객체지향프로그래밍

Lecture 12 : 가상함수와 추상클래스

충북대 소프트웨어학부

이 태 겸(showm321@gmail.com)

목차

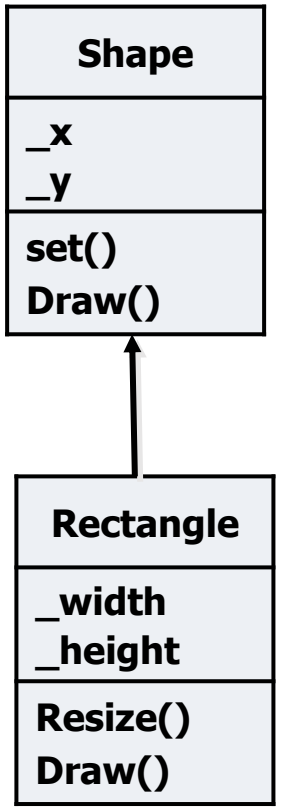
클래스 형 변환(casting) 규칙

❖ 캐스팅(casting) == 타입의 형 변환
- 데이터를 보는 관점의 변화
- int a = 1.0; => 변수 a 관점에서는 1로 간주
- float f = 1 => 변수 f 관점에서는 1.0로 간주

❖ 클래스 객체의 형 변환(casting)
- 상속의 관계에 있는 클래스들의 포인터 객체는 타입 변경이 허용됨

❖ 기본클래스 : Shape 클래스
- 직사각형과 원의 공통된 특징 제공

❖ 파생클래스 : Rectangle 클래스
