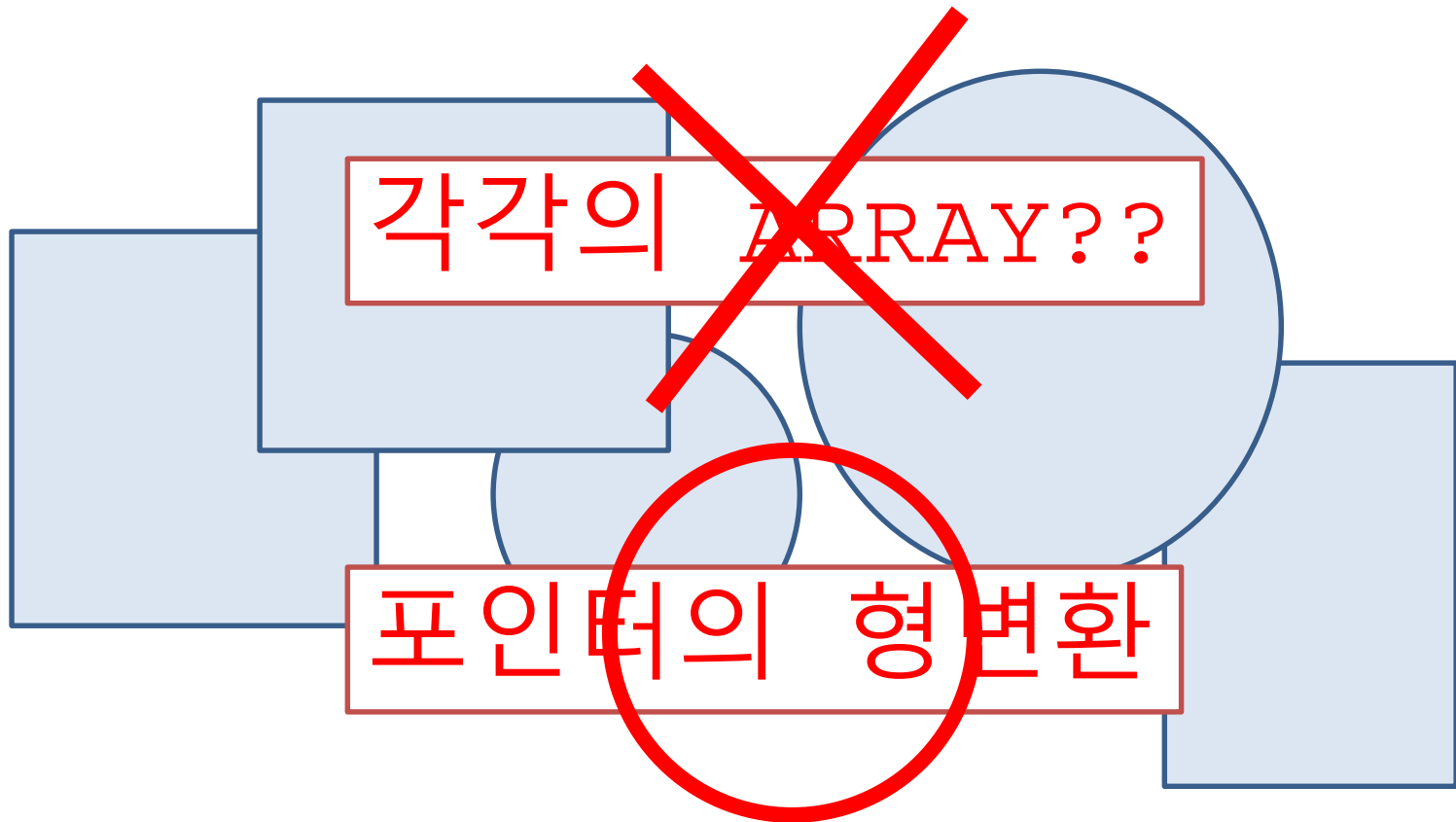


가상함수와 오버라이딩

(virtual function & overriding)

