

# 객체 지향 프로그래밍 응용

## Chap 1. 사전학습



2013.09.06.

오 병 우

컴퓨터공학과

# 기술 동향

## □ 프로그래밍 방식의 변천

### ◆ 기계중심의 Stored-Procedure

- 기계어, 어셈블리어

### ◆ 구조적 프로그래밍

- Pascal, C
- 잘 정의된 제어구조, 코드 블록, GOTO문 사용억제, 순환호출 (recursion)과 지역변수를 지원하는 독립형 부 프로그램
- 기능별 모듈화

### ◆ 객체지향 프로그래밍

- C++
- 객체 단위로 모듈화
- Data와 Control 통합
- 복잡도 감소 목표
- Run-time시 정보 처리가 많아 구조적 프로그래밍 방식보다 실행속도가 느림

# C vs. C++

□ “계산기 프로그램”으로 살펴보는 C와 C++ 프로그래밍의 차이

◆ C: 기능에 따라 세분화

- 입력, 계산, 출력 기능 → 각 기능을 함수로 구현

◆ C++: 객체에 따라 세분화

- 키패드, LCD, CPU → 각 객체가 가져야할 특성에 따라 클래스로 구현

```
main()
{
    input();
    compute();
    output();
}
```



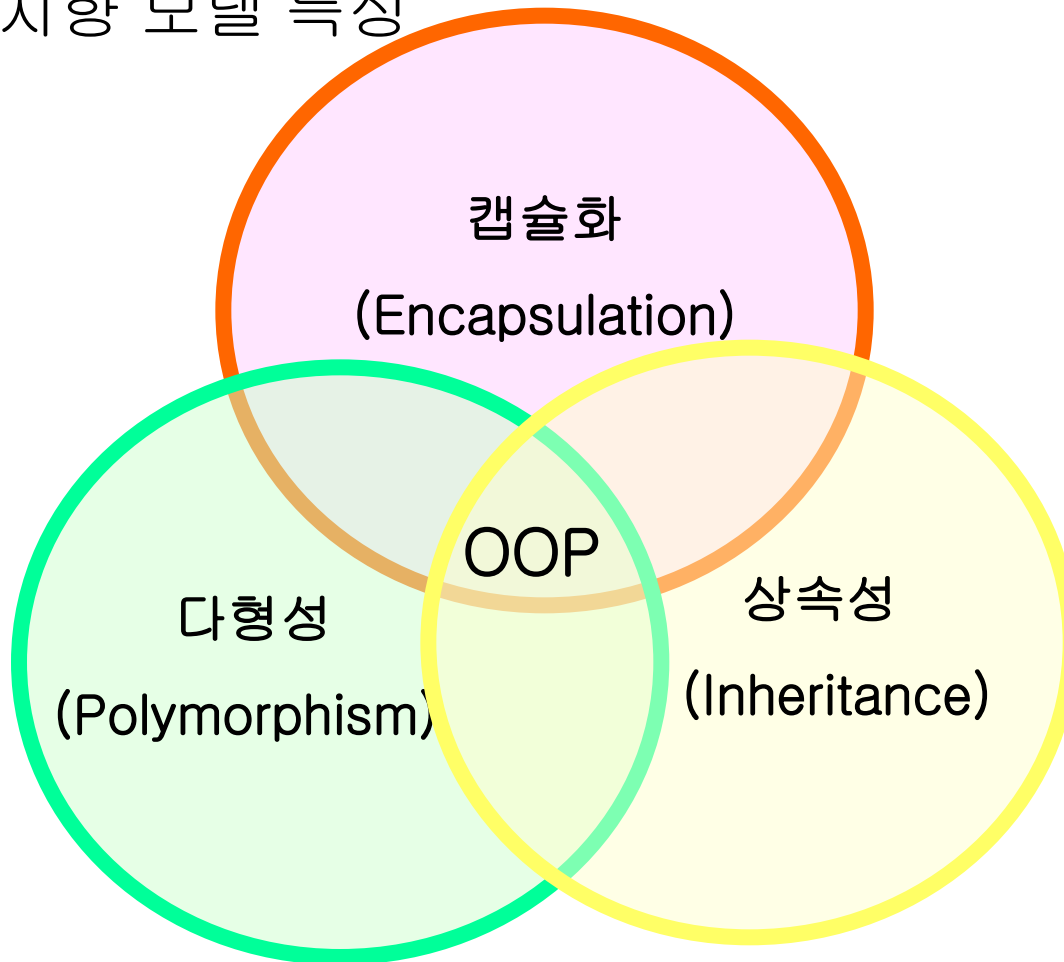
```
Class Keypad{
... // Keypad의 특성과 해야할 일 구현
};

Class LCD {
... // LCD의 특성과 해야할 일 구현
};

Class CPU {
... // CPU의 특성과 해야할 일 구현
};
```

