



09

가상 함수와 추상 클래스

학습 목표

1. 상속에서 함수 재정의를 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.

예제 9-1 파생클래스에서 함수를재정의하는사례

3

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

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

함수 재정의

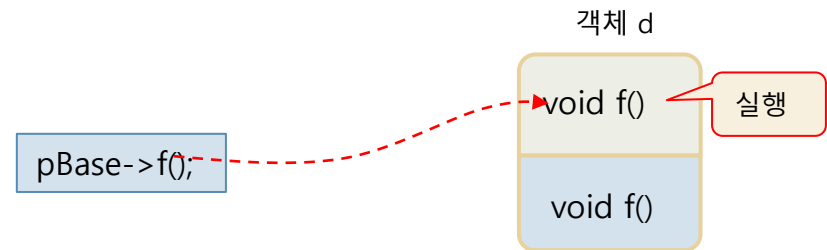
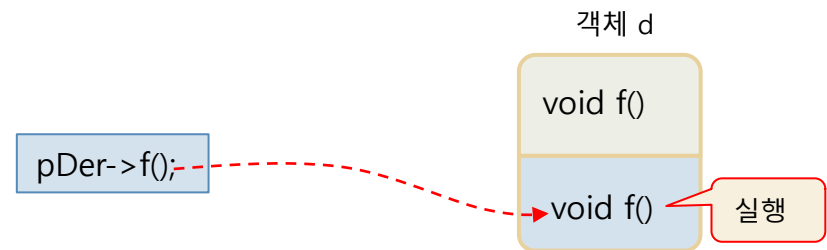
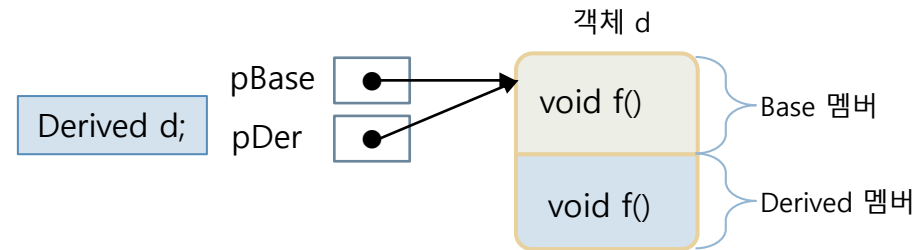
```
class Derived : public Base {
public:
    void f() { cout << "Derived::f() called" << endl; }
};
```

함수 중복

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

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

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



가상 함수와 오버라이딩

4

□ 가상 함수(virtual function)

- ▣ virtual 키워드로 선언된 멤버 함수
- ▣ virtual 키워드의 의미

- 동적 바인딩 지시어
- 컴파일러에게 함수에 대한 호출 바인딩을 실행 시간까지 미루도록 지시

Static } 바인딩
dynamic

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

□ 함수 오버라이딩(function overriding)

- ▣ 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

오버라이딩 개념

5



함수 재정의와 오버라이딩 사례 비교

6

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

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

함수 재정의

함수 재정의(redefine)

