



객체지향언어

제9장 가상함수와 추상 클래스

2025. 4. 15

컴퓨터공학

허훈식

Learning Objectives



1. 가상 함수의 개념과 함수 중복의 차이점을 이해한다.
2. 가상 함수와 오버라이딩, 동적바인딩의 개념을 이해한다.
3. 가상 소멸자의 중요성을 이해한다.
4. 가상 함수를 활용하여 프로그램을 작성할 수 있다.
5. 순수가상함수와 추상 클래스를 이해하고 작성할 수 있다.

9.1 상속 관계에서의 함수 중복

➤ 상속 관계에서 함수 재정의하는 경우

- 파생 클래스에서 기본 클래스와 동일한 형식의 함수를 작성 경우

- 기본 클래스에 대한 포인터는 기본 클래스의 멤버 함수를 호출하고
- 파생 클래스에 대한 포인터는 파생 클래스에서 작성된 멤버 함수를 호출한다.

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

```
// 파생 클래스에 대한 포인터로 호출  
Derived d, *pDer;  
pDer = &d;  
pDer->f(); // Derived::f() 호출  
  
// 기본 클래스에 대한 포인터로 호출  
Base* pBase;  
pBase = pDer; // 업캐스팅  
pBase->f(); // Base::f() 호출
```

이러한 호출 관계는 컴파일시에 결정된다.(정적 바인딩)

제9장. 가상 함수와 추상 클래스

9.1 상속 관계에서의 함수 중복

➤ [예제 9-1] 상속 관계에서 함수를 재정의하는 경우

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

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

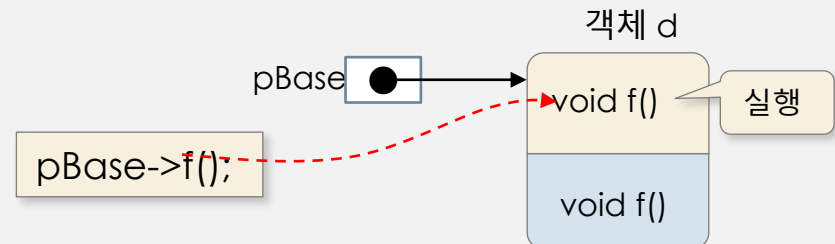
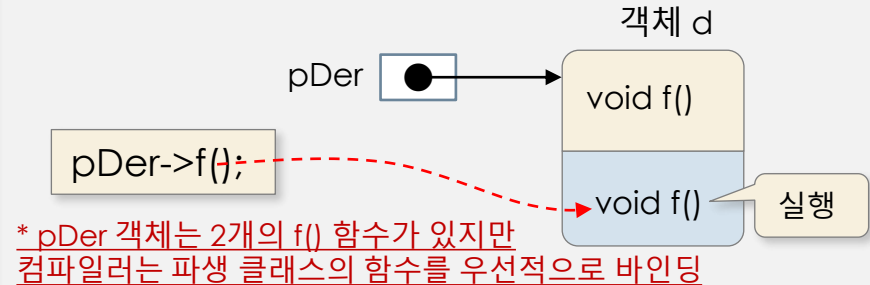
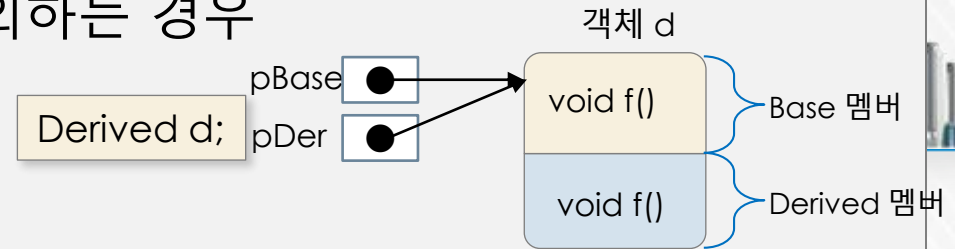
class Derived : public Base {
public:
    void f() { cout << "Derived::f() called" << endl; }
};

void main() {
    Derived d, *pDer;
    pDer = &d;
    pDer->f(); // Derived::f() 호출

    Base* pBase;
    pBase = pDer; // 업캐스팅
    pBase->f(); // Base::f() 호출
}
```

함수 중복

Derived::f() called
Base::f() called



9.2 가상 함수와 오버라이딩

- 가상 함수(virtual function)
 - virtual 키워드로 선언된 멤버 함수
- virtual 키워드의 의미
 - 동적 바인딩 지시어
 - 컴파일러에게 함수에 대한 호출 바인딩을 실행 시간까지 미루도록 지시

