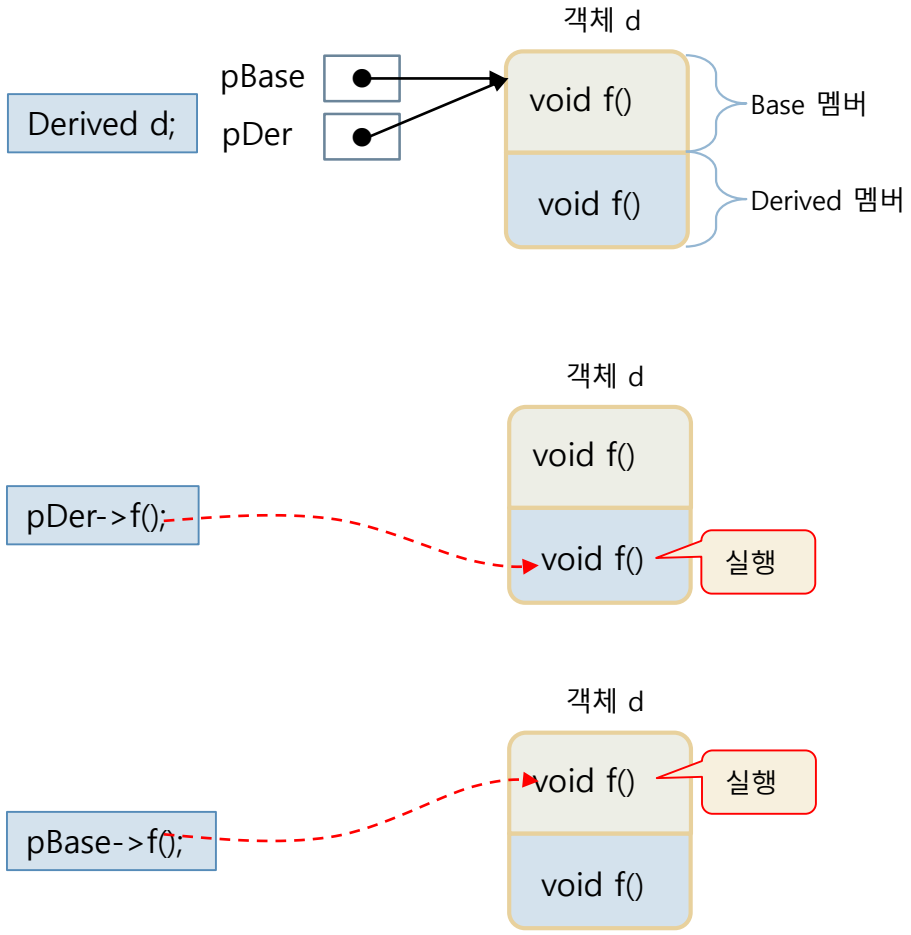
Ch13 오버라이딩

예제 9-1 파생클래스에서 함수를재정의하는사례


```
#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
```



