원리를 알면 IT가 맛있다
IT COOKBOOK

IT CookBook, C++ 하이킹 객체지향과 만나는 여행

[강의교안 이용 안내]

C++ 하이킹

객체지향과 만나는 여행

Chapter 12. 상속성

한빛아카데미 Hanbit Academy, Inc.

목차

학습목표

- 클래스간의 상속 관계를 갖도록 하는 방법을 익힌다.
- 상속 관계에서의 생성자의 특징을 살펴본다.
- 함수의 오버라이딩에 대해서 학습한다.
- 업 캐스팅과 다운 캐스팅의 개념과 그 활용법을 익힌다.
- 가상함수에 대해서 학습한다.
- 완전 가상함수를 갖는 추상 클래스를 설계한다.
- 객체지향 프로그래밍의 다형성을 학습한다.

01 클래스간의 상속관계

상속의 의미

❶ 프로그램을 개발하다 보면 중복되는 내용이 많아서 이 코드를 재활용하려고 나온 개념이 상속이다.

❷ 상속은 부모에게 무엇인가를 물려받는 것을 의미한다.

❸ 새로 만드는 클래스를 이미 정의된 훌륭한 클래스에서 상속받는다면 많은 기능을 모두 물려받아 사용할 수 있다.

❹ 부모가 되는 클래스를 기반 클래스, 기초 클래스, 베이스 클래스라 하고 자식 클래스에 해당되는 클래스를 파생 클래스(derived class)라고 한다.

기반 클래스와 파생 클래스 만들기

- 상속을 이용해서 프로그램 코드를 어떻게 재활용할 수 있는지 Add와 Mul 클래스를 통해 살펴보자.
- 기반 클래스 만들기
  - 공통적으로 기술된 멤버변수와 멤버함수들로 구성된 기반 클래스를 정의해 보자.

[표 12-1] **접근 지정자의 접근 허용 범위**

접근 지정자	자신의 클래스	파생 클래스	클래스 외부
private	O	X	X
protected	O	O	X
public	O	O	O

01 클래스간의 상속관계

- Add와 Mul 클래스의 공통점만 가진 기반 클래스를 설계할 때는 Add와 Mul 클래스에서 private로 선언된 멤버를 다음과 같이 protected로 변경해야 한다. 그래야 파생 클래스 Calc에서 상속받아 사용할 수 있기 때문이다.

```
class Calc{
protected:
int a;
int b;
int c;
public:
void Init(int new_A,int new_B);
void Prn();
};
void Calc::Init(iint new_A,int new_B)
{
a=new_A;
b=new_B;
c=0;
}
void Calc::Prn()
{
cout<<a<<"₩t"<<b<<"₩t"<<c<<endl;
}
```

01 클래스간의 상속관계

- 파생 클래스 만들기
  - 파생 클래스를 만들려면 우선 기반 클래스가 존재해야 한다. 파생 클래스를 새롭게 정의할 때는 다음 형식처럼 파생 클래스 뒤에 콜론(:)을 붙이고 기반 클래스를 기술한다.

파생 클래스 기본 형식

```
class 파생 클래스 : 접근 지정자 기반 클래스 {
  멤버변수:
  멤버함수;
};
```

```
class Add    :    public Calc
```

새롭게 생성되는 파생 클래스 — 이미 존재하는 기반 클래스

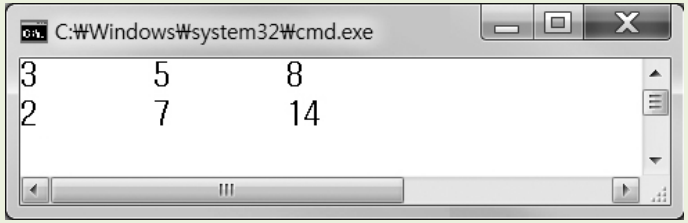
  - Add 클래스와 Mul 클래스는 Calc 클래스의 상속을 받도록 하고 각 클래스에만 있는 특징을 추가한다.

```
class Add : public Calc{
public:
void Sum();
};
void Add::Sum()
{
c=a+b;
}
```

```
class Mul : public Calc{
public:
void Gob();
};
void Mul::Gob()
{
c=a*b;
}
```

예제 12-1. 기반 클래스와 파생 클래스 설계하기(12_01.cpp)

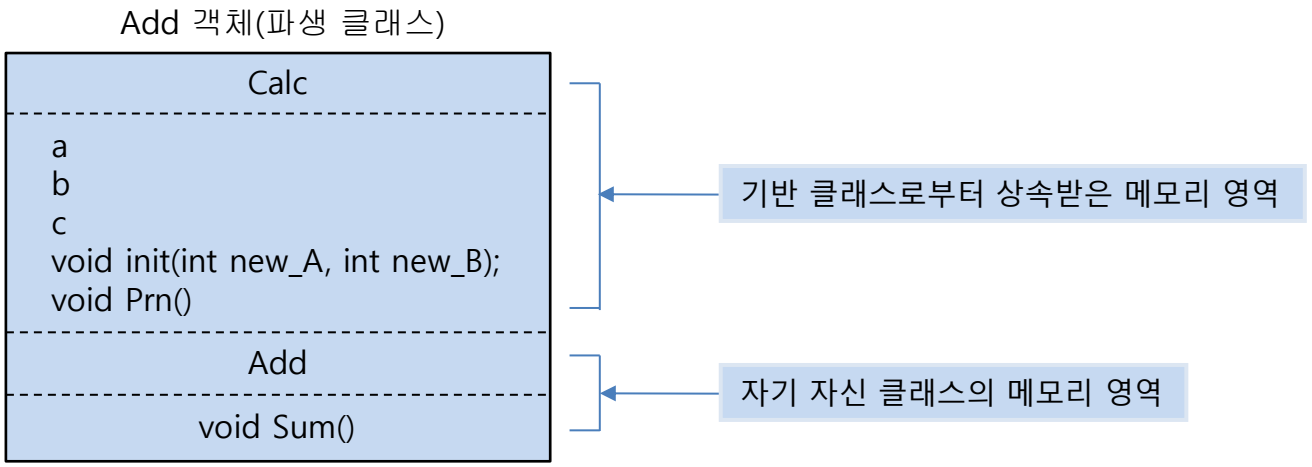
책의 소스코드 참고



02 상속 관계에서의 생성자

- 상속 관계에 있는 클래스에서 생성자는 다음과 같은 특징이 있으므로 조심스럽게 다루어야 한다.
  ❶ 생성자는 멤버함수지만 상속할 수 없다.
  ❷ 파생 객체가 생성되어도 기반 클래스의 생성자까지 연속적으로 자동 호출된다.

■ 생성자와 소멸자 호출 순서



[그림 12-1] 파생 클래스로 선언된 객체의 메모리 구조

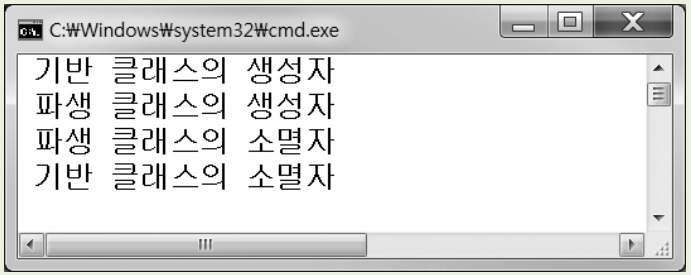
- 상속 관계에서 생성자와 소멸자에 대해 정리하면 다음과 같다.

❶ 파생 클래스의 객체가 생성될 때 기반 클래스의 멤버변수에 대한 메모리 영역에 파생 클래스의 멤버변수에 대한 메모리 영역을 포함한 형태로 메모리를 할당한다.

❷ 상속 관계에 있는 파생 클래스로 객체를 생성하면 자신의 생성자 뿐만 아니라 기반 클래스의 생성자까지 연속으로 자동 호출되는데, 기반 클래스의 생성자가 먼저 호출되고 파생 클래스의 생성자가 나중에 호출된다.

