

Chapter 07. 상속과 다형성

박종혁 교수

UCS Lab

Tel: 970-6702

Email: jhpark1@seoultech.ac.kr

07-1. 객체 포인터의 참조관계

상속된 객체와 포인터 관계

- 객체 포인터
 - 객체의 주소 값을 저장할 수 있는 포인터
 - A클래스의 포인터는 A 객체의 주소 뿐만 아니라 A 클래스를 상속하는 파생 클래스의 객체의 주소 값도 저장 가능
 - A클래스의 참조자는 A객체 뿐만 아니라 A클래스를 상속하는 파생 클래스의 객체도 참조 가능

상속된 객체의 포인터, 참조자

- 객체 포인터의 권한
 - 포인터를 통해서 접근할 수 있는 객체 멤버의 영역
 - A클래스의 객체 포인터는 A클래스의 멤버와 A클래스가 상속받은 베이스 클래스의 멤버만 접근 가능
 - A클래스의 참조자는 A클래스의 멤버와 A클래스가 상속받은 베이스 클래스의 멤버만 접근 가능

상속된 객체와 참조 관계

- 객체의 레퍼런스
 - 객체를 참조할 수 있는 레퍼런스
 - 클래스 포인터의 특성과 일치
- 객체 레퍼런스의 권한
 - 객체를 참조하는 레퍼런스의 권한
 - 클래스 포인터의 권한과 일치

객체의 주소 값을 저장하는 객체 포인터 변수

“ C++에서, **A형 포인터 변수**는 **A객체** 또는 **A를 직접** 혹은 **간접적으로 상속하는 모든 객체**를 가리킬 수 있다 (객체의 주소 값을 저장할 수 있다).”

```
class Person
{
    . . . . .
};
```

```
class Student : public Person
{
    . . . . .
};
```

```
class PartTimeStudent public Student
{
    . . . . .
};
```

```
Person * ptr=new Student();
```

```
Person * ptr=new PartTimeStudent();
```

```
Student * ptr=new PartTimeStudent();
```

예제

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

class Person
{
public:
    void Sleep() { cout<<"Sleep"<<endl; }
};

class Student : public Person
{
public:
    void Study() { cout<<"Study"<<endl; }
};

class PartTimeStudent : public Student
{
public:
    void Work() { cout<<"Work"<<endl; }
};
```

```
int main(void)
{
    Person * ptr1=new Student();

    Person * ptr2=new
    PartTimeStudent();

    Student * ptr3=new
    PartTimeStudent();

    ptr1->Sleep();
    ptr2->Sleep();
    ptr3->Study();

    delete ptr1; delete ptr2; delete
    ptr3;

    return 0;
}
```

파생 클래스의 객체도 가리키는 포인터!

IS-A 관계

“학생(Student)은 사람(Person)의 일종이다.”

“근로학생(PartTimeStudent)은 학생(Student)의 일종이다.”

“근로학생(PartTimeStudent)은 사람(Person)의 일종이다.”



파생 클래스 객체를 베이스 클래스 객체로 바라볼 수 있는 근거

“학생(Student)은 사람(Person)이다.”

“근로학생(PartTimeStudent)은 학생(Student)이다.”

“근로학생(PartTimeStudent)은 사람(Person)이다.”

함수 오버라이딩(Function Overriding)

개념



함수 오버라이딩

- 파생 클래스에서 기본 클래스의 가상 함수와 동일한 이름의 함수 선언
 - 기본 클래스의 가상 함수의 존재감 상실시킴
 - 파생 클래스에서 오버라이딩한 함수가 호출되도록 동적 바인딩
 - 함수 재정의라고도 부름
 - 다형성의 한 종류
- 함수 오버라이딩 되면, 오버라이딩 된 베이스 클래스의 함수는 오버라이딩을 한 파생클래스의 함수에 가려짐
- 주의) 함수 오버로딩과 혼동 금지 !!

C++ 오버라이딩의 특징

- 오버라이딩의 성공 조건
 - 가상 함수 이름, 매개 변수 타입과 개수, 리턴 타입이 모두 일치

```
class Base {
public:
    virtual void fail();
    virtual void success();
    virtual void g(int);
};

class Derived : public Base {
public:
    virtual int fail(); // 오버라이딩 실패. 리턴
타입이 다름
    virtual void success(); // 오버라이딩 성공
};
```

```
class Base {
public:
    virtual void f();
};

class Derived : public Base {
public:
    virtual void f(); // virtual void f()와
동일한 선언 → 생략 가능
};
```

- 오버라이딩 시 `virtual` 지시어 생략 가능
 - 가상 함수의 `virtual` 지시어는 상속됨, 파생 클래스에서 `virtual` 생략 가능
- 가상 함수의 접근 지정
 - `private`, `protected`, `public` 중 자유롭게 지정 가능

함수 오버라이딩 vs 함수 오버로딩

- 베이스 클래스와 동일한 이름의 함수를 파생 클래스에서 정의한다고 해서 무조건 함수 오버라이딩이 되는 것은 아님.
- 함수 오버로딩
 - 매개변수의 자료형 및 개수가 다를 경우
 - 전달되는 인자에 따라서 호출되는 함수가 결정
 - 함수 오버로딩은 상속의 관계에서도 구성 가능

07-2. 가상함수(Virtual Function)

가상함수(virtual function)

- 가상함수는 베이스클래스 내에서 정의된 멤버함수를 파생클래스에서 재정의하고자 할때 사용
 - 베이스클래스의 멤버함수와 같은 이름을 갖는 함수를 파생클래스에서 재정의함으로써 각 클래스마다 고유의 기능을 갖도록 변경할때 이용
 - 파생클래스에서 재정의되는 가상함수는 함수중복과 달리 베이스클래스와 함수의 반환형, 인수의 갯수, 형이 같아야함
- 가상함수를 정의하기 위해서는 가장 먼저 기술되는 베이스클래스의 멤버 함수 앞에 **virtual**이라는 키워드로 기술

가상함수 예