```
class Base {  
public:  
    virtual void f();    // f()는 가상 함수  
};
```

9.2 가상 함수와 오버라이딩

➤ 오버라이딩(overriding)

- 파생 클래스에서 기본 클래스의 가상 함수와 동일한 이름의 함수 선언
- 기본 클래스의 가상 함수의 존재감 상실시킴
- 파생 클래스에서 오버라이딩한 함수가 호출되도록 동적 바인딩
- 다형성의 한 종류

➤ 가상 함수를 재정의하는 오버라이딩 vs 그외 함수를 재정의

- 동적 바인딩 vs 정적 바인딩과 같다.
- 자바에서는 모든 함수의 재정의는 동적 바인딩이 일어난다.

9.2 가상 함수와 오버라이딩

➤ 오버라이딩 개념



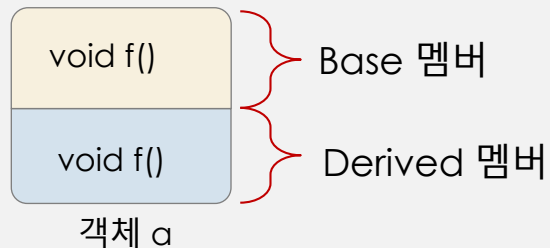
9.2 가상 함수와 오버라이딩

➤ 함수 재정의(redefine)와 오버라이딩 사례 비교

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

함수 재정의

Derived a;

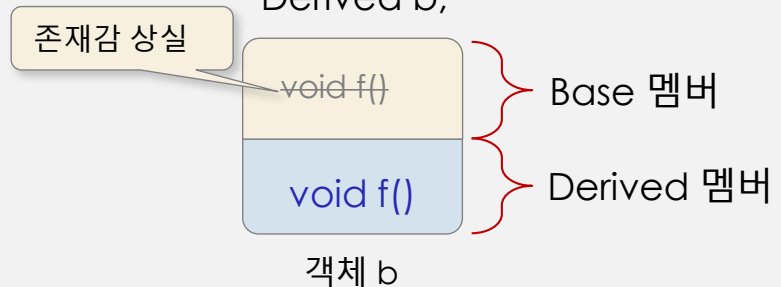


(a) a 객체에는 동등한 호출 기회를 가진 함수 f()가 두 개 존재

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

오버라이딩

Derived b;



(b) b 객체에는 두 개의 함수 f()가 존재하지만, Base의 f()는 존재감을 잃고, 항상 Derived의 f()가 호출됨

9.2 가상 함수와 오버라이딩

➤ 함수 재정의와 오버라이딩 용어의 정리

가상 함수를 재정의하는 오버라이딩의 경우 함수가 호출되는 실행 시간에 동적 바인딩이 일어나지만, 그렇지 않은 함수 재정의의 경우는 컴파일 시간에 결정된 함수가 호출되는 정적 바인딩이 일어난다.

Java의 경우 멤버 함수가 가상이나 아니냐로 구분되지 않으며, 함수 재정의는 곧 오버라이딩이며, 무조건 동적 바인딩이 일어난다.

9.2 가상 함수와 오버라이딩

➤ [예제 9-2] 오버라이딩과 가상 함수 호출

```

#include <iostream>
using namespace std;

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

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

int main() {
    Derived d, *pDer;
    pDer = &d;
    pDer->f(); // Derived::f() 호출