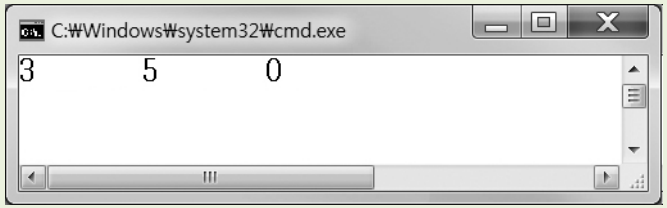
예제 12-2. 상속 관계에서의 생성자와 소멸자 알아보기(12_02.cpp)


```
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 Derived obj;
33 }
```



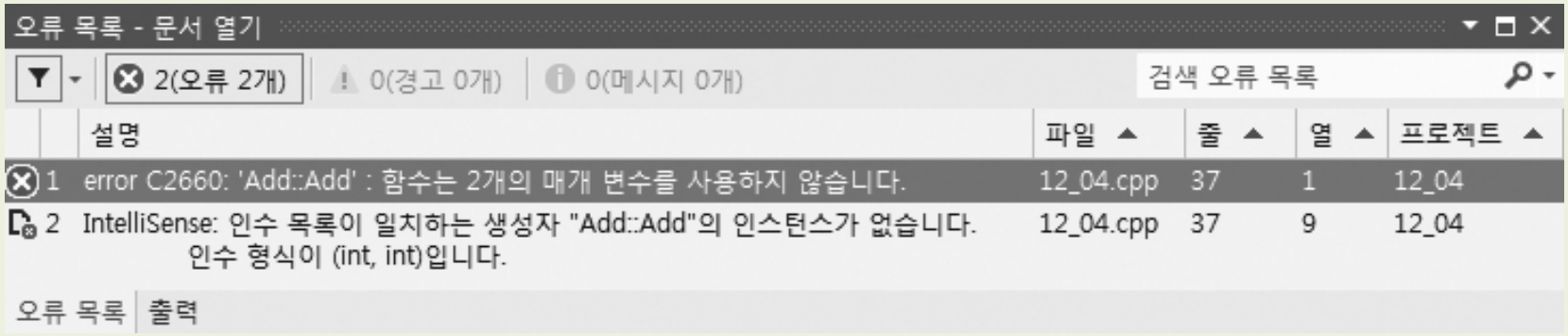
예제 12-3. Calc 클래스에서 생성자 정의하기(12_03.cpp)


```
01 #include<iostream>
02 using namespace std;
03 class Calc{
04 protected:
05 int a;
06 int b;
07 int c;
08 public:
09 void Prn();
10 Calc(int new_A,int new_B); // 기반 클래스에 생성자 추가
11 };
12
13 void Calc::Prn()
14 {
15 cout<<a<<"₩t"<<b<<"₩t"<<c<<endl;
16 }
17
18 Calc::Calc(int new_A,int new_B) // 기반 클래스에 생성자 추
가
19 {
20 a=new_A;
21 b=new_B;
22 c=0;
23 }
24
25 void main()
26 {
27 Calc x(3, 5);
28 x.Prn();
29 }
```



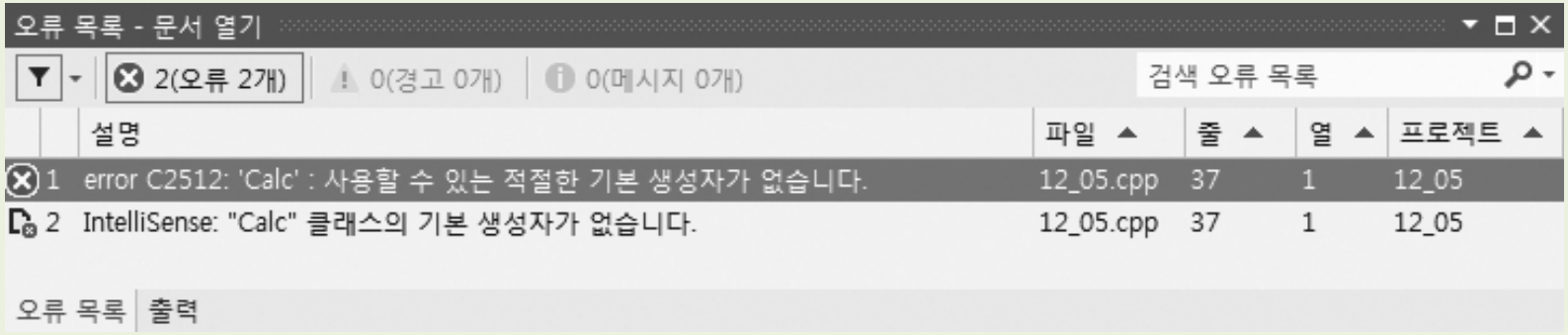
예제 12-4. 생성자를 정의한 Calc 클래스의 파생 클래스 설계하기(12_04.cpp)

책의 소스코드 참고



예제 12-5. Add 클래스에 생성자 정의하기(12_05.cpp)

책의 소스코드 참고

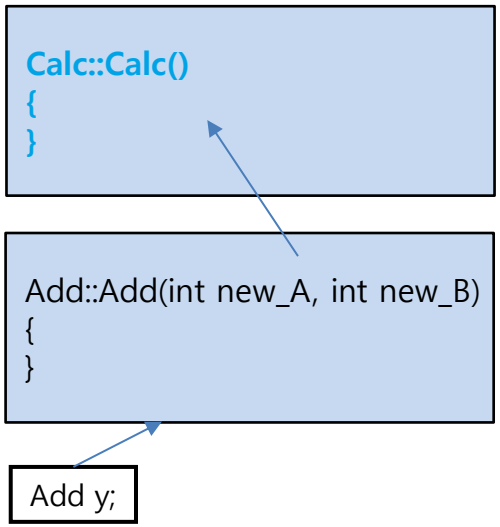


02 상속 관계에서의 생성자

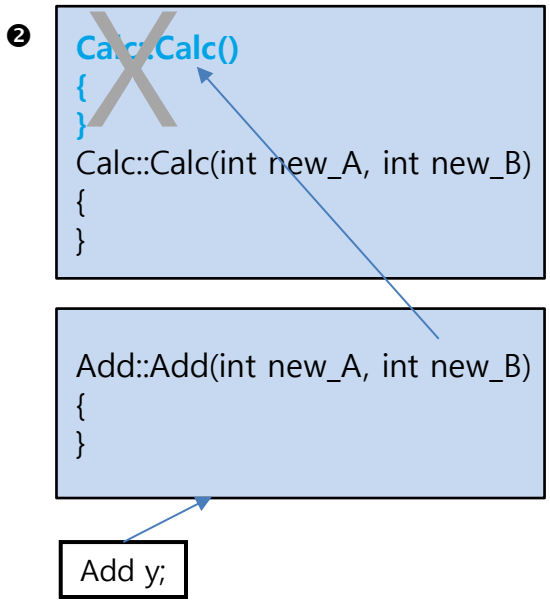
상속 관계에서 생성자 문제

- ❶은 파생 클래스의 생성자가 기반 클래스의 암시적으로 컴파일러에 의해 제공되는 기본 생성자를 호출하는 것이다. ❷처럼 매개변수를 한 개라도 갖는 생성자를 정의하면 더 이상 C++ 컴파일러가 제공하지 않게 된다.

❶



❷



02 상속 관계에서의 생성자

파생 클래스에서 기반 클래스의 생성자를 명시적으로 호출하기

- 기반 클래스의 생성자를 파생 클래스에서 명시적으로 호출하는 것은 콜론(:) 초기화의 형태다. 파생 클래스의 생성자를 정의할 때 함수의 머리 부분 맨 마지막에 :을 기술한 후 기반 클래스의 생성자를 호출한다.
- 명시적으로 기술하지 않아도 Add 클래스의 생성자는 자신의 기반 클래스인 Calc 클래스의 생성자를 자동 호출한다.

```
Add::Add() : Calc()
{
....
}
```

→ 기반 클래스의 생성자가 명시적으로 호출된다.

```
Add::Add()
{
....
}
```

→ 명시적으로 호출하지 않아도 기반 클래스의 생성자가 암시적으로 호출된다.

- 만약 기반 클래스 내에 매개변수를 갖는 생성자를 추가로 정의할 경우에는 매개변수가 없는 기본 생성자를 프로그래머가 반드시 명시적으로 정의하는 습관을 들여야 한다.

```
class Calc{
public :
  Calc::Calc(int new_A, int new_B)
  {
    a=new_A;
    b=new_B;
    c=0;
  }
  Calc::Calc()
  {
    a=0;
    b=0;
    c=0;
  }
}
```

→ 기본 생성자를 프로그래머가 명시적으로 정의하는 습관을 들인다.