```
#include <iostream>
using std::endl;
using std::cout;

class AAA
{
public:
    virtual void fct(){ // 가상함수
    cout<<"AAA"<<endl;
    }
};

class BBB : public AAA
{
public:
    void fct(){
    cout<<"BBB"<<endl;
    }
};
```

```
int main(void)
{
    BBB* b=new BBB;
    b->fct();

    AAA* a=b;
    a->fct();

    delete b;
    return 0;
}
```

```
BBB
BBB
```

- 오버라이딩 된 함수가 virtual이면 오버라이딩 한 함수도 자동 virtual

```

class First
{
public:
    virtual void MyFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    virtual void MyFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    virtual void MyFunc() { cout<<"ThirdFunc"<<endl; }
};

```

```

int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    fptr->MyFunc();
    sptr->MyFunc();
    tptr->MyFunc();
    delete tptr;
    return 0;
}

```

실행결과

```

ThirdFunc
ThirdFunc
ThirdFunc

```

- 포인터의 형에 상관 없이 포인터가 가리키는 객체의 마지막 오버라이딩 함수를 호출

가상함수 특성 상속 예

```

#include <iostream>
using std::endl;
using std::cout;
class AAA
{
public:
    virtual void fct(){ // 가상함수
        cout<<"AAA"<<endl;
    }
};
class BBB : public AAA
{
public:
    void fct(){ // virtual void fct()
        cout<<"BBB"<<endl;
    }
};
class CCC : public BBB
{
public:
    void fct(){
        cout<<"CCC"<<endl;
    }
};

```

```

int main(void)
{
    BBB* b=new BBB;
    b->fct();

    AAA* a=b;
    a->fct();

    delete b;
    return 0;
}

```

```

CCC
CCC

```

순수 가상함수

- **순수 가상함수(pure virtual function)**
 - 베이스 클래스에서는 어떤 동작도 정의되지 않고 함수의 선언만을 하는 가상함수
 - 순수 가상함수를 선언하고 파생 클래스에서 이 가상함수를 중복 정의하지 않으면 컴파일 시에 에러가 발생
 - 하나 이상의 멤버가 순수 가상함수인 클래스를 **추상 클래스(abstract class)**라 함
 - 완성된 클래스가 아니기 때문에 객체화되지 않는 클래스
 - 베이스 클래스에서 다음과 같은 형식으로 선언

```
virtual 자료형 함수명(인수 리스트) = 0;
```

추상 클래스의 목적

- 추상 클래스의 목적
 - 추상 클래스의 인스턴스를 생성할 목적 아님
 - 상속에서 기본 클래스의 역할을 하기 위함
 - 순수 가상 함수를 통해 파생 클래스에서 구현할 함수의 형태(원형)을 보여주는 인터페이스 역할
 - 추상 클래스의 모든 멤버 함수를 순수 가상 함수로 선언할 필요 없음

순수 가상 함수 예 (1)

```
#include <iostream>
using std::endl;
using std::cout;
class Date {          // 베이스 클래스
protected:
    int year,month,day;
public:
    Date(int y,int m,int d)
        { year = y; month = m; day = d; }
    virtual void print() = 0; // 순수 가상함수
};
class Adate : public Date {
        // 파생 클래스 Adate
public:
    Adate(int y,int m,int d) : Date(y,m,d)
        { /* no operation */ }
    void print()          // 가상함수
        { cout << year << '.' << month << '.' << day << ".\n"; }
};
class Bdate : public Date {
        // 파생 클래스 Bdate
public:
    Bdate(int y,int m,int d) : Date(y,m,d)
        { /* no operation */ }
    void print();        // 가상함수
};
```

void Bdate::print()

```
{
    static char *mn[] = {
        "Jan.", "Feb.", "Mar.", "Apr.", "May",
        "June","July", "Aug.", "Sep.", "Oct.",
        "Nov.,"Dec." };
    cout << mn[month-1] << ' ' << day
        << ' ' << year << '\n';
}

