

# 가상함수와 추상클래스

# 가상 함수와 추상 클래스

- 이번 강의에서는 가상 함수와 추상클래스에 대하여 학습합니다
- 상속과 함께 객체 지향에서 중요한 개념이 다형성 입니다.
- 이번 시간에는 다형성 구현과 관련하여 다음을 잘 이해하셔야 합니다
  - 가상 함수
  - 동적 바인딩
  - 함수 오버라이딩
- 시작하도록 하겠습니다.

# 학습 목차

- 상속 관계에서 함수 재정의
- 가상 함수와 오버라이딩
- 런타임 타입 정보와 동적 자료형 변환
- 추상 클래스

# 학습 목표

- 함수 오버라이딩과 동적 바인딩을 이해한다
- 가상 함수를 이해한다
- 다형성을 이해한다
- 다형성을 적용한 프로그램을 구현할 수 있다.
- 추상 클래스의 목적을 이해하고 프로그램에 활용할 수 있다.

# 상속 관계에서 함수 재정의

- 정적 바인딩 – 함수 재정의(컴파일 시간 다형성)

```

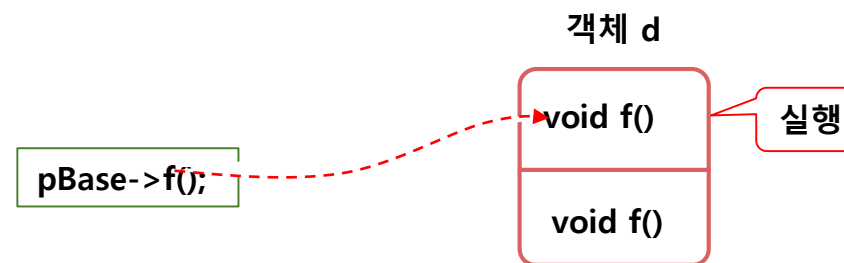
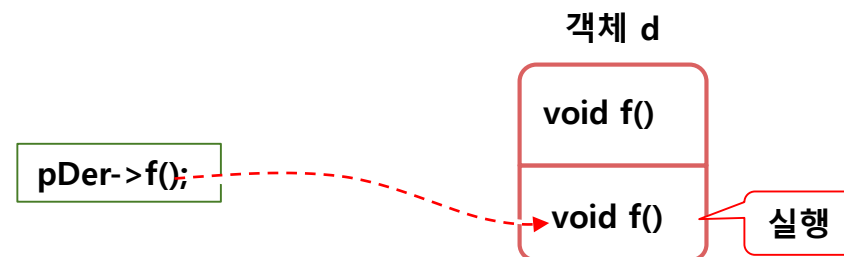
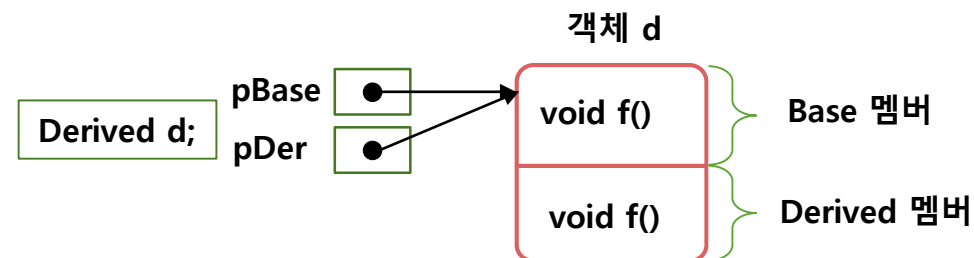
class Base {
public:
    void f() { cout << "Base::f() called" << endl; }
};

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

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

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

```



# 다형성

- 다형성은 같은 이름을 갖는 여러 형태의 함수를 클래스 별로 만들 수 있게 해주는 기능
- 삼각형 객체의 printArea는 삼각형의 넓이, 사각형 객체의 printArea는 사각형의 넓이를 구하는 형태로 사용될 수 있게 하는 것이 다형성
- C++ 다형성 요소
  - 포인터
    - 베이스 클래스를 가리키는 포인터를 사용 베이스클래스와 파생 클래스 모두를 가리키게 함
  - 호환 객체
    - 상속을 하는 객체
  - 가상 함수