02 상속 관계에서의 생성자

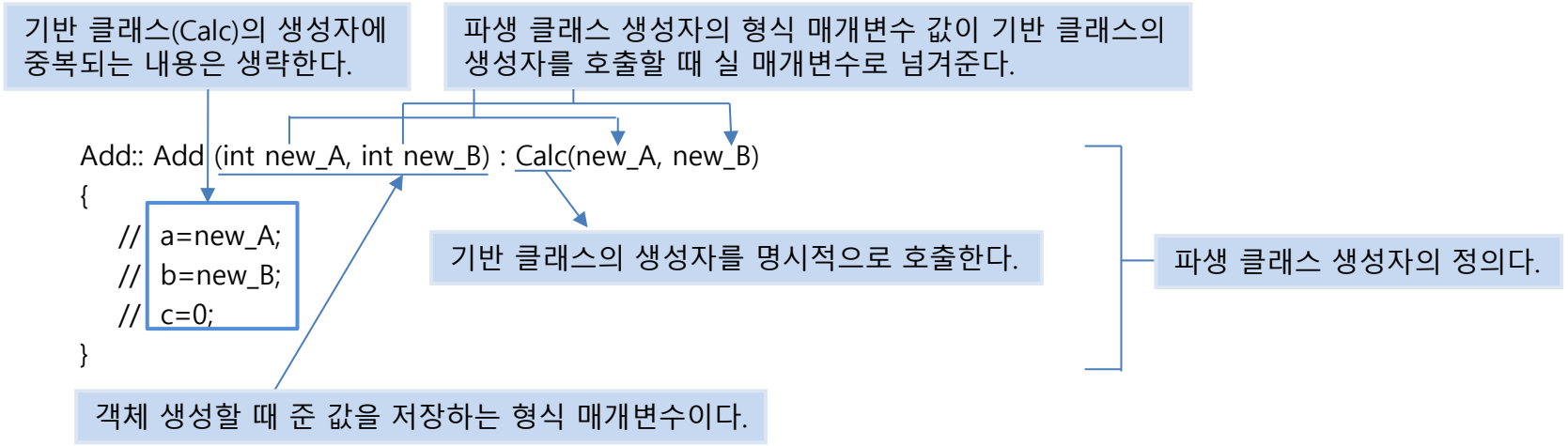
기반 클래스의 생성자에 매개변수 전달하기

- 파생 클래스(Add)에도 동일한 작업이 중복되어 기술되어 있다.

```
Calc::Calc(int new_A, int new_B)
{
  a=new_A;
  b=new_B;
  c=0;
}
```

```
Add::Add(int new_A, int new_B)
{
  a=new_A;
  b=new_B;
  c=0;
}
```

- 파생 클래스의생성자가 받은 매개변수의 값을 기반 클래스의 생성자를 호출하면서 전달해주도록 하는 예다.



02 상속 관계에서의 생성자

- : 다음에 기술된 기반 클래스의 생성자는 함수의 호출이다. 함수를 호출할 때 소괄호 내부에 기술된 내용은 실매개변수이므로 변수나 값을 적어주어야 한다. 함수의 머리 부분에 기술되었다고 해서 자료형을 기술하는 경우가 많은데 잘못된 것이다.

```
Add:: Add (int new_A, int new_B) : Calc(int new_A, int new_B)
{
}
```

잘못된 표현

- 함수를 정의할 때는 소괄호 안에 형식 매개변수를 기술해야 하므로 자료형과 함께 변수 선언문 형태를 취하고 : 다음에 나오는 부분은 분명히 함수 호출이므로 변수만 기술해야 한다. 자료형을 기술해서는 안 된다.

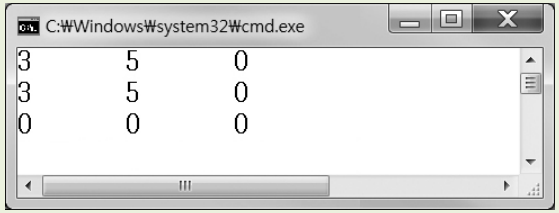
함수의 오버라이딩

- 기반 클래스에 정의되어 있는 함수와 동일한 형태로 파생 클래스에서 다시 정의하는 것을 함수의 오버라이딩(overridding)이라 한다. 이때 기반 클래스에 정의되어 있는 함수의 원형과 동일한 형태로 정의해야 한다.
- 오버라이딩되어 은폐된 기반 클래스의 멤버함수가 수행되어야 한다면 다음 예처럼 기반 클래스명을 스코프 연산자(::)와 함께 함수명 앞에 명시적으로 기술해서 호출해야 한다.

```
void Add::Prn()
{
  Calc::Prn(); // 은폐된 기반 클래스의 멤버함수 호출
  cout<<a<<" + "<<b<<" = "<<c<<endl;
}
```

예제 12-6. 상속 관계에서 생성자 문제 해결하기(12_06.cpp)

책의 소스코드 참고



03 업 캐스팅과 다운 캐스팅

형변환

- C++에서 서로 다른 자료형에 대해서 대입 연산을 할 경우 형변환이 일어난다. 암시적 형변환은 ⓐ처럼 프로그래머가 모르는 사이에 C++ 컴파일러에 의해 형변환이 일어나는 것이다.
- 명시적 형변환은 ⓑ처럼 프로그래머가 캐스트 연산자를 사용해서 직접 형변환을 하는 것이다.

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

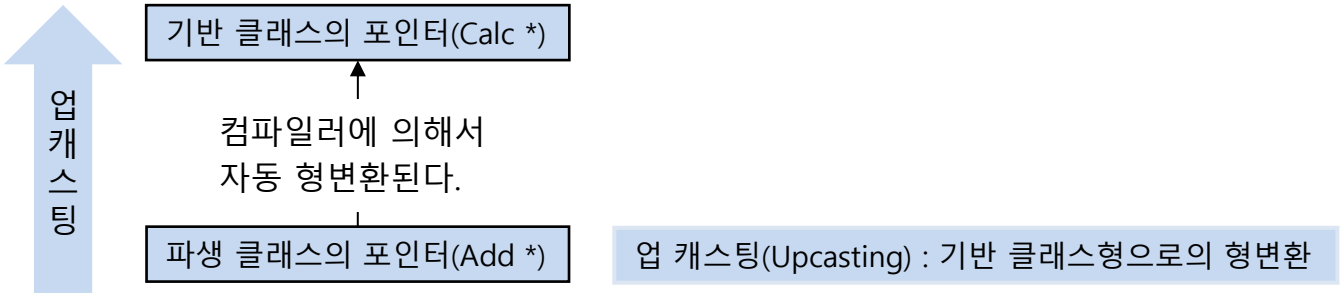
ⓑ
```
int i= 5;
double d= 10.6;
i = (int) d;
```

- 상속 관계에 있는 클래스 사이의 형변환은 업 캐스팅(UpCasting)과 다운 캐스팅(DownCasting)로 구분된다.

업 캐스팅

- 기반 클래스의 포인터 변수가 파생 클래스의 인스턴스를 가리킬 수 있는데, 이를 업 캐스팅이라 하며 컴파일러에서 자동으로 형변환이 이루어진다.