int main()
{
    Adate a(1994,6,1);
    Bdate b(1945,8,15);
    Date &r1 = a, &r2 = b; // 참조자
    r1.print();
    r2.print();
    return 0;
}
```

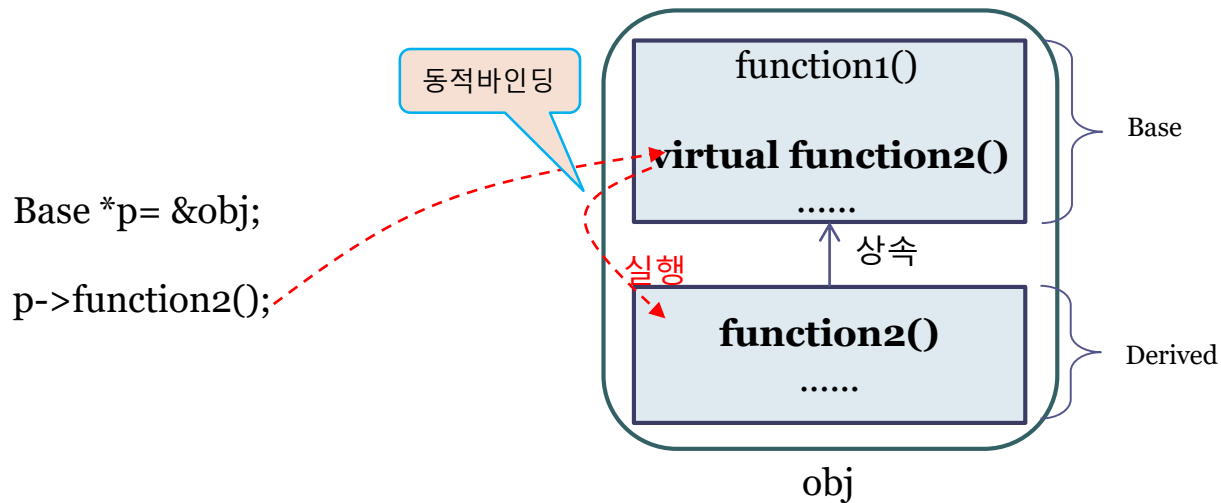
1994.6.1.
Aug. 15 1945

바인딩(binding)과 다형성

- 바인딩
 - 정적 바인딩(static binding)
 - 컴파일 시(compile-time) 호출되는 함수를 결정
 - 동적 바인딩(dynamic binding)
 - 실행 시(run-time) 호출되는 함수를 결정
- 다형성(polymorphism)
 - 같은 모습의 형태가 다른 특성
 - a->fct() 예
 - a라는 포인터(모습)가 가리키는 대상에 따라 호출되는 함수(형태)가 다름
 - 함수 오버로딩, 동적 바인딩 등이 다형성의 예

동적 바인딩

- 동적 바인딩
 - 파생 클래스에 대해
 - 베이스 클래스에 대한 포인터로 가상 함수를 호출하는 경우
 - 객체 내에 오버라이딩한 파생 클래스의 함수를 찾아 실행
 - 실행 중에 이루어짐
 - 실행시간 바인딩, 런타임 바인딩, 늦은 바인딩으로 불림



예제) 상속이 반복되는 경우 가상 함수 호출

```
class Base {  
public:  
    virtual void f() { cout << "Base::f() called" << endl; }  
};  
  
class Derived : public Base {  
public:  
    void f() { cout << "Derived::f() called" << endl; }  
};  
  
class GrandDerived : public Derived {  
public:  
    void f() { cout << "GrandDerived::f() called" << endl; }  
};  
  
int main() {  
    GrandDerived g;  
    Base *bp;  
    Derived *dp;  
    GrandDerived *gp;  
  
    bp = dp = gp = &g;  
  
    bp->f();  
    dp->f();  
    gp->f();  
}
```

Base, Derived, GrandDerived가
상속 관계에 있을 때,
다음 코드를 실행한 결과는
무엇인가?

예제) 상속이 반복되는 경우 가상 함수 호출

```

class Base {
public:
    virtual void f() { cout << "Base::f() called" << endl; }
};

class Derived : public Base {
public:
    void f() { cout << "Derived::f() called" << endl; }
};

class GrandDerived : public Derived {
public:
    void f() { cout << "GrandDerived::f() called" << endl; }
};

int main() {
    GrandDerived g;
    Base *bp;
    Derived *dp;
    GrandDerived *gp;

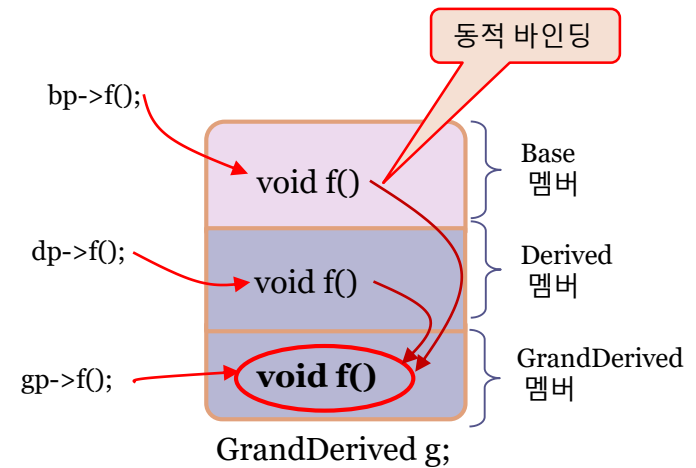
    bp = dp = gp = &g;

    bp->f();
    dp->f();
    gp->f();
}

```

동적 바인딩에 의해 모두
GrandDerived의 함수 f()
호출

Base, Derived, GrandDerived가
상속 관계에 있을 때,
다음 코드를 실행한 결과는
무엇인가?



GrandDerived::f() called
GrandDerived::f() called
GrandDerived::f() called

다형성(Polymorphism)