# 객체 지향 모델

## □ 객체지향 모델 특성



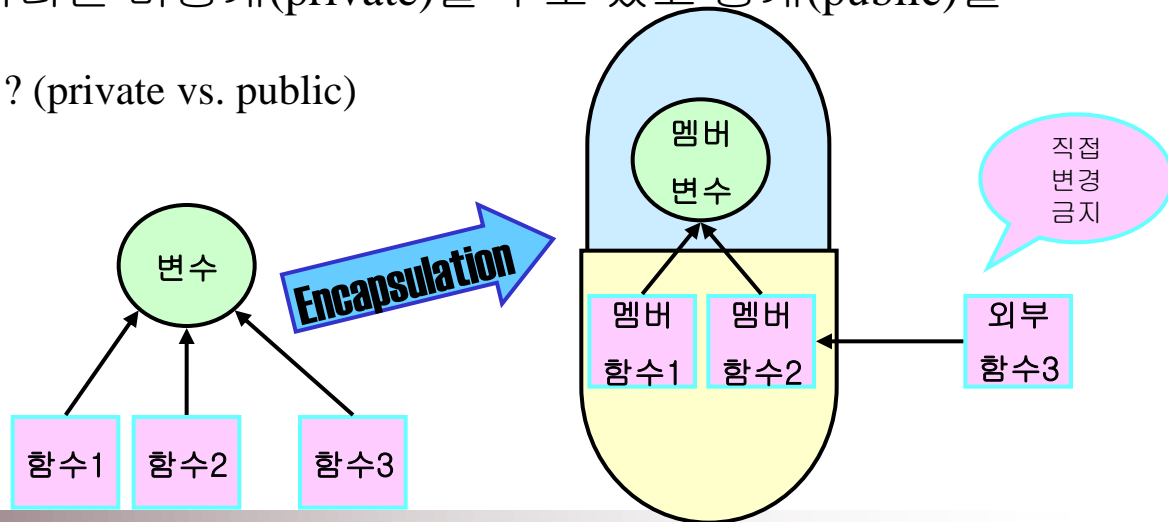
# Encapsulation

## □ 목적

- ◆ Information hiding
  - 복잡한 내부 구조 숨김
- ◆ Divide and Conquer
  - 프로그램 전체적인 관점에서 문제를 잘게 나누고 해결
- ◆ 프로그래머 실수 방지
  - 반드시 필요한 함수만 공개

## □ 설명

- ◆ 데이터(data)와 이를 조작하는 코드(code, control, function) 통합
- ◆ 객체 안의 코드와 데이터는 비공개(private)될 수도 있고 공개(public)될 수도 있음
  - class와 struct의 차이? (private vs. public)
- ◆ 예제) 오디오



# Polymorphism

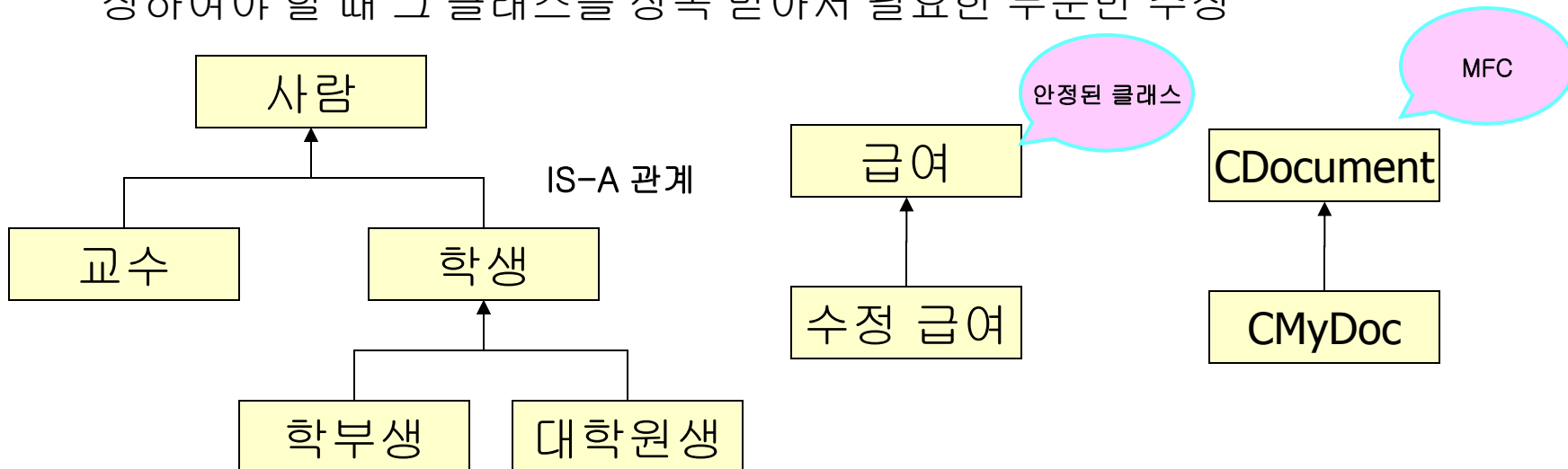
## □ 다형성 (polymorphism)

- ◆ 하나의 이름으로 두 가지 이상의 작동을 수행할 수 있는 성질
- ◆ 예제
  - 하나의 인터페이스에 여러 방법 지원
  - `abs()`, `labs()`, `fabs()` → `abs()`
- ◆ 함수 이름과 연산자 이름에 사용할 수 있음
  - 함수 오버로딩 (overloading)
  - 연산자 오버로딩
- ◆ Mangling을 통해 자연스럽게 구현됨
  - 리턴값, 파라미터, \_ 등을 함수 이름에 덧붙임

