



Chapter 12. 상속성

목차

1. 클래스간의 상속 관계
2. 상속 관계에서의 생성자
3. 업 캐스팅과 다운 캐스팅
4. 동적 바인딩과 가상함수
5. 완전 가상함수와 추상 클래스

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

- ① 코드의 재활용을 목적으로 하는 개념이 상속이다.
- ② 상속은 부모에게 무엇인가를 물려받는 것을 의미한다.
- ③ 새로 만드는 클래스를 이미 정의된 클래스로부터 많은 기능을 모두 상속받아 정의한다.
- ④ 부모가 되는 클래스를 **기반(base) 클래스**, 자식 클래스에 해당되는 클래스를 **파생(derived) 클래스**라 한다.

■ 기반 클래스와 파생 클래스 만들기

예) 상속을 이용해서 프로그램 코드를 어떻게 재활용할 수 있는지 Calc와 Add 및 Mul 클래스를 통해 살펴보자.

① 기반 클래스 Calc 만들기

- Add와 Mul에서 **공통적으로 기술되는 멤버변수와 멤버함수들로 구성된 기반 클래스 Calc**를 정의하고, 기반 클래스의 상속을 받는 파생 클래스 Add와 Mul을 만들어 두 클래스의 **공통된 내용이 아닌 서로 다른 내용만 각 파생 클래스에서 정의한다.**

[표 12-1] 접근 지정자의 접근 허용 범위

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

01 클래스간의 상속관계

- Add와 Mul 클래스의 공통점만 가진 기반 클래스 Calc를 설계할 때는 Add와 Mul 클래스에서 private로 선언되는 멤버를 다음과 같이 **protected**로 변경하여 기반 클래스 Calc에서 상속받아 사용할 수 있게 해준다.

```
class Calc {  
protected:  
    int a;  
    int b;  
    int c;  
public:  
    void Init(int new_A, int new_B);  
    void Prn();  
};  
  
void Calc::Init(int 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

② 파생 클래스 Add와 Mul 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.

```
class 파생_클래스 : 접근_지정자 기반_클래스 {  
    멤버변수;  
    멤버함수;  
};
```

파생 클래스 기본 형식

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)

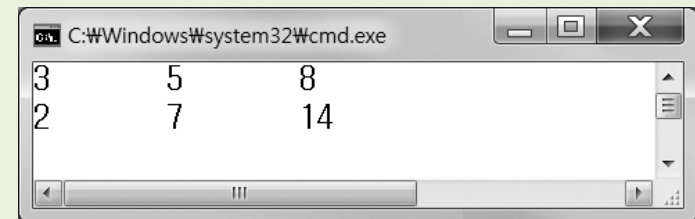
```
#include <iostream>
using namespace std;
class Calc {
protected :
    int a;
    int b;
    int c;
public :
    void Init(int new_A, int new_B);
    void Prn();
};
void Calc::Init(int new_A, int new_B) {
    a=new_A;
    b=new_B;
    c=0;
}
void Calc::Prn() {
    cout<<a<<"\t"<<b<<"\t"<<c<<endl;
}
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;
}