```
class First
{
public:
    virtual void SimpleFunc() { cout<<"First"<<endl; }
};

class Second: public First
{
public:
    virtual void SimpleFunc() { cout<<"Second"<<endl; }
};

int main(void)
{
    First * ptr=new First();
    ptr->SimpleFunc();    // 아래에 동일한 문장이 존재한다.
    delete ptr;
    ptr=new Second();
    ptr->SimpleFunc();    // 위에 동일한 문장이 존재한다.
    delete ptr;
    return 0;
}
```

지금까지 공부한 가상함수와 관련된 내용을 가리켜 '다형성'이라 한다!

다형성은 동질이상의 의미를 갖는다.

모습은 같은데 형태는 다르다.

문장은 같은데 결과는 다르다!

ptr->Simplefunc 함수의 호출이 다형성의 예!

다형성이란?

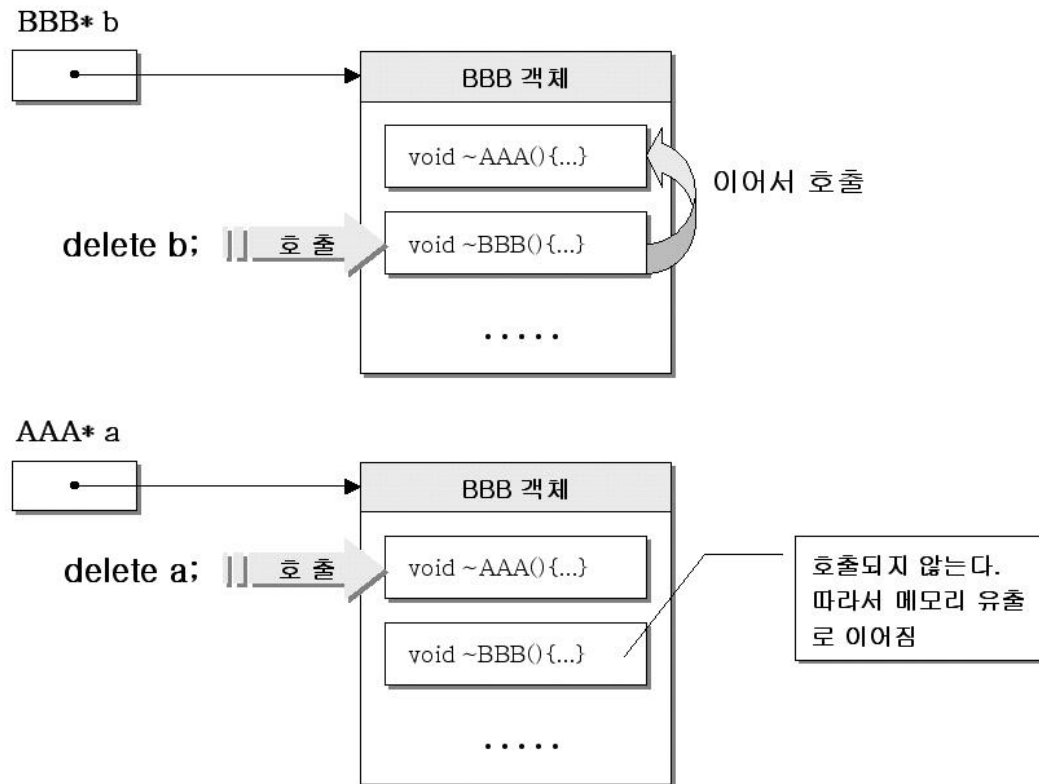
- 다형성(polymorphism)이란 객체들의 타입이 다르면 똑같은 메시지가 전달되더라도 서로 다른 동작을 하는 것
- 다형성은 객체 지향 기법에서 하나의 코드로 다양한 타입의 객체를 처리하는 중요한 기술



07-3. 가상 소멸자와 참조자의 참조 가능성

Virtual 소멸자의 필요성

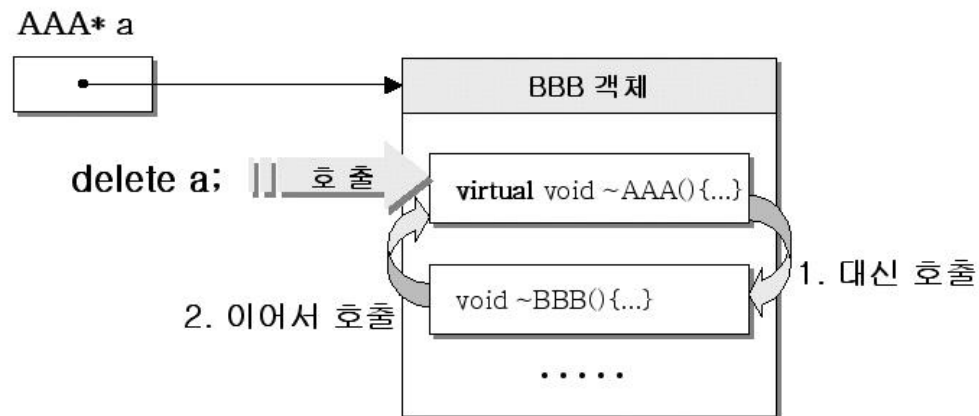
- 상속하고 있는 클래스 객체 소멸 문제점



Virtual 소멸자의 필요성

- virtual 소멸자

```
virtual ~AAA(){
cout<<"~AAA() call!"<<endl;
delete []str1;
}
```



virtual 소멸자

- 파생클래스를 가리키는 베이스클래스의 포인터가 가리키는 객체의 소멸 시에는 파생 클래스의 소멸자를 호출하지 않음
- virtual 소멸자
 - 객체 소멸 시 베이스클래스 뿐만 아니라 파생클래스의 소멸자도 호출
 - 소멸자 앞에 virtual 키워드

virtual 소멸자 필요성 예

```
#include <iostream>
using std::endl;
using std::cout;
class AAA
{
    char* str1;
public:
    AAA(char* _str1){
        str1= new char[strlen(_str1)+1];
        strcpy(str1, _str1);
    }
    ~AAA(){
        cout<<"~AAA() call!"<<endl;
        delete []str1;
    }
    virtual void ShowString(){
        cout<<str1<<' ';
    }
};
class BBB : public AAA
{
    char* str2;
public:
    BBB(char* _str1, char* _str2) : AAA(_str1){
        str2= new char[strlen(_str2)+1];
        strcpy(str2, _str2);
    }
};
```