- 기본 클래스가 제공해주는 공통된 특징 외에 직사각형의 구체적인 특징 가짐



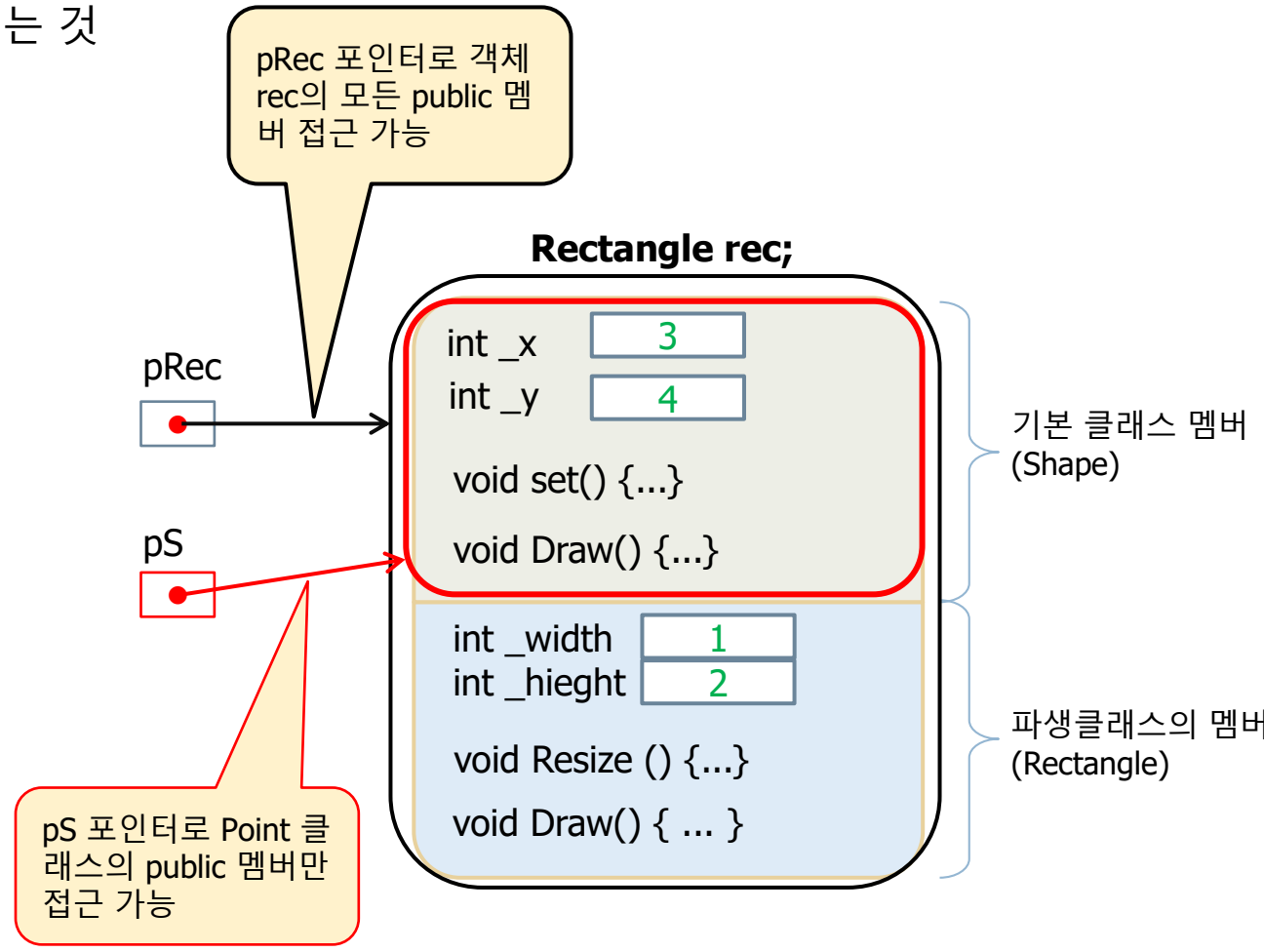
상속과 객체 포인터 – 업 캐스팅

❖ 업 캐스팅(up-casting)

- 파생 클래스 포인터가 기본 클래스 포인터에 치환 되는 것
- 하위 개념에서 상위 개념으로 치환

```
int main() {
    Rectangle rec;
    Rectangle *pRec = &rec;
    Shape* pS = pRec; // 업캐스팅

    pRec→set(3,4);
    pS→Draw();
    pRec→Resize(1,2);
    pRec→Draw();
    pS→Resize();// 컴파일 오류
}
```



상속과 객체 포인터 – 다운 캐스팅

❖ 다운 캐스팅(down-casting)