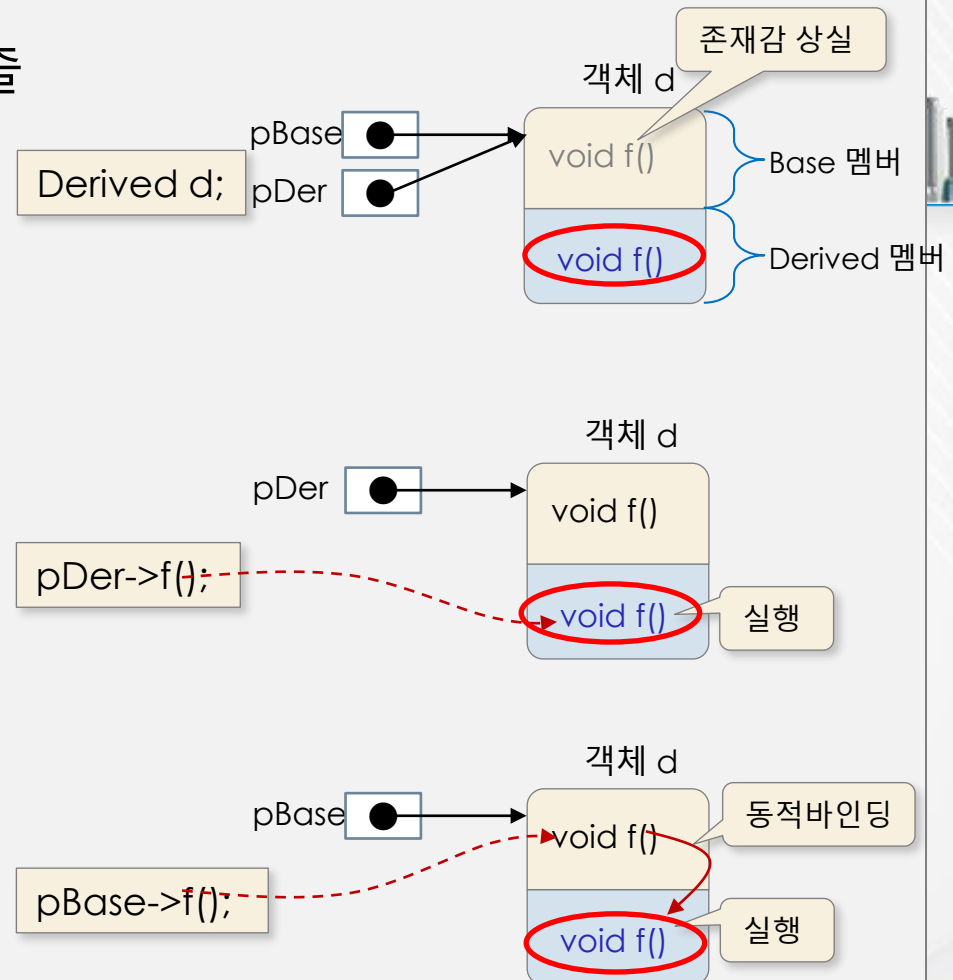
    Base *pBase;
    pBase = pDer; // 업 캐스팅
    pBase->f(); // 동적바인딩 발생 Derived::f() 실행
}

```

```

Derived::f() called
Derived::f() called

```



9.2 가상 함수와 오버라이딩

- 오버라이딩의 목적 : 파생 클래스들이 자신의 목적에 맞게 가상 함수를 재정의
- 가상함수는 파생 클래스에서 구현할 함수 인터페이스 제공(파생 클래스의 다형성)

다형성의 실현

- draw() 가상 함수를 가진 기본 클래스 Shape
- 오버라이딩을 통해 Circle, Rect, Line 클래스에서 자신만의 draw() 구현

```
class Shape {  
protected:  
    virtual void draw() {}  
};
```

가상 함수 선언.
파생 클래스에서 재정의할
함수에 대한 인터페이스 역할

```
class Circle : public Shape  
{  
protected:  
    virtual void draw() {  
        // Circle을 그린다.  
    }  
};
```

오버라이딩,
다형성 실현

```
class Rect : public Shape {  
protected:  
    virtual void draw() {  
        // Rect을 그린다.  
    }  
};
```