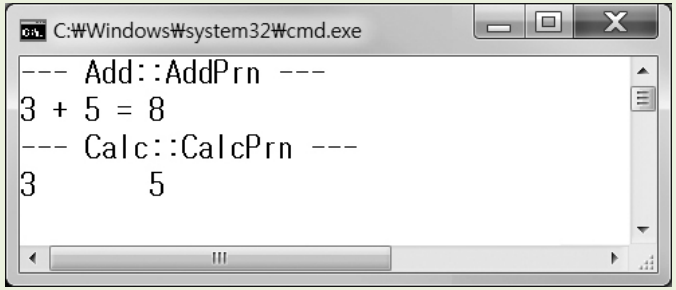
```
Add AddObj(3, 5);
Calc *CalcPtr;
CalcPtr = &AddObj;
```



[그림 12-2] 업 캐스팅

예제 12-8. 업 캐스팅하기(12_08.cpp)

책의 소스코드 참고



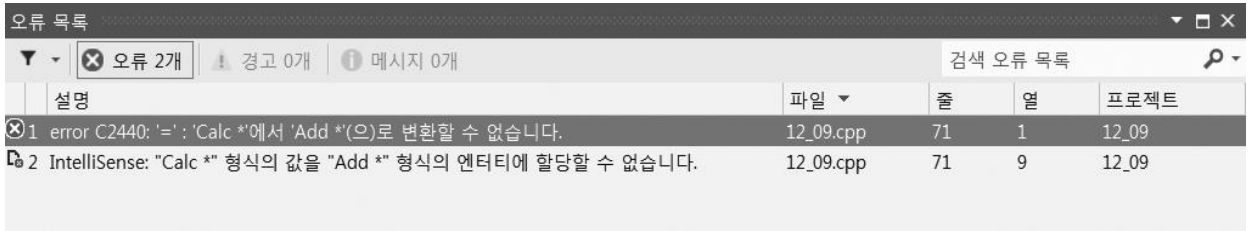
03 업 캐스팅과 다운 캐스팅

- 업 캐스팅에 대해 정리해 보자.
  - ❶ 업 캐스팅은 파생 객체의 포인터가 기반 객체의 포인터로 형변환하는 것이다.
  - ❷ 업 캐스팅을 하면 참조 가능한 영역이 축소된다.
  - ❸ 업 캐스팅은 컴파일러에 의해서 자동 형변환된다.

다운 캐스팅

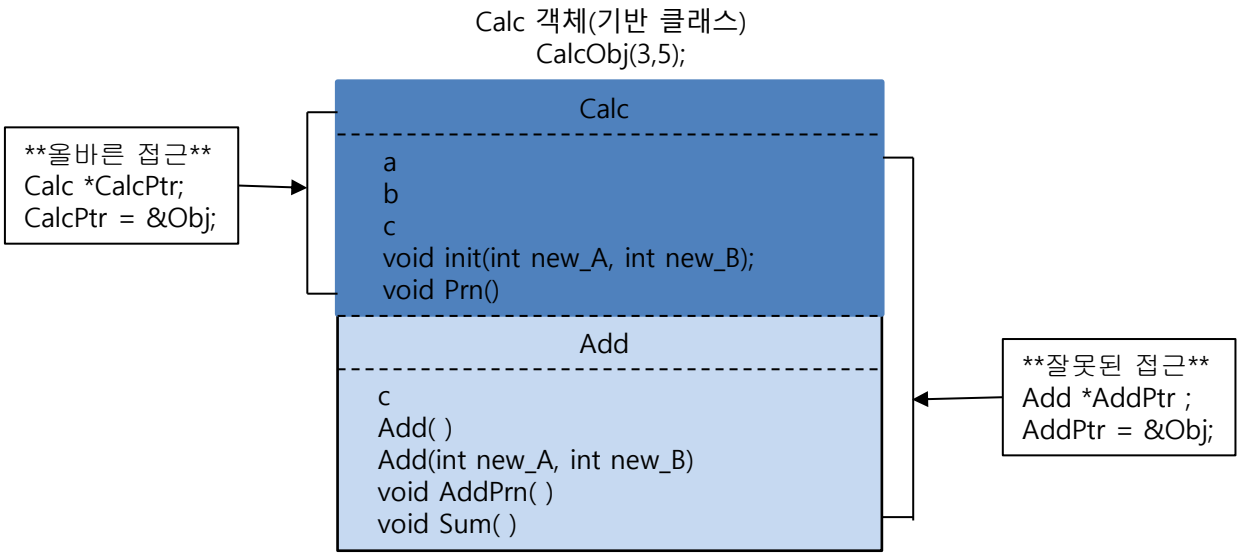
- 다운 캐스팅이란 파생 클래스로 선언된 포인터 변수에 기반 클래스로 선언된 객체의 주소를 저장하는 것인데, 다운 캐스팅에 대해서는 컴파일러가 자동으로 형변환하지 않기 때문에 컴파일 에러가 발생한다.

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



- Calc(기반 클래스)가 새로 생성된 객체라면 Calc 클래스에 의해서 생성된 영역은 Calc 객체 영역 뿐이지 Add 객체에 대한 영역은 존재하지 않는다.

03 업 캐스팅과 다운 캐스팅



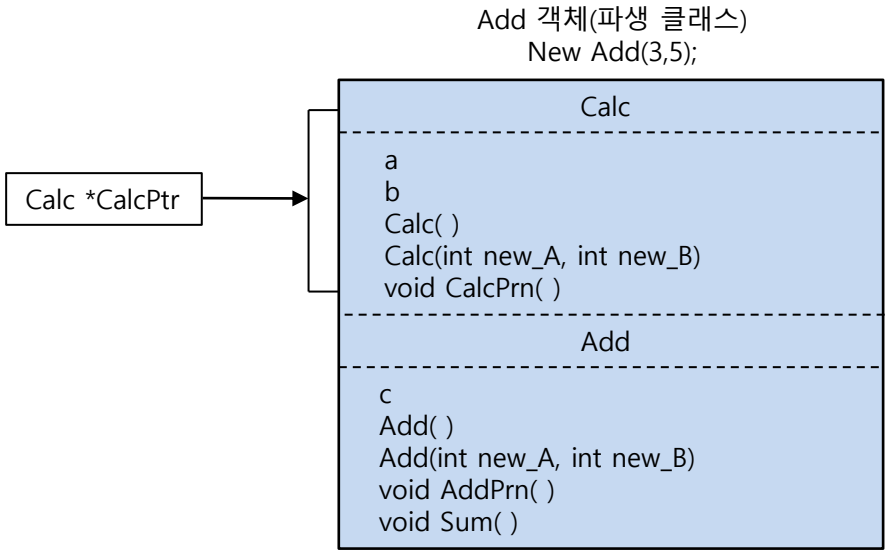
[그림 12-3] **잘못된 다운 캐스팅**

- 다운 캐스팅을 위한 강제 형변환
  - 컴파일러에 의해서 자동으로 다운 캐스팅되지는 않지만 명시적으로 캐스트 연산자를 기술해서 강제로 형변환을 하는 경우에 다운 캐스팅을 허용해 준다.

```
Calc *CalcPtr = new Add(3, 5); // ---------- ❶
```

  - 포인터 변수 CalcPtr이 Add 클래스형 객체를 가리키고 있다. 파생 클래스형 객체의 주소를 기반 클래스로 선언된 포인터 변수에 대입했다. 이 과정에서 업 캐스팅이 일어났다.

03 업 캐스팅과 다운 캐스팅



[그림 12-4] **업 캐스팅된 메모리 구조**

- 만약 클래스 Calc형 포인터 변수인 CalcPtr로 클래스 Add형 객체에 접근해서 Add 클래스에 정의된 멤버함수를 호출하면 컴파일 에러가 발생한다.

```
CalcPtr->AddPrn();     // 에러
CalcPtr->Sum();        // 에러
```

