

## 9장 상속과 다형성

- # 기본형의 형변환 복습
- # 서로 다른 클래스 객체들 사이의 대입
- # 상속 관계인 객체와 포인터의 관계
- # 가상 함수
- # 가상 함수의 동작 원리
- # 추상 클래스와 순수 가상 함수
- # virtual 소멸자
- # 클래스 멤버 변수로서의 클래스 객체
- # 다중 파일 프로그래밍

# 1. 기본형의 형변환 복습

## # C/C++ 형변환의 종류

- 자동 형변환(묵시적 형변환)
- 강제 형변환(명시적 형변환)

## # 기본형의 자동 형변환의 예

### 1. 배열 to 포인터 변환

- 배열명은 해당 배열의 첫 번째 원소의 주소로 변환됨

```
int ary[10];  
int *pAry = ary;
```

### 2. 함수 to 포인터 변환

- 함수명은 해당 함수에 대한 주소로 변환됨

```
int Sum(int a, int b);  
int (*pSum)(int, int);  
pSum = Sum;
```

### 3. 정수 승격

- 정수 계열 타입의 값은 수식에서 사용될 경우 int값으로 변환되어 처리됨

```
char c1 = 1, c2 = 2;  
char c3 = c1 + c2;
```

# 1. 기본형의 형변환 복습

## ✦ 기본형의 자동 형변환의 예 (계속)

### 4. 실수 승격

- 실수 계산시 float형의 값은 double형의 값으로 변환되어 처리됨

```
float pi = 3.14f;  
float area = pi * 5.0 * 5.0;
```

### 5. 정수 변환

- 정수 계열 타입의 값은 또 다른 정수 계열 타입의 값으로 자동 변환됨

```
int a = 5;  
char c1 = a;
```

### 6. 실수 변환

- 실수 계열 타입 값 사이의 변환

```
float pi = 3.14f;  
double dPi = pi;
```

### 7. 부동소수점수와 정수 사이의 변환

- 실수, 정수값 사이의 변환
- 변환 후의 표현 범위에 주의

```
int a = 123456;  
float b = a;
```

### 8. 포인터 변환

- 널 포인터 상수(NULL)는 평가값이 0인 int 형 포인터임. 널 포인터는 다른 타입의 포인터로 자동 형변환됨


```
double *ptr = NULL;
```

C++ : 이외의 포인터 자동 형변환은 불가  
비교 : C는 void \*와 다른 포인터 사이의 형변환 가능

# 1. 기본형의 형변환 복습

## # 강제 형변환이 필요한 상황

```
int a = 1, b = 2;  
double c = a / b;    // 실행 결과는?
```



```
int a = 1, b = 2;  
double c = (double) a / (double) b;
```

## # C++ : 강제 형변환 문법

### ■ 다음 둘 다 사용 가능

```
double c = (double) a;    // 기존 스타일  
double c = double (a);    // 함수 호출 스타일
```

## 2. 서로 다른 클래스 객체들 사이의 대입

### ⌘ 서로 다른 클래스 객체들 사이의 자동 형변환?

- ① 서로 다른 클래스 A와 B가 있으며 각각에 대한 객체 ObjA와 ObjB가 있다.  
ObjA = ObjB와 같은 대입문이 자동으로 수행될 수 있는 경우는 언제인가?
- ② ①과 같이 ObjA = ObjB가 자동으로 수행 가능한 경우 이외에 ObjA = ObjB가 가능하도록 만들려면 어떻게 해야 되는가?
- ③ ObjA의 값을 클래스 B 값으로 만들 수 있을까? 즉, ObjA 객체를 기반으로 클래스 B 객체를 만들 수 있을까?

### ⌘ 위의 세 가지 경우 중에서

- ① → 자동 형변환처럼 보이지만 동작 원리 상 자동 형변환과 무관
- ② → 한 가지 해결 방안 : 대입 연산자 오버로딩 (7.10절)
- ③ → 실질적인 형 변환 → ②도 해결 됨 (15.2절)

### ⌘ 본 절의 주제 : ① 디폴트로 대입문(ObjA = ObjB)이 가능한 경우

- 원리 : derived 클래스 객체는 base 클래스의 참조 객체로 대입될 수 있다. 그 외의 관계에 있는 클래스 객체와 참조 객체 사이의 대입은 허용되지 않는다.