```
class Line : public Shape {  
protected:  
    virtual void draw() {  
        // Line을 그린다.  
    }  
};
```

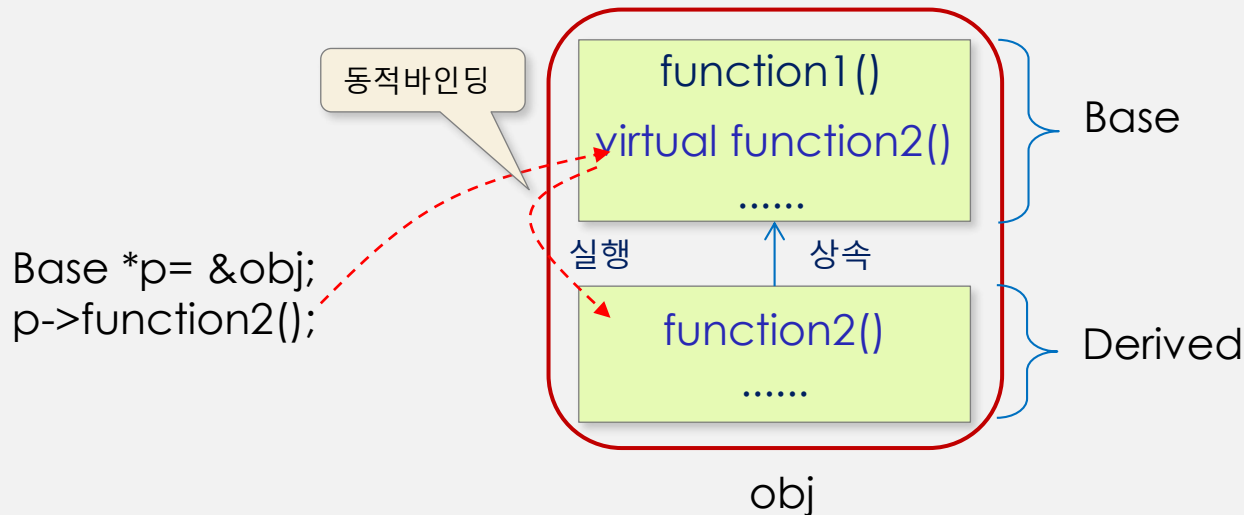
```
void paint(Shape* p) {  
    p->draw();  
}  
paint(new Circle()); // Circle을 그린다.  
paint(new Rect()); // Rect을 그린다.  
paint(new Line()); // Line을 그린다.
```

p가 가리키는 객체에
오버라이딩된 draw()
호출

9.2 가상 함수와 오버라이딩

➤ 동적 바인딩

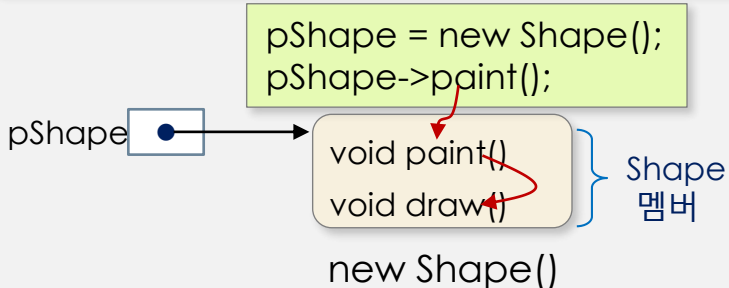
- 파생 클래스의 객체에 대해 기본 클래스에 대한 포인터로 가상 함수를 호출하는 경우 동적 바인딩이 일어남
 - 기본 클래스의 객체에 대해 가상 함수가 호출된다 하더라도 동적 바인딩은 일어나지 않음
- 객체 내에 오버라이딩한 파생 클래스의 함수를 찾아 실행
 - 실행 중에 이루어져 실행시간 바인딩, 런타임 바인딩, 늦은 바인딩으로 불림



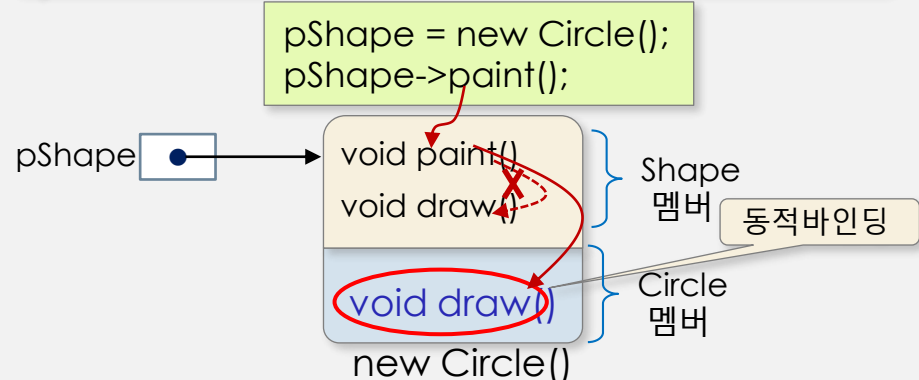
9.2 가상 함수와 오버라이딩

➤ 오버라이딩된 함수호출(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;
}
```



```
#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;
}
```



9.2 가상 함수와 오버라이딩

➤ 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. 리턴타입다름
    virtual void success(); //오버라이딩 성공
    virtual void g(int, double); //오버로딩 사례
};
```