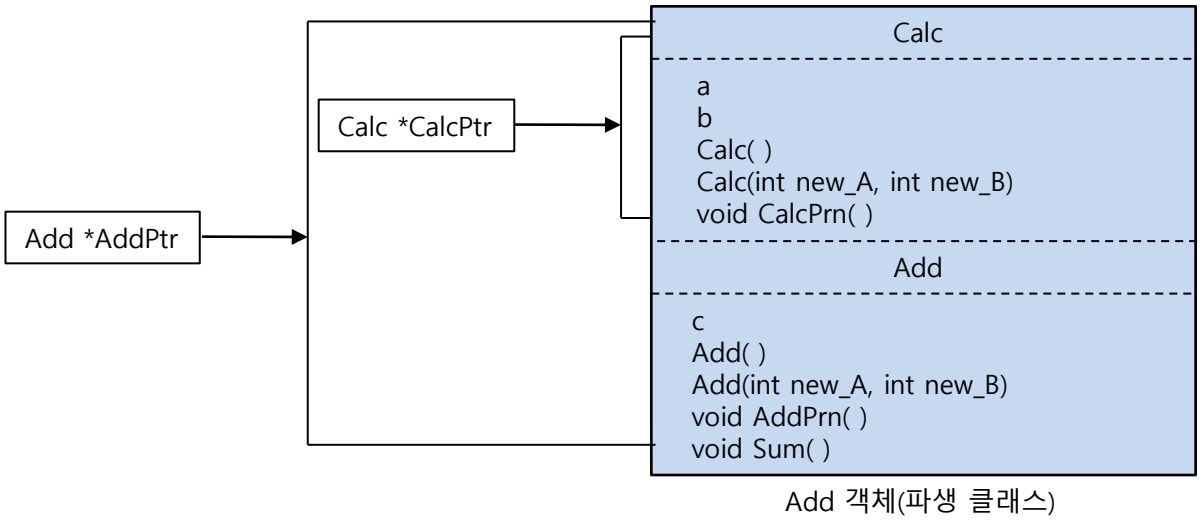
- Add 클래스에 정의된 멤버함수를 사용하려면 Add형 객체를 Add형 포인터 변수가 가리키도록 해야 한다. 클래스 Add형 포인터 변수 AddPtr을 선언해서 이 포인터 변수가 Add형 객체를 가리키도록 해 보자.

```
Calc *CalcPtr = new Add(3, 5);
Add *AddPtr = CalcPtr; // 에러!!!!
```

03 업 캐스팅과 다운 캐스팅

- ❷와 같이 CalcPtr 앞에 (Add *)와 같이 캐스트 연산자를 명시적으로 기술해서 강제 형변환해야 컴파일 에러가 발생하지 않는다.

```
Calc *CalcPtr = new Add(3, 5);
Add *AddPtr = (Add *)CalcPtr; // ------------------ ❷
```



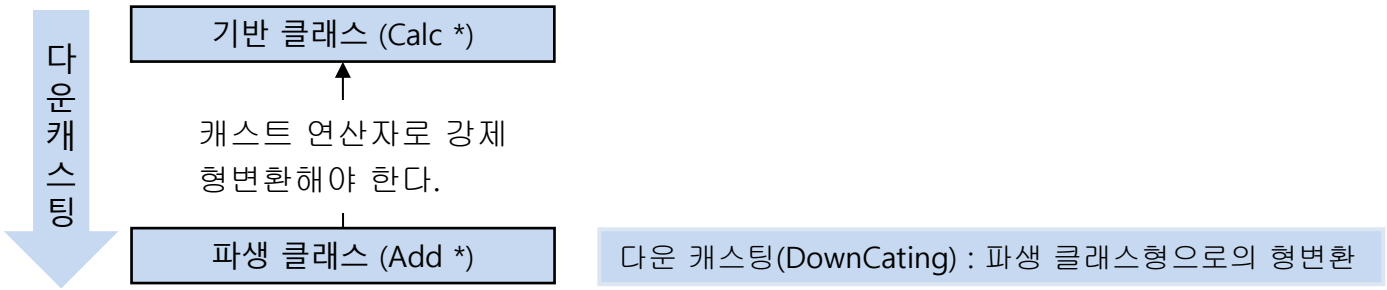
Add 객체(파생 클래스)

[그림 12-4] **업 캐스팅된 메모리 구조**

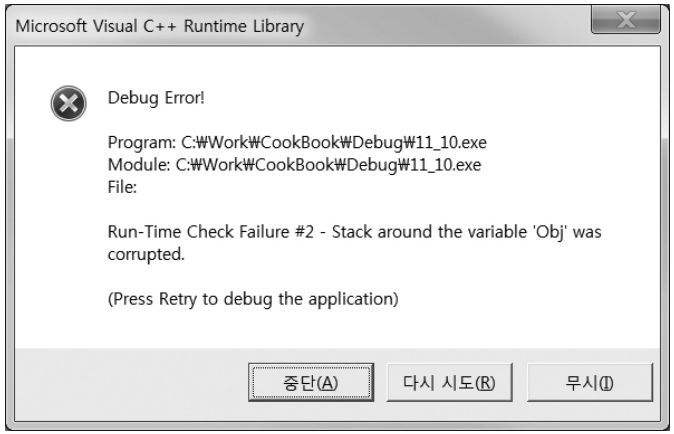
- Add형 포인터 변수 AddPtr이 Add 객체를 무사히 가리키게 되면 부모로부터 상속받는 멤버함수 CalcPrn은 물론 자신의 멤버함수인 AddPrn, Sum까지 모두 사용할 수 있게 된다.

```
AddPtr->CalcPrn();
AddPtr->AddPrn();
AddPtr->Sum();
```

03 업 캐스팅과 다운 캐스팅



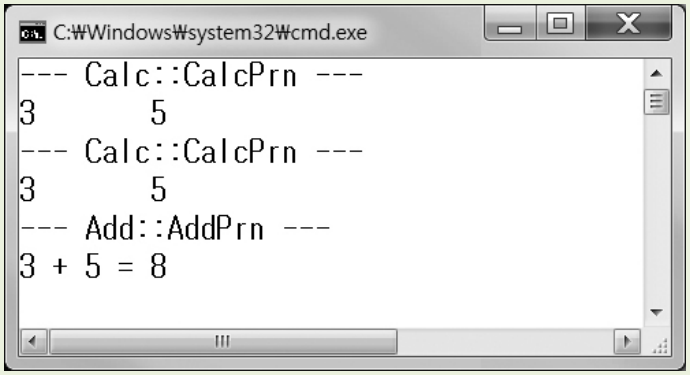
[그림 12-6] **다운 캐스팅**



- 다운 캐스팅을 할 때는 강제 형변환을 해서 컴파일 에러를 피할 수 있지만 프로그램을 실행할 때 에러가 발생하므로 한번 업 캐스팅이 된 객체만 다운 캐스팅해야 한다.

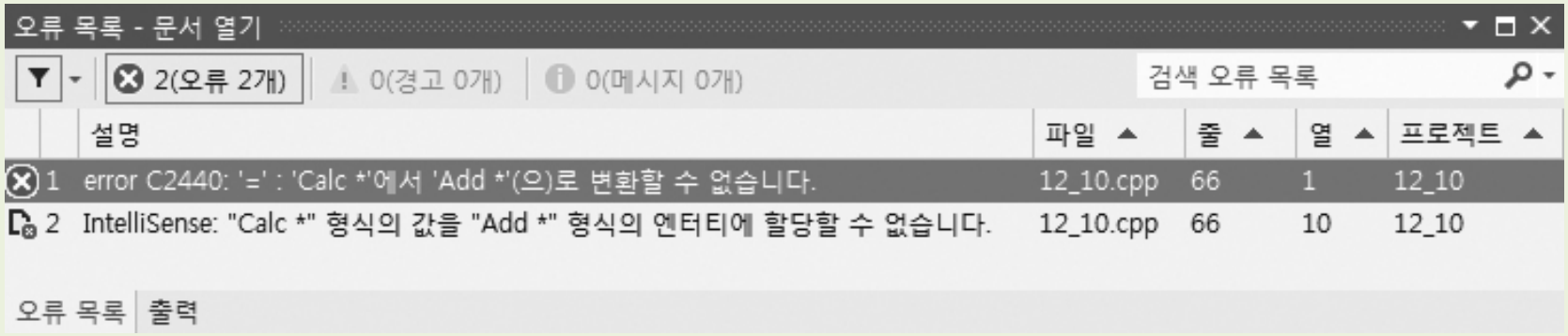
예제 12-9. 다운 캐스팅하기(12_09.cpp)


```
63 void main()
64 {
65 Calc *CalcPtr;
66 CalcPtr = new Add(3, 5); // 업 캐스팅
67 CalcPtr->CalcPrn();
68 // CalcPtr->Sum();
69
70 Add *AddPtr;
71 AddPtr=(Add *)CalcPtr; // 다운 캐스팅
72 AddPtr->CalcPrn();
73 AddPtr->Sum();
74 AddPtr->AddPrn();
75 }
```



예제 12-10. 다운 캐스팅의 잘못된 예 알아보기(12_10.cpp)


```
63 void main()
64 {
65 Calc Obj(3, 5);
66 Add *AddPtr ;
67 AddPtr = &Obj;
68 // AddPtr = (Add *)&Obj;
69 AddPtr->CalcPrn();
70 AddPtr->Sum();
71 }
```



03 업 캐스팅과 다운 캐스팅

- 다운 캐스팅을 다시 한 번 정리해 보자.
  - ❶ 다운 캐스팅은 파생 클래스로 형변환하는 것이다.
  - ❷ 다운 캐스팅은 참조 가능한 영역이 확대되는 것을 의미한다.
  - ❸ 다운 캐스팅에 대해서는 컴파일러가 자동으로 형변환하지 않는다.
  - ❹ 다운 캐스팅은 프로그래머가 명시적으로 형변환해줘야만 컴파일상의 에러를 방지할 수 있다.
  - ❺ 다운 캐스팅은 강제 형변환한 후에도 실행 시 예외사항이 발생할 수 있으므로 인스턴스의 클래스형과 참조하는 포인터 변수의 상속 관계를 생각해서 명시적 형변환을 해야 한다(한번 업 캐스팅이 된 포인터 값을 다운 캐스팅하는 경우에만 안전하다).