# Inheritance

## 상속성(inheritance)

- ◆ 클래스가 상위 클래스의 특성을 이어받을 수 있는 특성
- ◆ 서로 연관 있는 클래스들을 계층적인 구조로 표현할 수 있으며, 자식 클래스(child class)로 갈수록 일반적인 것에서 특수한 것으로 이동
- ◆ 실제 업무에서는 개발과 테스트를 거친 안정된 클래스를 사용하다가 수정하여야 할 때 그 클래스를 상속 받아서 필요한 부분만 수정



Class Diagram

# Inheritance (계속)

◆ Base Class, Super Class 또는 Parent Class

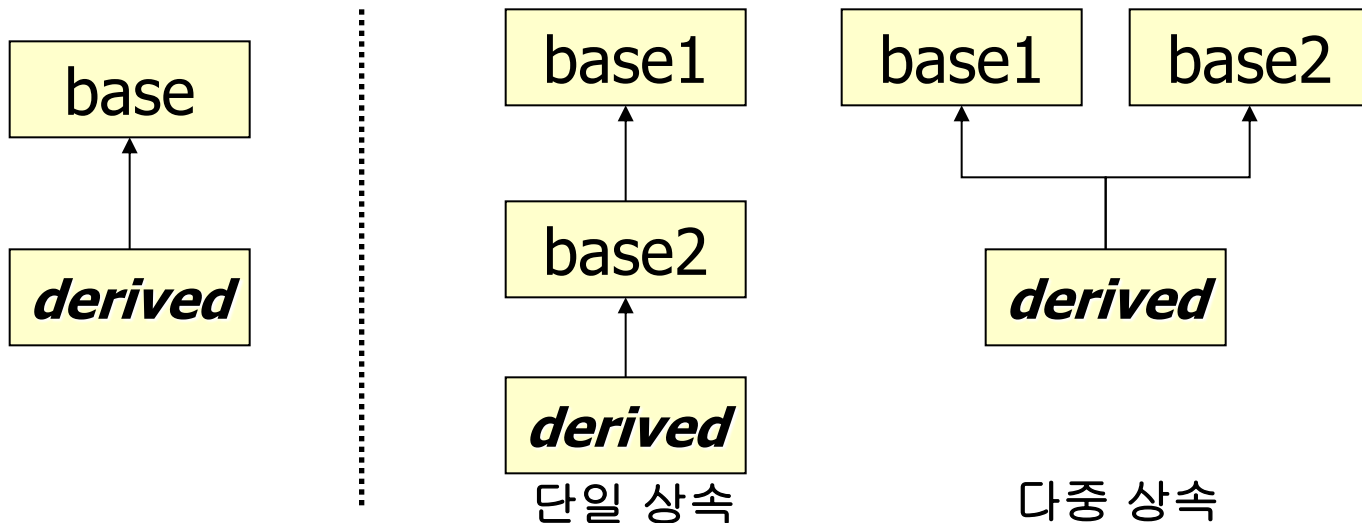
- 상속 해주는 측

◆ Derived Class 또는 Child Class

- 상속 받는 측

◆ 상속의 유형

- 단일상속 : 파생 클래스가 하나의 기본 클래스를 상속
- 다중상속 : 파생 클래스가 여러 개의 기본 클래스 들을 상속





# 클래스 및 인스턴스

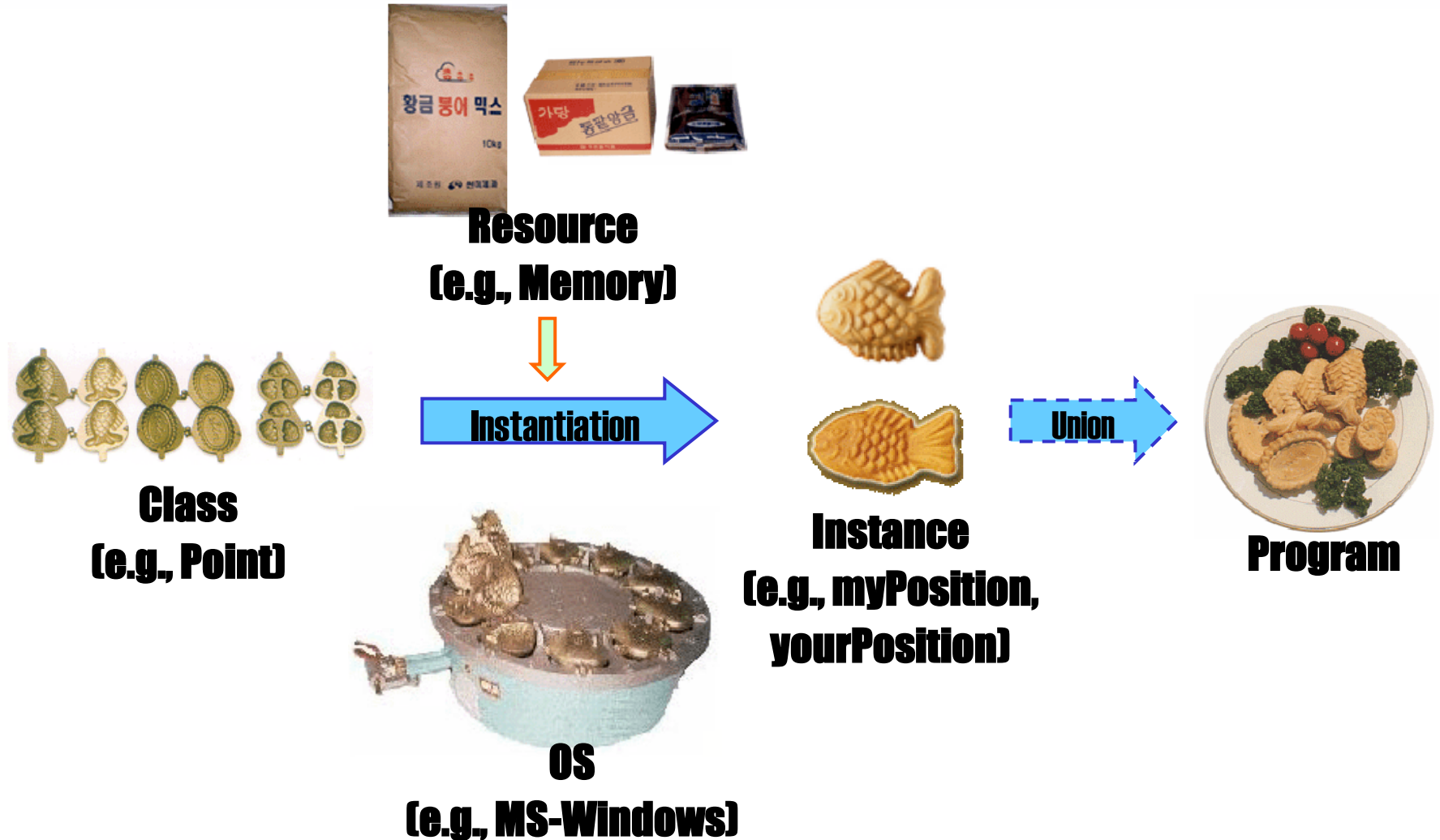
## □ 클래스

- ◆ 관심 대상의 특성을 정의
  - Data 및 Control (Function) 통합
- ◆ 사용자 정의 타입
- ◆ 예제) 붕어빵 찍는 기계

## □ 인스턴스(Instance) 또는 객체(Object)

- ◆ 클래스에 의해 실체화되어 생성된 것
  - 클래스만 정의해 놓고 쓰지 않는다면 효용없음
  - Notepad.exe라는 프로그램을 만들어 놓고 실행한 메모장 프로그램들
- ◆ 메모리에 할당된 공간을 가지는 변수
- ◆ 예제) 붕어빵

# 클래스 및 인스턴스 예제



# C++에서의 클래스 및 인스턴스



```
// Point.h
class Point {