# 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
- 함수 오버라이딩(function overriding)
  - 파생 클래스에서 기본 클래스의 가상 함수와 동일한 이름의 함수 선언
    - 기본 클래스의 가상 함수의 존재감 상실시킴
    - 파생 클래스에서 오버라이딩 한 함수가 호출되도록 동적 바인딩 - 실행시간 다형성
    - 함수 재정의라고도 부름 - 다형성의 한 종류
- 오버라이딩의 목적
  - 파생 클래스에서 구현할 함수 인터페이스 제공(파생 클래스의 다형성)

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

# 가상 함수와 오버라이딩

- 오버라이딩과 가상 함수 호출

```

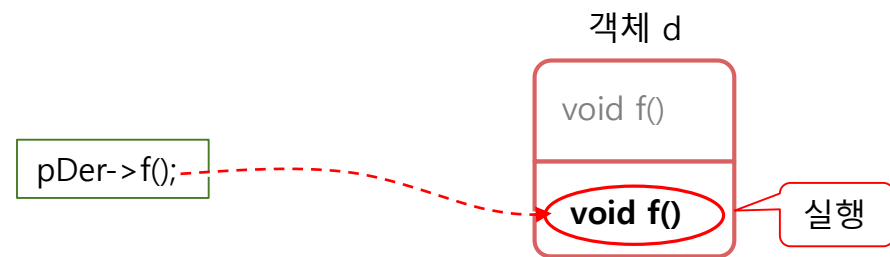
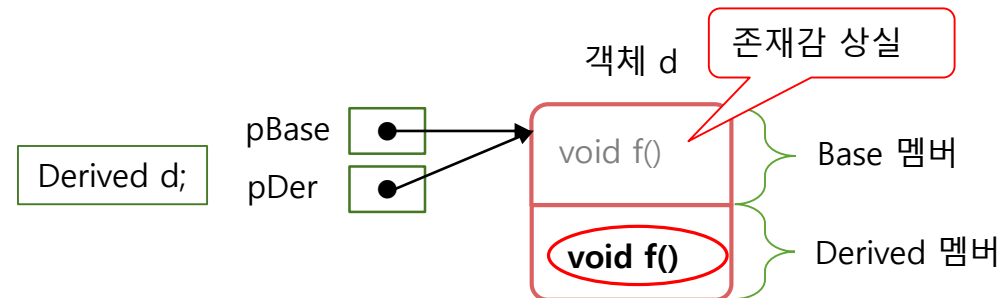
class Base {
public: //가상 함수 선언
    virtual void f() { cout << "Base::f() called" << endl; }
};

class Derived : public Base {
public:
    //오버라이딩, virtual 생략 가능, 동적 바인딩
    virtual void f() { cout << "Derived::f() called" << endl; }
};

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

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

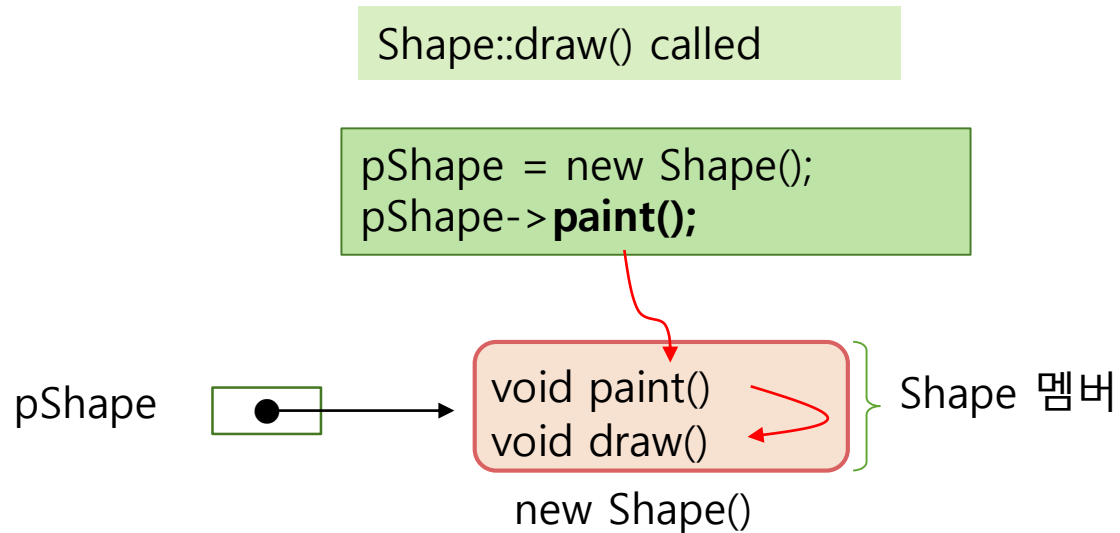
```