```
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

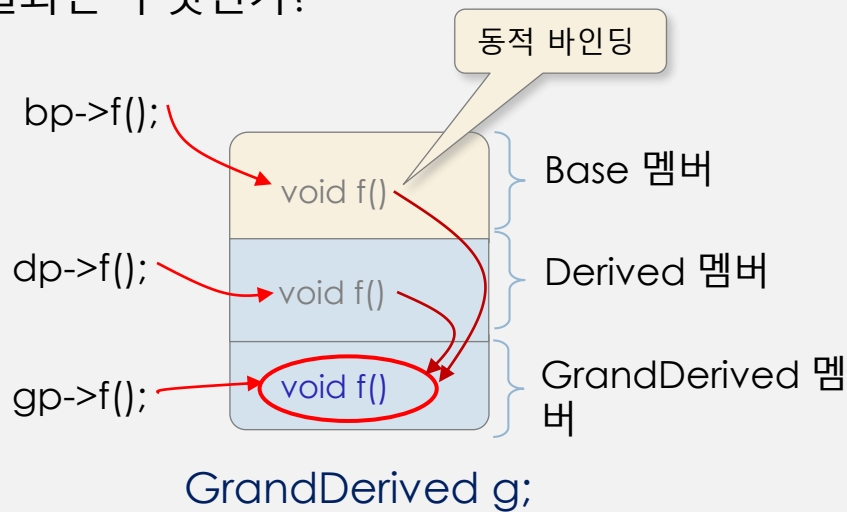
- 가상 함수의 접근 지정

- private, protected, public 중 자유롭게 지정 가능

9.2 가상 함수와 오버라이딩

➤ [예제 9-3] 상속이 반복되는 경우 가상 함수 호출

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



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

```
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();
}
```

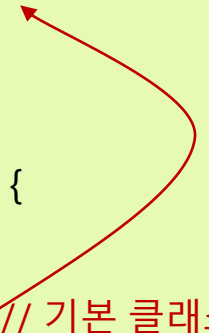
동적 바인딩에 의해 모두 GrandDerived의 함수 f() 호출

9.2 가상 함수와 오버라이딩

➤ 범위 지정 연산자(::)

- 정적 바인딩 지시
- 기본클래스::가상함수() 형태로 기본 클래스의 가상 함수를 정적 바인딩으로 호출
 - Shape::draw();

```
class Shape {
public:
    virtual void draw() {
        ...
    }
};
class Circle : public Shape {
public:
    virtual void draw() {
        Shape::draw(); // 기본 클래스의 draw()를 실행한다.
        ....           // 기능을 추가한다.
    }
};
```



9.2 가상 함수와 오버라이딩

➤ [예제 9-4] 범위 지정 연산자(::)를 이용한 기본 클래스의 가상 함수 호출

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

class Shape {
public:
    virtual void draw() {
        cout << "--Shape--";
    }
};

class Circle : public Shape {
public:
    virtual void draw() {
        Shape::draw(); // 기본 클래스의 draw() 호출
        cout << "Circle" << endl;
    }
};

int main() {
    Circle circle;
    Shape * pShape = &circle;

    pShape->draw();
    pShape->Shape::draw();
}
```

정적 바인딩

동적바인딩

정적 바인딩

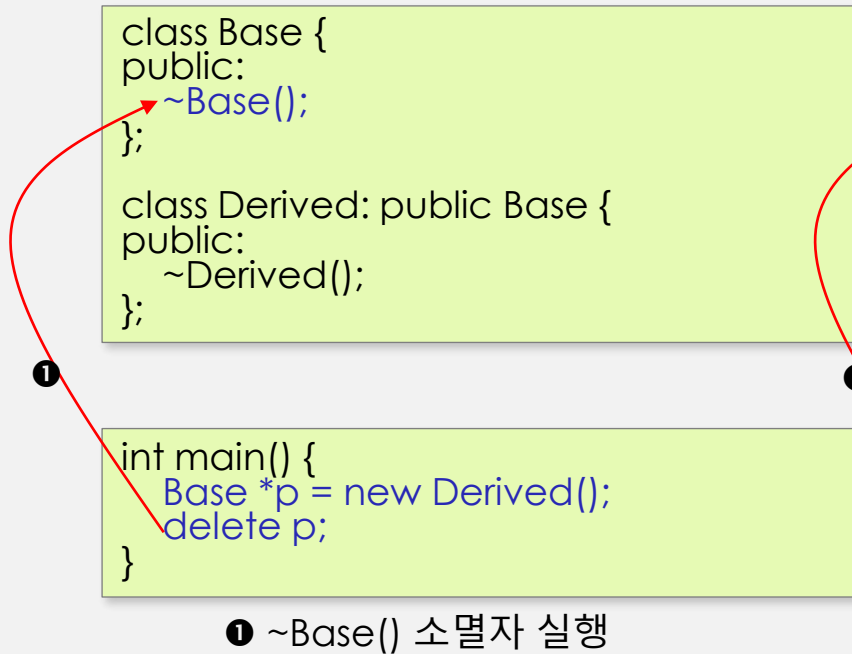
동적 바인딩을
포함하는 호출

--Shape--Circle
--Shape--

9.2 가상 함수와 오버라이딩

➤ 가상 소멸자

- 소멸자를 **virtual** 키워드로 선언
- 소멸자 호출 시 동적 바인딩 발생



소멸자가 가상함수가 아닌 경우



가상 소멸자 경우

9.2 가상 함수와 오버라이딩

➤ [예제 9-5] 소멸자를 가상 함수로 선언

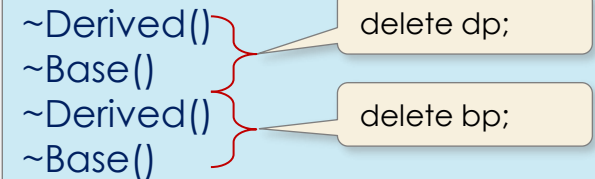
```
#include <iostream>
using namespace std;