## 2. 서로 다른 클래스 객체들 사이의 대입

```
#define PI 3.14
```

대입이 가능한 예

```
class CCircle {  
protected :
```

```
    int x, y;          // 중심  
    double Radius;     // 반지름
```

```
public :
```

```
    CCircle(int a, int b, double r) : x(a), y(b), Radius(r) { }
```

```
    double GetArea() { return (PI * Radius * Radius); }
```

```
};
```

```
class CSphere : public CCircle {
```

```
private :
```

```
    int z;
```

```
public :
```

```
    CSphere(int a, int b, int c, double r) : CCircle(a, b, r), z(c) { }
```

```
    double GetArea() { return (4 * PI * Radius * Radius); }
```

```
    double GetVolume() { return ((4.0/3.0) * PI * Radius * Radius * Radius); }
```

```
};
```

```
int main(void)
```

```
{
```

```
    CSphere Sph(1, 1, 1, 1);
```

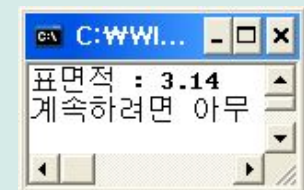
```
    CCircle &Cir = Sph;          // ok! derived 객체를 base 참조로 대입
```

```
    cout << "표면적 : " << Cir.GetArea() << endl;          // 어떤 GetArea 함수?
```

```
    //cout << Cir.GetVolume() << endl;          // CCircle은 GetVolume을 볼 수 없음
```

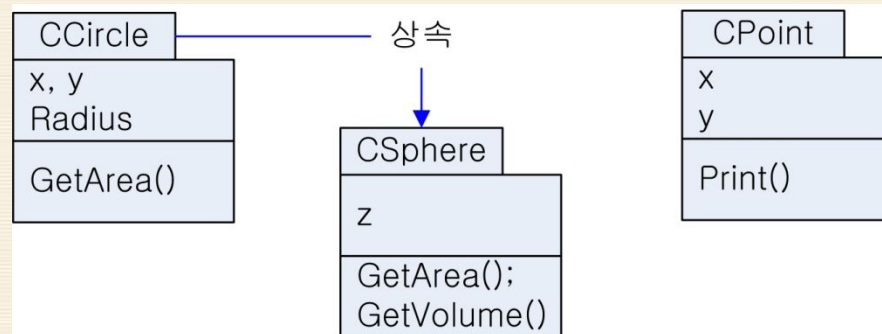
```
    return 0;
```

```
}
```



## 2. 서로 다른 클래스 객체들 사이의 대입

✦ 다음과 같은 클래스들이 있을 때 가능한 대입은?



- ① `Po = Cir;` // 서로 무관한 클래스 객체 사이의 대입
- ② `Cir = Sph;` // derived 클래스 객체로부터 base 클래스 객체로의 대입
- ③ `Sph = Cir;` // base 클래스 객체로부터 derived 클래스 객체로의 대입

✦ 힌트 : 각 클래스에 대한 대입 연산자 멤버 함수를 생각해 보라.

```
CPoint &operator=(const CPoint &P) { ... }
CCircle &operator=(const CCircle &C) { ... }
CSphere &operator=(const CSphere &S) { ... }
```

**derived 객체는 base 객체로 대입 가능. 그 외는 불가능!**

- ① `Po = Cir;` → `Po.operator=(Cir);` // 불가능
- ② `Cir = Sph;` → `Cir.operator=(Sph);` // **가능**
- ③ `Sph = Cir;` → `Sph.operator=(Cir);` // 불가능