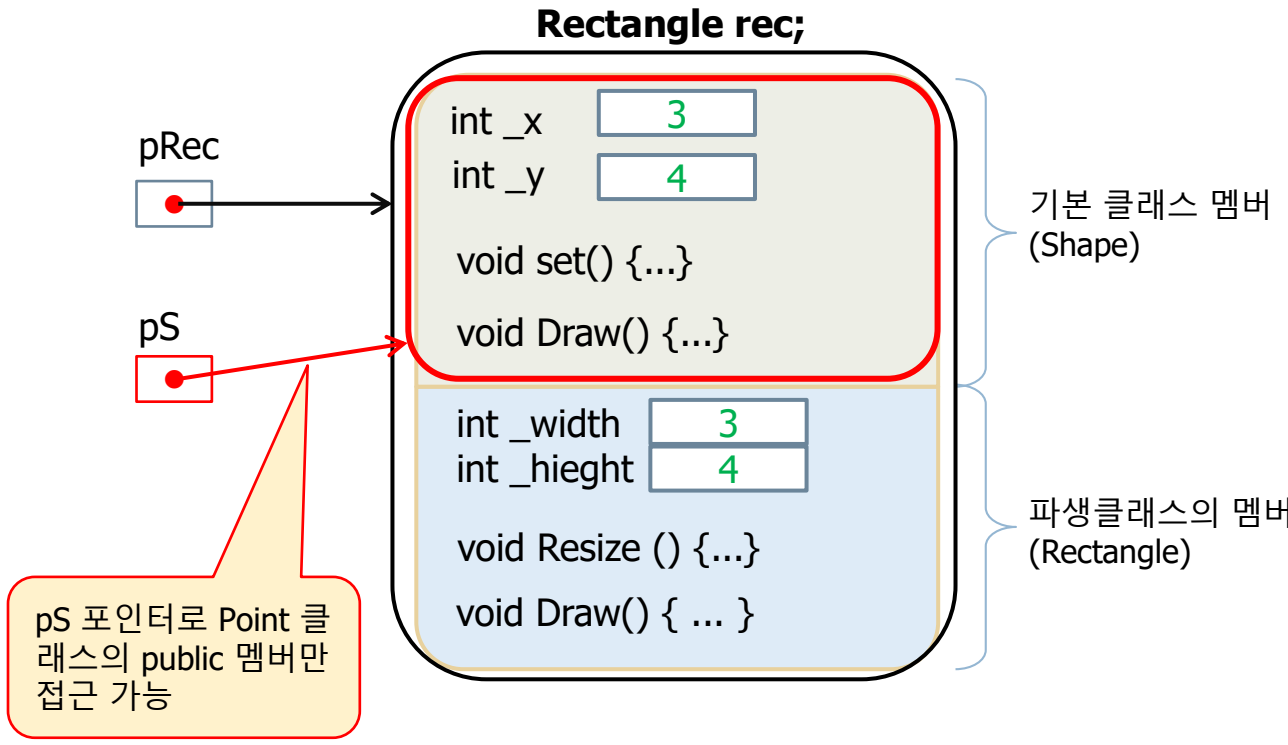
- 기본 클래스의 포인터가 파생 클래스의 포인터에 치환되는 것
- 포인터 객체 pS는 rec객체를 가리키고 있지만 기본 클래스 멤버에만 접근 가능
- 이 상태를 pRec로 치환하게 되면 **심각한 오류를 야기** (강제 형 변환 필요)

```
int main() {
    Rectangle rec;
    Rectangle *pRec = &rec;
    Shape* pS = pRec; // 업캐스팅

    pRec→set(3,4);
    pS→Draw();
    pRec→Resize(1,2);
    pRec→Draw();

    pRec = (Rectange*)pS; // 다운캐스팅
    pRec→Resize(3,4); // 정상 컴파일
    pRec→Draw(); // 정상 컴파일
}
```

강제 타입 변환 반드시 필요



목차

함수 재정의(overriding)

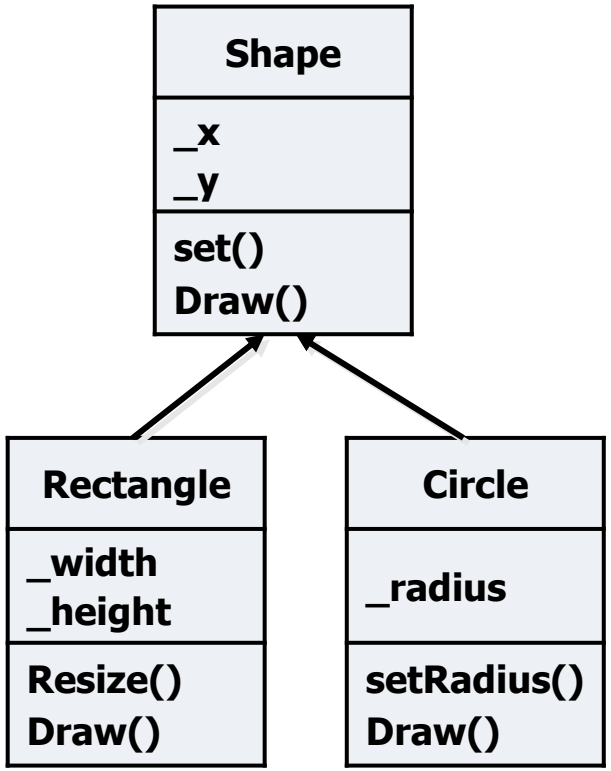
❖파생 클래스에서 기본 클래스의 동일한 이름의 함수 선언하는 것

❖파생 클래스에서 기본 클래스의 함수를 재정의 하려면,

- 함수 이름, 매개변수, 반환형이 **동일**해야 함

❖Shape* pS = new Rectangle(); //업 캐스팅

- pS->Draw();
  - Shape::Draw() ?
  - Rectangle::Draw() ?