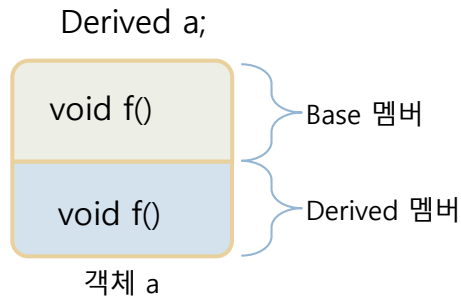
```
class Base {
public:
    virtual void f() {
        cout << "Base::f() called" << endl;
    }
};

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

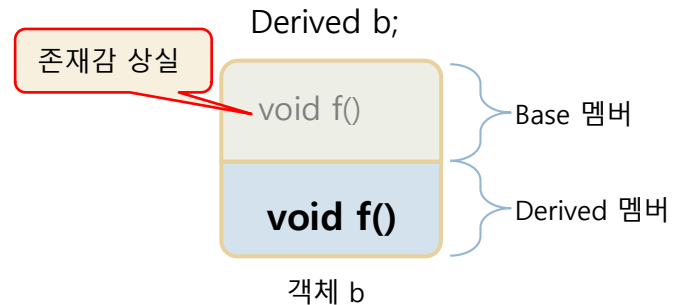
가상 함수

오버라이딩

오버라이딩(overriding)



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



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

함수 재정의와 오버라이딩 용어의 혼란 정리

함수 재정의라는 용어를 사용할 때 신중을 기해야 한다. 가상 함수를 재정의하는 경우와 아닌 경우에 따라 프로그램의 실행이 완전히 달라지기 때문이다([그림 9-3] 참고).

가상 함수를 재정의하는 **오버라이딩**의 경우 함수가 호출되는 실행 시간에 **동적 바인딩**이 일어나지만, 그렇지 않은 경우 컴파일 시간에 결정된 함수가 단순히 호출된다(정적 바인딩).

저자는 가상 함수를 재정의하는 것을 **오버라이딩**으로, 그렇지 않는 경우를 **함수 재정의**로 구분하고자 한다.

Java의 경우 이런 혼란은 없다. 멤버 함수가 가상이나 아니냐로 구분되지 않으며, 함수 재정의는 곧 오버라이딩이며, **무조건 동적 바인딩**이 일어난다.

예제 9-2 오버라이딩과 가상 함수 호출

8

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

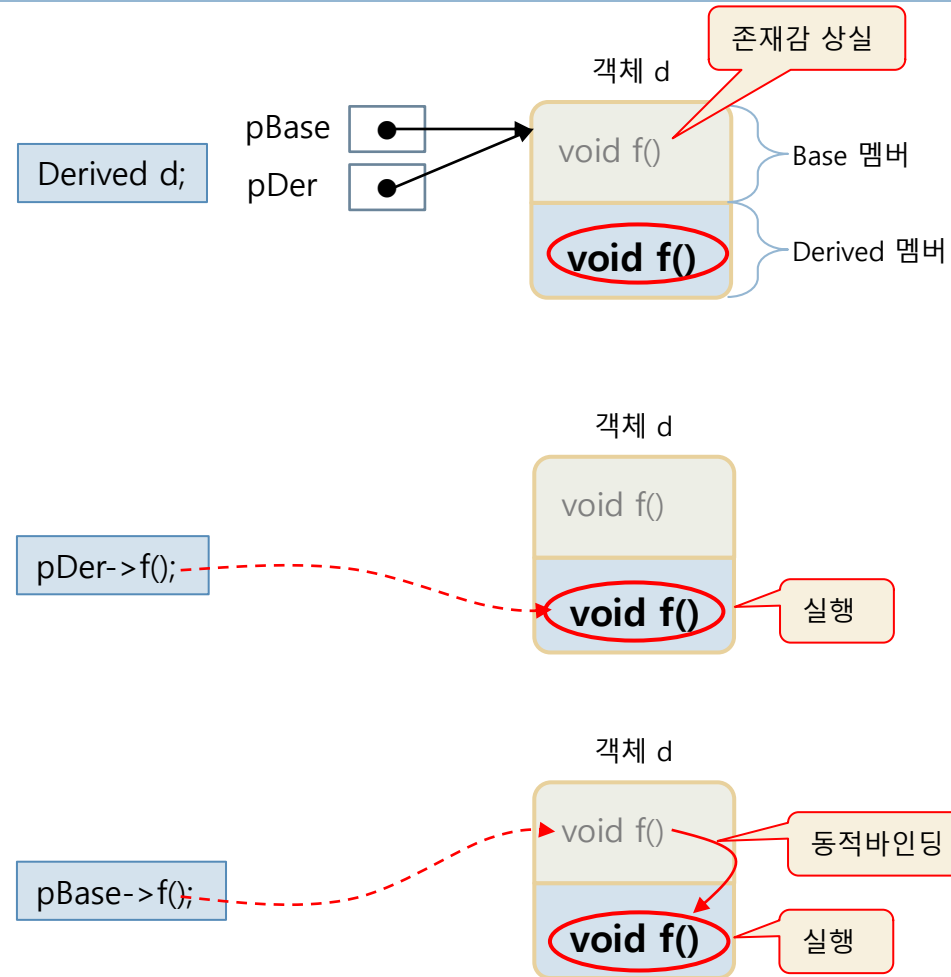
class Base {
public:
    virtual void f() { cout << "Base::f() called" << endl; }
};

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(); // 동적 바인딩 발생!! Derived::f() 실행
}
```