# 가상 함수와 오버라이딩

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



```
#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;
}
```

# 가상 함수와 오버라이딩

- 오버라이딩된 함수를 호출하는 동적 바인딩

```
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 함수 호출

Circle::draw() called

pShape = new Circle();  
pShape->**paint()**;

pShape



void paint()

void draw()

**void draw()**

new Circle()

Shape 멤버

동적바인딩

Circle 멤버

# 가상 함수와 오버라이딩

- override 와 final 지시어

```
class Shape {
public:
    void paint() { draw(); }
    virtual void draw();
};
class Circle : public Shape {
public:
    void draw(); //오버라이딩이 아닌 새로운 멤버 함수로 인식
};
//void draw() override; //파생 클래스 원형 뒤에 작성, 컴파일 오류 발생

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

- ❖ final 지시어
  - 가상 함수 뒤에 사용 - 파생 클래스에서 오버라이딩 금지
  - 클래스 이름 뒤에 사용 - 상속 금지

# 가상 함수와 오버라이딩

- 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  
    void success();             // 오버라이딩 성공, virtual 생략 가능  
    virtual void g(int, double); // 오버로딩 사례. 정상 컴파일  
};
```

- 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

# 가상 함수와 오버라이딩

- 상속이 반복되는 경우 가상 함수 호출

[실행 결과]

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
}
```