포인터 pS를 통해
Ractangle::Draw()가 호출되게 하려면?

가상함수(virtual function)

❖ pS->Draw(); == Rectangle::Draw(); 가 되려면,
- 기본 클래스의 멤버 함수를 가상함수로 선언

❖ 가상함수의 사용
- virtual 키워드를 기본 클래스의 멤버 함수에 지정
- 함수 선언에만 지정하고 함수 정의에는 쓰지 않음 (써도 문제는 없다)
- 기본 클래스의 멤버 함수를 가상 함수로 지정하면 파생 클래스에서 재정의되는 함수는 virtual 키워드를 지정하지 않아도 자동으로 가상 함수가 됨

```
class Shape  {
    ...
    virtual void Draw() const;      // Draw()는 가상함수
};
```

```
class Rectangle  {
    ...
    void Draw() const;  // Draw()는 재정의된 함수
};
```

동적 바인딩

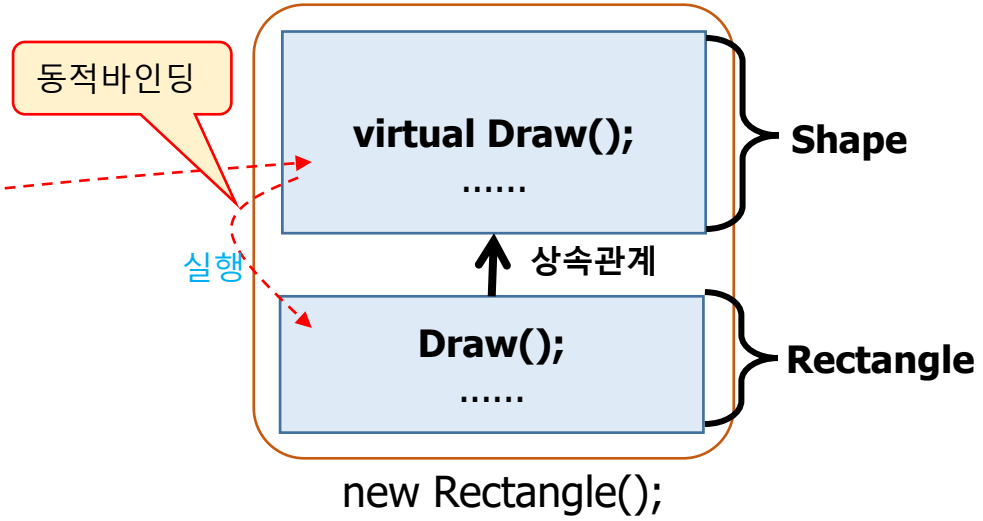
❖ 바인딩(binding): 프로그램의 어떤 이름(예: 함수 이름, 변수 이름)이 실제 메모리 상의 대상과 연결되는 과정

❖ 동적 바인딩 (Dynamic)

- 함수 호출이 실행시간에 결정되는 것
- **virtual 함수** 일 때만 적용 **<-> 정적 바인딩(함수 호출이 컴파일시간에 결정됨)**
- 기본 클래스의 포인터/참조로 파생 클래스의 함수를 호출하는 경우 사용됨
- 포인터/참조가 가리키고 있는 객체의 함수를 호출

Shape* pS= new Rectangle();

pS→Draw();