public:
    // 멤버 변수
    int m_nX;
    int m_nY;

    // 멤버 함수
    void show();
};
```

class



```
// main.cpp
#include "Point.h"

Point myPosition,
yourPosition;
```

instance

# 멤버 접근 지시자

## 접근 지시자

- ◆ 3종류: public, protected (상속), private
- ◆ 멤버 접근 허용 지정
- ◆ 클래스의 캡슐화를 고도화

Default  
cf) struct

```
class Point {
private:
    // 멤버 변수
    int m_nX;
    int m_nY;

public:
    // 멤버 함수
    void show();
};
```

# 클래스 내부와 외부

## 내부

- ◆ 변수명 또는 함수명으로 사용 가능

```
void Point::show()
{
    cout << "X=" << m_nX << " Y=" << m_nY << "\n";
}
```

## 외부

- ◆ 인스턴스명 + 점(.) + 공개된 (public) 변수명 또는 함수명

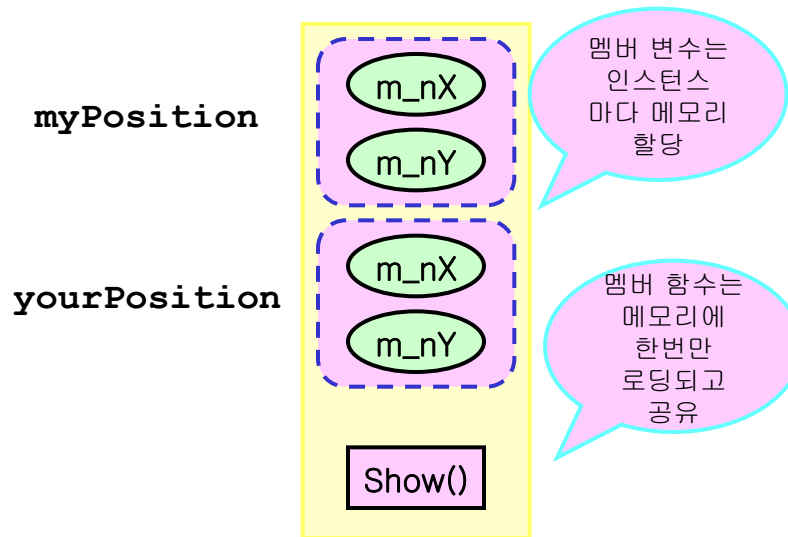
```
void main()
{
    Point myPosition;
    myPosition.show();
    myPosition.m_nX = 10;
}
```

포인터일  
경우는 ->

Error

# 인스턴스들 사이의 멤버 함수 공유

- 클래스의 인스턴스 생성
  - ◆ 메모리 공간 할당
- 인스턴스 생성
  - ◆ 멤버변수는 각 인스턴스에 따로 생성됨
  - ◆ 멤버함수는 각 인스턴스들이 공유