가상 함수 선언



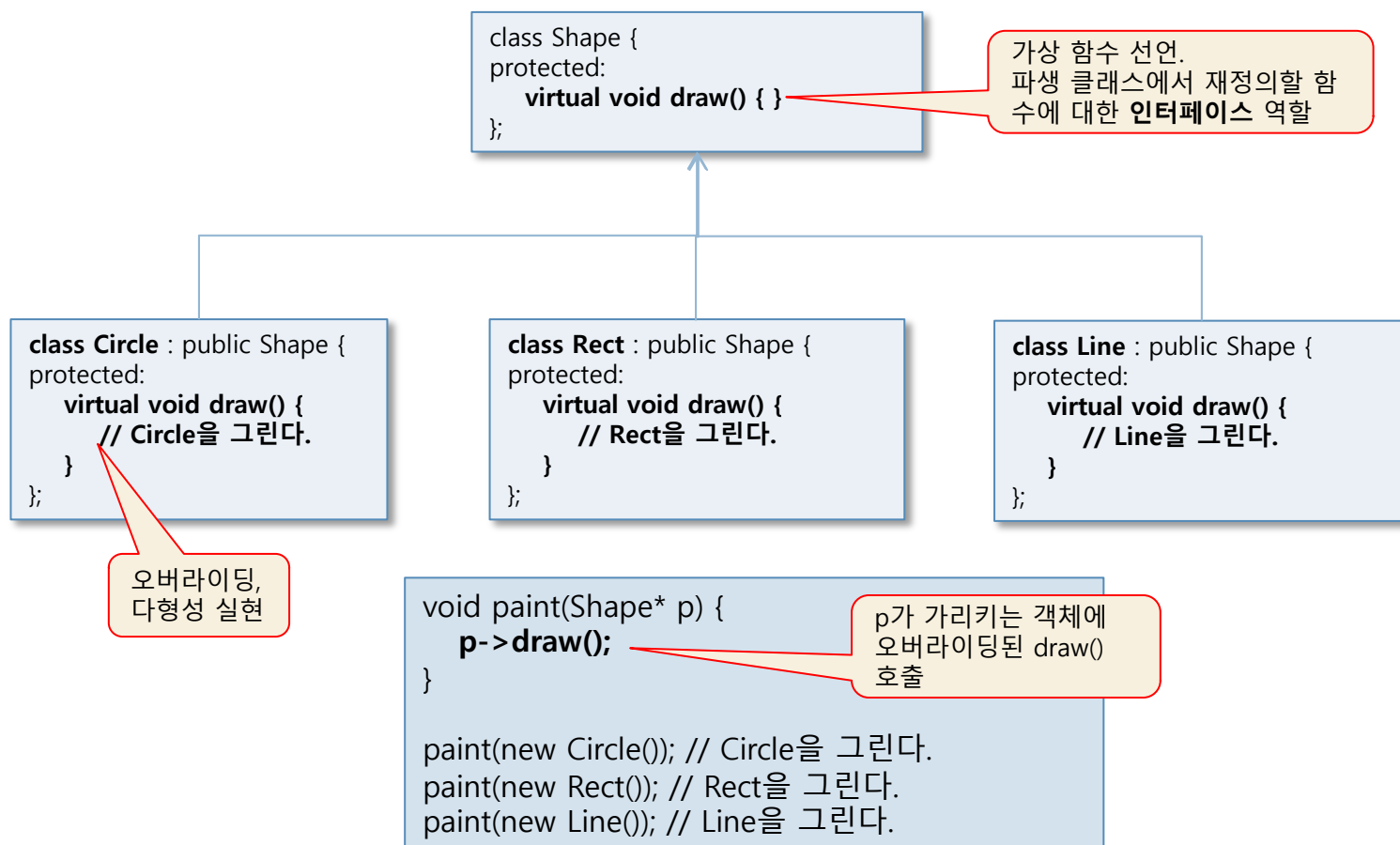
Derived::f() called
Derived::f() called

오버라이딩의 목적 -파생 클래스에서 구현할 함수 인터페이스 제공(파생 클래스의 다형성)