```
~BBB(){
    cout<<"~BBB() call!"<<endl;
    delete []str2;
}
virtual void ShowString(){
    AAA::ShowString();
    cout<<str2<<endl;
}
};
int main()
{
    AAA * a=new BBB("Good", "evening");
    BBB * b=new BBB("Good", "morning");
    a->ShowString();
    b->ShowString();
    cout<<"-----객체 소멸 직전-----"<<endl;
    delete a; // AAA 소멸자만 호출
    delete b; // BBB,AAA 소멸자 호출
    return 0;
}
```

```
Good evening
Good morning
-----객체 소멸 직전-----
~AAA() call!
~BBB() call!
~AAA() call!
```

virtual 소멸자 예

```
#include <iostream>
using std::endl;
using std::cout;
class AAA
{
    char* str1;
public:
    AAA(char* _str1){
        str1= new char[strlen(_str1)+1];
        strcpy(str1, _str1);
    }
    virtual ~AAA(){
        cout<<"~AAA() call!"<<endl;
        delete []str1;
    }
    virtual void ShowString(){
        cout<<str1<<' ';
    }
};
class BBB : public AAA
{
    char* str2;
public:
    BBB(char* _str1, char* _str2) : AAA(_str1){
        str2= new char[strlen(_str2)+1];
        strcpy(str2, _str2);
    }
};
```

```
~BBB(){
    cout<<"~BBB() call!"<<endl;
    delete []str2;
}
virtual void ShowString(){
    AAA::ShowString();
    cout<<str2<<endl;
}
};
int main()
{
    AAA * a=new BBB("Good", "evening");
    BBB * b=new BBB("Good", "morning");
    a->ShowString();
    b->ShowString();
    cout<<"-----객체 소멸 직전-----"<<endl;
    delete a; // BBB, AAA 소멸자만 호출
    delete b; // BBB,AAA 소멸자 호출
    return 0;
}
```

```
Good evening
Good morning
-----객체 소멸 직전-----
~BBB() call!
~AAA() call!
~BBB() call!
~AAA() call!
```


참고문헌

- 뇌를 자극하는 C++ 프로그래밍, 이현창, 한빛미디어1
- 열혈 C++ 프로그래밍(개정판), 윤성우, 오렌지미디어
- C++ ESPRESSO, 천인국 저, 인피니티북스
- 명품 C++ Programming, 황기태, 생능출판사

추가 자료



오버라이딩된 함수를 호출하는 동적 바인딩

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

class Shape {
public:
    void paint() {
        draw();
    }
    virtual void draw() {
        cout << "Shape::draw() called" << endl;
    }
};

int main() {
    Shape *pShape = new Shape();
    pShape->paint();
    delete pShape;
}
```

Shape::draw() called

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

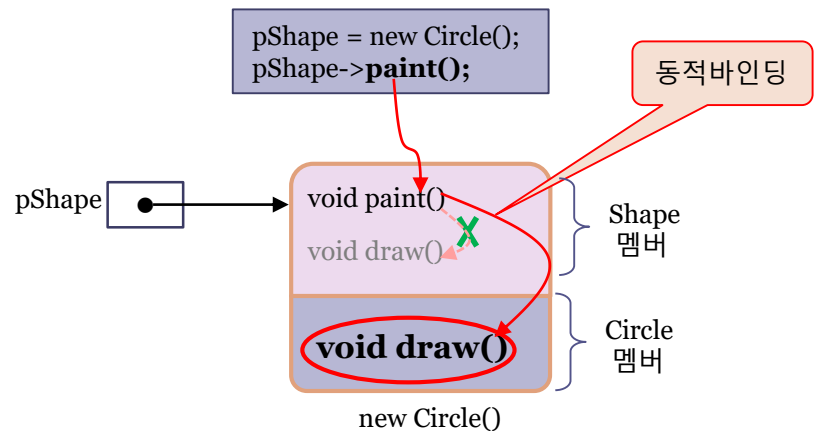
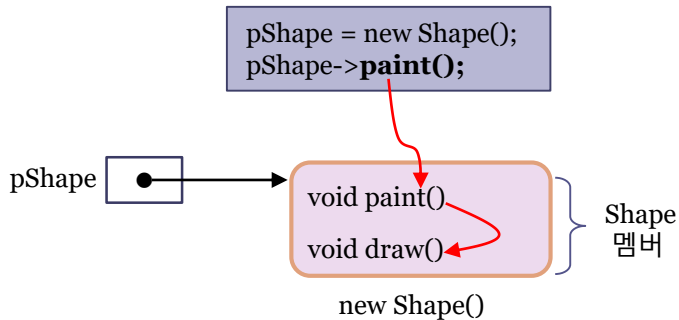
class Shape {
public:
    void paint() {
        draw();
    }
    virtual void draw() {
        cout << "Shape::draw() called" << endl;
    }
};

class Circle : public Shape {
public:
    virtual void draw() {
        cout << "Circle::draw() called" << endl;
    }
};

int main() {
    Shape *pShape = new Circle(); // 업캐스팅
    pShape->paint();
    delete pShape;
}
```

기본 클래스에서 파생 클래스의 함수를 호출하게 되는 사례

Circle::draw() called



C++ 컴파일러는 포인터를 이용한 연산의 가능성 여부를 판단할 때, 포인터의 자료형을 기준으로 판단함

- 실제 가리키는 객체의 자료형을 기준으로 판단하지 **않음**
- 포인터 형에 해당하는 클래스의 멤버에만 접근이 **가능**

