객체지향프로그래밍 정리노트(Week6)

202004029 김정호, 202004040 노성민

Q1. 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(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; }
};
```

- 오버라이딩(overriding) -

김정호: 위의 함수 재정의 예시는 만약 Derived a; 선언하여 a 객체를 생성했을 때, a 객체에게 동등한 호출 기회를 가진 함수 f()가 두 개 존재하고, 오버라이딩 사례를 보면 마찬가지로 함수 f()가 두 개 존재하지만 virtual을 사용한 함수는 오버라이딩되어 기본 클래스 f() 함수가 존재감을 잃고 Derived의 f()가 호출돼.

Q2. 9-3 상속이 반복되는 경우 가상 함수 호출

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

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();
}
```

노성민: 우선 코드를 보면 f() 함수가 오버라이딩 된 것을 알 수 있어. 메인 함수를 보면 GrandDerived 클래스의 g 객체가 생성되었고 각 클래스의 포인터를 선언하여 g의 주소를 가리키고 있어. 따라서 결과가 모두 GrandDrived::f() called가 출력될 것이야.

Q3. 예제 9-6 소멸자를 가상 함수로 선언

```
#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
}
```

김정호: 소멸자를 가상 함수로 선언하였는데, 우선 delete dp; 이 부분에서는 가상 함수로 선언하지 않았어도 소멸자 실행 시 두 개의 소멸자가 실행될 거야. 하지만 delete bp; 이 부분은 소멸자가 가상 함수로 선언되었기 때문에 파생 클래스가 자신의 클래스를 실행 후 기본 클래스의 소멸자를 호출하도록 컴파일하기 때문에 여기서도 마찬가지로 두 개의 소멸자가 실행돼. 따라서 결과는

~Derived()
~Base()
~Derived()
~Base() 이런 식으로 출력될 거야.

Q4. 동적 바인딩 실행 : 파생 클래스의 가상 함수실행

```
#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
	}
}
```

김정호: 우선 pStart, pLast 포인터 변수를 선언하고 초기화 해준 후 처음에는 Circle을 이용하여 객체를 동적으로 생성했어. 다음에 이제 해당 객체들을 생성해주고 파란색 부분을 통해 생성한 객체의 도형을 그리고 빨간색 부분을 통해 생성된 도형을 삭제해주는 코드야. 여기서 가상 함수를 오버라이딩하여 어떤 경우에도 자신의 draw 함수가 실행될 수 있도록 했어.

Q5. 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();
}
```

```
정수 2 개를 입력하세요>> 5 3
계산된 값은 8
정수 2 개를 입력하세요>> 5 3
계산된 값은 2
```

노성민: 우선 Adder, Subtractor 클래스는 Calculator를 상속받아야만 결과처럼 출력이 가능해져. 그리고 Calculator가 추상 클래스이므로

```
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;
	}
};
```

이렇게 순수 가상 함수로 작성할 수 있어.