void main() {
    Add x;
    x.Init(3, 5);
    Mul y;
    y.Init(2, 7);
    x.Sum();
    x.Prn();
    y.Gob();
    y.Prn();
}
```



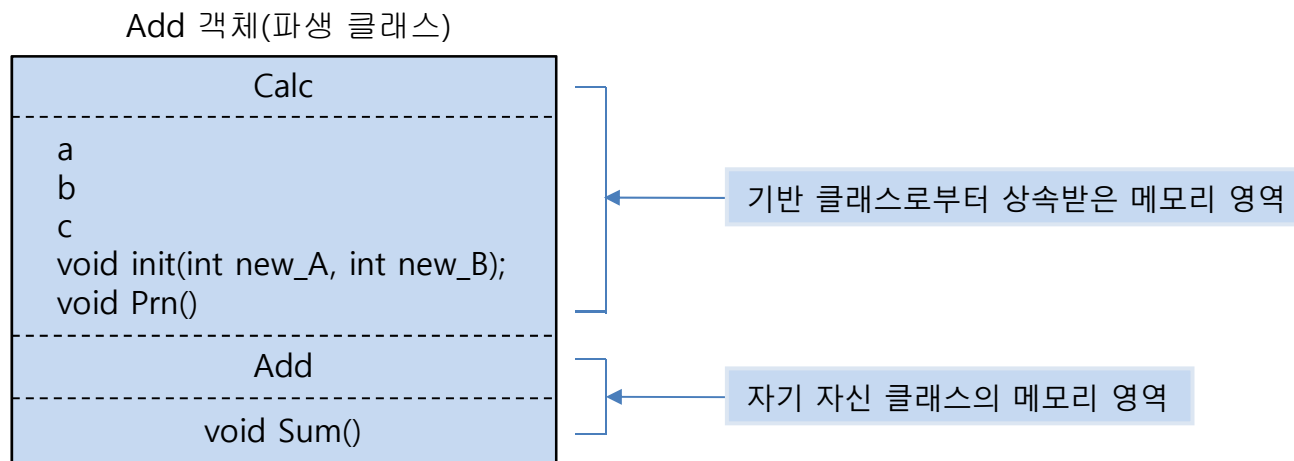
```
C:\Windows\system32\cmd.exe
3      5      8
2      7      14
```

02 상속 관계에서의 생성자

- 상속 관계에 있는 클래스에서 생성자는 다음과 같은 특징을 갖는다.
 - ① 생성자는 멤버함수지만 상속할 수 없다 (파생 클래스의 생성자는 별도로 정의해야 한다).
 - ② 파생 객체가 생성되어도 기반 클래스의 생성자까지 연속적으로 자동 호출된다.

■ 생성자와 소멸자 호출 순서

- 상속 관계에 있는 파생 클래스의 객체를 생성하면, 자신의 생성자 뿐만 아니라 기반 클래스의 생성자도 자동 호출되는데, 기반 클래스의 생성자가 먼저 호출되고 파생 클래스의 생성자가 나중에 호출된다.

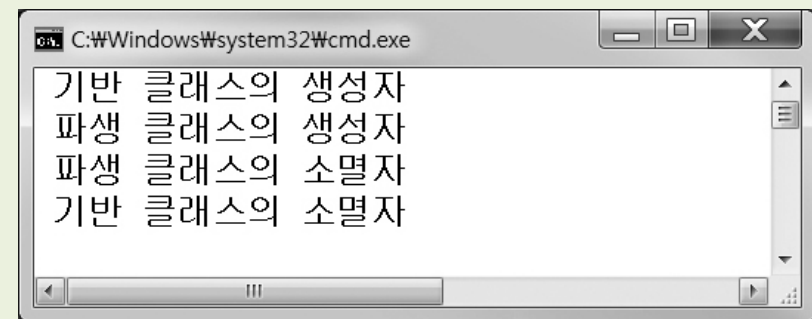


[그림 12-1] 파생 클래스로 선언된 객체의 메모리 구조

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()
```

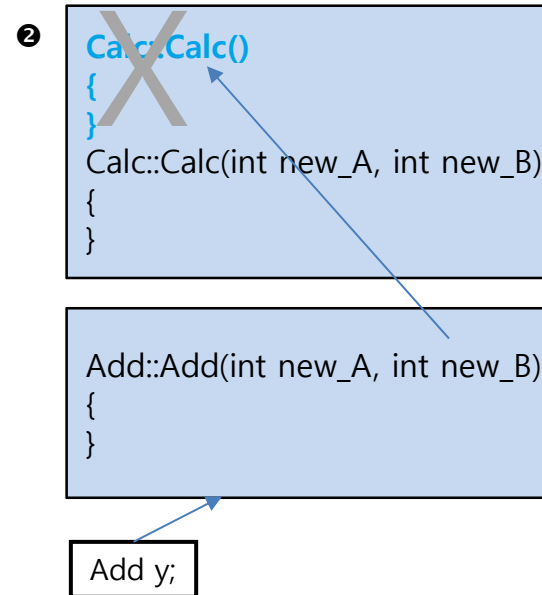
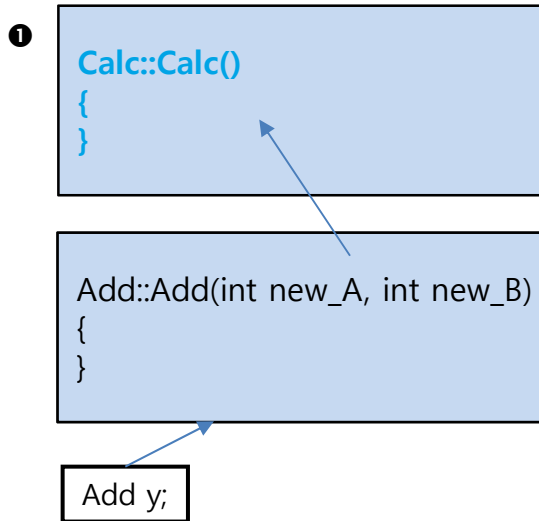
```
22 {
23     cout<<" 파생 클래스의 생성자 "<<endl;
24 }
25 Derived::~Derived()
26 {
27     cout<<" 파생 클래스의 소멸자 "<<endl;
28 }
29
30 void main()
31 {
32     Derived obj;
33 }
```



02 상속 관계에서의 생성자

■ 상속 관계에서 생성자 문제

- ❶은 파생 클래스의 생성자가 암시적으로 기반 클래스의 기본 생성자를 자동으로 호출한다.
- ❷처럼 매개변수를 갖는 생성자를 정의하면 C++ 컴파일러는 더 이상 기본 생성자를 제공하지 않게 된다.



