

창의적 소프트웨어 프로그래밍 (Creative Software Design)

Polymorphism

2018.07.09.

담당교수 이 효 섭

- **형변환**

- **명시적 형변환 : 프로그래머가 캐스트 연산자를 사용해 직접 형변환**

① `int i= 5;`
`double d= 10.6;`
`d = i;`

② `int i= 5;`
`double d= 10.6;`
`i = (int) d;`

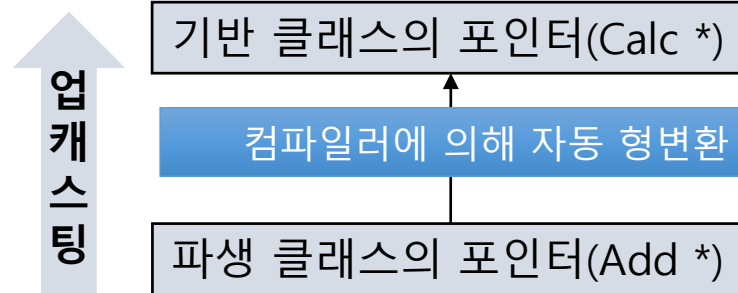
- **상속관계에 있는 클래스 사이의 형변환은**

- **업 캐스팅(UpCasting)과 다운 캐스팅(DownCasting)으로 구분**

● 업 캐스팅

- 기반 클래스의 포인터 변수가 파생 클래스의 인스턴스를 가리킬 때

```
Add AddObj(3, 5);  
Calc *CalcPtr;  
CalcPtr = &AddObj;
```



업 캐스팅(Upcasting)
: 기반 클래스형으로의 형변환

- 특징

- ① 업 캐스팅은 파생 객체의 포인터가 기반 객체의 포인터로 형변환하는 것
- ② 업 캐스팅을 하면 참조 가능한 영역이 축소
- ③ 업 캐스팅은 컴파일러에 의해서 자동 형변환

● 다운 캐스팅

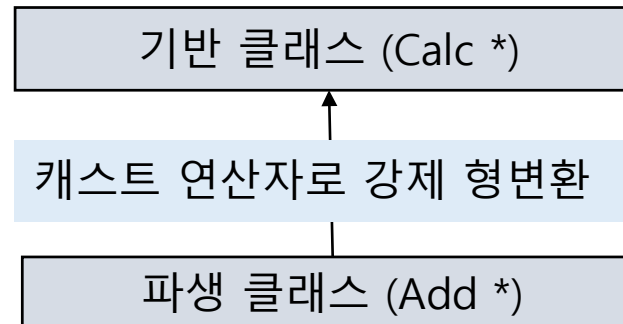
- 파생 클래스로 선언된 포인터 변수에 기반 클래스로 선언된 객체의 주소를 저장
- 다운 캐스팅은 컴파일러가 자동으로 형변환하지 않음 → 컴파일 에러 발생

```
Calc Obj(3, 5);  
Add *AddPtr ;  
AddPtr = &Obj;
```

오류 목록				
오류 2개 경고 0개 메시지 0개				
설명	파일	줄	열	프로젝트
1 error C2440: '=' : 'Calc *'에서 'Add *'(으)로 변환할 수 없습니다.	12_09.cpp	71	1	12_09
2 IntelliSense: "Calc *" 형식의 값을 "Add *" 형식의 엔티티에 할당할 수 없습니다.	12_09.cpp	71	9	12_09

```
Calc Obj(3, 5);  
Add *AddPtr ;  
AddPtr = (Add *)&Obj;
```