업 캐스팅과 멤버함수 오버라이딩

- 기반 클래스인 Calc의 Prn 멤버함수를 파생 클래스인 Add 클래스에 오버라이딩해 보자.

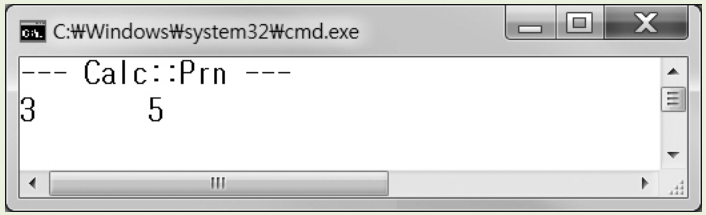
```
class Calc{
public :
  void Prn();
};
class Add : public Calc{
public :
  void Prn(); // 오버라이딩
};
void main()
{
  Calc *CalcPtr;
  CalcPtr= new Add(3, 5); // 업 캐스팅 ------- ❶
  CalcPtr->Prn(); // ---------------- ❷
}
```

03 업 캐스팅과 다운 캐스팅

- 파생 클래스로 객체를 생성해서 기반 클래스형 포인터 변수가 가리키도록 한다(❶). 암시적으로 업 캐스팅되므로 컴파일 에러는 발생하지 않는다. 이런 관계에서 오버라이딩된 멤버함수를 기반 클래스로 선언된 포인터 변수로 호출하면(❷) 파생 클래스의 오버라이딩된 멤버함수(Add::Prn)가 호출되지 못하고 기반 클래스의 멤버함수(Calc::Prn)가 호출된다.

예제 12-11. 기반 클래스형 포인터 변수로 오버라이딩된 멤버함수 호출하기(12_11.cpp)

책의 소스코드 참고



04 동적 바인딩과 가상함수

- 먼저 다음 2가지 사항을 설정해 보자.
  ❶ 파생 클래스에서 기반 클래스의 멤버함수를 오버라이딩한다.
  ❷ 파생 클래스형 객체의 주소를 기반 클래스형 포인터 변수가 저장한다(자동 형변환에 의한 업 캐스팅).
- 이런 사항에서 기반 클래스형 객체 포인터로 접근해서 오버라이딩된 멤버함수를 호출할 경우 포인터 변수를 선언한 클래스형에 의해서 호출되는 멤버함수가 어느 클래스 소속인지 결정된다. 예를 통해 알아보자.
- 멤버함수를 호출하는 객체(x)가 기반 클래스(Calc)이면 기반 클래스의 멤버함수(Calc::Prn)가 호출되고 멤버함수를 호출하는 객체(y)가 파생 클래스(Add)이면 파생 클래스의 멤버함수(Add::Prn)가 호출된다.

```
Calc x(3, 5);
x.Prn(); // Calc::Prn 멤버함수 호출
Add y(3, 5);
y.Prn(); // Add::Prn 멤버함수 호출
```

- 객체 포인터로 접근해 보자.

```
Calc *CalcPtr = &x;;
CalcPtr->Prn(); // Calc::Prn 멤버함수 호출
Add *AddPtr = &y;
AddPtr->Prn(); // Add::Prn 멤버함수 호출
```

- 기반 클래스형 포인터 변수(CalcPtr)에 기반 클래스형 객체(x) 주소를 저장했다면 기반 클래스형 포인터 변수로 호출되는 멤버함수는 당연히 기반 클래스의 멤버함수(Calc::Prn)가 호출된다. 파생 클래스형 포인터 변수(AddPtr)에 파생 클래스형 객체(y) 주소를 저장했다면 파생 클래스형 포인터 변수로 호출되는 멤버함수는 당연히 파생 클래스의 멤버함수(Add::Prn)가 호출된다.

04 동적 바인딩과 가상함수

- 업 캐스팅되었을 때 즉, 기반 클래스형 포인터 변수에 파생 클래스형 객체의 주소를 저장했을 때를 살펴보자.

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

- 컴파일러가 포인터 변수의 자료형을 존중한다면 기반 클래스의 멤버함수(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. 이런 결과가 나타나는 이유를 살펴보자.

정적 바인딩과 동적 바인딩

- 바인딩은 함수 호출을 해당 함수의 정의와 결합해 둔 것이다.

EX) CalcPtr->Prn();

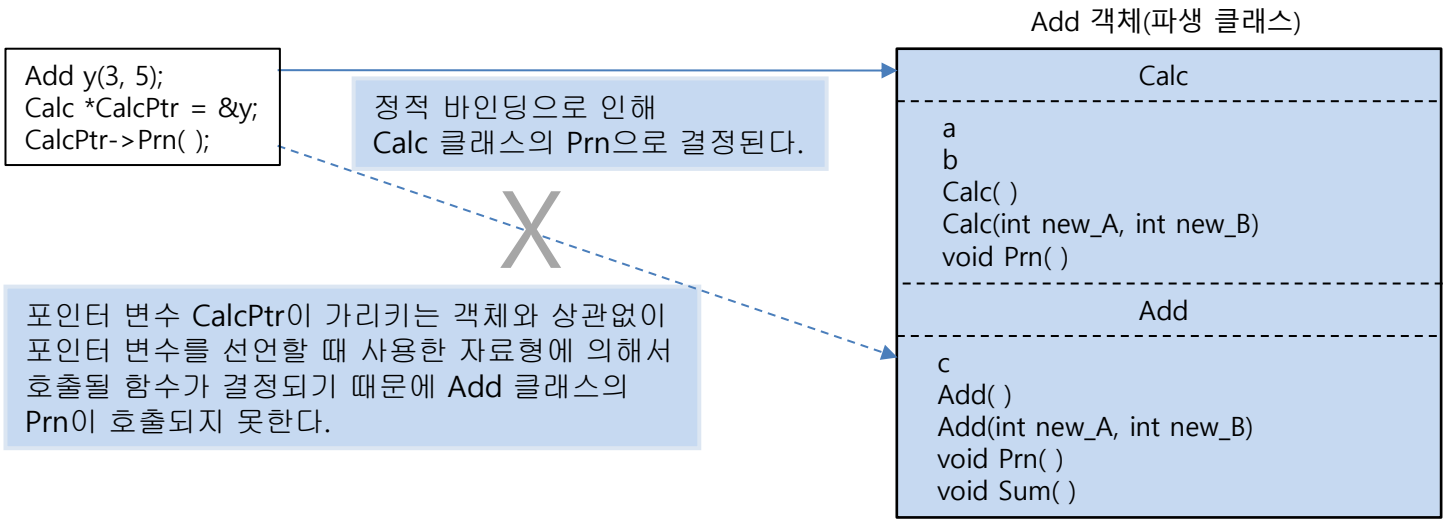
[표 12-2] **컴파일 시점과 실행 시점에 결정되는 작업**

컴파일 시점	실행 시점
변수의 자료형이 결정된다.	변숫값이 저장된다.
호출될 함수가 결정된다.	함수가 실

- 컴파일할 때 미리 호출될 함수를 결정하는 것을 '정적 바인딩(static binding)' 또는 '이른 바인딩(early binding)'이라고 한다.

04 동적 바인딩과 가상함수

- 변수 값이 저장되는 시점은 실행 시점이다. 컴파일 시점에서는 선언된 포인터 변수의 자료형에 대한 정보만 있을 뿐 그 포인터가 어떤 인스턴스를 가리키는지에 대한 정보는 없다.