# 가상 함수와 오버라이딩

- 범위 지정 연산자(::)
  - 정적 바인딩 지시
  - 기본클래스::가상함수()** 형태로 기본 클래스의 가상 함수를 정적 바인딩으로 호출

정적바인딩

정적바인딩

```
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;

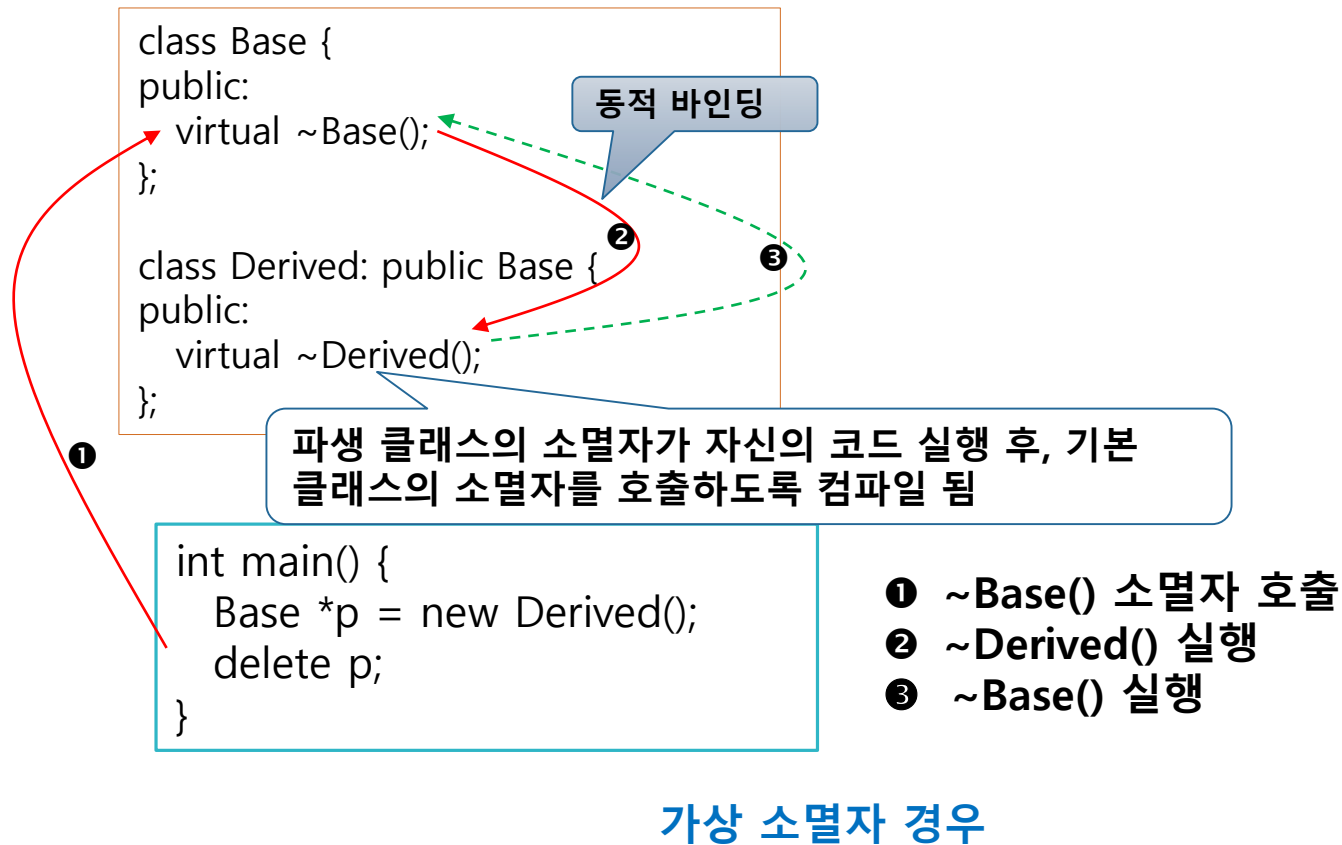
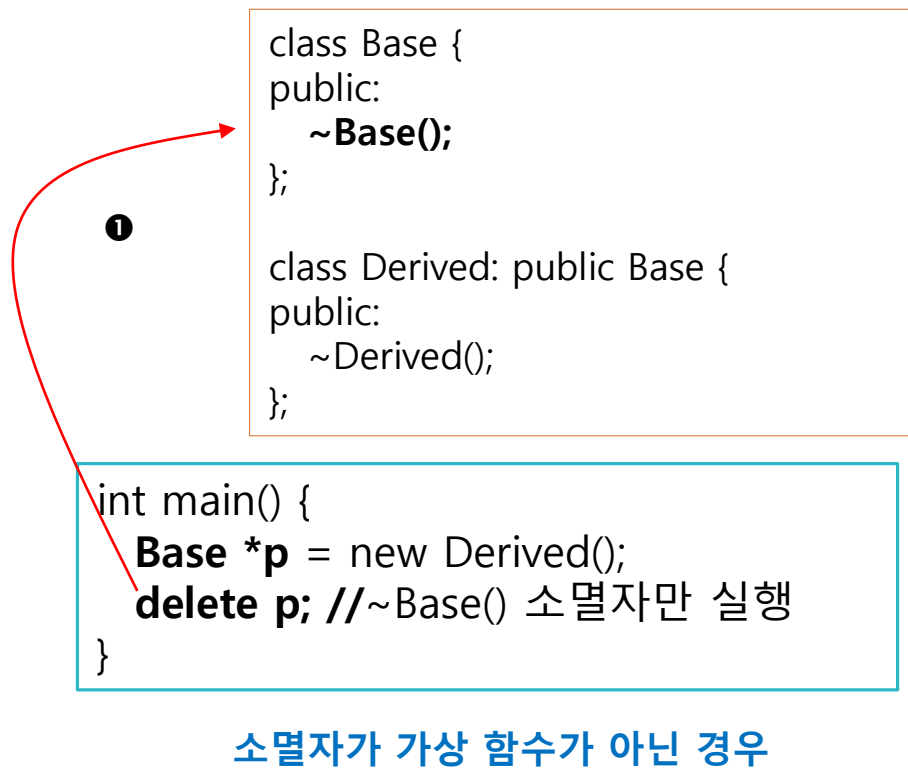
    pShape->draw(); //동적 바인딩
    pShape->Shape::draw(); //범위 지정 연산자로 정적 바인딩
}
```