class Base {
public:
    virtual ~Base() { cout << "~Base()" << endl; }
};

class Derived: public Base {
public:
    virtual ~Derived() { cout << "~Derived()" << endl; }
};

int main() {
    Derived *dp = new Derived();
    Base *bp = new Derived();

    delete dp;    // Derived의 포인터로 소멸
    delete bp;    // Base의 포인터로 소멸
}
```



9.2 가상 함수와 오버라이딩

➤ 오버로딩과 함수 재정의, 오버라이딩 비교

비교 요소	오버로딩	함수 재정의(가상 함수가 아닌 멤버에 대해)	오버라이딩
정의	<u>매개 변수 타입이나 개수가 다르지만, 이름이 같은 함수들이 중복 작성되는 것</u>	기본 클래스의 멤버 함수를 파생 클래스에서 이름, 매개 변수 타입과 개수, 리턴 타입까지 <u>완벽히 같은 원형으로 재작성</u> 하는 것	기본 클래스의 <u>가상 함수</u> 를 파생 클래스에서 이름, 매개 변수 타입과 개수, 리턴 타입까지 <u>완벽히 같은 원형으로 재작성</u> 하는 것
존재	클래스의 멤버들 사이, 외부 함수들 사이, 그리고 기본 클래스와 파생 클래스 사이에 존재 가능	상속 관계	상속 관계
목적	이름이 같은 여러 개의 함수를 중복 작성하여 사용의 편의성 향상	기본 클래스의 멤버 함수와 별도로 파생 클래스에서 필요하여 재작성	기본 클래스에 구현된 가상 함수를 무시하고, 파생 클래스에서 새로운 기능으로 재작성하고자 함
바인딩	정적 바인딩. 컴파일 시에 중복된 함수들의 호출 구분	정적 바인딩. 컴파일 시에 함수의 호출 구분	동적 바인딩. 실행 시간에 오버라이딩된 함수를 찾아 실행
객체 지향 특성	컴파일 시간 다형성	컴파일 시간 다형성	실행 시간 다형성

9.3 가상 함수와 오버라이딩의 활용 사례

➤ 가상 함수를 가진 기본 클래스의 목적

Shape은 상속을 위한 기본 클래스로의 역할

- 가상 함수 draw()로 파생 클래스의 인터페이스를 보여줌
- Shape 객체를 생성할 목적 아님
- 파생 클래스에서 draw() 재정의.
자신의 도형을 그리도록 유도

Shape.h

```
class Shape {
    Shape* next;
protected:
    virtual void draw();
public:
    Shape() { next = NULL; }
    virtual ~Shape() {}
    void paint();
    Shape* add(Shape* p);
    Shape* getNext() { return next; }
};
```

Shape.cpp

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

void Shape::paint() {
    draw();
}

void Shape::draw() {
    cout << "--Shape--" << endl;
}

Shape* Shape::add(Shape *p) {
    this->next = p;
    return p;
}
```

Circle.h

```
class Circle : public Shape {
protected:
    virtual void draw();
};
```

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

void Circle::draw() {
    cout << "Circle" << endl;
}
```

Circle.cpp

Rect.h

```
class Rect : public Shape {
protected:
    virtual void draw();
};
```

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

void Rect::draw() {
    cout << "Rectangle" << endl;
}
```

Rect.cpp

Line.h

```
class Line : public Shape {
protected:
    virtual void draw();
};
```

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

void Line::draw() {
    cout << "Line" << endl;
}
```

Line.cpp

9.3 가상 함수와 오버라이딩의 활용 사례

- 가상 함수 오버라이딩 : 파생 클래스마다 다르게 구현하는 다형성

```
void Circle::draw() { cout << "Circle" << endl; }  
void Rect::draw() { cout << "Rectangle" << endl; }  
void Line::draw() { cout << "Line" << endl; }
```

- 파생 클래스에서 가상 함수 draw()의 재정의
 - 어떤 경우에도 동적 바인딩에 의해 자신이 만든 draw()가 호출됨을 보장 받음

9.3 가상 함수와 오버라이딩의 활용 사례

➤ 동적 바인딩 실행 : 파생 클래스의 가상 함수실행