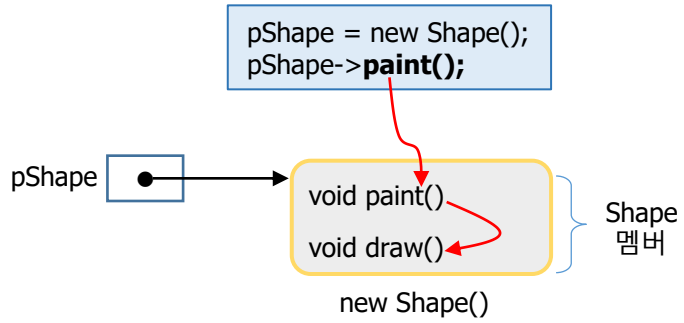
동적 바인딩

```
#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



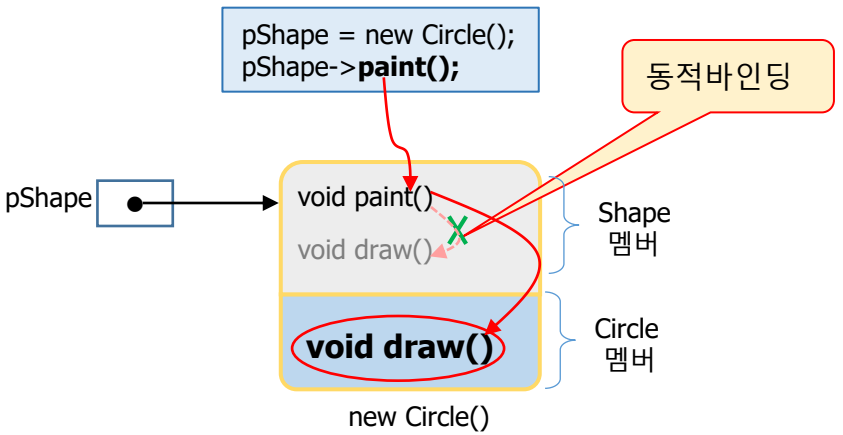

```
#include <iostream>
using namespace std;

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();
    pShape->paint();
    delete pShape;
}
```

Circle::draw() called



가상 소멸자

❖ 소멸자도 가상함수로 선언해야 하는 경우

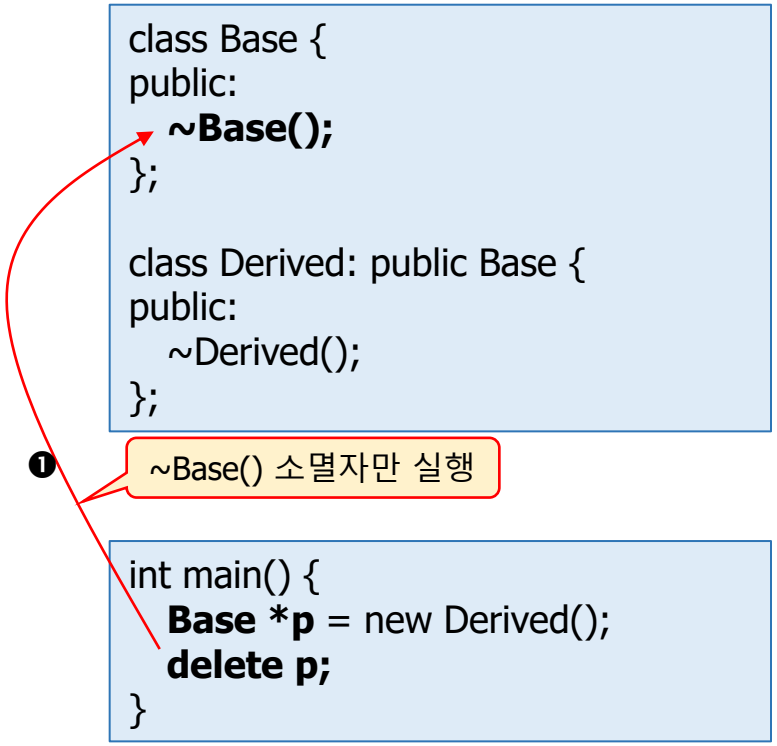
- 동적 바인딩의 상황