02 상속 관계에서의 생성자

■ 파생 클래스에서 기반 클래스의 생성자를 명시적으로 호출하기

- 기반 클래스의 생성자를 파생 클래스에서 명시적으로 호출하기 위해서는 파생 클래스의 생성자를 정의할 때 : 을 기술한 후 기반 클래스의 생성자를 호출한다.
- 명시적으로 기술하지 않아도 Add 클래스의 생성자는 자신의 기반 클래스인 Calc 클래스의 기본 생성자를 자동 호출한다.

```
Add::Add() : Calc()
```

```
{  
  ....  
}
```

기반 클래스의 생성자가
명시적으로 호출된다.

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

명시적으로 호출하지 않아도 기반 클래스의
생성자가 암시적으로 호출된다.

02 상속 관계에서의 생성자

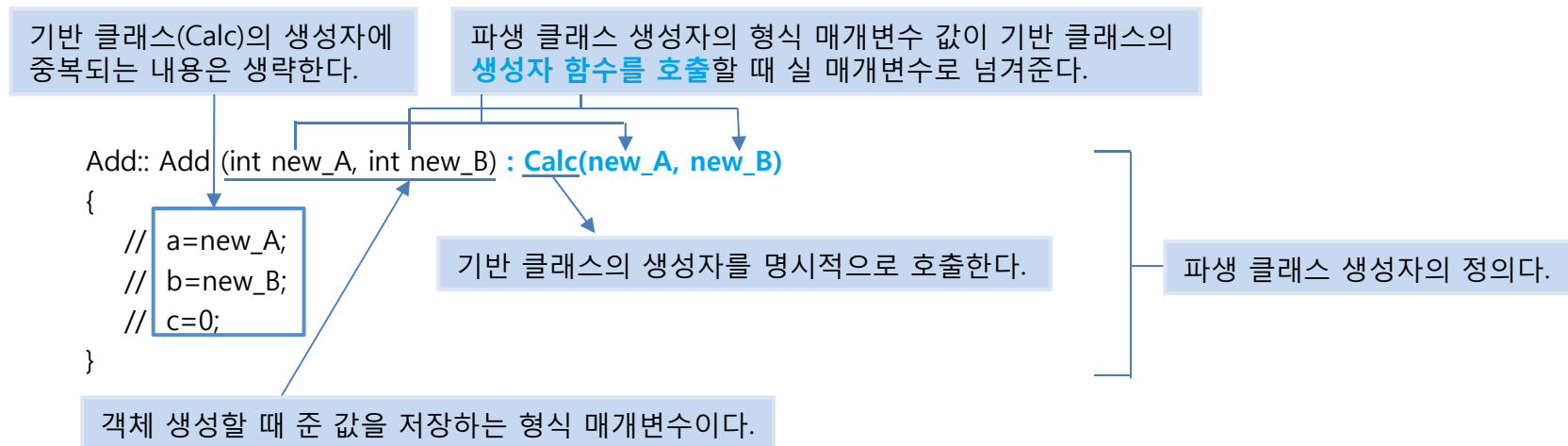
■ 기반 클래스의 생성자에 매개변수 전달하기

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



예제 12-6. 상속 관계에서 생성자 문제 해결하기(12_06.cpp)_1

```
#include<iostream>
using namespace std;
class Calc {
protected:
    int a;
    int b;
    int c;
public:
    void Prn();
    Calc(int new_A,int new_B); // 매개변수가 있는 생성자
    Calc();                   // 기본 생성자
};

void Calc::Prn() {
    cout<<a<<"\t"<<b<<"\t"<<c<<endl;
}

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

```
Calc::Calc() {
    a=0;
    b=0;
    c=0;
}

class Add : public Calc {
public :
    void Sum();
    Add(int new_A,int new_B); // 매개변수가 있는 생성자
    Add();                   // 기본 생성자
};

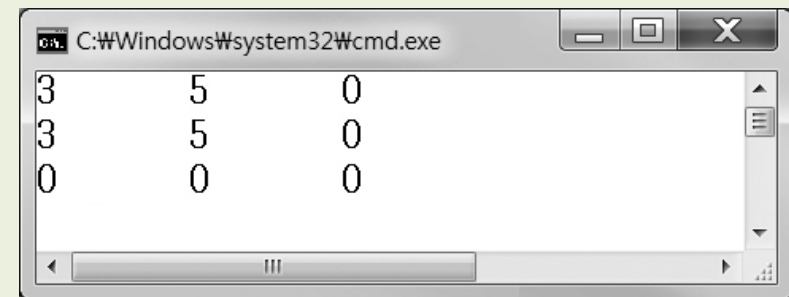
void Add::Sum {
    c=a+b;
}
```

예제 12-6. 상속 관계에서 생성자 문제 해결하기(12_06.cpp)_2

```
Add::Add(int new_A, int new_B) : Calc(new_A, new_B) { // 베이스 클래스의 매개변수가 있는 생성자 함수 호출
    // a=new_A;
    // b=new_B;
    // c=0;
}