그림 그리는 프로그램

- 여러 가지 형태의 도형을 저장하고 싶다

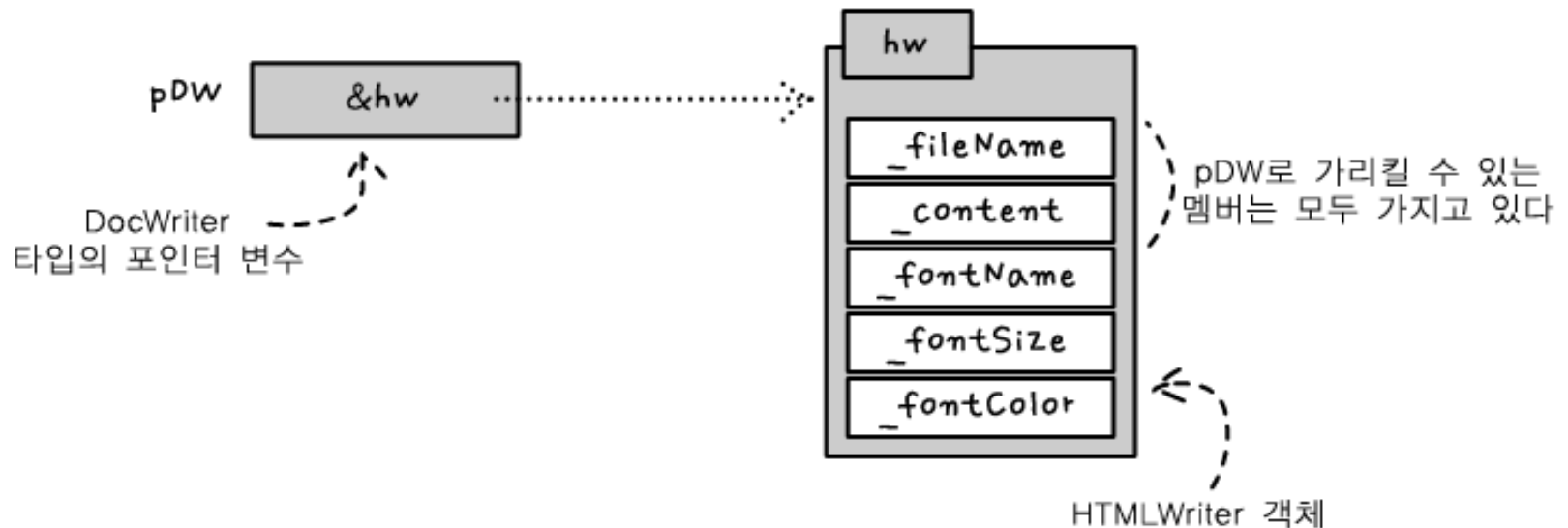


포인터의 형변환

- 부모 클래스의 포인터로 자식 객체를 가리킬 수 **있음**

```
HTMLWriter hw; // 자식 클래스의 객체 생성

// 부모 클래스의 포인터 변수로 자식 객체를 가리킨다.
DocWriter* pDW = &hw; // OK
```



포인터의 형변환 이용

- 1. 부모(base) 클래스 포인터의 배열을 설정
- 2. 필요 시마다 new를 이용 자식class 생성
- 3. 생성된 자식 class의 주소를 부모 클래스 포인터에 할당
- 4. 다 사용 했으면 delete를 이용 메모리 해제

다양한 클래스의 객체를 배열에 담기(1)