## 2. 서로 다른 클래스 객체들 사이의 대입

✦ 예 : CCircle 클래스에 명시적 대입 연산자 오버로딩 구현, 호출 확인

```
#define PI 3.14

class CCircle {
protected :
    int x, y;          // 중심
    double Radius;     // 반지름

public :
    CCircle(int a, int b, double r) : x(a), y(b), Radius(r) { }
    double GetArea() { return (PI * Radius * Radius); }
    CCircle &operator=(const CCircle &Cir) { // 대입 연산자 오버로딩
        cout << "CCircle 대입연산자" << endl;
        x = Cir.x; y = Cir.y; Radius = Cir.Radius;
        return (*this);
    }
};

class CSphere : public CCircle {
private :
    int z;

public :
    CSphere(int a, int b, int c, double r) : CCircle(a, b, r), z(c) { }
    double GetArea() { return (4 * PI * Radius * Radius); }
    double GetVolume() { return ((4.0/3.0) * PI * Radius * Radius * Radius); }
};
```



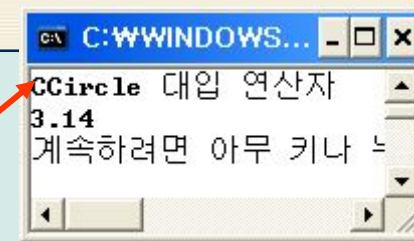
## 2. 서로 다른 클래스 객체들 사이의 대입

### ✦ 코드 계속

```
int main(void)
{
    CSphere Sph(1, 1, 1, 1);
    CCircle Cir(2, 2, 100);
    Cir = Sph;           // ok! derived 객체를 base 객체로 대입

    cout << Cir.GetArea() << endl;

    return 0;
}
```



### ✦ 응용 : 다음 중 수행 가능한 것은?

```
CCircle Cir(1, 1, 1);
CSphere Sph = Cir; // 복사 생성
```

**CSphere(const CSphere &Sph) { ... }**  
→ 불가능

```
CSphere Sph(1, 1, 1, 1);
CCircle Cir = Sph; // 복사 생성
```

**CCircle(const CCircle &Cir) { ... }**  
→ 가능

### 3. 상속 관계인 객체와 포인터의 관계

#### ✦ 서로 다른 타입의 포인터들 사이의 자동 형변환

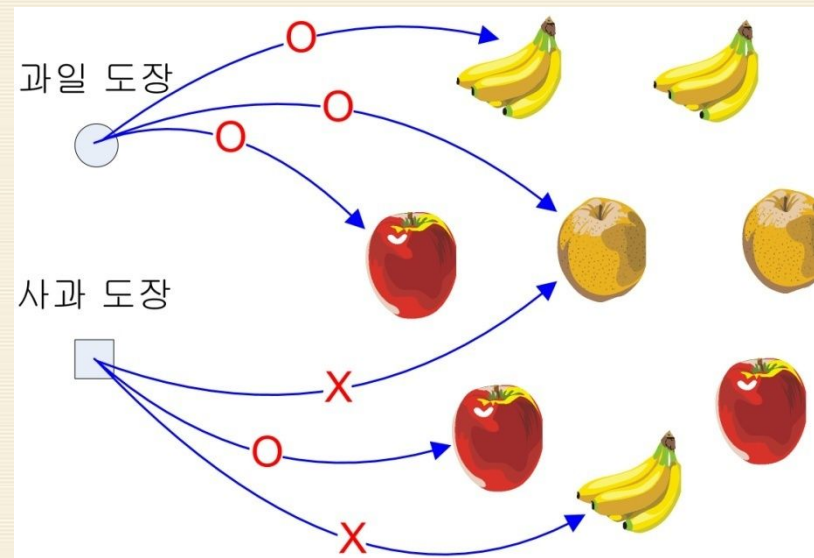
- 널 포인터와 다른 타입 사이의 형변환 외에는 자동 형변환 불가
- 예외 : derived 클래스 타입의 포인터는 묵시적으로 base 클래스 타입의 포인터로 변환될 수 있다!