Add::Add() : Calc()
{
}

void main() {
    Calc x(3, 5);
    x.Prn();
    Add y(3, 5);
    y.Prn();
    Add z;      // 기본 생성자 함수들 호출
    z.Prn();
}
```



02 상속 관계에서의 생성자

■ 함수의 오버라이딩

- 기반 클래스에 정의되어 있는 함수와 동일한 형태로 파생 클래스에서 다시 정의하는 것을 함수의 오버라이딩 (overriding)이라 한다. 이때 기반 클래스에 정의되어 있는 함수의 원형과 동일한 형태로 정의해야 한다.
- 오버라이딩 되어 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 12-7. 함수 오버라이딩 하기(12_07.cpp)_1

```
#include<iostream>
using namespace std;
class Calc {
protected:
    int a;
    int b;
public:
    Calc();
    Calc(int new_A, int new_B);
    void Prn();
};
```

```
Calc::Calc() {
    a = 0;
    b = 0;
}
```

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

```
void Calc::Prn() {    // 파생 클래스에서 오버라이딩 될 함수
    cout << a << "Wt" << b << endl;
}
```

```
class Add : public Calc {
protected:
    int c;
public:
    Add();
    Add(int new_A, int new_B);
    void Sum();
    void Prn();// 함수 오버라이딩
};
```

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

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


예제 12-7. 함수 오버라이딩 하기(12_07.cpp)_2

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

// 함수 오버라이딩: 기반 클래스의 동일이름 함수와 다르게
// 정의됨
void Add::Prn() {
    cout << a << " + " << b << " = " << c << endl;
}

class Mul : public Calc {
protected:
    int c;
public:
    Mul();
    Mul(int new_A, int new_B);
    void Gob();
    void Prn();    // 함수 오버라이딩
};

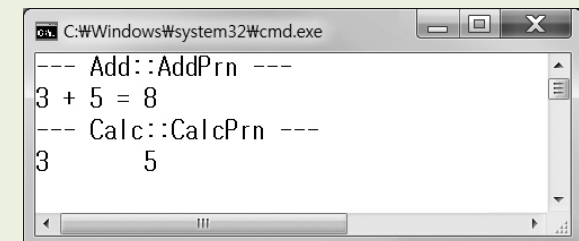
Mul::Mul()
{
}
```

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

void Mul::Gob() {
    c = a*b;
}

void Mul::Prn() {    // 함수 오버라이딩
{
    cout << a << " * " << b << " = " << c << endl;
}

void main() {
    Calc x(3, 5);
    x.Prn();
    Add y(3, 5);
    y.Sum();
    y.Prn();
    Mul z(3, 5);
    z.Gob();
    z.Prn();
}
```



```
C:\Windows\system32\cmd.exe
--- Add::AddPrn ---
3 + 5 = 8
--- Calc::CalcPrn ---
3      5
```

03 업 캐스팅과 다운 캐스팅

■ 형변환

- 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.

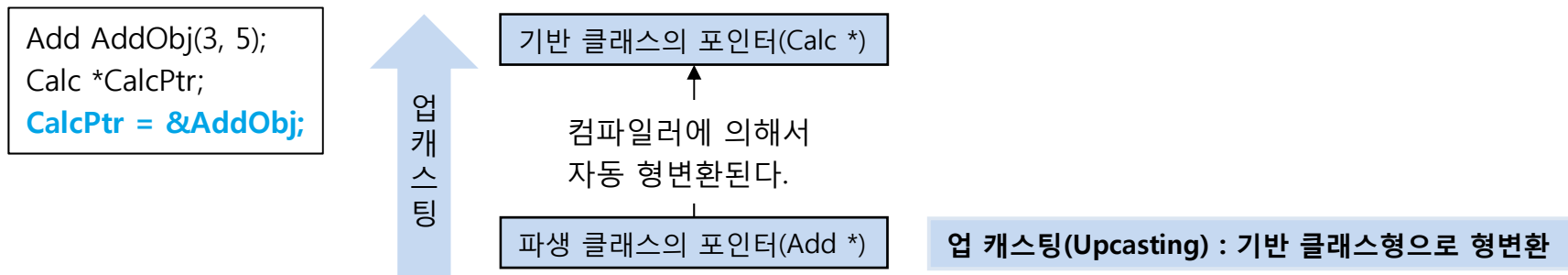
㉠ int i= 5;
double d= 10.6;
d = i;

㉡ 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.

① 업 캐스팅

- 기반 클래스의 포인터 변수가 파생 클래스의 인스턴스를 가리키는 경우, 이를 업 캐스팅이라 하며 컴파일러에서 자동으로 형변환이 이루어진다. 즉, 파생 객체의 포인터를 기반 객체의 포인터로 형변환 하는 것이다.



[그림 12-2] 업 캐스팅

예제 12-8. 업 캐스팅하기(12_08.cpp)_1

```
#include<iostream>
using namespace std;
class Calc {
protected:
    int a;
    int b;
public:
    Calc();
    Calc(int new_A, int new_B);
    void CalcPrn();
};