# Static Member Variable & Function

## 인스턴스 생성

- ◆ 멤버변수는 각 인스턴스에 따로 생성됨
  - 정적 멤버 변수(static member variable)는 공유됨 (메모리도 절약)
- ◆ 멤버함수는 각 인스턴스들이 공유
  - 정적 멤버 함수(static member function)는 인스턴스 없이 호출 가능

## Member Function

```
class Point {
public:
    void Dimension();
};

void Point::Dimension()
{
    printf("2D\n");
}

int main(int argc, char* argv[])
{
    Point t;
    t.Dimension();
    return 0;
}
```

## Static Member Function

```
class Point {
public:
    static void Dimension();
};

void Point::Dimension()
{
    printf("2D\n");
}

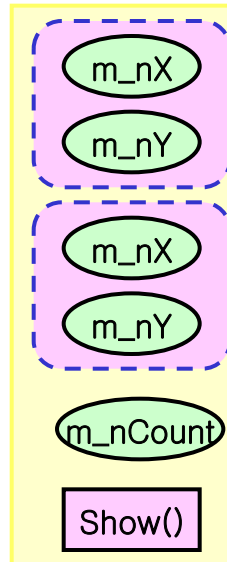
int main(int argc, char* argv[])
{
    Point::Dimension();
    return 0;
}
```

```
// Point.h
class Point {
private:
    // 멤버 변수
    int m_nX;
    int m_nY;
    static int m_nCount;
public:
    // 멤버 함수
    void show();
};
```

Static Member Variable

별도의 변수 선언 필요

```
// Point.cpp
int Point::m_nCount = 0;
// 또는 int Point::m_nCount = 0;
```



정적 멤버 변수는 메모리에 한번만 로딩되고 공유

# 데이터 감추기

## □ 지역변수 (Local Variable)

### ◆ 함수(블록)내에서 선언된 변수

- 함수 시작하는 순간에 자동으로 메모리 (스택) 할당
- 함수가 종료되면 자동으로 메모리 해제

```
for (int i = 0; i < 3; i++) {
    int j = 0;
    printf("%d ", j++);
}

printf("%d\n", i);
printf("%d\n", j); // error C2065: 'j' : undeclared identifier

// Output (if possible): 0 0 0 3 ?
```

## □ 전역변수 (Global Variable)

### ◆ 함수 밖에서 선언된 변수

- 프로그램 시작시 할당 종료시 해제
- 여러 함수에서 참조

## □ 영속변수 (Persistent Variable)

- ◆ 프로그램이 종료되어도 해제되지 않음
- ◆ 객체 지향 데이터베이스

Variable	Life Cycle
Local	Function
Global	Program Run
Persistent	Program Installation

```
int global;

void function()
{
    int local;
}
```

Stack에 공간 할당

```
void function()
{
    static int count = 0;

    printf("%d\n", count++);
}
```

Static Variable - Heap에 할당



# 전역 변수의 문제점

## □ 전역변수 증가

### ◆ 관리 어려움

- Naming Convention

- 많은 변수 이름들은 프로그래머에게 부담 가중 및 실수 유발 증가

### ◆ 변경 어려움

- 여러 함수에서 사용하므로 type 변경, 이름 변경, 구조 변경시에 사용하는 모든 함수 변경 필요

### ◆ 가능한한 사용하지 말 것

# 클래스의 데이터 캡슐화

## ■ 전역 변수 문제점 해결

- ◆ 관련 있는 데이터 묶어서 캡슐화

## ■ 캡슐화된 데이터 감추기

### ◆ private, protected

- 외부에서 사용 불가능
- protected : 상속시 child클래스에서 사용 가능
- 클래스 선언시 접근 지시자를 지정하지 않으면 default는 private

### ◆ public

- 외부에서 사용 가능

### ◆ 변수는 private 또는 protected로 선언

### ◆ public getter 및 setter 함수 만들어서 사용

- e.g.) GetPosition(), SetPosition(x, y)

# 생성자 및 소멸자 함수

## □ 클래스의 생성자와 소멸자

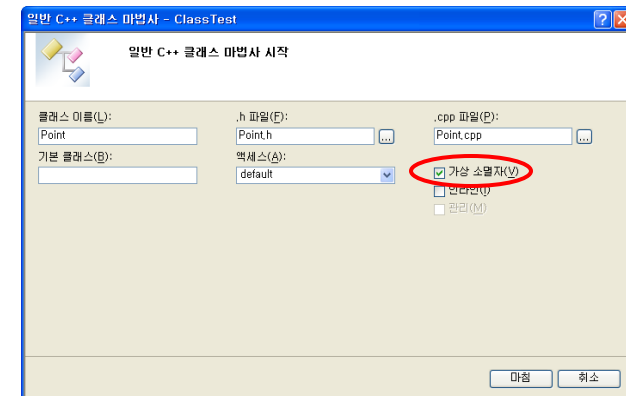
### ◆ 생성자 (Constructor)

- 클래스 이름과 같은 이름의 함수
- 개체가 만들어질 때 자동으로 호출되어 주로 초기화 수행
- return값 없음
- 매개인자 가질 수 있음
- 여러 개의 생성자 함수가 존재할 수 있음 (Overloading)
- 상위 클래스 생성자 자동 호출

