객체프로그래밍 정리노트

202204103 ict융합공학부 이승훈
202204023 ict융합공학부 김보민

[가상 함수]
- virtual 키워드로 선언된 멤버 함수
ㄴ 동적 바인딩 지시어
ㄴ virtual 키워드는 컴파일러에게 함수에 대한 호출을 실행시간까지 미루도록 지시

[함수 오버라이딩]
-파생 클래스에서 기본 클래스의 가상 함수와 동일한 이름의 함수 선언
ㄴ 기본 클래스의 가상 함수의 존재감 상실
ㄴ 파생 클래스에서 오버라이딩한 함수가 호출되도록 동적 바인딩

[동적 바인딩]
ㄴ 파생 클래스에 대해 기본 클래스에 대한 포인터로 가상 함수를 호출하는 경우 객체 내에 오버라이딩한 파생 클래스의 함수를 찾아 실행
ㄴ 실행 중에 이루어짐

[추상 클래스]
ㄴ 기본 클래스의 가상 함수의 목적 :
파생클래스에서 재정의할 함수를 알려주는 역할 (코드 실행 목적 x)

[순수 가상 함수]
ㄴ 함수의 코드가 없고 선언만 있는 가상 멤버 함수
ㄴ 선언 방법: (멤버 함수의 원형) = 0;

[실행시킬 수 있는 메인함수 만들기]
(오버라이딩)

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

```
                cout << "Derived::f() called" << endl;
        }
};
```

[메인 함수]

```
//보민 , ->승훈
int main() {
        Derived d, * pDer;
        pDer = &d;
        pDer->f();        ->// Derived::f() 호출해서 pDer가 가리키는 객체의 f() 메서드를 호출하니까 Derived 클래스의 f()가 실행된다고 볼 수 있어
        Base* pBase;
        pBase = pDer; //이 코드로 업캐스팅해서 pDer 포인터 값을 pBase 포인터에 할당해줘야 되는 거까지 이해했어. 그럼 pBase는 Derived 객체를 가리키는게 되나 ? -> 이해하고 있는게 맞아
        pBase->f(); ->그럼 이 코드는 Base::f() 호출하게 되고 pBase가 가리키는 객체의 f() 메서드를 호출하므로 Base 클래스의 f()가 실행돼
}
}
```

[9-3 예제 풀이]

```
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();
}
```
-> 이 코드는 상속이 반복되는 것을 보여주는 다형성을 보여주는 예시야. 다형성은 하나의 인터페이스나 기본클래스가 주어지면 그에 따라 다양한 형태의 객체를 다룰 수 있다는 특징이 있어.

// 아 그래서 GrandDerived 객체를 가리키는 포인터들을 초기화 시켜주고 방금 배운 동적 바인딩에 의해 f() 함수를 호출한 거구나.

[예제 9-6]

```
#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;
        delete bp;
}
```
// 이 코드는 가상 소멸자를 이용한 예시야. 가상 소멸자는 기본 클래스에서 파생된 클래스의 객체를 올바르게 소멸시키기 위해 사용되지.

-> 그럼 여기서는 delete 연산자를 사용해서 객체 타입을 기반으로 소멸자를 호출했다고 볼 수 있겠네.

// 맞아 근데 만약에 여기서 Base 클래스의 소멸자가 가상 함수로 선언되어 있지 않았다면 bp로 소멸될 때 Derived의 소멸자가 호출되지 않았을 거야.

[22쪽 클래스를 구분해서 파일생성]

< Shape.h >

```
#ifndef SHAPE_H
#define SHAPE_H

class Shape {
```

```
public:
        virtual void paint() const = 0;
        virtual Shape* add(Shape* shape) = 0;
        virtual Shape* getNext() const = 0;
        virtual ~Shape() {}
};
#endif
```

< Circle.h >

```
#ifndef CIRCLE_H
#define CIRCLE_H
#include "Shape.h"

class Circle : public Shape {
public:
        virtual void paint() const;
        virtual Shape* add(Shape* shape);
        virtual Shape* getNext() const;
        virtual ~Circle();
private:
        Shape* next;
};
#endif
```