Calc::Calc() {
    a=0;
    b=0;
}

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

```
void Calc::CalcPrn() {
    cout<<"--- Calc::CalcPrn ---"<<endl;
    cout<<a<<"\t"<<b<<endl;
}

class Add : public Calc {
protected:
    int c;
public :
    Add();
    Add(int new_A, int new_B);
    void Sum();
    void AddPrn();
};

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

예제 12-8. 업 캐스팅하기(12_08.cpp)_2

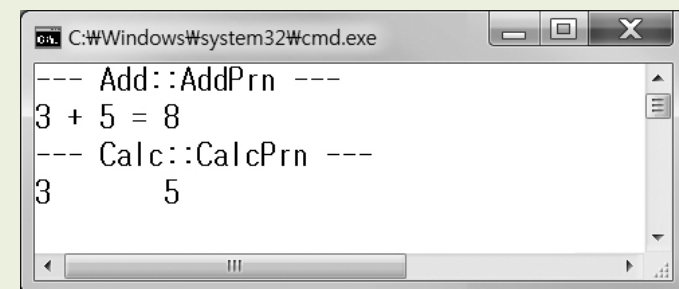
```
Add::Add(int new_A,int new_B) :Calc(new_A, new_B)
{
    a=new_A;
    b=new_B;
    c=0;
}

void Add::Sum()
{
    c=a+b;
}

void Add::AddPrn()
{
    cout<<"--- Add::AddPrn ---"<<endl;
    cout<<a<<" + "<<b<<" = "<<c<<endl;
}
```

```
void main()
{
    Add AddObj(3, 5);
    Add *AddPtr;
    AddPtr= &AddObj;
    AddPtr->Sum();
    AddPtr->AddPrn();

    Calc *CalcPtr;
    CalcPtr = &AddObj; // 업캐스팅
    //CalcPtr->Sum();    // 이미 형변환이 일어났으므로 에러발생
    CalcPtr->CalcPrn();
}
```



```
C:\Windows\system32\cmd.exe
--- Add::AddPrn ---
3 + 5 = 8
--- Calc::CalcPrn ---
3      5
```

03 업 캐스팅과 다운 캐스팅

② 다운 캐스팅

- 다운 캐스팅이란 파생 클래스로 선언된 포인터 변수에 기반 클래스로 선언된 객체의 주소를 저장하는 것인데, 다운 캐스팅에서는 컴파일러가 자동으로 형변환 시키지 않으므로 컴파일 에러가 발생한다.
- 다운 캐스팅을 할 때는 강제 형변환을 통해 컴파일 에러를 피할 수 있지만, 프로그램을 실행할 때 다시 에러가 발생하므로 이미 한번 업 캐스팅이 된 객체에 대해서만 다시 다운 캐스팅한다.

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

오류 목록				
오류 2개 경고 0개 메시지 0개				
검색 오류 목록				
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 엔터티에 할당할 수 없습니다.	12_09.cpp	71	9	12_09



[그림 12-6] 다운 캐스팅

03 업 캐스팅과 다운 캐스팅

■ 업 캐스팅과 멤버함수 오버라이딩

- 기반 클래스인 Calc의 Prn() 멤버함수를 파생 클래스인 Add 클래스에서 오버라이딩 해보자.

```
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
}
```

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

예제 12-11. 기반 클래스형 포인터 변수로 오버라이딩 된 멤버함수 호출하기(12_11.cpp)_1

```
#include<iostream>
using namespace std;
class Calc {
protected:
    int a;
    int b;
public:
    Calc();
    Calc(int new_A, int new_B);
    void Prn();    // 파생 클래스에서 오버라이딩 할 멤버함수
};

Calc::Calc() {
    a = 0;
    b = 0;
}

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

```
void Calc::Prn()    // 파생 클래스에서 오버라이딩 할 멤버함수
{
    cout << "--- Calc::Prn ---" << endl;
    cout << a << "Wt" << b << endl;
}

class Add : public Calc {
protected:
    int c;
public:
    Add();
    Add(int new_A, int new_B);
    void Sum();
    void Prn();    // 기반 클래스의 멤버함수를 오버라이딩
};

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

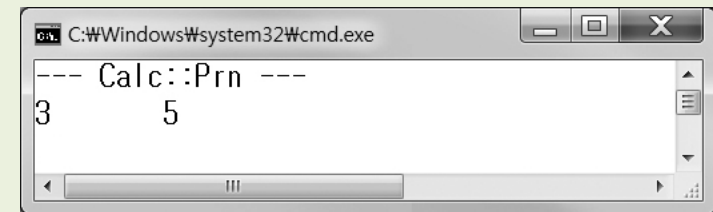
예제 12-11. 기반 클래스형 포인터 변수로 오버라이딩 된 멤버함수 호출하기(12_11.cpp)_2

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

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

```
void Add::Prn()    // 기반 클래스의 멤버함수를 오버라이딩
{
    cout << "--- Add::Prn ---" << endl;
    cout << a << " + " << b << " = " << c << endl;
}
```