9

다형성의 실현

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



동적 바인딩

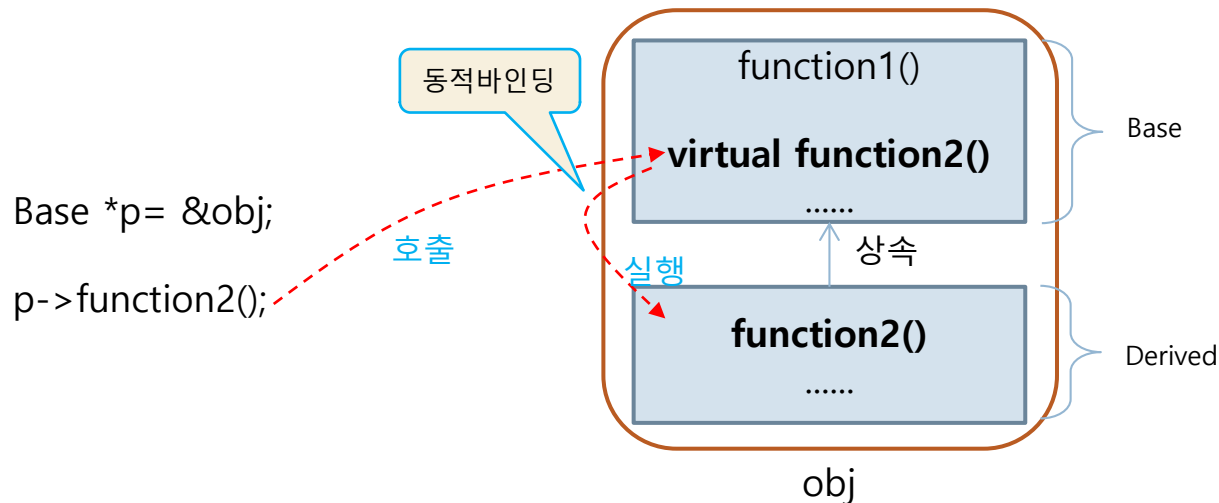
10

□ 동적 바인딩

- 파생 클래스에 대해
- 기본 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



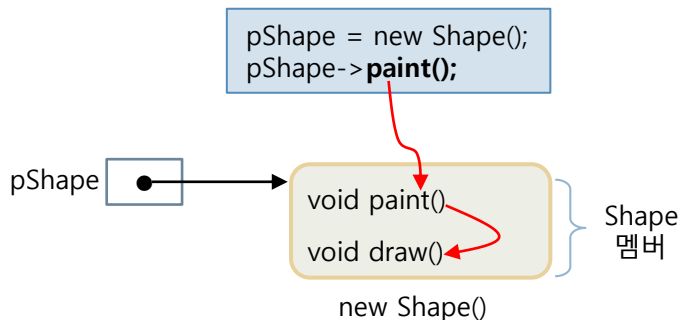
오버라이딩된 함수를 호출하는 동적 바인딩

```
#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;

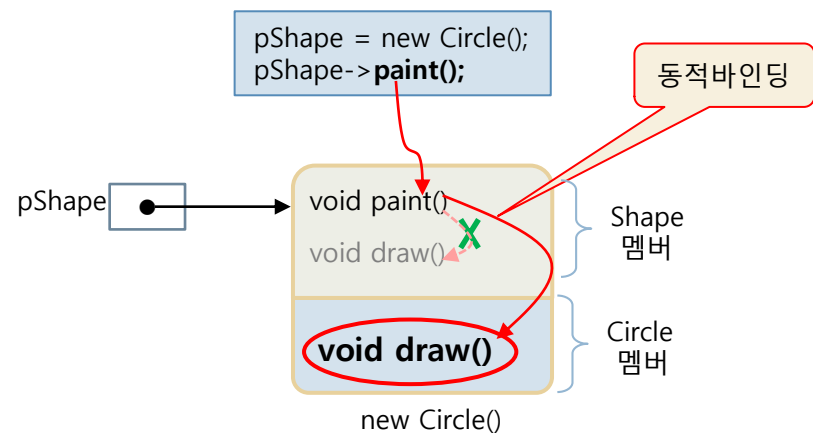
class Shape {
public:
    void paint() {
        draw();
    }
    virtual void draw() {
        cout << "Shape::draw() called" << endl;
    }
};

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++ 오버라이딩의 특징

12

- 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
    virtual void g(int, double); // 오버로딩 사례. 정상 컴파일
};
```