< Rect.h >

```
#ifndef RECT_H
#define RECT_H
#include "Shape.h"

class Rect : public Shape {
public:
        virtual void paint() const;
        virtual Shape* add(Shape* shape);
        virtual Shape* getNext() const;
        virtual ~Rect();
private:
        Shape* next;
};
#endif
```

< Line.h >

```
#ifndef LINE_H
#define LINE_H
#include "Shape.h"

class Line : public Shape {
public:
        virtual void paint() const;
        virtual Shape* add(Shape* shape);
        virtual Shape* getNext() const;
        virtual ~Line();
private:
        Shape* next;
};
#endif
```

< Shape.cpp >

```
#include "Shape.h"

Shape::~Shape() {
}

Shape* Shape::add(Shape* shape) {
        return nullptr;
}

Shape* Shape::getNext() const {

        return nullptr;
}
```

< Circle.cpp >

```
#include "Circle.h"
#include <iostream>

void Circle::paint() const {
        std::cout << "Drawing Circle" << std::endl;
```

```
}

Shape* Circle::add(Shape* shape) {
        next = shape;
        return shape;
}

Shape* Circle::getNext() const {
        return next;
}

Circle::~Circle() {
        delete next;
}
```

< Rect.cpp >

```
#include "Rect.h"
#include <iostream>

void Rect::paint() const {
        std::cout << "Drawing Rectangle" << std::endl;
}

Shape* Rect::add(Shape* shape) {
        next = shape;
        return shape;
}

Shape* Rect::getNext() const {
        return next;
}

Rect::~Rect() {
        delete next;
}
```

< Line.cpp >

```
#include "Line.h"
#include <iostream>
```

```
void Line::paint() const {
	std::cout << "Drawing Line" << std::endl;
}

Shape* Line::add(Shape* shape) {
	next = shape;
	return shape;
}

Shape* Line::getNext() const {
	return next;
}

Line::~Line() {
	delete next;
}
```

< main.cpp >

```
#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();
	pLast = pStart;
	pLast = pLast->add(new Rect());
	pLast = pLast->add(new Circle());
	pLast = pLast->add(new Line());
	pLast = pLast->add(new Rect());

	Shape* p = pStart;
	while (p != NULL) {
		p->paint();
		p = p->getNext();
```

```
        }

        p = pStart;
        while (p != NULL) {
                Shape* q = p->getNext();
                delete p;
                p = q;
        }

        return 0;
}
```

[9-7 예제]

```
#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;
public:
        void run() {
                input();
                cout << "계산된 값은 " << calc(a, b) << endl;
        }
};
int main() {
        Adder adder;
        Subtractor subtractor;
        adder.run();
        subtractor.run();
}
```

-> 먼저 이 문제를 풀기 전에 순수 가상 함수에 대해서 알아보자.
순수 가상함수는 함수의 코드가 없고 선언만 있는 가상 멤버 함수라고 볼 수 있어. 선언 방법은 멤버 함수의 원형=0;으로 선언해 우리는 순수 가상 함수를 사용하면 이는 자신의 인터페이스를 유지하면서도 기본 클래스에서 파생 클래스한테 특정 동작을 구현하게 할 수 있기에 우리는 이걸 사용해. 이 개념을 가지고 이 문제를 해결해보자.

// 이 코드를 실행 시키려면 추상 클래스 Calculator를 상속받는 클래스이고 순수 가상 함수 clac를 선언하고 있으니까 우리는 이 함수를 파생 클래스에서 구현해야 돼. Adder와 Subractor클래스가 Calculator 클래스를 상속하고 있으니까 기본 클래스의 멤버 변수랑 함수에 접근이 가능해 그럼 우리는 이제 각각의 클래스에서 clac를 구현할 수 있을 것 같아.
// 심지어 Claculator 클래스에서 선언된 a,b 는 protected로 선언되어 있으니까 이 변수들에 Adder랑 Substractor 클래스에서 직접적으로 사용할 수 있겠다. 그럼 코드는 이런 식으로 나올거야.

```
class Adder : public Calculator {
protected:
	int calc(int a, int b) {
		return a + b;
	}
};
class Subtractor : public Calculator {
protected:
	int calc(int a, int b) {
		return a - b;
	}
};
```