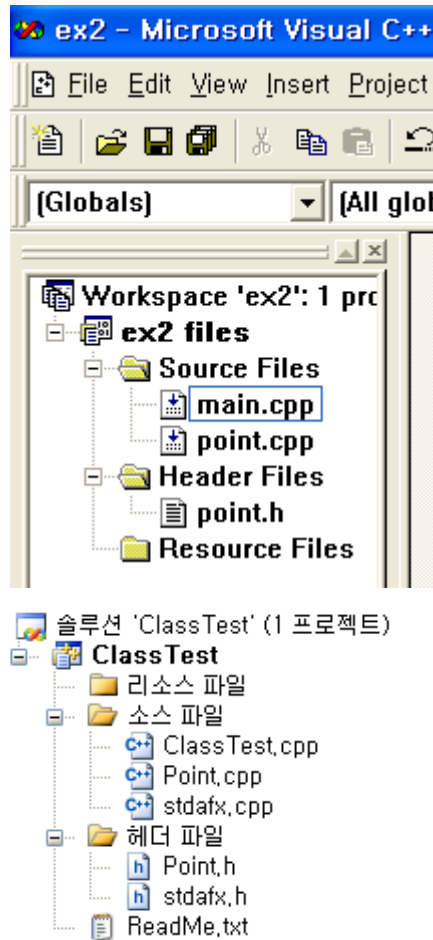
```
class Point {
...
    // Constructor
    Point();
    Point(int x, int y);
    // Destructor
    ~Point();
...
};
```

### ◆ 소멸자 (Destructor)

- 클래스 이름 앞에 '~' 붙은 함수
- 개체가 소멸될 때 자동으로 수행되어 주로 사용한 리소스 해제 수행
- return값 없음
- 매개인자 가질 수 없음
- 상위 클래스 소멸자 자동 호출
- 상속이 될 경우에 발생할 수 있는 문제를 대비하려면 virtual 사용 (슬라이드 p.28 참조)



# 클래스 정의 실습 #1



```
// main.cpp
#include <iostream.h>
#include "Point.h"
void main()
{
    Point p;
    cout << "X=" << p.m_nX << " Y=" << p.m_nY << "\n";
}
```

```
// Point.h
class Point {
public:
    int m_nX, m_nY;
    Point();
};
```

```
// Point.cpp
#include "point.h"
Point::Point()
{
    m_nX = 0;
    m_nY = 0;
}
```

# 클래스 정의 실습 #2

```
// main.cpp
#include <iostream.h>
#include "point.h"

void ShowCount()
{
    if (!Point::GetCount())
        cout << "Not Created\n";
    else
        cout << Point::GetCount() << ((Point::GetCount() > 1)? " Objects" : " Object") << " Created\n";
}

void main()
{
    ShowCount();

    Point p(7, 7), k;
    cout << "X = " << p.GetX() << ", Y = " << p.GetY() << "\n";
    ShowCount();
}
```

```
// point.h
class Point {
private:
    int m_nX, m_nY;

    static int m_nSharedCount;

public:
    Point() { m_nX = 0; m_nY = 0; m_nSharedCount++; };
    Point(int x, int y) { m_nX = x; m_nY = y; m_nSharedCount++; };

    int GetX() { return m_nX; };
    int GetY() { return m_nY; };
    static int GetCount() { return m_nSharedCount; };
};
```

```
// point.cpp
#include "point.h"

int Point::m_nSharedCount = 0;
```

# 연산자 오버로딩 (Overloading)

■ 기존 연산자의 기능을 다른 방식으로 작동하도록 재정의

◆ Polymorphism

■ 연산자의 유형

◆ 단항 연산자 (unary operator) : 오퍼랜드(operand)가 한 개로 구성되는 연산자

- ++, --, +, - ...
  - 전위형: ++a
  - 후위형: a++

◆ 이항 연산자 (binary operator): 오퍼랜드가 두 개로 구성되는 연산자

- +, -, \*, /, = ...
  - a + b

■ 불가능

◆ ., \*, ::, ?:

# 연산자 오버로딩 실습

```
// main.cpp
#include <iostream.h>
#include "point.h"

void ShowCount()
{
    if (!Point::GetCount())
        cout << "Not Created\n";
    else
        cout << Point::GetCount() << ((Point::GetCount() > 1)? " Objects" : " Object") << " Created\n";
}

void main()
{
    ShowCount();
    Point p(7, 7), k;
    ShowCount();

    cout << "p: X = " << p.GetX() << ", Y = " << p.GetY() << "\n";
    cout << "k: X = " << k.GetX() << ", Y = " << k.GetY() << "\n";
    ++k; // 단항 연산자 전위형
    cout << "k: X = " << k.GetX() << ", Y = " << k.GetY() << "\n";
    k++; // 단항 연산자 후위형
    cout << "k: X = " << k.GetX() << ", Y = " << k.GetY() << "\n";
    k = k + p; // 이항 연산자
    // k += p; // 이항 연산자
    cout << "k: X = " << k.GetX() <<

```

```
// point.cpp
#include "point.h"

int Point::m_nSharedCount = 0;

Point Point::operator+(Point &point)
{
    Point temp;
    temp.m_nX = m_nX + point.m_nX;
    temp.m_nY = m_nY + point.m_nY;
    return temp;
}