```
void main()
{
    Calc *CalcPtr;
    CalcPtr = new Add(3, 5); // 자동 업 캐스팅
    CalcPtr->Prn();          // 기반 클래스의 Prn() 함수를 호출
}
```



04 동적 바인딩과 가상함수

- (예제 12-11과 같이) 먼저 다음 2가지 상황을 설정해 보자.
 - ① 파생 클래스에서 기반 클래스의 멤버함수를 오버라이딩 한다.
 - ② 파생 클래스형 객체의 주소를 기반 클래스형 포인터 변수에 저장한다 (자동 형변환에 의한 업 캐스팅 경우)
- 이런 상황에서 기반 클래스형 객체 포인터로 접근해서 오버라이딩 된 멤버함수를 호출하는 경우
선언된 포인터 변수의 클래스형이 무엇인지에 따라서 호출되는 멤버함수가 어느 클래스 소속인지 결정된다.

(예제)

- 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
```

04 동적 바인딩과 가상함수

- 여기서 업 캐스팅되었을 때, 즉 기반 클래스형 포인터 변수에 파생 클래스형 객체의 주소를 저장했을 때는??
 - 기반 클래스의 멤버함수인 Calc::Prn()가 호출된다. 즉, 오버라이딩 된 파생 클래스의 멤버함수를 호출할 수 없다.

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

■ 정적 바인딩과 동적 바인딩

- '바인딩'이란 함수 호출을 해당 클래스의 멤버함수 정의와 결합해 둔 것이다. 즉, EX) CalcPtr->Prn();

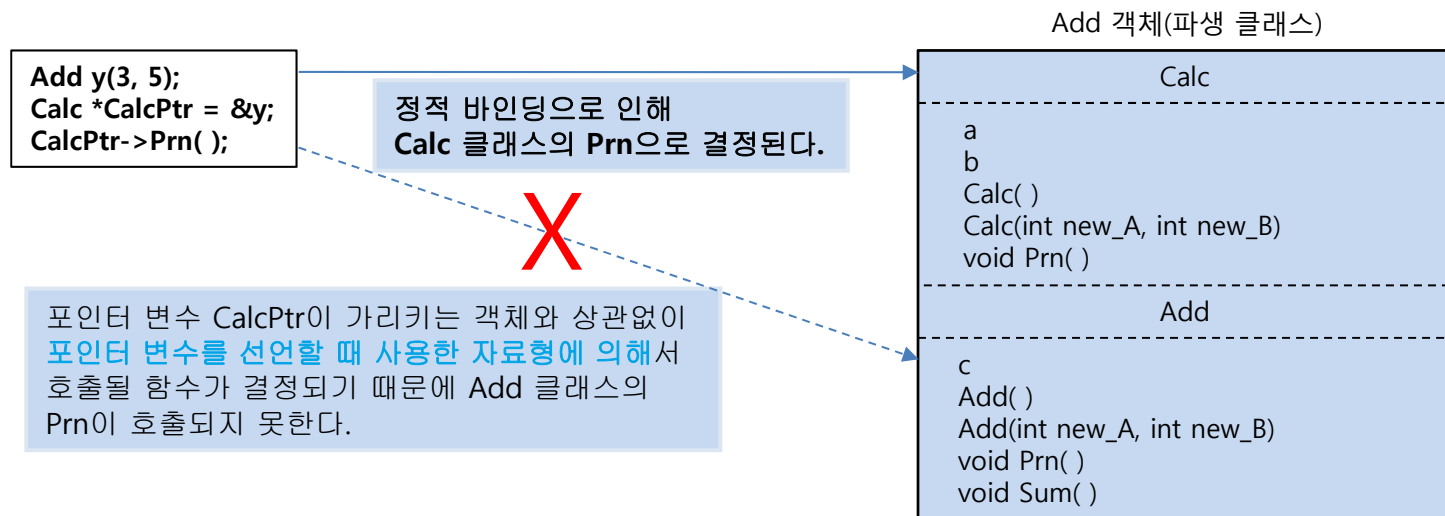
[표 12-2] 컴파일 시점과 실행 시점에 결정되는 작업

컴파일 시점	실행 시점
변수의 자료형이 결정된다.	변수값이 저장된다.
호출될 함수가 결정된다.	함수가 실행된다.

04 동적 바인딩과 가상함수

① 정적 바인딩

- 컴파일 때 미리 호출될 함수를 결정하는 것을 **정적 바인딩(static binding)**이라고 한다.
- 변수 값이 저장되는 시점은 실행 시점이다. 컴파일 시점에서는 선언된 포인터 변수의 자료형에 대한 정보만 있을 뿐 그 포인터가 실제 어떤 인스턴스를 가리키는지에 대한 정보는 없다.