- 도형 클래스의 객체들을 배열에 담아서 사용하는 예

```
Shape* shapes[5] = {NULL};

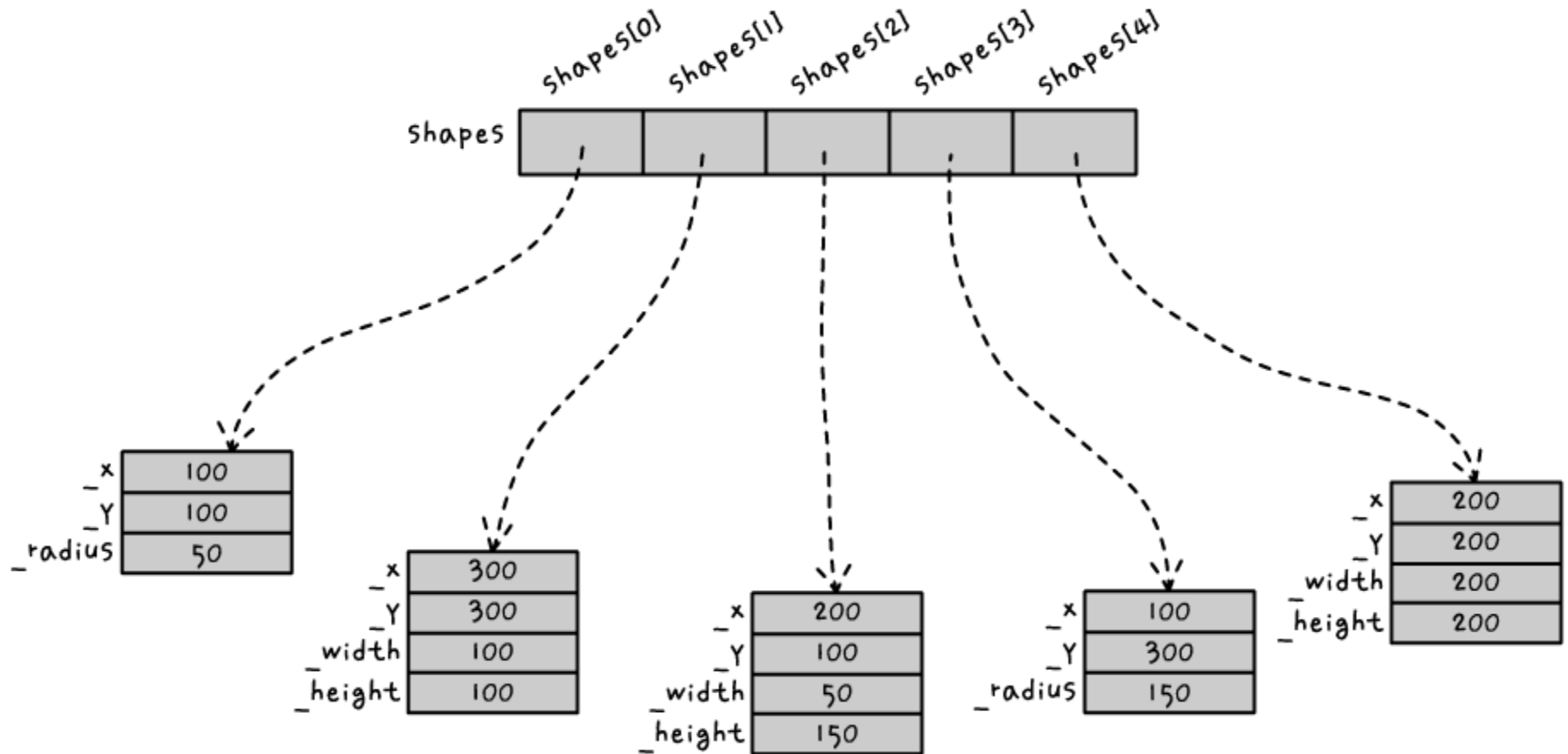
shapes[0] = new Circle( 100, 100, 50);
shapes[1] = new Rectangle( 300, 300, 100, 100);
shapes[2] = new Rectangle( 200, 100, 50, 150);
shapes[3] = new Circle(100, 300, 150);
shapes[4] = new Rectangle( 200, 200, 200, 200);

for (int i = 0; i < 5; ++i)
    shapes[i]->Draw();

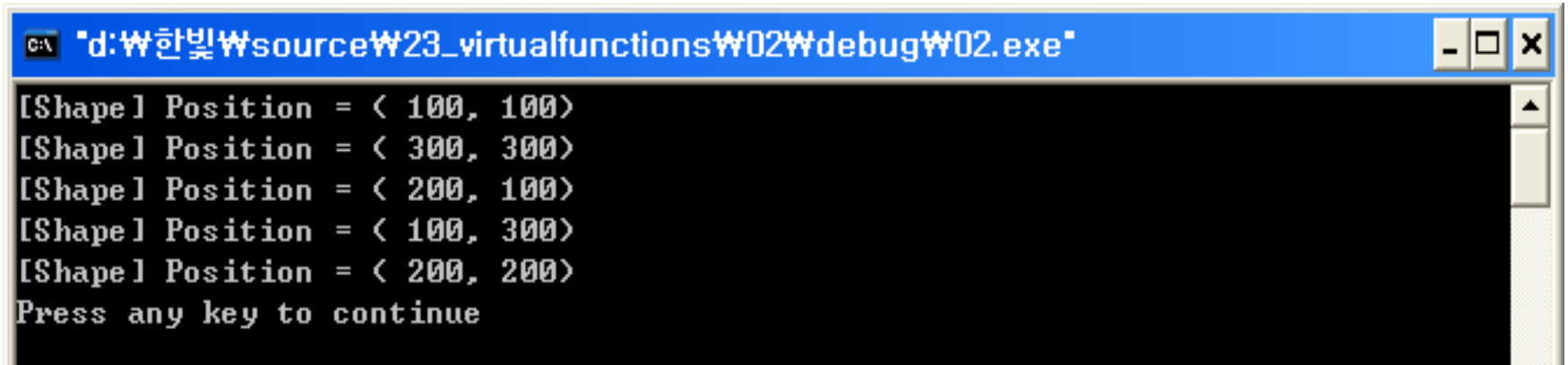
for (i = 0; i < 5; ++i)
{
    delete shapes[i];
    shapes[i] = NULL;
}
```


다양한 클래스의 객체를 배열에 담기(2)

- 배열과 객체들의 메모리 구조



Draw의 실행 결과가 이상하다??



```
C:\> "d:\한빛\source\23_virtualfunctions\02\debug\02.exe"

[Shape] Position = < 100, 100>
[Shape] Position = < 300, 300>
[Shape] Position = < 200, 100>
[Shape] Position = < 100, 300>
[Shape] Position = < 200, 200>
Press any key to continue
```

- 부모(base) 클래스의 함수가 호출됐다?!