동적바인딩

--Shape--Circle  
--Shape--

# 가상 함수와 오버라이딩

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



# 가상 함수와 오버라이딩

- 소멸자를 가상 함수로 선언

- 소멸자를 가상함수로 선언
  - ✓ 파생클래스와 기본 클래스의 소멸자를 모두 실행
  - ✓ 기본 클래스의 소멸자를 가상함수로 선언하여 정상적인 소멸 과정이 진행되도록 하는 것이 좋음

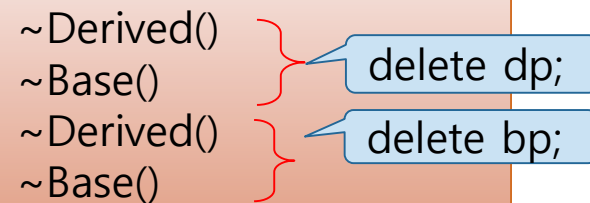
```
#include <iostream>
using namespace std;

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
}
```





# 오버로딩과 함수 재정의, 오버라이딩 비교

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

# 런타임 타입 정보와 동적 자료형 변환

- 런타임 타입 정보
  - 클래스가 계층 구조를 가질 때는 객체가 어떤 객체인지 확인하거나, 객체의 자료형을 변경해야 하는 경우가 있음 - 런타임 타입 정보(RTTI, Run-Time Type Information) 이용
  - typeid 연산자
    - 런타임 때 객체의 자료형 확인
    - <typeinfo> 헤더에 있는 type\_info 클래스 사용
    - type\_info 클래스는 생성자, 소멸자, 복사 생성자가 없음
    - type\_info 객체는 typeid라는 연산자를 호출해서 생성
- 동적 자료형 변환
  - 베이스 클래스에 대한 포인터에 파생 클래스의 객체 대입- 업 캐스트(upcast)  
`Shape *pShape = new Circle(); // 업캐스팅`
  - 파생 클래스에 대한 포인터에 베이스 클래스의 객체 대입 - 다운 캐스트 (downcast)
    - dynamic\_cast 연산자 사용 - virtual 함수가 없으면 오류`Circle *pCircle = dynamic_cast<Circle *>(pShape); //다운캐스팅`

# 런타임 타입 정보와 동적 자료형 변환

```
#include <typeinfo>
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; }
    void write() { cout << "Circle::write()" << endl; }
};
int main() {
    Shape *pShape = new Circle(); // 업캐스팅
    if (typeid(*pShape) == typeid(Circle)) { //type_info 객체에 ==, != 연산자 사용하여 자료형 비교
        //name() : type_info 객체의 자료형 반환, 컴파일러 마다 결과가 다름
        cout << "typeid(*pShape).name() : " << typeid(*pShape).name() << endl;
        Circle *pCircle = dynamic_cast<Circle *>(pShape); //다운캐스팅
        pCircle->write();
    }
    cout << "typeid(pShape).name() : " << typeid(pShape).name() << endl;
    delete pShape;
}
```

visual studio 2019

```
typeid(*pShape).name() : class Circle
Circle::write()
typeid(pShape).name() : class Shape *
```

visual studio code

```
typeid(*pShape).name() : 6Circle
Circle::write()
typeid(pShape).name() : P5Shape
```

# 가상 함수와 오버라이딩 활용

## 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.
    }
};
```

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

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

오버라이딩,  
다형성 실현

```
void paint( Shape* p) {
    p->draw(); //p가 가리키는 객체에 오버라이딩된 draw() 호출
}

paint( new Circle() ); // Circle을 그린다.
paint( new Rect() ); // Rect을 그린다.
paint( new Line() ); // Line을 그린다.
```

# 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 있을까?*
- 순수 가상 함수
  - pure virtual function
  - 함수의 **코드가 없고** 선언만 있는 **가상 멤버 함수**
  - 선언 방법
    - 멤버 함수의 원형 **=0;**으로 선언

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

# 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.
}
```