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
};
```

- 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

기태네 집

함수 재정의와 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

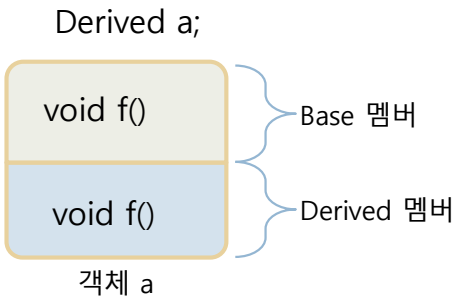
함수 재정의(redefine)

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

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

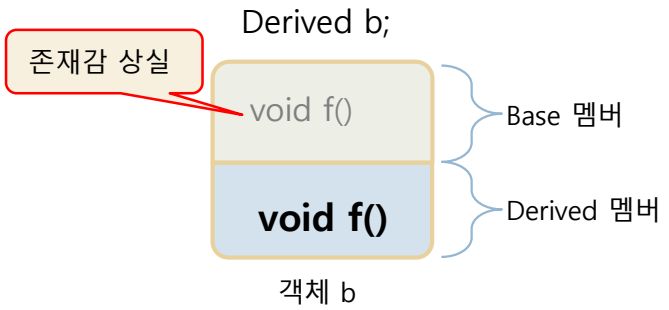
가상 함수

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.

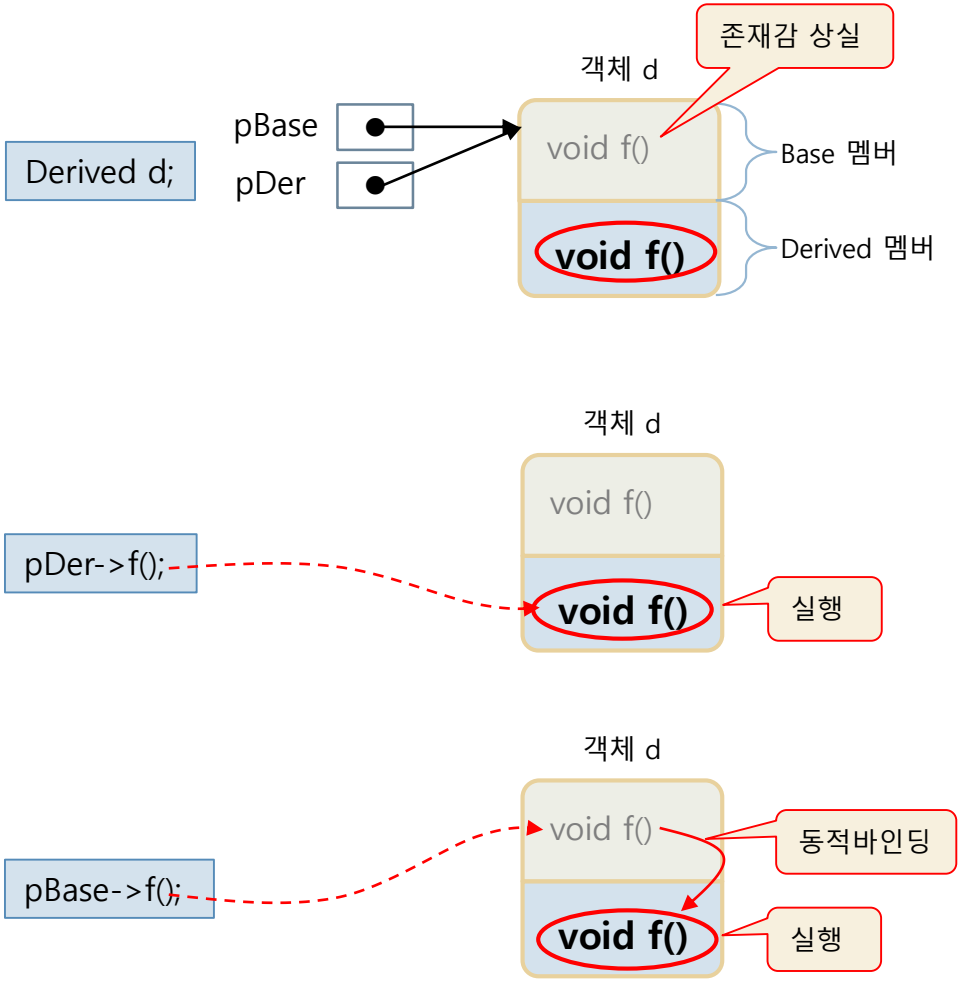
Java의 경우 이런 혼란은 없다. 멤버 함수가 가상이냐 아니냐로 구분되지 않으며, 함수 재정의는 곧 오버라이딩이며, **무조건 동적 바인딩**이 일어난다.