#### ✦ 다음 중 가능한 것은

- ① `CCircle *pCir = &Sph;`      // 가능
- ② `CSphere *pSph = &Cir;`      // 불가능

#### ✦ 비유적 표현

- 과일 : base 클래스
- 사과 : derived 클래스



### 3. 상속 관계인 객체와 포인터의 관계

예 : CCircle 클래스와 CSphere 클래스 사이의  
포인터와 객체의 관계

```
#define PI 3.14

class CCircle {
protected :
    int x, y;           // 중심
    double Radius;      // 반지름

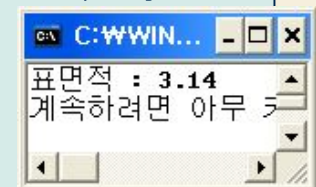
public :
    CCircle(int a, int b, double r) : x(a), y(b), Radius(r) { }
    double GetArea() { return (PI * Radius * Radius); }
};

class CSphere : public CCircle {
private :
    int z;

public :
    CSphere(int a, int b, int c, double r) : CCircle(a, b, r), z(c) { }
    double GetArea() { return (4 * PI * Radius * Radius); }
    double GetVolume() { return ((4.0/3.0) * PI * Radius*Radius*Radius); }
};

int main(void)
{
    CSphere Sph(1, 1, 1, 1);
    CCircle *pCir = &Sph; // ok! derive 객체 주소를 base 포인터로 대입

    cout << "표면적 : " << pCir->GetArea() << endl; // 어떤 GetArea 함수?
    // cout << pCir->GetVolume() << endl; // CCircle 포인터는 GetVolume 안보임
    return 0;
}
```



### 3. 상속 관계인 객체와 포인터의 관계

#### ‡ CCircle과 CSphere의 예에서

- CSphere Sph(1, 1, 1, 1); CCircle \*pCir = &Sph;
- pCir->GetArea(); // 어떤 함수가 수행되는가?
  - 디폴트 : CCircle 클래스의 GetArea 함수 → 겉으로 보이는 대로!
  - CSphere 클래스의 GetArea 함수가 수행되도록 하려면 → 가상 함수(다음 절)

#### ‡ 다음은 수행 가능한가?

- pCir->GetVolume(); // 불가능, CCircle 클래스에서는 보이지 않음
- 수행되도록 하려면 → 강제 형변환
  - cout << ((CSphere \*) pCir)->GetVolume() << endl;

#### ‡ 참조 : 동작 방식은 포인터일 때와 동일함

```
int main(void)
{
    CSphere Sph(1, 1, 1, 1);
    CCircle &Cir = Sph;           // derived 객체를 base 참조로 대입

    cout << Cir.GetArea() << endl; // 어떤 GetArea 함수? 디폴트 CCircle
    return 0;
}
```

CSphere의 GetArea 함수가 수행되도록 하려면? → 가상 함수

## 4. 가상 함수

### ✦ 프로그래밍 언어론에서의 정적 바인딩과 동적 바인딩

- 바인딩 : 프로그램을 구성하는 요소들의 속성을 결정하는 것
- 정적 바인딩 : 어떤 속성이 컴파일 시에 결정되는 것
  - C/C++ : 변수의 타입 → 선언(예, int Var) 후 사용 가능
- 동적 바인딩 : 어떤 속성이 실행 중에 결정되는 것
  - javascript : 변수 Var의 타입 → 실행 중에 변경 가능

### ✦ pCir->GetArea()에서 GetArea 함수의 결정 시기

- 디폴트 : 정적 바인딩 → 컴파일 시 결정 → 호출하는 객체 또는 포인터의 타입

```
CCircle Cir(1, 1, 1); CSphere Sph(1, 1, 1, 1);
```

```
CCircle *pCir; CSphere *pSph;
```

```
① Cir.GetArea(); // CCircle의 GetArea 함수
```

```
② Sph.GetArea(); // CSphere의 GetArea 함수
```

```
③ pCir = &Cir; pCir->GetArea(); // CCircle의 GetArea 함수
```

```
④ pCir = &Sph; pCir->GetArea(); // CCircle의 GetArea 함수
```

```
⑤ CCircle &rCir = Cir; rCir.GetArea(); // CCircle의 GetArea 함수
```

```
⑥ CCircle &rCir = Sph; rCir.GetArea(); // CCircle의 GetArea 함수
```