- 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 클래스

- 추상 클래스의 목적
  - 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 클래스

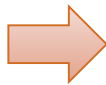
- 추상 클래스의 상속
  - 추상 클래스를 단순 상속하면 자동 추상 클래스
- 추상 클래스의 구현
  - 추상 클래스를 상속받아 순수 가상 함수를 오버라이딩
    - 파생 클래스는 추상 클래스가 아님

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

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

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

추상 클래스의 단순 상속



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

    추상 클래스의 구현
class Circle : public Shape { //추상 클래스 아님
public:
    virtual void draw() { //순수 가상 함수 오버라이딩
        cout << "Circle";
    }
    string toString() { return "Circle 객체"; }
};

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



# 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
};
```

```
class GoodCalc : public Calculator {
public:
    //순수 가상 함수 구현
    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;
    }
};
```

```
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 연습

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

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

int main() {
    Calculator *cp = new Adder();
    cp->run();
    delete cp;

    cp = new Subtractor();
    cp->run();
    delete cp;
    return 0;
}
```

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

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

# 활용 1. 가상 함수와 오버라이딩(1/2)

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()=0;
public:
    Shape() { next = nullptr; }
    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

# 활용 1. 가상 함수와 오버라이딩(2/2)

- 동적 바인딩 실행 : 파생 클래스의 가상 함수 실행

```
#include <iostream>
#include "Shape.h"
#include "Circle.h"
#include "Rect.h"
#include "Line.h"
using namespace std;
int main() {
    Shape *pStart=nullptr;
    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

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

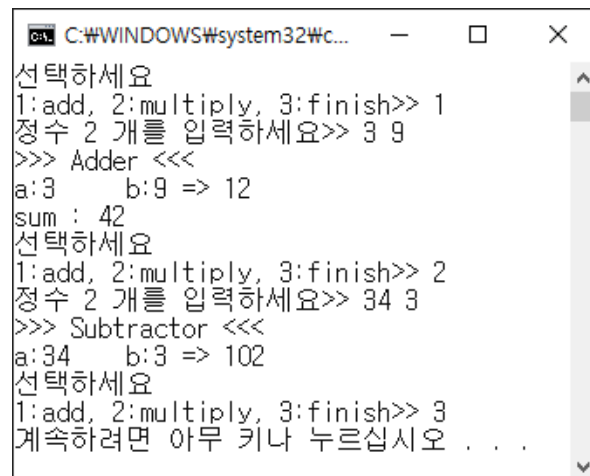
## 활용 2. 추상 클래스

- 다음과 같은 클래스를 상속받는 Adder, Mul 클래스를 구현하고 다형성을 사용하여 제시된 결과처럼 실행되는 프로그램을 완성 하시오. 단, 1을 선택하면 입력된 두 정수 a~b까지의 합을 계산하여 반환하는 sum() 함수를 호출하여 결과를 출력한다.

```
class Calculator {
protected:
    int a, b;
public:
    Calculator(int a, int b) : a(a), b(b) {}
    virtual int calc() = 0;
    virtual void write() {
        cout << "a:" << a << " * b:" << b << " => " ;
    }
};

class Adder : public Calculator {
public:
    int sum(); //Adder 구현 시 추가
};

int main() {
    Manage::run();
}
```



```
C:\WINDOWS\system32\cmd.exe
선택하세요
1:add, 2:multiply, 3:finish>> 1
정수 2 개를 입력하세요>> 3 9
>>> Adder <<<
a:3      b:9 => 12
sum : 42
선택하세요
1:add, 2:multiply, 3:finish>> 2
정수 2 개를 입력하세요>> 34 3
>>> Subtractor <<<
a:34     b:3 => 102
선택하세요
1:add, 2:multiply, 3:finish>> 3
계속하려면 아무 키나 누르십시오 . . .
```

# Q & A

- “가상 함수와 추상 클래스”에 대한 학습이 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
- 과제와 퀴즈(제한 기간내 2회 응시 가능)가 있습니다.
- 다음은 ‘템플릿과 STL’에 대하여 학습하도록 하겠습니다.
- 수고하셨습니다.^ ^