```
Shape* pShape = new Rectangle;
delete pShape;          // Rectangle 소멸자가 호출되도록 하려면
                        // Shape 소멸자로 가상함수로 만든다.
```

```
class Shape {
    ...
    virtual void Draw() const;
    ...
    virtual ~Shape();
};
```

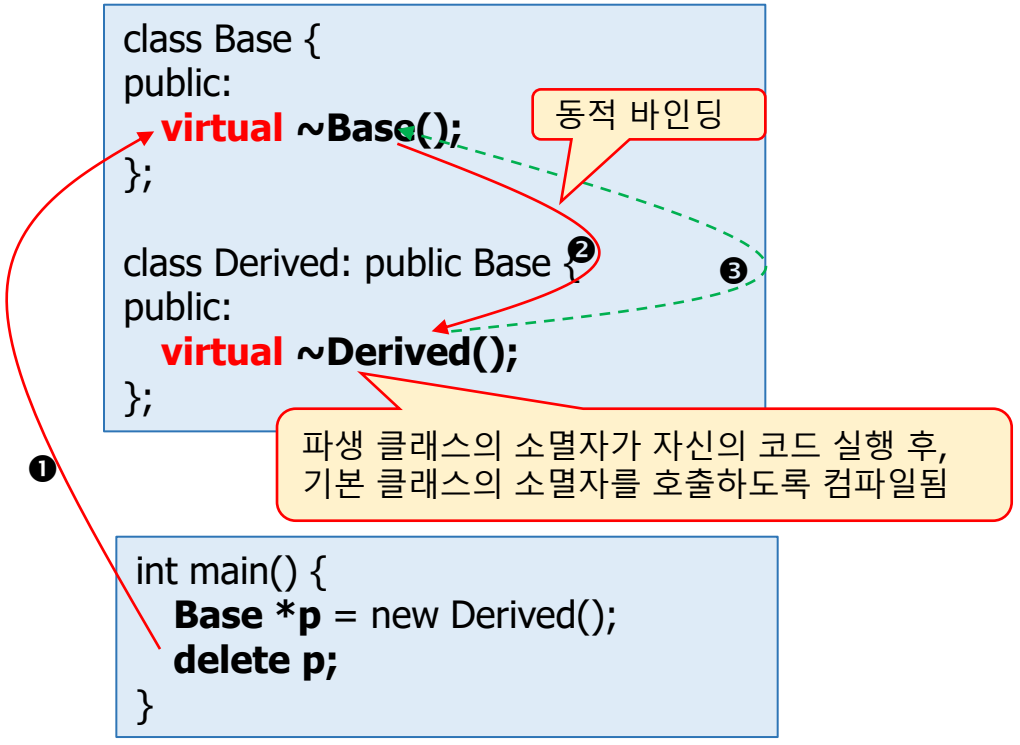
가상 소멸자가 필요한 이유

❖Derived 클래스 객체의 두 부분(Base의 멤버, Derived의 멤버)이 순서대로 정리되어야 하기 때문



❶ ~Base() 소멸자 실행

소멸자가 가상 함수가 아닌 경우



❶ ~Base() 소멸자 호출
❷ ~Derived() 실행
❸ ~Base() 실행

가상 소멸자 경우

오버로딩과 오버라이딩 비교

비교 요소	오버로딩	오버라이딩
정의	매개 변수 타입이나 개수가 다르지만, 이름이 같은 함수들이 중복 작성되는 것	기본 클래스에 선언된 가상 함수를 파생 클래스에서 이름, 매개 변수 타입, 매개 변수 개수, 리턴 타입까지 완벽히 같은 원형으로 재작성하는 것
존재	외부 함수들 사이. 한 클래스의 멤버들. 상속 관계	상속 관계. 가상 함수에서만 적용
목적	이름이 같은 여러 개의 함수를 중복 작성하여 사용의 편의성 향상	기본 클래스에 구현된 가상 함수를 무시하고, 파생 클래스에서 새로운 기능으로 재정의하고자 함
바인딩	정적 바인딩. 컴파일 시에 중복된 함수들의 호출 구분	동적 바인딩. 실행 시간에 오버라이딩된 함수를 찾아 실행
관련 객체 지향 특성	다형성	다형성

목차

순수 가상함수

❖ 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 가상함수

- 함수의 **코드가 없고** 선언만 있는 **가상 멤버** 함수
- "파생 클래스에서 재정의하도록 원형만 미리 준비해두는 함수"
- 파생 클래스는 상속받은 순수 가상 함수를 **반드시 재정의**해야 한다.

❖ 순수 가상 함수 선언 방법

- 함수 선언의 끝 부분에 "= 0"을 적어줌

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

추상 클래스

❖ 추상 클래스(abstract class)

- 최소한 하나의 순수 가상 함수를 갖는 클래스