= 함수 중복 정의

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

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

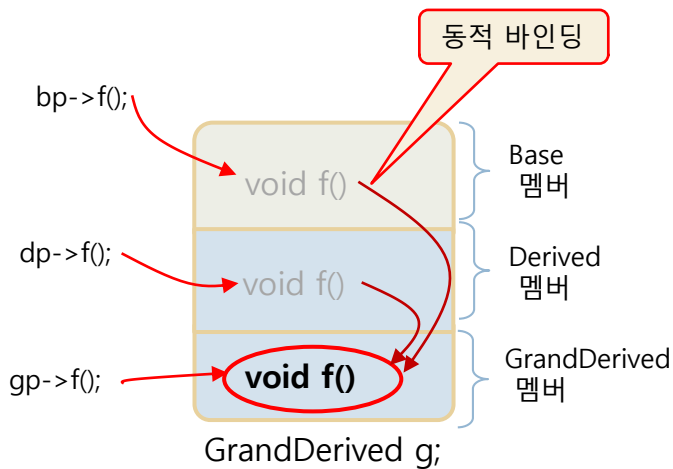
생략 가능

- 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

예제 9-3 상속이 반복되는 경우 가상 함수 호출

13

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



정적 바인딩

Base
Derived
GrandDerived

동적 바인딩

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; }
}; (virtual)

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

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

    bp = dp = gp = &g; //업캐스팅

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

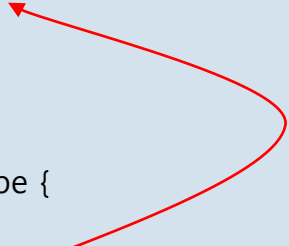
동적 바인딩에 의해 모두
GrandDerived의 함수 f()
호출

오버라이딩과 범위 지정 연산자(::)

14

- 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. ← 정적바인딩으로 호출  
        .... // 기능을 추가한다.  
    }  
};
```



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

15