```
#include <iostream>
#include "Shape.h"
#include "Circle.h"
#include "Rect.h"
#include "Line.h"
using namespace std;

int main() {
    Shape *pStart=NULL;
    Shape *pLast;
    pStart = new Circle(); //처음에 원 도형을 생성
    pLast = pStart;

    pLast = pLast->add(new Rect()); //사각형 생성
    pLast = pLast->add(new Circle()); // 원 생성
    pLast = pLast->add(new Line()); // 선 생성
    pLast = pLast->add(new Rect()); // 사각형 생성

    // 현재 연결된 모든 도형을 화면에 그린다.
    Shape* p = pStart;
    while(p != NULL) {
        p->paint();
        p = p->getNext();
    }
```

```
// 현재 연결된 모든 도형을 삭제한다.
p = pStart;
while(p != NULL) {
    // 다음 도형 주소 기억
    Shape* q = p->getNext();

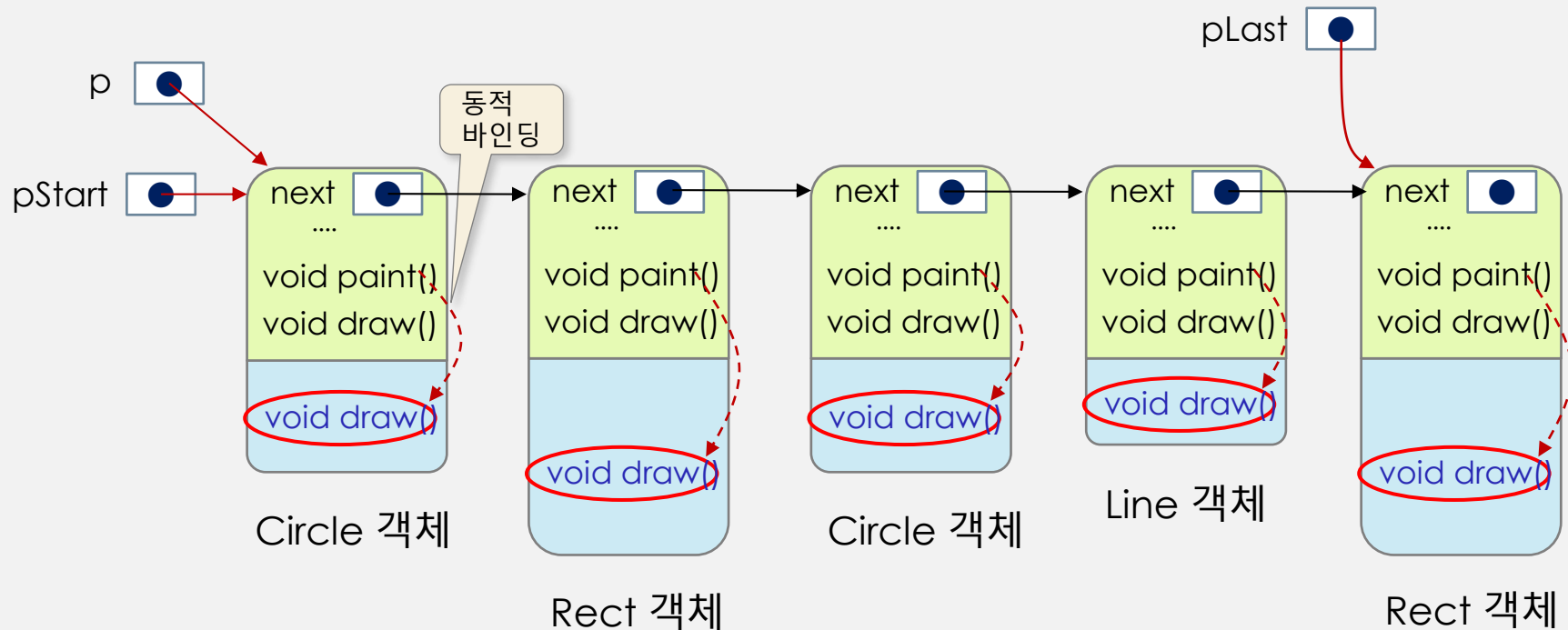
    // 기본 클래스의 가상 소멸자 호출
    delete p;