## 4. 가상 함수

※ 실객체의 타입이 실행 시점에 결정되는 예

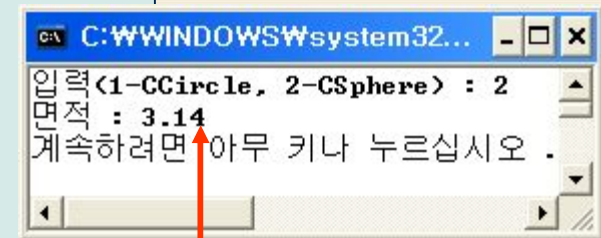
```
int main(void)
{
    int input;
    CCircle *pCir;

    cout << "입력(1-CCircle, 2-CSphere) : ";
    cin >> input;

    if (input == 1)
        pCir = new CCircle(1, 1, 1);
    else
        pCir = new CSphere(1, 1, 1, 1);

    cout << "면적 : " << pCir->GetArea() << endl;

    return 0;
}
```



디폴트 CCircle의 GetArea 함수 호출

실객체(CSphere)의 GetArea 함수가  
호출되도록 하려면  
CCircle의 GetArea 함수를 가상 함수로 선언  
→ CCircle을 다형적 클래스라고 함

## 4. 가상 함수

▣ 가상 함수 선언 : 함수 선언 시 virtual 키워드 추가

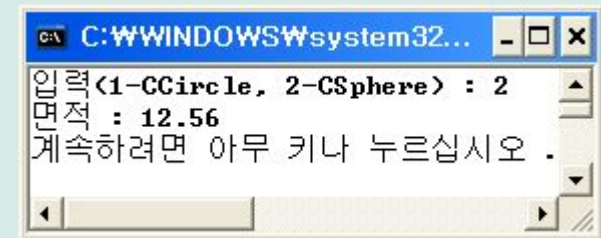
```
#define PI 3.14
```

```
class CCircle {  
protected :  
    int x, y;        // 중심  
    double Radius;   // 반지름
```

```
public :  
    CCircle(int a, int b, double r) : x(a), y(b), Radius(r) { }  
    virtual double GetArea() { return (PI * Radius * Radius); } // 가상 함수  
};
```

```
class CSphere : public CCircle {  
private :  
    int z;
```

```
public :  
    CSphere(int a, int b, int c, double r) : CCircle(a, b, r), z(c) { }  
    virtual double GetArea() { return (4 * PI * Radius * Radius); }  
    double GetVolume()  
    { return ((4.0/3.0) * PI * Radius * Radius * Radius * z); }  
};
```



base 클래스의 함수를 가상 함수로 만드는 경우  
derived 클래스의 함수는 자동으로 가상 함수가  
됨. CSphere에는 virtual이 없어도 가상 함수임.

## 4. 가상 함수

### ■ 가상 함수의 활용 예

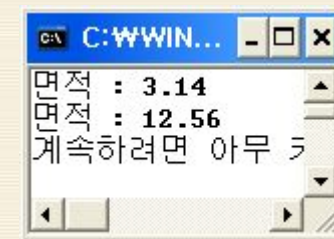
- 함수의 형식매개변수로 base 클래스의 참조로 받음
  - 참조 역시 포인터와 동일하게 동작함

```
void PrintArea(CCircle &Cir)
{
    cout << "면적 : " << Cir.GetArea() << endl;
}

int main(void)
{
    CCircle Cir(1, 1, 1);
    CSphere Sph(1, 1, 1, 1);

    PrintArea(Cir);
    PrintArea(Sph);

    return 0;
}
```



상황에 따라 실객체의 **GetArea** 함수가 호출됨

## 5. 가상 함수의 동작 원리

### ⌘ 가상 함수의 동작 원리

- 일반적으로 가상 함수 테이블(virtual function table) 사용
  - 가상 함수를 1개 이상 포함하는 클래스는 모든 가상 함수에 대한 주소를 저장
    - 가상 함수 테이블 ← 해당 클래스 객체는 자신의 클래스에 대한 가상 함수 테이블을 가리킴
- 실제로는 컴파일러의 구현에 따라 달라질 수 있음

### ⌘ 동작 원리 이해를 위한 예

```
class base {  
private :  
    int x;  
  
public :  
    void func1() { cout << "base::func1" << endl; }  
    virtual void func2() { cout << "base::func2" << endl; }  
    virtual void func3() { cout << "base::func3" << endl; }  
};
```

```
class derived : public base {  
private :  
    int y;  
  
public :  
    void func1() { cout << "derived::func1" << endl; }  
    void func2() { cout << "derived::func2" << endl; }  
    void func4() { cout << "derived::func4" << endl; }  
};
```