```
class First
{
public:
    void FirstFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    void SecondFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    void ThirdFunc() { cout<<"ThirdFunc"<<endl; }
};
```

```
int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    tptr->FirstFunc();    (○)
    tptr->SecondFunc();   (○)
    tptr->ThirdFunc();    (○)

    sptr->FirstFunc();    (○)
    sptr->SecondFunc();   (○)
    sptr->ThirdFunc();    (×)

    fptr->FirstFunc();    (○)
    fptr->SecondFunc();   (×)
    fptr->ThirdFunc();    (×)
    . . . . .
}
```

베이스클래스의 포인터로 객체를 참조하면,

C++ 컴파일러는 포인터 연산의 가능성 여부를 판단할 때, **포인터의 자료형을 기준으로 판단하지, 실제 가리키는 객체의 자료형을 기준으로 판단하지 않는다.**

```
class Base
{
public:
    void BaseFunc() { cout<<"Base Function"<<endl; }
};

class Derived : public Base
{
public:
    void DerivedFunc() { cout<<"Derived Function"<<endl; }
};
```

```
int main(void)
{
    Base * bptr=new Derived(); // 컴파일 OK!
    bptr->DerivedFunc();      // 컴파일 Error!
    . . . .
}

int main(void)
{
    Base * bptr=new Derived(); // 컴파일 OK!
    Derived * dptr=bptr;      // 컴파일 Error!
    . . . .
}

int main(void)
{
    Derived * dptr=new Derived(); // 컴파일 OK!
    Base * bptr=dptr;           // 컴파일 OK!
    . . . .
}
```

함수의 오버라이딩과 포인터 형

```

class First
{
public:
    void MyFunc() { cout<<"FirstFunc"<<endl; }
};

class Second: public First
{
public:
    void MyFunc() { cout<<"SecondFunc"<<endl; }
};

class Third: public Second
{
public:
    void MyFunc() { cout<<"ThirdFunc"<<endl; }
};

int main(void)
{
    Third * tptr=new Third();
    Second * sptr=tptr;
    First * fptr=sptr;

    fptr->MyFunc();
    sptr->MyFunc();
    tptr->MyFunc();
    delete tptr;
    return 0;
}

```

실행결과

```

FirstFunc
SecondFunc
ThirdFunc


```

함수를 호출할 때 사용이 된 포인터의 형에 따라서 호출되는 함수가 결정된다!

포인터의 형에 정의된 함수가 호출된다.

순수 가상함수 추가 예제)

```
class Shape {  
protected:  
    int x, y;  
  
public:  
    ...  
    virtual void draw() = 0;  
};  
  
class Rectangle : public Shape {  
private:  
    int width, height;  
  
public:  
    void draw() {  
        cout << "Rectangle Draw" << endl;  
    }  
};
```



```
int main()
```

```
{
```

```
    Shape *ps = new Rectangle();
```

```
    // OK!
```

```
    ps->draw();
```

```
    // Rectangle의 draw()가 호출된다.
```

```
    delete ps;
```

```
    return 0;
```

```
}
```



Rectangle Draw

추상 클래스의 상속과 구현

- 추상 클래스의 상속
 - 추상 클래스를 단순 상속하면 자동 추상 클래스
- 추상 클래스의 구현
 - 추상 클래스를 상속받아 순수 가상 함수를 오버라이딩
 - 파생 클래스는 추상 클래스가 아님

Shape은
추상 클래스

```
class Shape {
public:
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    string toString() { return "Circle 객체"; }
};

Shape shape; // 객체 생성 오류
Circle waffle; // 객체 생성 오류
```

추상 클래스의 단순 상속

Circle도
추상 클래스



```
class Shape {
public:
    virtual void draw() = 0;
};

class Circle : public Shape {
public:
    virtual void draw() {
        cout << "Circle";
    }
    string toString() { return "Circle 객체"; }
};

Shape shape; // 객체 생성 오류
Circle waffle; // 정상적인 객체 생성
```

Shape은
추상 클래스

Circle은
추상 클래스 아님

순수 가상 함수
오버라이딩

추상 클래스의 구현

추상 클래스 구현 - 추가 예제)

다음 추상 클래스 `Calculator`를 상속받아 `GoodCalc` 클래스를 구현하라.

```
class Calculator {
public:
    virtual int add(int a, int b) = 0; // 두 정수의 합 리턴
    virtual int subtract(int a, int b) = 0; // 두 정수의 차 리턴
    virtual double average(int a [], int size) = 0; // 배열 a의 평균 리턴. size는 배열의 크기
};
```

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

// 이 곳에 Calculator 클래스 코드 필요

class GoodCalc : public Calculator {
public:
    int add(int a, int b) { return a + b; }
    int subtract(int a, int b) { return a - b; }
    double average(int a [], int size) {
        double sum = 0;
        for(int i=0; i<size; i++)
            sum += a[i];
        return sum/size;
    }
};
```

순수 가상 함수 구현

```
int main() {
    int a[] = {1,2,3,4,5};
    Calculator *p = new GoodCalc();
    cout << p->add(2, 3) << endl;
    cout << p->subtract(2, 3) << endl;
    cout << p->average(a, 5) << endl;
    delete p;
}
```

```
5
-1
3
```

추상 클래스를 상속받는 파생 클래스 구현 - 추가 예제)

다음 코드와 실행 결과를 참고하여 추상 클래스 Calculator를 상속받는 Adder와 Subtractor 클래스를 구현하라.