```
class Shape { // Shape은 추상 클래스
 protected :
     int _x;
     int _y;
 public:
     void paint();
     virtual void draw() const= 0; // 순수 가상 함수
};

void Shape::paint() {
   draw(); // 순수 가상 함수라도 자식 클래스를 통해 호출은 할 수 있다.
}
```

추상 클래스

❖추상 클래스 특징

- **온전한 클래스가 아니므로 추상 클래스는 객체를 생성할 수 없다.**

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

- **추상 클래스의 포인터 변수나 레퍼런스 변수는 정의할 수 있다.**
  - 파생 클래스 객체를 가리키는 용도로 사용
  - 파생 클래스 객체에 접근하는 인터페이스 역할을 제공

```
Shape *p;
```

추상 클래스의 목적

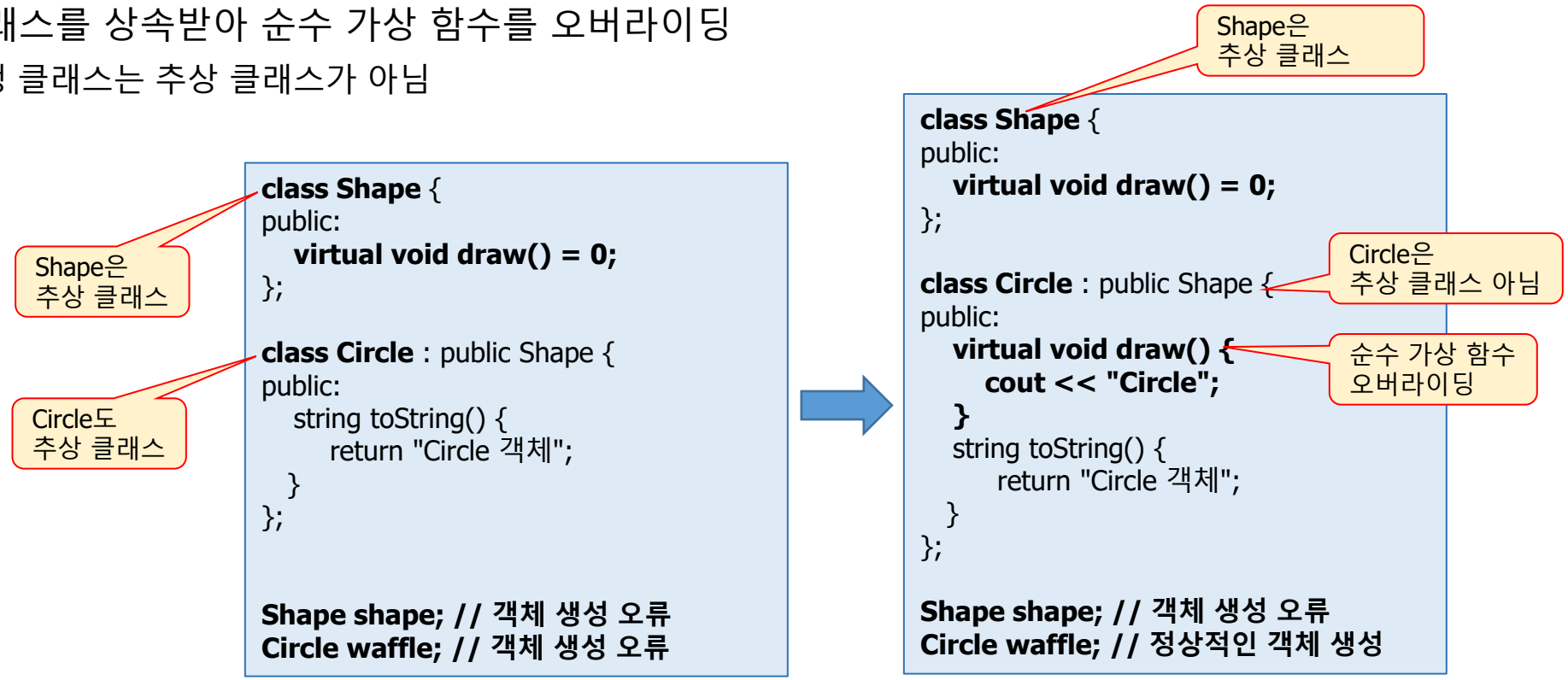
❖ 추상 클래스의 목적

- 추상 클래스의 인스턴스를 생성할 목적이 아님
- 상속에서 기본 클래스의 역할을 하기 위함
- **순수 가상 함수를 통해** 파생 클래스에서 **구현할 함수의 형태(원형)을 보여주는 인터페이스 역할**

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



추상 클래스의 단순 상속

추상 클래스의 구현

구현 상속과 인터페이스 상속

구분	기본 클래스의 함수	파생 클래스의 의무	특징
구현 상속	일반 함수 (non-virtual)	재정의 ❌ 선택도 ❌	함수 그대로 상속받아 사용함
디폴트 구현 상속	가상 함수 (virtual)	재정의 ❌ (선택 가능)	기본 동작을 상속, 필요 시 재정의 가능