## 5. 가상 함수의 동작 원리

### # 코드 계속

```
int main(void)
{
    base *pBase1 = new base();
    base *pBase2 = new derived();

    pBase2->func1();
    pBase2->func2();
    pBase2->func3();
    //pBase2->func4();    // 호출 불가

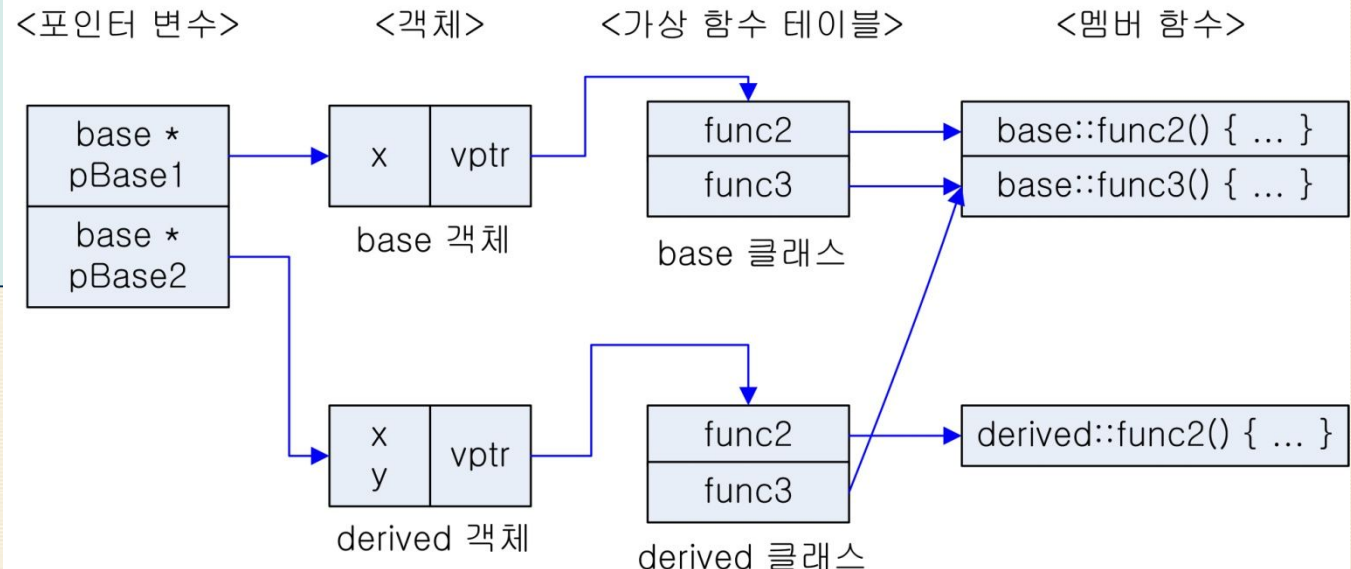
    delete pBase1;
    delete pBase2;

    return 0;
}
```

```
C:\WWIN...
base::func1
derived::func2
base::func3
계속하려면 아무 키를 누르세요
```

#### \* 가상 함수 테이블 구조 및 동작 방식

1. **base, derived** 클래스의 가상함수테이블 작성
2. **base** 객체 생성(**vpitr: base** 테이블 지시)
3. **derived** 객체 생성 (**vpitr: derived** 테이블 지시)
4. 해당 객체를 가리키는 포인터에 관계없이 항상 그 객체의 클래스 함수가 수행됨





## 6. 추상 클래스와 순수 가상 함수

✦ 예 : 원(CCircle), 직사각형(CRect) 클래스 작성

- 둘 다 좌표(x, y), Move 함수, GetArea 함수 포함

```
class CCircle {
private :
    int x, y;
    double Radius;

public :
    CCircle(int a, int b, double r) : x(a), y(b), Radius(r) { }
    void Move(int a, int b) { x += a; y += b; }
    double GetArea() { return (3.14 * Radius * Radius); }
};

class CRect {
private :
    int x, y;
    int Garo, Sero;

public :
    CRect(int a, int b, int g, int s) : x(a), y(b), Garo(g), Sero(s) { }
    void Move(int a, int b) { x += a; y += b; }
    double GetArea() { return (Garo * Sero); }
};
```