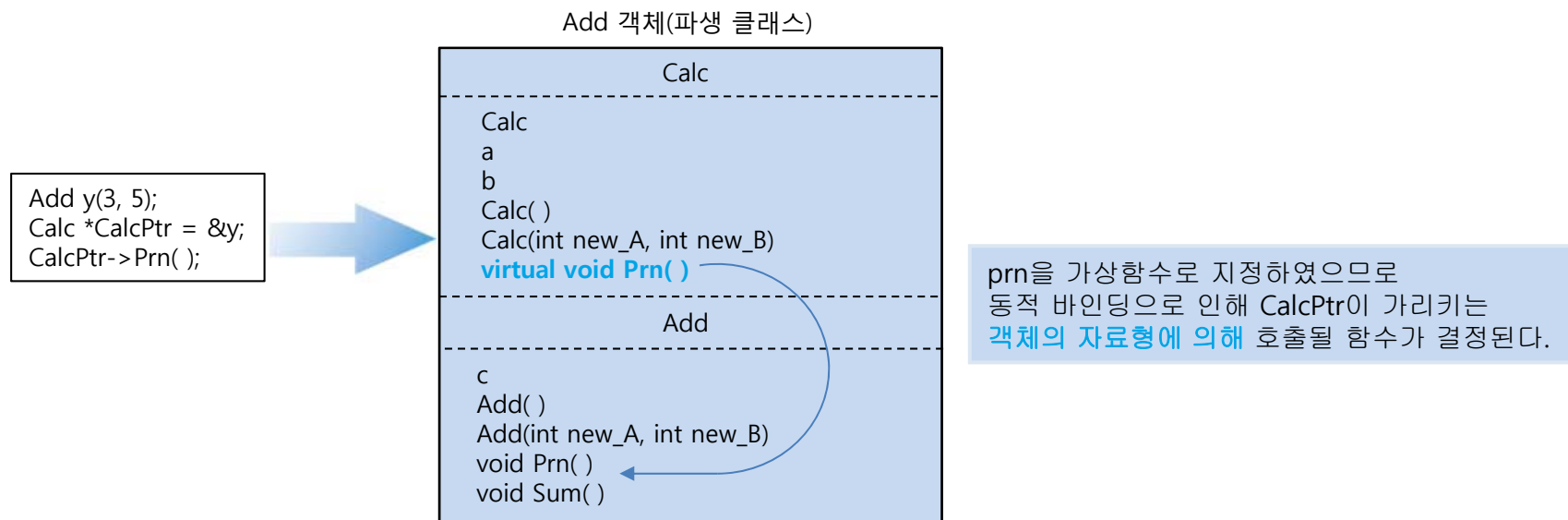
[그림 12-7] 정적 바인딩된 메모리 구조

- 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.

04 동적 바인딩과 가상함수

② 동적 바인딩

- 동적 바인딩(dynamic binding)은 '늦은 바인딩(lately binding)'이라고도 한다.
이는 호출할 함수가 컴파일할 때 결정되지 않고, 프로그램이 실행되는 동안에 결정된다.
- C++에서 상속 관계에 있는 클래스에서 오버라이딩 된 멤버함수가 존재하는 경우,
동적 바인딩을 위하여 기반 클래스의 함수 원형에 **virtual** 키워드를 붙인다. 이때 **virtual**을 붙인
함수를 **가상함수**라고 한다. 가상함수는 클래스 내의 멤버함수일 경우에만 지정할 수 있다.



[그림 12-8] 동적 바인딩 된 메모리 구조

예제 12-12. 가상함수로 동적 바인딩 하기(12_12.cpp)_1

```
#include<iostream>
using namespace std;
class Calc {
protected:
    int a;
    int b;
public:
    Calc();
    Calc(int new_A, int new_B);
    virtual void Prn();    // 기반 클래스에서 가상함수 선언
    //void Prn();
};
Calc::Calc() {
    a = 0;
    b = 0;
}
Calc::Calc(int new_A, int new_B) {
    a = new_A;
    b = new_B;
}
```

```
void Calc::Prn() {
    cout << "--- Calc::Prn ---" << endl;
    cout << a << "Wt" << b << endl;
}

class Add : public Calc {
protected:
    int c;
public:
    Add();
    Add(int new_A, int new_B);
    void Sum();
    void Prn();
};
Add::Add() : Calc() {
}
Add::Add(int new_A, int new_B) : Calc(new_A, new_B) {
    a = new_A;
    b = new_B;
    c = 0;
}
```

예제 12-12. 가상함수로 동적 바인딩 하기(12_12.cpp)_2

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

void Add::Prn() {
    cout << "--- Add::Prn ---" << endl;
    cout << a << " + " << b << " = " << c << endl;
}

void main() {
    Calc *CalcPtr;
    CalcPtr = new Add(3, 5); // Add 클래스의 객체를 가리키는 포인터
    //CalcPtr->Sum();
    CalcPtr->Prn();          // 실행 시 동적 바인딩에 의해 Add 클래스의 Prn()함수 호출
}
```

05 완전 가상함수와 추상 클래스

- 완전 가상함수(pure virtual function)는 함수의 정의없이 함수의 유형만을 기반 클래스에 제시해 놓는 것.
`virtual`을 선언문의 맨 앞에 붙이고 함수의 선언문 마지막 부분에 `'=0'`을 덧붙인다.
이렇게 선언된 멤버함수는 함수의 몸체 부분이 없다.