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

가상 함수 선언

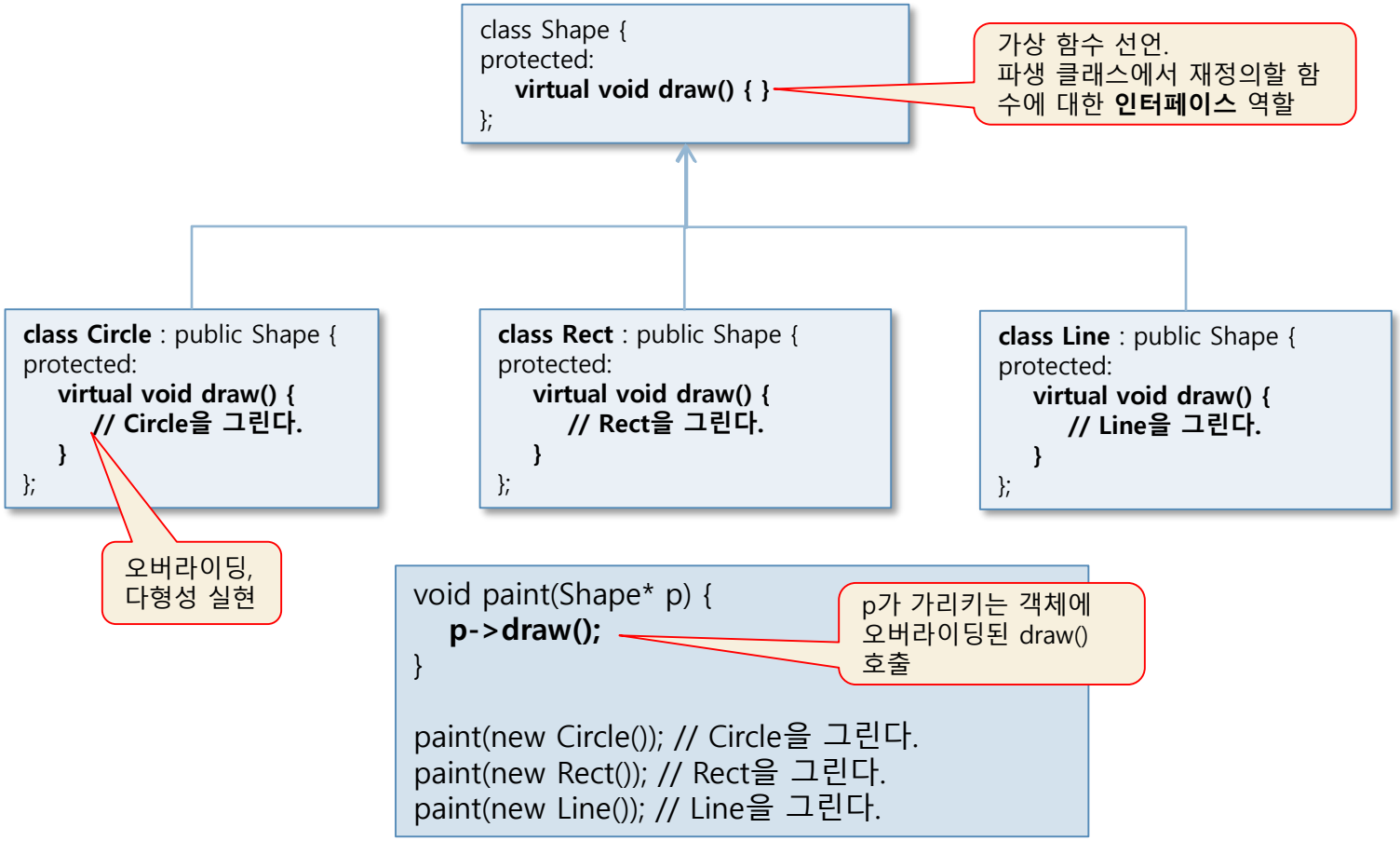
```
Derived::f() called
Derived::f() called
```



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

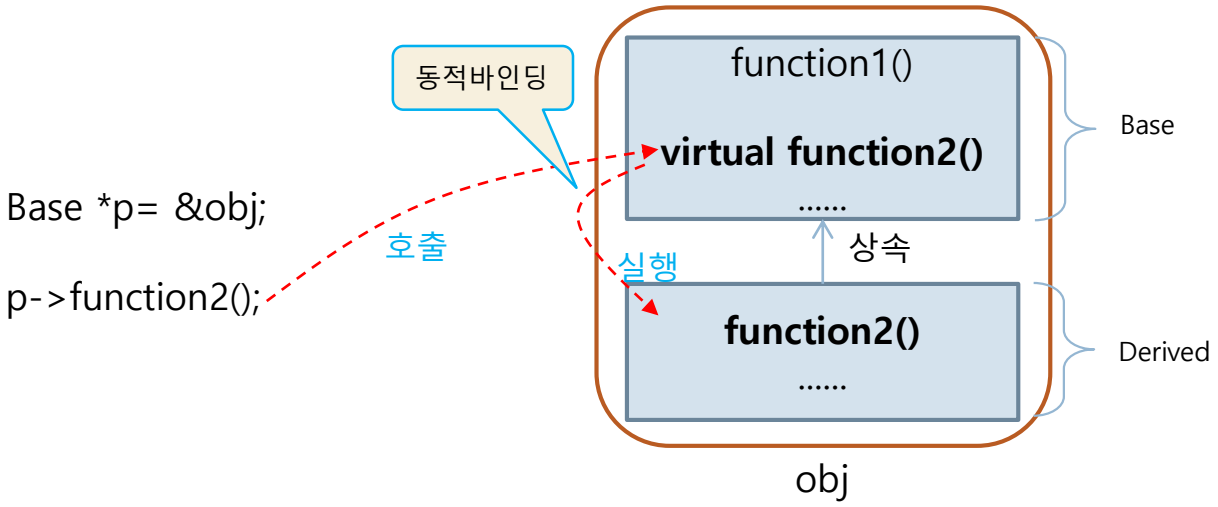
다형성의 실현

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



동적 바인딩

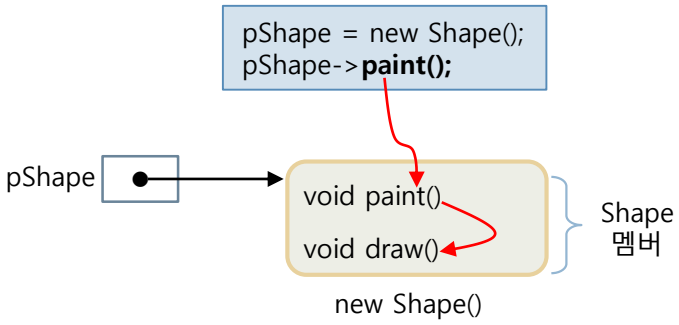
- 동적 바인딩
  - 파생 클래스에 대해
  - 기본 클래스에 대한 포인터로 가상 함수를 호출하는 경우
  - 객체 내에 오버라이딩한 파생 클래스의 함수를 찾아 실행
    - 실행 중에 이루어짐
      - 실행시간 바인딩, 런타임 바인딩, 늦은 바인딩으로 불림



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;
}
```