```
#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; // 업캐스팅

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

정적바인딩

동적바인딩

정적바인딩

동적 바인딩을 포함하는 호출

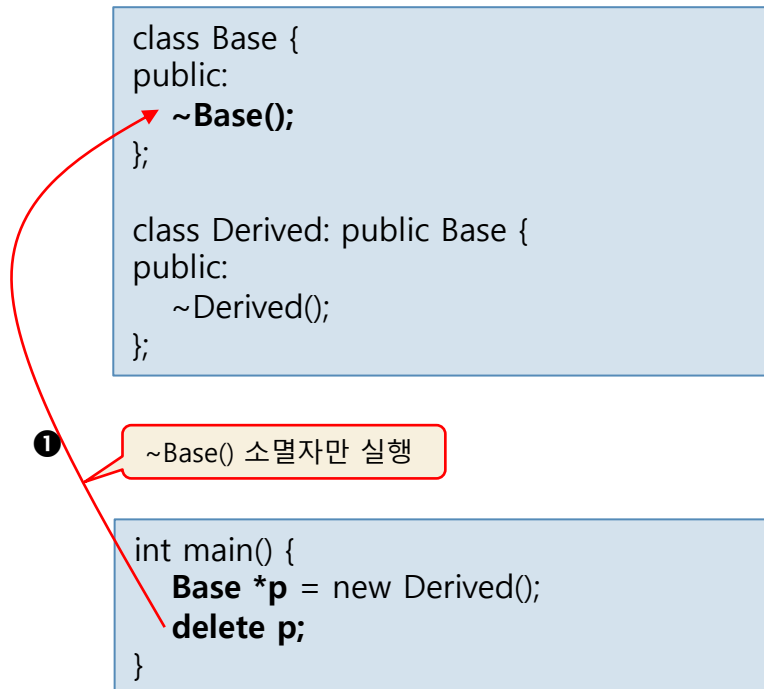
--Shape--Circle
--Shape--

가상 소멸자

16

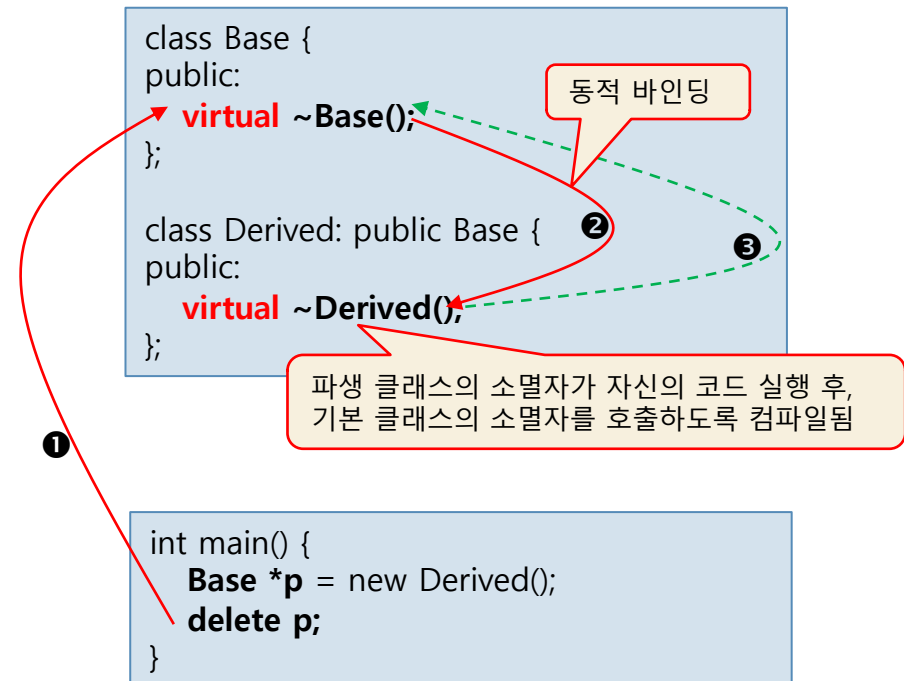
가상 소멸자

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



① ~Base() 소멸자 실행

소멸자가 가상 함수가 아닌 경우



① ~Base() 소멸자 호출

② ~Derived() 실행

③ ~Base() 실행

가상 소멸자 경우

예제 9-6 소멸자를 가상 함수로 선언

17

```
#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
}
```

~Base() 소멸자 호출
~Derived() 실행
~Base() 실행

The diagram illustrates the sequence of destructor calls for two delete operations. For `delete dp;`, the destructors `~Derived()` and `~Base()` are called in sequence. For `delete bp;`, the destructors `~Derived()` and `~Base()` are also called in sequence. Red curly braces group the destructor calls for each delete operation, and red callout boxes point to the corresponding delete statement.

```
~Derived()
~Base()
~Derived()
~Base()
```

delete dp;

delete bp;

오버로딩과 함수 재정의, 오버라이딩 비교

18

비교 요소	오버로딩	함수 재정의(가상 함수가 아닌 멤버에 대해)	오버라이딩
정의	매개 변수 타입이나 개수가 다르지만, 이름이 같은 함수들이 중복 작성되는 것	기본 클래스의 멤버 함수를 파생 클래스에서 이름, 매개 변수 타입과 개수, 리턴 타입까지 완벽히 같은 원형으로 재작성하는 것	기본 클래스의 가상 함수를 파생 클래스에서 이름, 매개 변수 타입과 개수, 리턴 타입까지 완벽히 같은 원형으로 재작성하는 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

가상 함수와 오버라이딩 활용 사례

- 가상 함수를 가진 기본 클래스의 목적
- 가상 함수 오버라이딩
- 동적 바인딩 실행
- 기본 클래스의 포인터 활용

1. 가상 함수를 가진 기본 클래스의 목적

20

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

2. 가상 함수 오버라이딩

21

- 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 자신이 만든 draw()가 호출됨을 보장 받음
 - 동적 바인딩에 의해