다운 캐스팅



다운 캐스팅(DownCating)
: 파생 클래스형으로의
형변환

- 다운 캐스팅

- 특징

- ❶ 다운 캐스팅은 파생 클래스로 형변환하는 것
- ❷ 다운 캐스팅은 참조 가능한 영역이 확대되는 것을 의미
- ❸ 다운 캐스팅에 대해서는 컴파일러가 자동으로 형변환하지 않음
- ❹ 다운 캐스팅은 프로그래머가 명시적으로 형변환해야만 컴파일상의 에러 방지
- ❺ 다운 캐스팅은 강제 형변환한 후에도 실행 시 예외사항이 발생할 수 있으므로 인스턴스의 클래스형과 참조하는 포인터 변수의 상속 관계를 생각해서 명시적 형변환을 해야 함.
(한번 업 캐스팅이 된 포인터 값을 다운 캐스팅하는 경우에만 안전)

● 정적 바인딩

- 컴파일 할 때 미리 호출될 함수를 결정하는 것
- 정적 바인딩(static binding) 또는 이른 바인딩(early binding) 이라고 함

예)

```
Add y(3, 5);  
Calc *CalcPtr= &y;  
CalcPtr->Prn();
```

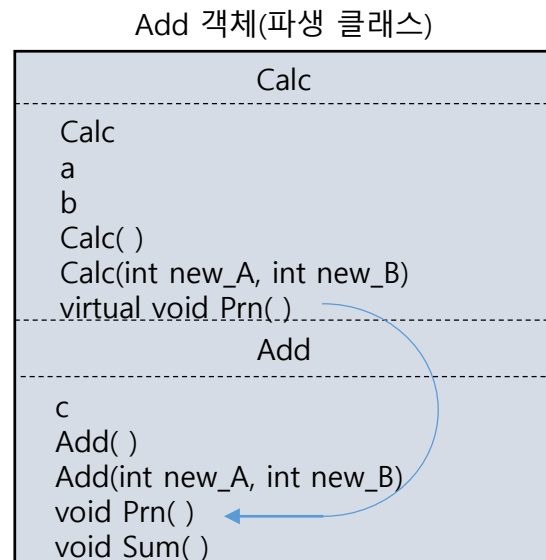
컴파일러가 포인터 변수의 자료형을 존중한다면 기반 클래스의 멤버 함수(Calc::Prn)가 호출되어야 한다. 하지만 포인터보다 그 포인터가 가리키는 객체의 자료형을 존중한다면 파생 클래스의 멤버함수인 Add::Prn이 호출될 것이다.
실제로는 기반 클래스의 멤버함수인 Calc::Prn이 호출된다.

컴파일 시점	실행 시점
변수의 자료형이 결정	변숫값이 저장
호출될 함수가 결정	함수가 실행

● 동적 바인딩

- 호출할 함수를 컴파일할 때 결정하지 않고 프로그램이 실행되는 동안 결정
- 동적 바인딩(dynamic binding) 또는 늦은 바인딩(lately binding) 이라고 함
- 방법
 - 동적 바인딩을 하고자 하는 함수가 선언되어 있는 기반 클래스에 virtual 붙이기
 - virtual 예약어를 붙인 함수를 가상함수라고 함

```
Add y(3, 5);  
Calc *CalcPtr = &y;  
CalcPtr->Prn();
```



prn을 가상함수로 지정하였으므로
동적 바인딩
→ CalcPtr이 가리키는
객체의 자료형에 의해
호출될 함수(Add 클래스의 Prn)가
결정

- 가상함수의 장단점

- ① 가상함수를 사용하려면 가상 테이블을 유지하기 위해 가상함수만큼의 주소 배열을 만들어야 하므로 메모리에 부담
- ② 각 객체도 가상함수 테이블의 주소를 저장하는 가상 포인터 변수를 위한 메모리를 할당해야 함
- ③ 가상함수가 호출될 때마다 가상함수 테이블을 이용해서 주소를 조사하기 때문에 처리 속도 지연

- **완전 가상함수(pure virtual function)**
 - 함수의 정의없이 함수의 유형만을 기반 클래스에 제시해 놓는 것
 - 가상함수처럼 멤버함수를 선언할 때 예약어 `virtual`을 선언문의 맨 앞에 붙이고 함수의 선언문 마지막 부분에 `'=0'` 을 덧붙여 사용
 - 이렇게 선언된 멤버함수는 함수의 몸체 부분이 없음

```
virtual 반환형 함수명() = 0;
```