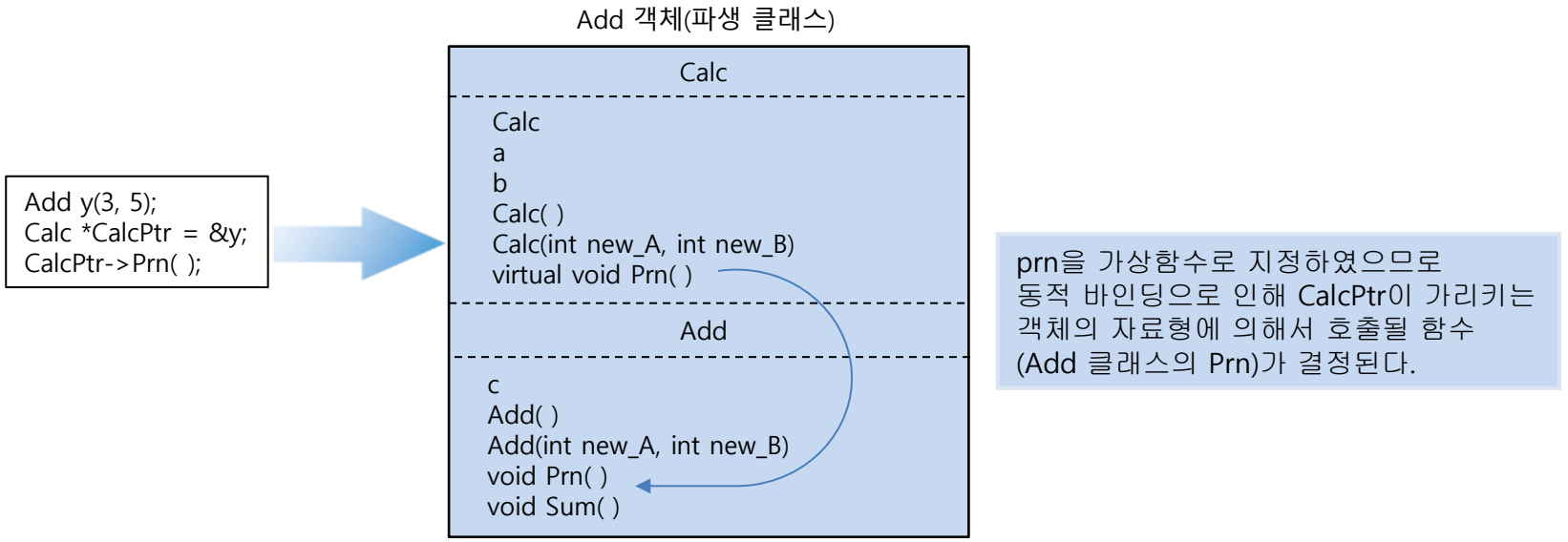
[그림 12-7] **정적 바인딩된 메모리 구조**

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

- 위 예의 출력 결과는 '3 5'이다. 함수는 기본적으로 정적 바인딩을 하기 때문에 객체의 자료형에 상관없이 그 객체를 가리키는 포인터 변수의 자료형에 의존하여 멤버함수가 호출된다. 따라서 '3 5'와 같이 멤버변수 2개만 출력된다.

04 동적 바인딩과 가상함수

- 동적 바인딩을 '늦은 바인딩(lately binding)'이라고도 하는데, 이는 호출할 함수가 컴파일할 때 결정되지 않고 프로그램이 실행되는 동안에 결정되기 때문이다.
- 동적 바인딩을 하려면 동적 바인딩을 하고자 하는 함수가 선언되어 있는 기반 클래스로 가서 함수의 원형 정의 앞에 virtual을 붙이면 된다. 이때 virtual 예약어를 붙인 함수를 가상함수라고 한다. 가상함수는 클래스 내의 멤버함수일 경우에만 지정할 수 있다. 기반 클래스에 예약어 virtual을 붙인 함수는 파생 클래스에서 예약어 virtual을 붙이지 않더라도 가상함수로 지정되고, 가상함수의 특징은 상속된다.

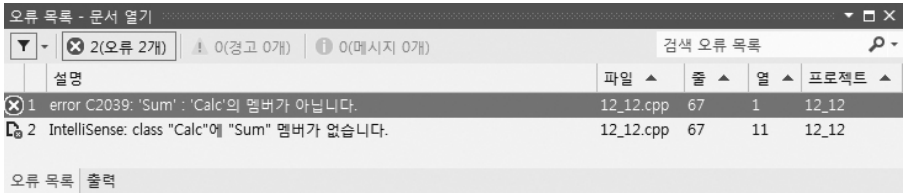


[그림 12-8] 동적 바인딩된 메모리 구조

04 동적 바인딩과 가상함수

동적 바인딩을 위한 조건

- [예제 12-12]의 결과가 '3 + 5 = 8'이 아닌 '3 + 5 = 0'으로 출력된 것은 두 정수값을 더하는 Sum 함수가 호출되지 않아서다. 67행의 주석 처리 부분을 없애고 컴파일하면 컴파일 에러가 발생한다.



- 동적 바인딩은 상속 관계에 있는 클래스에서 오버라이딩된 멤버함수가 존재할 경우 이 멤버함수를 기반 클래스에서 가상함수로 선언해 두었다는 조건에서 업 캐스팅이 된다. 따라서 파생 클래스 객체를 기반 클래스 객체 포인터가 가리켰을 때 기반 클래스 객체 포인터가 가상함수를 호출하게 되면 보다 적합하고 이치에 맞는 함수가 호출된다는 개념이다.
- 그러므로 Prn처럼 파생 클래스의 멤버함수에 접근하려면 Sum을 기반 클래스인 Calc의 가상함수로 선언해 두어야 한다.