3. 동적 바인딩 실행 : 파생 클래스의 가상 함수 실행

22

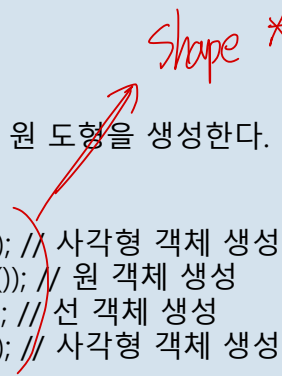
```
#include <iostream>
#include "Shape.h"
#include "Circle.h"
#include "Rect.h"
#include "Line.h"
using namespace std;

int main() {
    Shape *pStart=NULL;
    Shape *pLast;

    pStart = new Circle(); // 처음에 원 도형을 생성한다.
    pLast = pStart;

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

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

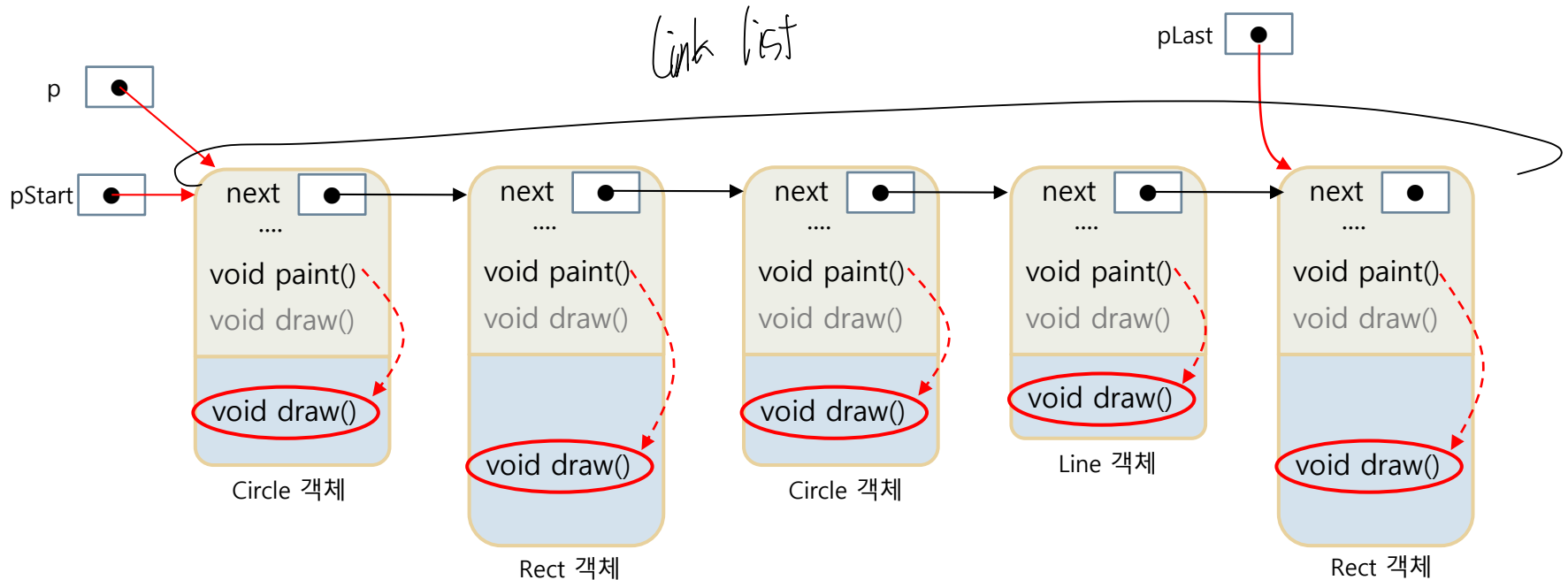


```
// 현재 연결된 모든 도형을 삭제한다.
p = pStart;
while(p != NULL) {
    Shape* q = p->getNext(); // 다음 도형 주소 기억
    delete p; // 기본 클래스의 가상 소멸자 호출
    p = q; // 다음 도형 주소를 p에 저장
}
```

Circle
Rectangle
Circle
Line
Rectangle

main() 함수가 실행될 때 구성된 객체의 연결

23



4. 기본 클래스의 포인터 활용

24

- 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

순수 가상 함수

25

- 기본 클래스의 가상 함수 목적
 - ▣ 파생 클래스에서 재정의할 함수를 알려주는 역할
 - 실행할 코드를 작성할 목적이 아님
 - ▣ *기본 클래스의 가상 함수를 굳이 구현할 필요가 있을까?*
- 순수 가상 함수
 - ▣ pure virtual function
 - ▣ 함수의 코드가 없고 선언만 있는 **가상 멤버** 함수
 - ▣ 선언 방법 ^{구현X}
 - 멤버 함수의 원형 **=0;**으로 선언

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

추상 클래스

26

- 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.
}
```

불완전
구현

- 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;
```

자식 클래스 객체

추상 클래스의 목적

27

□ 추상 클래스의 목적

- ▣ 추상 클래스의 인스턴스를 생성할 목적 아님
- ▣ 상속에서 기본 클래스의 ^{개별 객체} 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

추상 클래스의 상속과 구현

28

- 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;  
};
```