- 완전 가상함수를 최소 한 개 이상 갖는 클래스로는 객체를 생성하지 못함
→ '추상 클래스(abstract class)'

- 추상 클래스는 상속을 위한 기반 클래스로 사용

- 예제

- 사각형을 클래스로 정의해 보자. 사각형 클래스는 사각형을 그리기 위한 함수와 크기를 알려주기 위한 함수가 필요하다.
- 이번에는 원을 클래스로 정의해 보자. 원 클래스 역시 원을 그리기 위한 함수와 크기를 알려주기 위한 함수가 필요하다.
- 사각형과 원 클래스의 구체적인 내용은 다르지만 그린다라는 작업과 크기를 알아낸다는 작업은 목적이 동일하므로 함수명을 동일하게 해줄 수 있다.
- Shape 클래스에 완전 가상함수는 파생 클래스에게 표준안을 제공하기 위해 등록하는 것이며, Shape는 추상 클래스가 되어 파생 클래스를 설계하기 위한 기반 클래스로만 사용하게 된다.

rectangle.Draw(); // 사각형을 그린다.
circle.Draw(); // 원을 그린다.

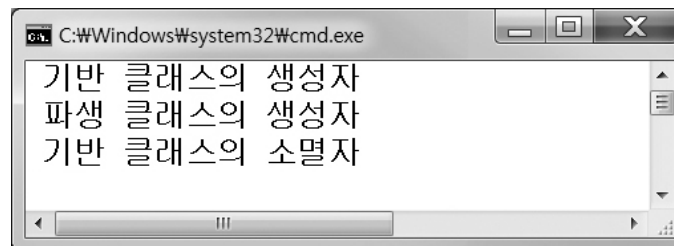
```
class Shape
{
public:
    virtual void Draw()=0;
    virtual double GetSize()=0;
};
```

```
class Rect : public Shape{
};
class Circle : public Shape{
};
```

일반 소멸자 사용시 문제점

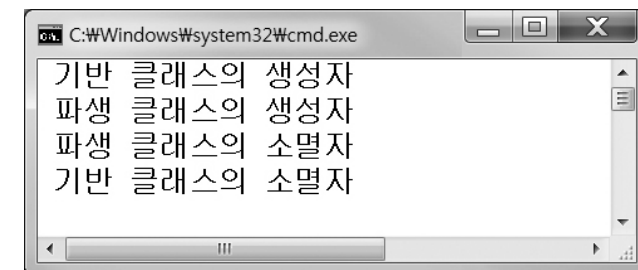
```
01 #include<iostream>
02 using namespace std;
03 class Base{
04 public:
05     Base();
06     ~Base();
07 };
08 Base::Base()
09 {
10     cout<<" 기반 클래스의 생성자 "<<endl;
11 }
12 Base::~~Base()
13 {
14     cout<<" 기반 클래스의 소멸자"<<endl;
15 }
16 class Derived : public Base{
17 public :
18     Derived();
19     ~Derived();
20 };
```

```
21 Derived::Derived()
22 {
23     cout<<" 파생 클래스의 생성자 "<<endl;
24 }
25 Derived::~~Derived()
26 {
27     cout<<" 파생 클래스의 소멸자 "<<endl;
28 }
29
30 void main()
31 {
32     Base *BasePtr = new Derived;
33     delete BasePtr;
34 }
```



```
C:\Windows\system32\cmd.exe
기반 클래스의 생성자
파생 클래스의 생성자
기반 클래스의 소멸자
```

이런 결과가 나오려면?



```
C:\Windows\system32\cmd.exe
기반 클래스의 생성자
파생 클래스의 생성자
파생 클래스의 소멸자
기반 클래스의 소멸자
```

Thank you!

Beyond The Engine of Korea

HANYANG UNIVERSITY



한양대학교
HANYANG UNIVERSITY