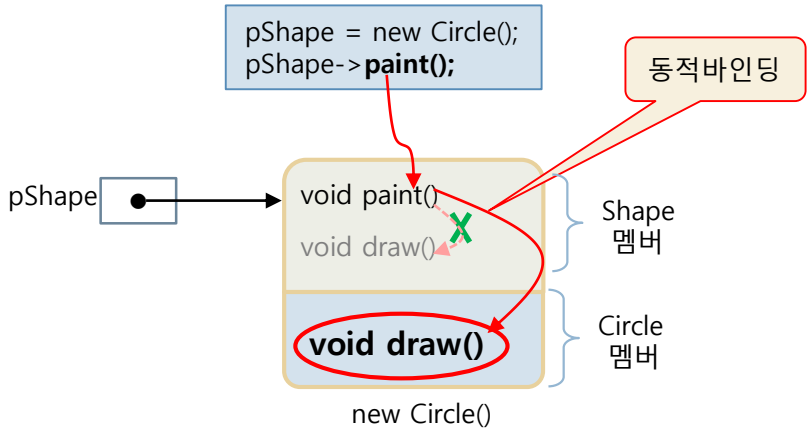
Shape::draw() called




```
#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

- 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
};
```

```
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

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

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

Circle도 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

```
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

Shape을 추상 클래스로 수정

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;
}
```

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;}
};
```

Shape은
추상 클래스

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

예제 9-6(실습) 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;
    }
};
```

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

다음 코드와 실행 결과를 참고하여 추상 클래스 Calculator를 상속받는 Adder와 Subractor 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


```
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;
   }
};
```