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

class Calculator {
    void input() {
        cout << "정수 2 개를 입력하세요>> ";
        cin >> a >> b;
    }
protected:
    int a, b;
    virtual int calc(int a, int b) = 0; // 두 정수의 합 리턴
public:
    void run() {
        input();
        cout << "계산된 값은 " << calc(a, b) << endl;
    }
};

int main() {
    Adder adder;
    Subtractor subtractor;
    adder.run();
    subtractor.run();
}
```

adder.run()에 의한 실행 결과

subtractor.run()에 의한 실행 결과

```
정수 2 개를 입력하세요>> 5 3
계산된 값은 8
정수 2 개를 입력하세요>> 5 3
계산된 값은 2
```

정답)

```
class Adder : public Calculator {
protected:
    int calc(int a, int b) { // 순수 가상 함수 구현
        return a + b;
    }
};

class Subtractor : public Calculator {
protected:
    int calc(int a, int b) { // 순수 가상 함수 구현
        return a - b;
    }
};
```

다형성 - 추가 예제)

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

class Animal
{
public:
    Animal() { cout <<"Animal 생성자" << endl; }
    ~Animal() { cout <<"Animal 소멸자" << endl; }
    virtual void speak() { cout <<"Animal speak()" << endl; }
};

class Dog : public Animal
{
public:
    Dog() { cout <<"Dog 생성자" << endl; }
    ~Dog() { cout <<"Dog 소멸자" << endl; }
    void speak() { cout <<"멍멍" << endl; }
};
```

```
class Cat : public Animal
{
public:
    Cat() { cout <<"Cat 생성자" << endl; }
    ~Cat() { cout <<"Cat 소멸자" << endl; }
    void speak() { cout <<"야옹" << endl; }
};

int main()
{
    Animal *a1 = new Dog();
    a1->speak();

    Animal *a2 = new Cat();
    a2->speak();
    return 0;
}
```



Animal 생성자

Dog 생성자

멍멍

Animal 소멸자

Animal 생성자

Cat 생성자

야옹

Animal 소멸자

가상 소멸자(Virtual Destructor)

```

class First
{
    . . . . .
public:
    virtual ~First() { . . . . . }
};

class Second: public First
{
    . . . . .
public:
    virtual ~Second() { . . . . . }
};

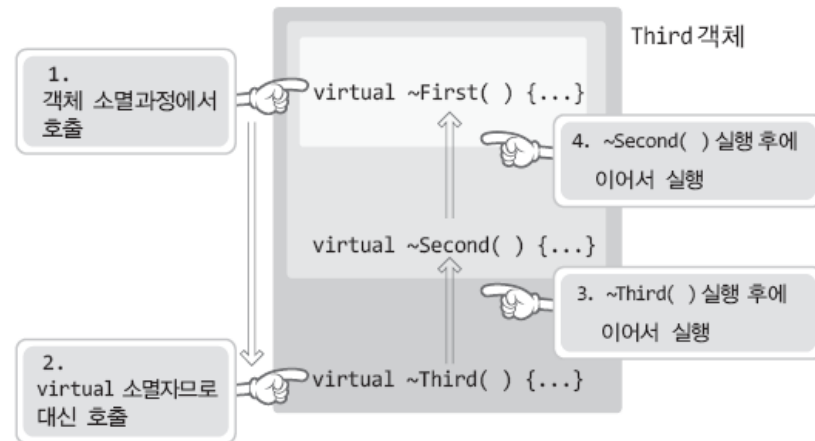
class Third: public Second
{
    . . . . .
public:
    virtual ~Third() { . . . . . }
};

```

```

int main(void)
{
    First * ptr=new Third();
    delete ptr;
    . . . . .
}

```



▶ [그림 08-3: 가상 소멸자의 호출과정]

소멸자를 가상으로 선언함으로써 각각의 생성자 내에서 할당한 메모리 공간을 효율적으로 해제할 수 있다.

참조자의 참조 가능성

“C++에서, AAA형 포인터 변수는 AAA 객체 또는 AAA를 직접 혹은 간접적으로 상속하는 모든 객체를 가리킬 수 있다(객체의 주소 값을 저장할 수 있다).”



“C++에서, AAA형 참조자는 AAA 객체 또는 AAA를 직접 혹은 간접적으로 상속하는 모든 객체를 참조할 수 있다.”

```
class First
{
public:
    void FirstFunc() { cout<<"FirstFunc()"<<endl; }
    virtual void SimpleFunc() { cout<<"First's SimpleFunc()"<<endl; }
};

class Second: public First
{
public:
    void SecondFunc() { cout<<"SecondFunc()"<<endl; }
    virtual void SimpleFunc() { cout<<"Second's SimpleFunc()"<<endl; }
};

class Third: public Second
{
public:
    void ThirdFunc() { cout<<"ThirdFunc()"<<endl; }
    virtual void SimpleFunc() { cout<<"Third's SimpleFunc()"<<endl; }
};
```

```
int main(void)
{
    Third obj;
    obj.FirstFunc();
    obj.SecondFunc();
    obj.ThirdFunc();
    obj.SimpleFunc();

    Second & sref=obj;
    sref.FirstFunc();
    sref.SecondFunc();
    sref.SimpleFunc();

    First & fref=obj;
    fref.FirstFunc();
    fref.SimpleFunc();

    return 0;
}
```

실행결과

```
FirstFunc()
SecondFunc()
ThirdFunc()
Third's SimpleFunc()
FirstFunc()
SecondFunc()
Third's SimpleFunc()
FirstFunc()
Third's SimpleFunc()
```