인터페이스 상속	순수 가상 함수 (`= 0`)	재정의 **반드시 필요**	무조건 구현해야 함 (설계 강제)

다양한 종류의 멤버 함수

❖ 상속과 관련해서 지금까지 살펴본 멤버 함수의 종류
- 일반적인 멤버 함수
- 가상 함수
- 순수 가상 함수

❖ 어떤 종류의 멤버 함수를 사용할 지에 대한 가이드 라인
- 처음엔 그냥 멤버 함수로 만든다.
- 다형성을 이용해야 하는 경우라면 가상 함수로 만든다.
- 다형성을 위해서 함수의 원형만 필요한 경우라면 순수 가상 함수로 만든다.

오버로드(overroad) 된 멤버 함수의 오버라이드(override)

❖ 오버로드(overroad) 된 멤버 함수

- 기본 클래스에서 오버로드 된 함수 중에서 어느 것 하나라도 오버라이드(override, 재정의) 하면 같은 이름을 가진 나머지 다른 함수들도 모두 사용할 수 없다
- **Name hiding 문제**: 자식 클래스에서 부모 클래스의 함수를 오버라이딩하면, 부모 클래스의 해당 이름을 가진 모든 함수가 가려짐

```
class Pet {
public:
	void Eat();
	void Eat(const string& it);

	string name;
};

class Dog : public Pet {
public:
	void Eat();
};

void Dog::Eat() {
	cout << name << "says, 'Growl~'□n";
}
```

```
int main() {
	// 강아지 생성
	Dog dog1;
	dog1.name = "Patrasche";

	// 두 가지 Eat() 함수를 호출한다.
	dog1.Eat();
	dog1.Eat( "milk" );	// Error!!
	dog1.Pet::Eat( "milk" );	// success!!

	return 0;
}
```

다음 수업

❖ 템플릿과 STL
- 1_ 클래스 템플릿
- 2_ STL