가상 함수

- 부모 클래스의 포인터로 자식 클래스를 가르킬 경우:
 - 같은 이름의 함수의 실행은 포인터 타입이 우선
- 객체의 실제 내용에 따라 불릴 순 없을까?
 - 함수가 컴파일 시에 결정되지 않고, 런타임 시 결정
 - 멤버 함수의 동적인 선택 (동적바인딩)!
 - 다형성 (Polymorphism): 객체가 자신의 본래 정보와 동적으로 연결되는 것

➔ 가상함수

가상함수의 선언

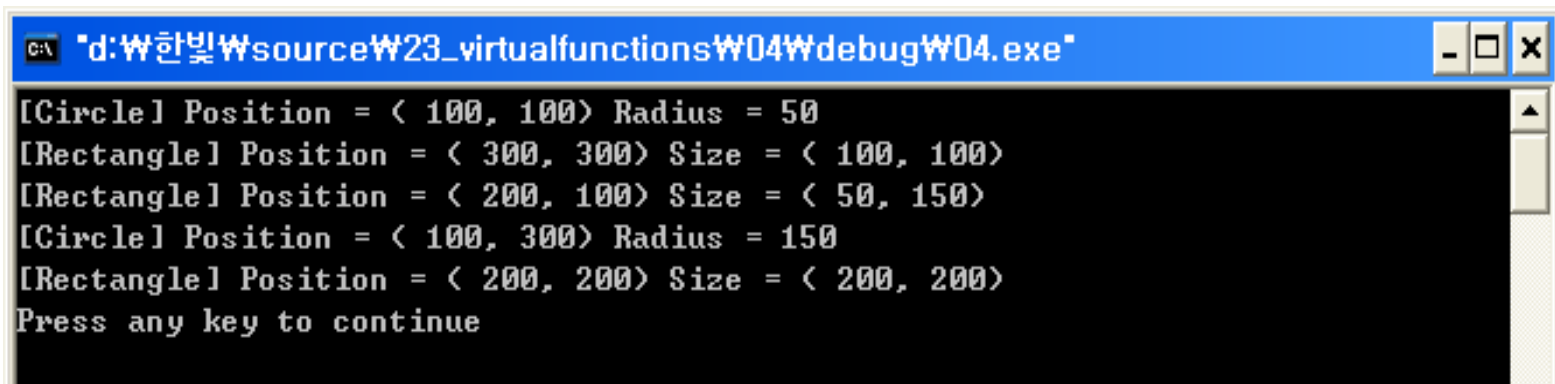
- 부모 클래스에서 함수 선언부분에 **virtual** 이라고 추가
- 가상 함수도 상속 되기 때문에 자식에게 키워드를 붙일 필요 없음

```
class parent
{
    public:
        virtual void DoIt();
};

class child : class parent
{
    public:
        void DoIt();
};
```

Draw 문제의 해결

- shape 클래스의 Draw() 선언부분 수정
 - **virtual** void Draw() const



```
C:\ "d:\한빛\source\23_virtualfunctions\04\debug\04.exe"
[Circle] Position = < 100, 100> Radius = 50
[Rectangle] Position = < 300, 300> Size = < 100, 100>
[Rectangle] Position = < 200, 100> Size = < 50, 150>
[Circle] Position = < 100, 300> Radius = 150
[Rectangle] Position = < 200, 200> Size = < 200, 200>
Press any key to continue
```

다형성(Polymorphism)과 가상함수(1)