가상 소멸자 - 추가 예제)

- 소멸자 문제

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

class String {
    char *s;
public:
    String(char *p){
        cout << "String() 생성자" << endl;
        s = new char[strlen(p)+1];
        strcpy(s, p);
    }
    ~String(){
        cout << "String() 소멸자" << endl;
        delete[] s;
    }
    virtual void display()
    {
        cout << s;
    }
};
```

```

class MyString : public String {
    char *header;
public:
    MyString(char *h, char *p) : String(p){
        cout << "MyString() 생성자" << endl;
        header = new char[strlen(h)+1];
        strcpy(header, h);
    }
    ~MyString(){
        cout << "MyString() 소멸자" << endl;
        delete[] header;
    }
    void display()
    {
        cout << header;        // 헤더출력
        String::display();
        cout << header << endl;    // 헤더출력
    }
};

```

```
int main()
{
    String *p = new MyString("----", "Hello World!");           // OK!
    p->display();
    delete p;

    return 0;
}
```

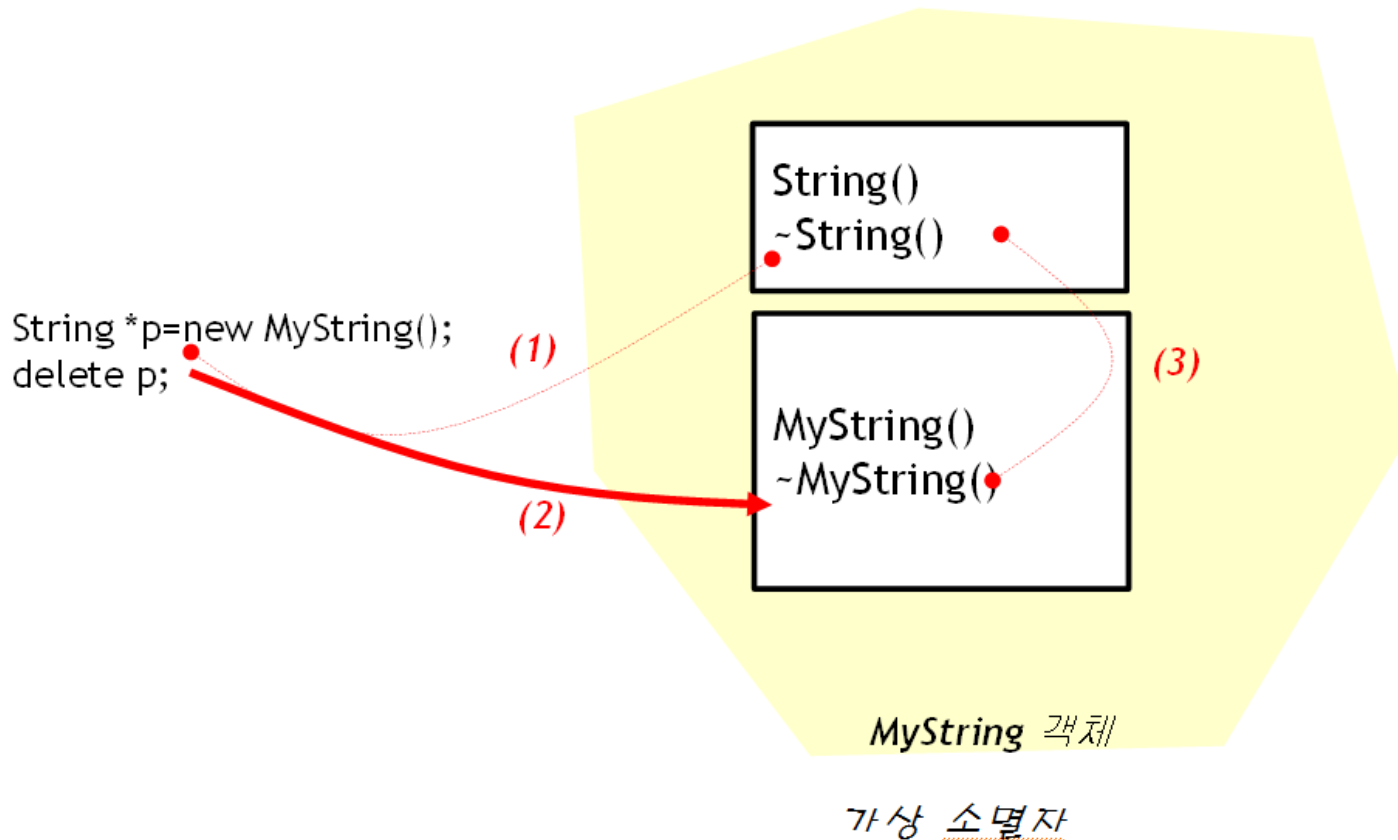


String() 생성자
MyString() 생성자
----Hello World!----
String() 소멸자

MyString의
소멸자가 호출되지 않음

→ 가상 소멸자 이용 해결

- 그렇다면 어떻게 하여야 MyString 소멸자도 호출되게 할 수 있는가?
- String 클래스의 소멸자를 virtual로 선언하면 된다.



```

class String {
    char *s;
public:
    String(char *p){
        ... // 앞과 동일

    }
    virtual ~String(){
        cout << "String() 소멸자" << endl;
        delete[] s;
    }
};

class MyString : public String {
    ...// 앞과 동일
};

int main()
{
    ...// 앞과 동일
}

```



```

String() 생성자
MyString() 생성자
----Hello World!----
MyString() 소멸자
String() 소멸자

```