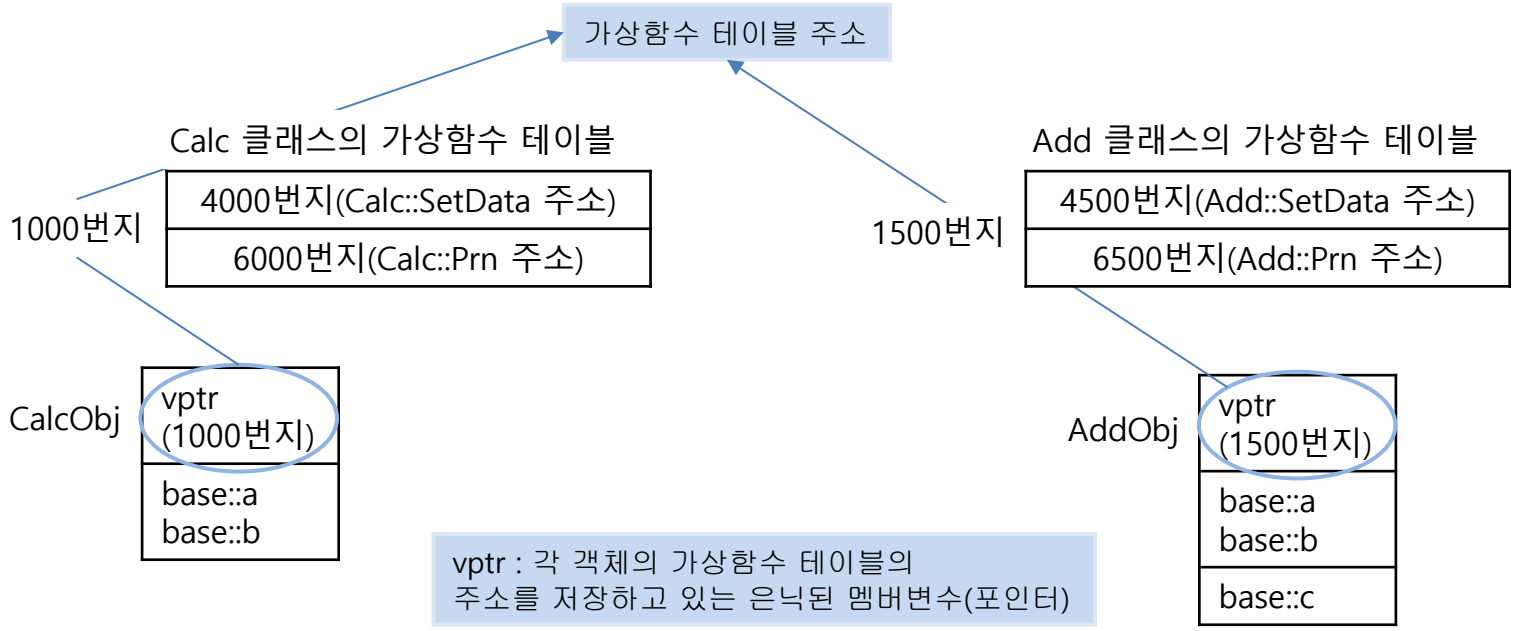
가상함수의 동작 원리

- C++에서 함수가 동적 바인딩을 하도록 하려면 예약어 virtual만 함수의 선언에 붙여 가상함수를 작성하면 된다. 가상함수는 호출될 함수가 실행 시점에 결정되어야 하므로 각 클래스의 가상함수 주소가 가상함수 테이블이라는 기억공간에 따로 저장된다. 가상함수 테이블은 클래스 단위로 만들어지며 가상함수들의 주소는 배열 형태로 저장된다. 해당 클래스로 선언된 객체는 가상함수 테이블의 주소를 저장하는 가상 포인터(vprt)를 갖게 된다. 그리고 가상 포인터는 가상함수 테이블을 가리키게 된다.

04 동적 바인딩과 가상함수

- Calc와 Add 클래스로 객체를 선언했을 때 객체와 가상 테이블이 어떤 형태로 메모리에 할당되는지 알아보자.

```
Calc CalcObj;
Add AddObj;
```



[그림 12-9] 가상함수 테이블

04 동적 바인딩과 가상함수

- 함수가 가상함수로 만들어지면 객체에 의해 오버라이딩된 함수 중 어떤 것을 호출해야 하는지를 알아야 한다. 컴파일러는 Calc 클래스와 Add 클래스에 의해 호출될 함수에 대한 정보를 저장할 수 있도록 가상함수 테이블을 만들어준다. 각 객체들은 가상함수 테이블의 주소를 유지하는 데 필요한 은닉 멤버변수(포인터, vptr)를 가지고 있으며 이 멤버변수는 해당 클래스의 가상함수 테이블을 가리킨다.

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

- 가상함수는 융통성 있게 함수를 호출한다는 장점이 있는 반면 다음과 같은 단점이 있다.
  ❶ 가상함수를 사용하려면 가상 테이블을 유지하려고 가상함수만큼의 주소 배열을 만들어 주므로 메모리에 부담을 줄 수 있다.
  ❷ 각 객체도 가상함수 테이블의 주소를 저장하는 가상 포인터 변수를 위한 메모리를 할당해야 한다.
  ❸ 가상함수가 호출될 때마다 가상함수 테이블을 이용해서 주소를 조사해야 하므로 처리속도가 지연된다.

05 완전 가상함수와 추상 클래스

- 완전 가상함수(pure virtual function)는 함수의 정의없이 함수의 유형만을 기반 클래스에 제시해 놓는 것이다. 완전 가상함수는 가상함수처럼 멤버함수를 선언할 때 예약어 virtual을 선언문의 맨 앞에 붙이고 함수의 선언문 마지막 부분에 '=0'을 덧붙인다. 이렇게 선언된 멤버함수는 함수의 몸체 부분이 없다.

완전 가상함수 기본 형식

```
virtual 반환형 함수명() = 0;
```

- 완전 가상함수를 최소 한 개 이상 갖는 클래스로는 객체를 생성하지 못한다. 이를 '추상 클래스(abstract class)'라 부른다.

추상 클래스와 다형성

- 추상 클래스는 상속을 위한 기반 클래스로 사용된다.
- 예를 들어 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 그린다는 작업과 크기를 알아낸다는 작업은 목적이 동일하므로 함수명을 동일하게 해줄 수 있다.

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

05 완전 가상함수와 추상 클래스

- 동일한 목적을 위한 함수가 하나의 이름으로 사용된다면 또 다른 도형을 위한 클래스에도 적용할 수 있다.

```
triangle.Draw();
pentagonal.Draw();
```

- 하지만 도형과 관련된 모든 클래스에서 그리는 작업을 위해 Draw 멤버함수를 정의했다는 보장이 없으므로 동일한 접근 방식을 제공하려면 추상 클래스를 반드시 설계해야 한다.
- 두 클래스의 공동 부분을 모아 Shape라는 클래스를 만들고 Draw와 GetSize 함수를 완전 가상함수로 선언해 둔다. Shape 클래스에 완전 가상함수는 파생 클래스에게 표준안을 제공하기 위해 등록하는 것이며, Shape는 추상 클래스가 되어 파생 클래스를 설계하기 위한 기반 클래스로만 사용하게 된다.

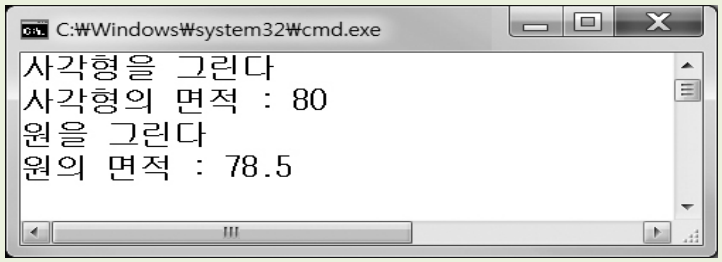
```
class Shape
{
public:
  virtual void Draw()=0;
  virtual double GetSize()=0;
};
```

- Shape 클래스를 기반 클래스로 하는 사각형을 위한 Rect 클래스와 원을 위한 Circle 클래스를 Shape 클래스의 파생 클래스로 설계한다.

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

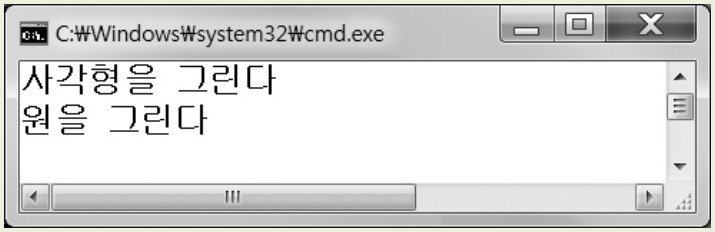
예제 12-13. 추상 클래스와 완전 가상함수를 이용해서 사각형과 원 클래스 설계하기(12_13.cpp)

책의 소스코드 참고



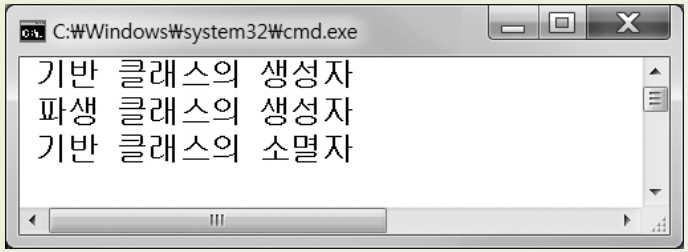
예제 12-14. 추상 클래스와 다형성 이용하기(12_14.cpp)


```
63 void CommonPrn(Shape *ptr)
64 {
65 ptr->Draw();
66 }
67 void main()
68 {
69 // Shape obj;
70
71 CommonPrn(new Rect);
72 CommonPrn(new Circle);
73 }
```



예제 12-15. 일반 소멸자 사용 시 문제점 알아보기(12_15.cpp)


```
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 }
```



05 완전 가상함수와 추상 클래스

- 소멸자를 가상 소멸자로 만들어 놓으면 파생 클래스의 인스턴스를 기반 클래스로 선언된 객체 포인터로 접근하더라도 동적 바인딩에 의해서 호출될 함수가 결정되므로 파생 클래스의 소멸자가 호출된다. 즉, 실체인 객체 인스턴스의 자료형에 맞는 소멸자가 호출된다.

예제 12-16. 가상 소멸자 사용 예 알아보기(12_16.cpp)


```
01 #include<iostream>
02 using namespace std;
03 class Base{
04 public:
05 Base();
06 virtual ~Base();
07 };
```