```
virtual 반환형 함수명() = 0;
```

완전 가상함수 기본 형식

- 완전 가상함수를 한 개 이상 갖는 클래스는 객체를 생성하지 못한다. 이를 추상 클래스(abstract class)라 함.

■ 추상 클래스와 다형성

- 추상 클래스는 상속을 위한 기반 클래스로 사용된다.

예)

- 사각형을 클래스로 정의해 보자: 사각형을 그리기 위한 함수와 크기를 알려주기 위한 함수로 구성.
- 원을 클래스로 정의해 보자: 원을 그리기 위한 함수와 크기를 알려주기 위한 함수로 구성.
- 사각형과 원 클래스의 구체적인 내용은 다르지만 그린다는 작업과 크기를 알아낸다는 작업은 목적이 동일하므로 함수명을 동일하게 해줄 수 있으며, 사각형과 원 이외의 다른 모양에도 적용 가능하다.

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

05 완전 가상함수와 추상 클래스

- 두 클래스의 공통 부분을 모아 Shape라는 기반 클래스를 만들고 Draw와 GetSize 함수를 완전 가상함수로 선언해 둔다. **Shape 클래스의 완전 가상함수는 파생 클래스에게 표준안을 제공하기 위해 등록하는 것이며, Shape는 모양이라는 추상적인 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)_1

```
#include<iostream>
using namespace std;
class Shape
{
protected:
    double area;
public:
    virtual void Draw() = 0;      // 완전 가상함수
    virtual double GetSize() = 0; // 완전 가상함수
};

class Rect : public Shape{
protected:
    int width;
    int height;
public:
    Rect(int w = 0, int h = 0);
    void Draw();      // 오버라이딩
    double GetSize(); // 오버라이딩
};
```

```
Rect::Rect(int w, int h)
{
    width = w;
    height = h;
}

void Rect::Draw()      // 오버라이딩
{
    cout << "사각형을 그린다" << endl;
}

double Rect::GetSize() // 오버라이딩
{
    area = width*height;
    return area;
}
```

예제 12-13. 추상 클래스와 완전 가상함수를 이용해서 사각형과 원 클래스 설계하기(12_13.cpp)_2

```
class Circle : public Shape{
protected:
    int radius;
public:
    Circle(int r = 0);
    void Draw();          // 오버라이딩
    double GetSize();     // 오버라이딩
};

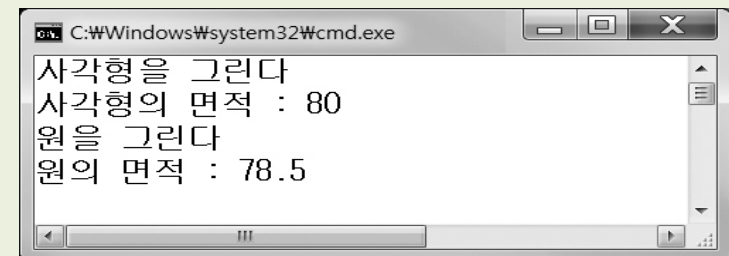
Circle::Circle(int r)
{
    radius = r;
}

void Circle::Draw()      // 오버라이딩
{
    cout << "원을 그린다" << endl;
}
```

```
double Circle::GetSize() // 오버라이딩
{
    area = radius*radius*3.14;
    return area;
}

void main()
{
    Rect recObj(8, 10);
    recObj.Draw();
    cout << "사각형의 면적 : " << recObj.GetSize() << endl;

    Circle cirObj(5);
    cirObj.Draw();
    cout << "원의 면적 : " << cirObj.GetSize() << endl;
}
```



예제 12-16. 가상 소멸자 사용 예 (12_16.cpp)

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

class Base {
public:
    Base();
    // 가상 소멸자 함수로 지정하여 파생클래스의 소멸자가
    // 호출될 수 있도록 함
    virtual ~Base();
};

Base::Base() {
    cout << " 기반 클래스의 생성자 " << endl;
}

Base::~~Base() {
    cout << " 기반 클래스의 소멸자 " << endl;
}
```

```
class Derived : public Base {
public:
    Derived();
    ~Derived();
};

Derived::Derived() {
    cout << " 파생 클래스의 생성자 " << endl;
}

Derived::~~Derived() {
    cout << " 파생 클래스의 소멸자 " << endl;
}

void main() {
    // 동적 메모리를 할당하여 파생 클래스의 객체를 생성하고
    // 포인터를 기반 클래스에 할당
    Base *BasePtr = new Derived;
    // 동적 메모리 해제 (가상함수에 의해 파생클래스 소멸자 호출)
    delete BasePtr;
}
```

Homework

- Chapter 12 Exercise: 3, 4, 5, 6, 7, 10, 11, 13, 14, 16, 18, 19, 20