Circle도
추상 클래스

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

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

추상 클래스의 단순 상속



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

Shape은
추상 클래스

```
class Circle : public Shape {  
public:
```

Circle은
추상 클래스 아님

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

순수 가상 함수
오버라이딩

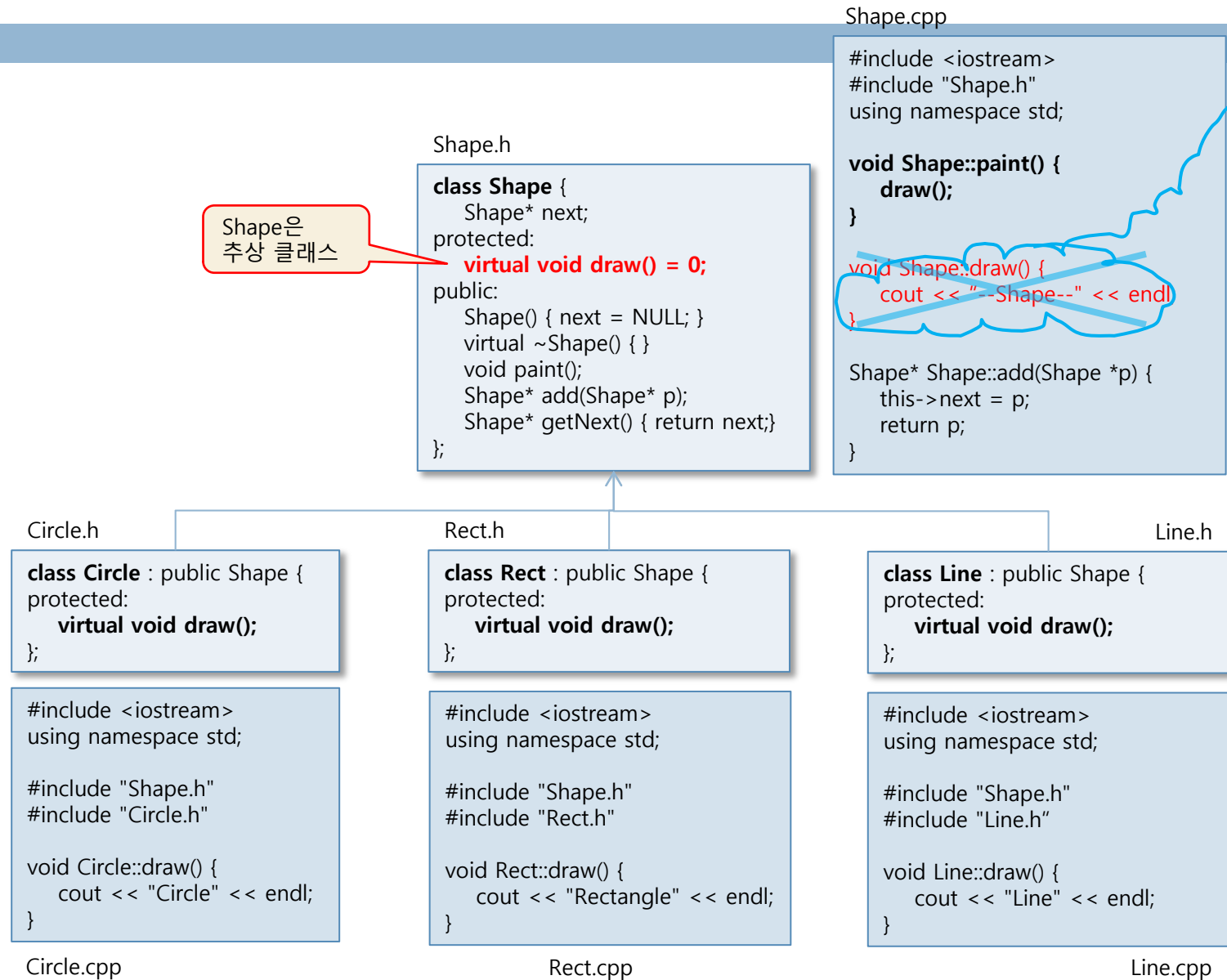
```
    string toString() { return "Circle 객체"; }  
};
```

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

추상 클래스의 구현

Shape을 추상 클래스로 수정

29



예제 9-6(실습) 추상 클래스 구현 연습

30

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;
    }
};
```

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

예제 9-7(실습) 추상 클래스를 상속받는 파생 클래스 구현 연습

31

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
```

예제 9-7 정답

32

```
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;  
    }  
};
```

오버로딩 = 함수 중복

함수 재정의 (가상함수가 아닌 멤버에 대해)

오버라이딩