```

```
// point.h

class Point {
private:
    int m_nX, m_nY;

    static int m_nSharedCount;

public:
    Point() { m_nX = 0; m_nY = 0; m_nSharedCount++; };
    Point(int x, int y) { m_nX = x; m_nY = y; m_nSharedCount++; };

    int GetX() { return m_nX; };
    int GetY() { return m_nY; };
    static int GetCount() { return m_nSharedCount; };

    Point operator++() { return Point(++m_nX, m_nY); }; // 전위형 (++a)
    Point operator++(int) { return Point(m_nX, ++m_nY); }; // 후위형 (a++)
    Point operator+(Point &point);
};

```

# 상속성 및 함수 Overriding

- 기반 클래스에서 상속받은 멤버함수의 기능 대체
  - ◆ 상속 받은 멤버함수의 Prototype과 똑 같이 재 선언하여 함수를 작성함
- Overriding
  - ◆ 기반 클래스의 같은 이름 함수가 먼저 호출되지 않음
  - ◆ 기반 클래스의 같은 이름 함수를 explicitly 호출 가능

생성자,  
소멸자는?

```
class Point {
public:
    void show() { cout << "Point\n"; };
};

class Point3D : public Point{
public:
    void show() { cout << "Point3D - "; Point::show(); };
};
```



# 가상 함수

## □ 함수의 바인딩 (binding)

- ◆ 함수 호출부분에 함수가 위치한 메모리 번지를 연결시키는 것
  - 정적 바인딩 : 컴파일 단계에서 수행될 함수의 번지가 결정됨
  - 동적 바인딩 : 프로그램 실행시 수행될 함수의 번지가 결정됨

## □ 가상 함수

- ◆ 기반 클래스의 함수 선언에 `virtual` 키워드 추가
- ◆ 파생 클래스에서 그 함수 오버라이딩
  - run-time polymorphism 지원
- ◆ 기반 클래스 포인터 변수에 파생 클래스의 객체를 대입
  - 객체의 타입에 따라 알맞은 멤버함수 (파생 클래스의 함수) 수행

# 가상 함수

## 가상함수 사용

```
#include <iostream.h>

class base1 {
public:
    virtual void show() { cout << "base1" << endl; }
};

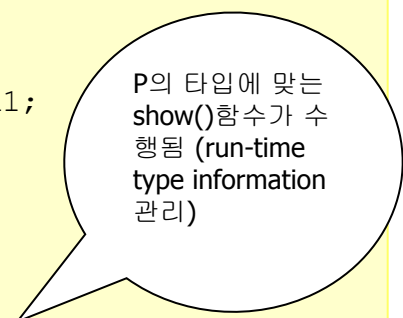
class derived1 : public base1{
public:
    void show() { cout << "derived1" << endl; }
};
```

```
Void main()
{
    base1 *p;
    base1 b1;
    derived1 d1;

    p = &b1;
    p->show();

    p = &d1;
    p->show();

}
```



P의 타입에 맞는  
show()함수가 수  
행됨 (run-time  
type information  
관리)

# 가상 함수 실습

class 로 변경하여 에러 확인

함수  
갖을 수  
있습니다

가상함수로 변경하여 결과 확인

상속도  
됩니다.

```

struct Point {
//class Point {
    int x, y;
    void show();
    //virtual void show();
};

void Point::show()
{
    printf("show\n");
}

struct Point3D : public Point {
    int z;
    void show();
};

void Point3D::show()
{
    printf("%d, %d, %d\n", x, y, z);
}

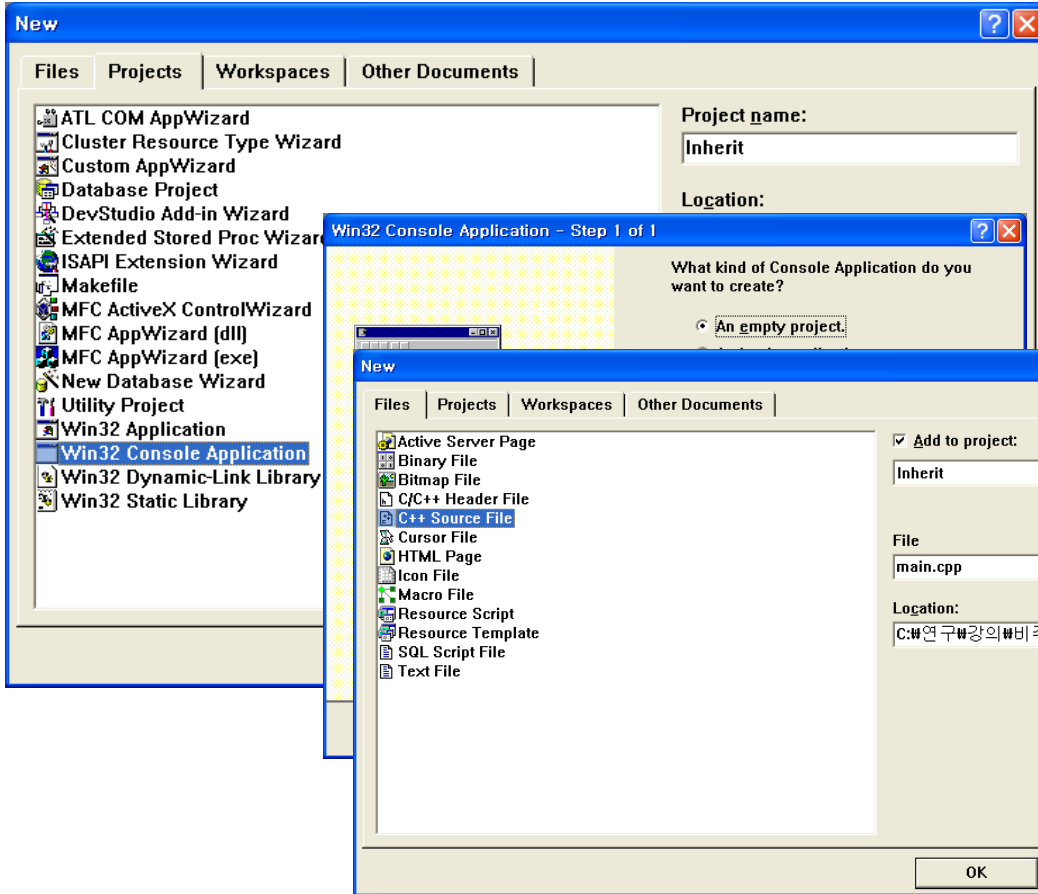
int main(int argc, char* argv[])
{
    Point3D t;
    t.x = 10;
    t.y = 20;
    t.z = 30;
    t.show();

    Point *p = &t;
    p->show();

    return 0;
}
    
```