    p = q; // 다음 도형 주소를 p에 저장
}
}
```

```
Circle
Rectangle
Circle
Line
Rectangle
```

9.3 가상 함수와 오버라이딩의 활용 사례

➤ main() 함수가 실행될 때 구성된 객체의 연결



9.3 가상 함수와 오버라이딩의 활용 사례

- 기본 클래스의 포인터 활용
 - 기본 클래스의 포인터로 파생 클래스 접근
 - pStart, pLast, p의 타입이 Shape*
 - 링크드 리스트를 따라 Shape을 상속받은 파생 객체들 접근
 - p->paint()의 간단한 호출로 파생 객체에 오버라이딩된 draw() 함수 호출

9.4 추상 클래스

- 기본 클래스의 가상 함수 목적
 - 파생 클래스에서 재정의할 함수를 알려주는 역할
 - 실행할 코드를 작성할 목적이 아님
 - 기본 클래스의 가상 함수를 굳이 구현할 필요가 없다.
- 순수 가상 함수(pure virtual function)
 - 함수의 코드가 없고 선언만 있는 가상 멤버 함수
 - 멤버 함수의 원형=0; 으로 선언

```
class Shape {  
public:  
    virtual void draw()=0;    // 순수 가상 함수 선언  
};
```


9.4 추상 클래스

➤ 추상 클래스

- 최소한 하나의 순수 가상 함수를 가진 클래스

```
class Shape {           // Shape은 추상 클래스
    Shape *next;
public:
    void paint() {
        draw();
    }
    virtual void draw() = 0; // 순수 가상 함수
};
void Shape::paint() {
    draw();              // 순수 가상 함수라도 호출은 할 수 있다.
}
```

9.4 추상 클래스

➤ 추상 클래스의 특징

- 온전한 클래스가 아니므로 객체 생성 불가능

```
Shape shape;           // 컴파일 오류  
Shape *p = new Shape(); // 컴파일 오류
```

→ error C2259: 'Shape' : 추상 클래스를 인스턴스화할 수 없습니다.

- 추상 클래스의 포인터는 선언 가능

```
Shape *p;
```

➤ 추상 클래스의 목적

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

9.4 추상 클래스

➤ 추상 클래스의 상속

- 추상 클래스를 단순 상속하면 자동 추상 클래스

➤ 추상 클래스의 구현

- 추상 클래스를 상속받아 순수 가상 함수를 오버라이딩 - 추상 클래스가 아님

추상 클래스의 단순 상속

```
class Shape {  
public:  
    virtual void draw() = 0;  
};  
  
class Circle : public Shape {  
public:  
    string toString() { return "Circle 객체"; }  
};
```

Shape은
추상 클래스

Circle도
추상 클래스

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

추상 클래스의 구현

```
class Shape {  
public:  
    virtual void draw() = 0;  
};  
  
class Circle : public Shape {  
public:  
    virtual void draw() {  
        cout << "Circle";  
    }  
    string toString() { return "Circle 객체"; }  
};
```

Shape은
추상 클래스

Circle은
추상 클래스 아님

순수 가상 함수
오버라이딩

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

9.4 추상 클래스

➤ Shape을 추상 클래스로 수정

Shape은 추상 클래스

Shape.h

```
class Shape {
    Shape* next;
protected:
    virtual void draw() = 0;
public:
    Shape() { next = NULL; }
    virtual ~Shape() {}
    void paint();
    Shape* add(Shape* p);
    Shape* getNext() { return next; }
};
```

Shape.cpp

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

void Shape::paint() {
    draw();
}

void Shape::draw() {
    cout << "--Shape--" << endl;
}

Shape* Shape::add(Shape *p) {
    this->next = p;
    return p;
}
```

Circle.h

```
class Circle : public Shape {
protected:
    virtual void draw();
};
```

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

```
#include "Shape.h"
#include "Circle.h"
```

```
void Circle::draw() {
    cout << "Circle" << endl;
}
```

Circle.cpp

Rect.h

```
class Rect : public Shape {
protected:
    virtual void draw();
};
```

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

```
#include "Shape.h"
#include "Rect.h"
```

```
void Rect::draw() {
    cout << "Rectangle" << endl;
}
```

Rect.cpp

Line.h

```
class Line : public Shape {
protected:
    virtual void draw();
};
```

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

```
#include "Shape.h"
#include "Line.h"
```

```
void Line::draw() {
    cout << "Line" << endl;
}
```

Line.cpp

9.4 추상 클래스

➤ [예제 9-6] 추상 클래스 구현 연습

- 다음 추상 클래스 Calculator를 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;  
    }  
};
```

순수가상함수
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

9.4 추상 클래스

➤ [예제 9-7] 추상 클래스를 상속받는 파생 클래스 구현 연습

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

adder.run()
에 의한 실행 결과

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

subtractor.run()
에 의한 실행 결과

```
#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
public:
    void run() {
        input();
        cout << "계산된 값은 " << calc(a, b) << endl;
    }
};

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


9.4 추상 클래스

➤ [예제 9-7] Adder와 Subractor 클래스

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