- 다른 분야에서의 다형성의 의미

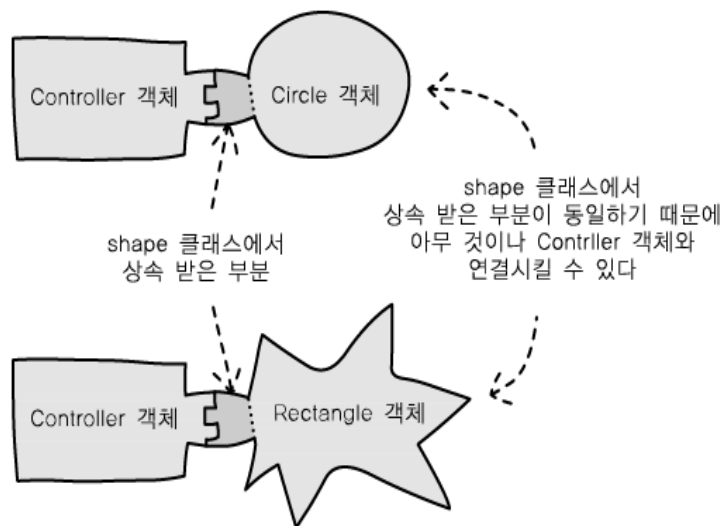
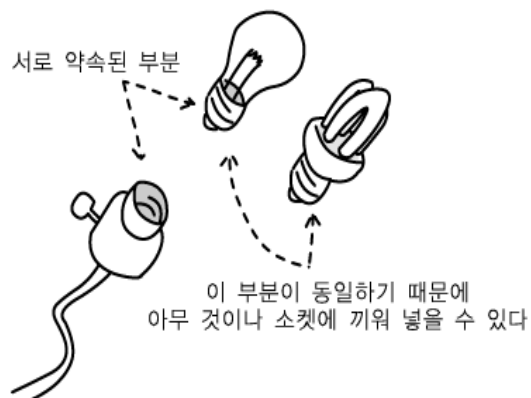
생물학	동일종의 생물이면서 형태나 성질이 다르게 보이는 다양성. 생물은 본래 동일종이라도 완전히 일치하는 개체는 거의 없으므로, 이 말은 상대적으로 현저한 차이가 있을 경우에 한해서 사용한다. 다만, 암수의 성별에 따른 2차 성징(二次性徵)의 차이에는 쓰지 않는다.
화학	화학조성이 같은 물질로써 결정구조를 달리하는 것. 다형(多形), 동질이형(同質異形), 동질다상(同質多像), 동질이정(同質異晶)이라고도 한다. 동질다상은 결정구조의 수에 따라 동질이상(同質二像), 동질삼상(同質三像) 등으로 나뉜다.

- 객체지향 프로그래밍에서의 다형성:
 - 타입에 관계 없이 동일한 방법으로 다룰 수 있는 능력
 - 예) Circle이나 Rectangle 객체들을 타입에 상관 없이 Shape 객체처럼 다룬다.

다형성(Polymorphism)과 가상함수(2)

- 다형성은 객체간의 결합(Coupling)을 약하게 만들어서, 객체 간의 연결을 유연하게 해준다.

```
// 도형을 원점으로 이동하는 함수  
void Controller::MoveToOrigin( Shape* p )  
{  
    p->Move( 0, 0 );  
    p->Draw();  
}
```



다양한 종류의 멤버 함수

- 상속과 관련된 멤버 함수의 종류
 - 일반적인 멤버 함수
 - 가상 함수
- 어떤 종류의 멤버 함수를 만들까?
 - 기본형태 → 멤버 함수
 - 다형성 → 가상 함수

오버로딩과 오버라이딩

- 오버로딩(Overloading)
 - 호출 인자가 다를 경우에 알맞은 것을 선택
 - ex) `void Dolt();`
 `void Dolt(int i);`
- 오버라이딩(Overriding)
 - 부모 객체의 함수를 자식 객체에서 재정의
 - ex) `parent::Dolt();`
 `child::Dolt();`

Homework #1 – Due: 03/24

- C++의 클래스 디자인 및 상속의 연습
 - 클래스 정의
 - 멤버 변수 (Member variables)
 - 멤버 함수 (Member functions)
 - 생성자 (Constructor) & 소멸자 (Destructor)
 - 클래스의 상속 (Inheritance)
- Code 제출시
 - 동작하는 code를 제출 (테스트 시 build가 안되면 **0점**)
 - Project file + source codes를 ZIP으로 압축하여 **ClassNet**에 제출
 - Debug 및 Release 폴더 제거
 - Visual Studio **2015**만 사용
- 주의 사항
 - 친구들과 활발한 토의와 토론은 권장. 단, 보고서나 코딩은 스스로 할 것
 - 보고서나 source code 의 copy 적발 시 원본 제공자와 복사자 모두 숙제 점수 **0점** 처리
 - 제출 시간 엄수: 1일 늦으면 전체 **50%** 감점, 2일이상은 **0점** 처리!

Q & A