# 소멸자의 가상함수 실습

소멸자에  
virtual을 붙여  
주어야 상위  
클래스 포인터로  
레퍼런스된  
객체도 상위  
소멸자 호출됨



```
#include <stdio.h>

class Point {
    int x, y;
public:
    // virtual이 되는지 확인
    Point() { printf("Point::Point()\n"); x = 0; y = 0; };
    // virtual로 변경하여 결과 확인
    ~Point() { printf("Point::~Point()\n"); };
    //virtual ~Point() { printf("Point::~~Point()\n"); };
};

class Point3D : public Point {
    int z;
public:
    Point3D() { printf("Point3D::Point3D()\n"); z = 0; };
    ~Point3D() { printf("Point3D::~~Point3D()\n"); };
};

main()
{
    printf("====Start - Point 3D====\n");
    Point3D location;
    printf("====End - Point 3D====\n");

    printf("====Start - Point*====\n");
    Point *p = new Point3D;
    printf("====End - Point*====\n");

    printf("====Start - delete Point*====\n");
    delete p;
    printf("====End - delete Point*====\n");

    printf("====Start - delete Point3D====\n");
}
```

# Inline 함수

## □ 코드가 통째로 복사

- ◆ 점프 아님
- ◆ 빈번히 호출되는 함수를 호출 처리 없이 바로 실행
- ◆ 함수 크기가 매우 작은 경우

## □ 장점 vs. 단점

- ◆ 수행속도 vs. 실행 파일 크기

## □ 클래스 헤더 파일에 함수 선언 & 구현

```
class Point {
    ...
    int GetCount() { return m_nSharedCount; };
};
```

## □ 클래스 소스 파일에 구현시 함수 앞에 inline 명시

```
inline int Point::GetCount()
{
    return m_nSharedCount;
}
```

# Reference

## Call by Reference

◆ 변수의 복사본을 만들지 않고 변수의 포인터를 넘김

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
};
```

## 가능한 사용하지 말 것

# Template

## □ 임의의 데이터 타입으로 교체 가능하도록

- ◆ 클래스 정의시에는 타입을 지정하지 않고
- ◆ 변수 선언시 타입 지정

```
template <class type> class Point
{
private:
    type m_X, m_Y;
public:
    void SetPosition(type x, type y);
    ...
};
```

```
template <class type>
void Point<type>::SetPosition(type x, type y)
{
    m_X = x;
    m_Y = y;
};
```

```
Point<double> dPosition;
Point<int> nPosition;
```

# Default Parameter Value

## □ 매개변수의 디폴트값 지정 가능

```
class Point {
public:
    int m_nX, m_nY;
    Point() { m_nX = 0; m_nY = 0; };
    Point(int x = 10, int y =20) { m_nX = x; m_nY = y; };
}; // warning C4520: 'Point' : multiple default constructors specified
```

적어도 한 개는  
디폴트값이  
없어야 성공

```
Point p; // error C2668: 'Point::Point' : ambiguous call to overloaded function
```



# 'this' Pointer

- 자기자신의 객체를 가리키는 포인터
  - ◆ 다른 클래스에 자기 자신을 매개 변수로 넘겨줘야 할 때 사용

```
class Where {
public:
    int data;
    void PrintPointer();
};
```

```
void Where::PrintPointer()
{
    cout << "주소 : " << this << "\n";
}
void main()
{
    Where a,b,c;

    a.PrintPointer();
    b.PrintPointer();
    c.PrintPointer();
}
```

# 'const' Variable & Function

## ▣ 프로그래머의 실수 방지

### ◆ Read-only Variable

```
const double pi = 3.141592;

pi = 10; // 에러
```

### ◆ Read-only Function

```
class Count {
    ...
    int GetCount() const { return m_nCount; };
    ...
};
```

# Summary

- Object Oriented Programming
- Encapsulation
- Polymorphism
- Inheritance
- Class and Instance
- Constructor and Destructor
- Static Member Variable and Function
- Function and Operator Overloading
- Function Overriding
- Virtual Function
- Inline Function
- Template
- Default Parameter Value
- 'this' Pointer
- 'const' Variable and Function