## 6. 추상 클래스와 순수 가상 함수

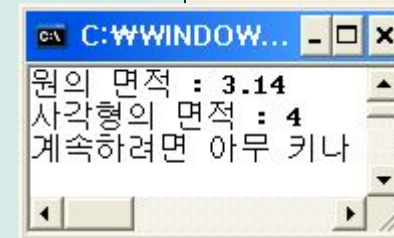
### ✦ 코드 계속

```
int main(void)
{
    CCircle Cir(1, 1, 1);
    CRect Rect(2, 2, 2, 2);

    Cir.Move(1, 1);
    Rect.Move(2, 2);

    cout << "원의면적: " << Cir.GetArea() << endl;
    cout << "사각형의면적: " << Rect.GetArea() << endl;

    return 0;
}
```



#### \* CCircle과 CRect의 공통 부분

- x, y 변수

- Move 함수 : 내용 동일

- GetArea 함수 : 내용 다름

→ 공통 부분을 포함하는 base 클래스를 하나 만들고  
CCircle과 CRect는 이 클래스로부터 상속받아 만들

## 6. 추상 클래스와 순수 가상 함수

✦ 예 : base 클래스인 CShape 클래스 추가 (main 함수는 동일)

```
class CShape {  
protected :  
    int x, y;  
  
public :  
    CShape(int a, int b) : x(a), y(b) { }  
    void Move(int a, int b) { x += a; y += b; }  
};  
  
class CCircle : public CShape {    // CShape로부터 상속  
private :  
    double Radius;  
  
public :  
    CCircle(int a, int b, double r) : CShape(a, b), Radius(r) { }  
    double GetArea() { return (3.14 * Radius * Radius); }  
};  
  
class CRect : public CShape {    // CShape로부터 상속  
private :  
    int Garo, Sero;  
  
public :  
    CRect(int a, int b, int g, int s) : CShape(a, b), Garo(g), Sero(s) { }  
    double GetArea() { return (Garo * Sero); }  
};
```

공통 부분인 x, y 변수와 Move 함수를  
CShape 클래스에서 구현

## 6. 추상 클래스와 순수 가상 함수

### ■ 본 예제에 대한 제약 사항

- CShape 클래스 객체는 존재할 수 없음
- 단지 다른 클래스의 base 클래스 역할만 담당
- 그러나 CShape 클래스 객체가 존재하지 않는다는 강제적 장치가 없음 → 추상 클래스로 만들면 됨

### ■ 추상 클래스

- 객체가 존재할 수 없는 클래스
- 추상 클래스 작성 방법
  - 순수 가상 함수 추가

\* 순수 가상 함수

- 몸체가 없음
- **derived** 클래스에서 반드시 재정의  
→ 재정의하지 않으면 그 클래스 역시 추상 클래스가 됨

주의 : 포인터는 존재할 수 있음

```
CShape *Spe = new CCircle(1, 1, 1, 1);  
cout << Spe->GetArea() << endl;
```

```
class CShape {  
protected :  
    int x, y;  
  
public :  
    CShape(int a, int b) : x(a), y(b) { }  
    void Move(int a, int b) { x += a; y += b; }  
    virtual double GetArea() = 0;  
};
```

## 7. virtual 소멸자

✦ 다음 프로그램의 문제점 및 해결 방안은?

```
class CString {
private :
    char *pStr;
    int len;

public :
    CString(char *str) { len = strlen(str);  pStr = new char[len + 1];
        strcpy(pStr, str); cout << "CString 생성자" << endl; }
    ~CString() { delete [] pStr; cout << "CString 소멸자" << endl; }
};

class CString : public CString {
private :
    char *pMyStr;
    int MyLen;

public :
    CString(char *str1, char *str2) : CString(str1) {
        MyLen = strlen(str2); pMyStr = new char[MyLen + 1];
        strcpy(pMyStr, str2); cout << "CString 생성자" << endl; }
    ~CString() { delete [] pMyStr; cout << "CString 소멸자" << endl; }
};
```



## 7. virtual 소멸자

### ✦ 코드 계속

```
int main(void)
{
    CString *pStr = new CMyString("CString", "CMyString");
    delete pStr;

    return 0;
}
```



생성자 호출 순서 : CString → CMyString  
소멸자 호출 순서 : ~CString?

- 올바른 동작 : ~CMyString → ~CString
- 그러나 pStr이 CString 포인터이므로 ~CString만 수행됨

실객체의 타입에 따라 소멸자가 호출되도록 하려면?  
→ 동적 바인딩  
→ CString 클래스의 소멸자를 가상함수로  
`virtual ~CString() { ... }`

## 8. 클래스 멤버 변수로서의 클래스 객체

✦ 어떤 클래스의 멤버 변수로서 다른 클래스의 객체가 오는 예

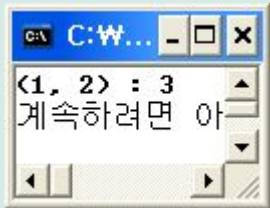
```
class CPoint {
private :
    int x, y;

public :
    CPoint(int a, int b) : x(a), y(b) { }
    void Print() { cout << "(" << x << ", " << y << ")"; }
};

class CCircle {
private :
    CPoint Center;    // CPoint 객체를 멤버 변수로 선언
    double Radius;

public :
    CCircle(int a, int b, int r) : Center(a, b), Radius(r) { } // 생성자
    void Print() {
        Center.Print();
        cout << " : " << Radius << endl;
    }
};

int main(void)
{
    CCircle Cir(1, 2, 3);
    Cir.Print();
    return 0;
}
```



\* 주의 사항 : 멤버 객체의 초기화  
→ 멤버 초기화 구문 사용

## 8. 클래스 멤버 변수로서의 클래스 객체

### ⌘ 상속 Vs. 멤버 객체

- is-a 관계 : 상속으로 표현
- has-a 관계 : 멤버 객체로 표현 (예 : 원은 점을 가지고 있다.)
- 절대적인 것은 아님

### ⌘ has-a 관계(원은 점을 가지고 있다)를 상속으로 표현한다면?

```
class CCircle : public CPoint {  
private :  
    double Radius;  
  
public :  
    CCircle(int a, int b, int r) : CPoint(a, b), Radius(r) { }  
    void Print() {  
        CPoint::Print();  
        cout << " : " << Radius << endl;  
    }  
};
```

### ⌘ CPoint 클래스의 멤버 변수로 CPoint 객체가 올 수 있을까? No

- CPoint 클래스에 대한 포인터는 올 수 있음

```
struct Node {  
    int data;  
    Node *next;  
};
```

## 9. 다중 파일 프로그래밍

✦ C++ 프로그램 : 주로 클래스 별로 소스 파일 작성

✦ 클래스 및 클래스 관련 구성 요소

- 단순 클래스 선언 : `class CPoint;`
  - 하나의 프로그램 내에 여러 번 등장 가능
  - 하나의 소스 파일 내에 여러 번 등장 가능
- 클래스 선언, 멤버 선언, 멤버 함수의 내부 정의 : `class CPoint { ... };`
  - 지금까지 사용한 방법
  - 하나의 프로그램 내에 여러 번 등장 가능
  - 하나의 소스 파일 내에 단 한번만 등장 가능
- 클래스 멤버 함수의 외부 정의 : `void CPoint::GetArea() { ... }`
  - 하나의 프로그램 내에 단 한 번만 등장 가능

## 9. 다중 파일 프로그래밍

### ※ 클래스 별 다중 파일 프로그래밍

#### Point.h : 클래스 선언

```
#ifndef __MY_CPOINT
#define __MY_CPOINT

class CPoint {
private :
    int x, y;

public :
    CPoint(int a = 0, int b = 0) : x(a), y(b) { }
    void Print();
};

#endif
```

하나의 소스파일 내에서 단  
한 번만 포함토록 보장

#### main.cpp

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

int main(void)
{
    CPoint Po(1, 2);
    Po.Print();

    return 0;
}
```

#### Point.cpp : 외부 함수 정의

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

void CPoint::Print()
{
    cout << "(" << x << ", " << y << ")" << endl;
}
```