Ⅰ. 객체 지향 프로그래밍 기초 지식

**프로그래밍 패러다임**은 프로그래밍을 어떻게 할 것인지에 대한 방법론을 의미합니다. **객체 지향 프로그래밍(OOP, Objected Oriented Programming)**은 컴퓨터 중심의 사고에서 벗어나 **문제 해결을 중점**으로 생각하는 프로그래밍 패러다임입니다.

여기서 말하는 문제 해결이란 큰 문제를 작은 문제들로 나누는 **모듈화** 작업을 말합니다. OOP에서는 이렇게 나눠진 작은 문제를 **객체**라고 표현합니다. 이 과정은 **Bottom-up** 방식이라고도 하며 때론 지금까지 해왔던 Top-down 방식으로 알려진 구조적 프로그래밍과 대조되어 많이 소개되곤 합니다.

모듈화를 이용한 장점으로는 분할된 부품을 이용하기 때문에 **재사용성**이 높아져서 코드의 중복을 줄일 수 있습니다. 중복이 줄어 수정 또한 쉬워져 유지보수가 쉬워지고 코드가 간결해집니다. 또한 **확장성**이 보장되고 **안정성**이 높아집니다. 다수가 작업을 할 때 분업이 편하다는 특징도 있습니다. 이런 특징의 결과로 개발 시간과 비용을 절감할 수 있습니다.

딱 봐도 객체 지향 프로그래밍 방식을 쓰면 많은 장점을 얻는다는 걸 알 수 있죠? C++에서 객체지향을 돕기 위해 새로 등장한 문법이 바로 **클래스**인데, 이 때문에 정말 다양한 문법들이 추가로 나오게 됩니다.

-OOP의 4대 특징

**추상화**는 객체의 도면, 즉 청사진을 말합니다. 하나의 기능을 위해 만들어진 객체들을 만들기 위한 설계도가 필요한데 이는 **클래스**로 구현할 수 있습니다.

**캡슐화**는 문제들은 하나로 묶음(캡슐화)으로써 외부로 노출할 필요가 없는 데이터는 **은닉**하게 되고 필요로 하는 데이터만 노출하게 됩니다. 또한 기능을 명확히 파악할 수 있다는 장점이 있습니다. 문법으로 예시를 들자면 **접근지정자**가 있습니다.

**상속**은 캡슐화된 기능의 일부분을 다른 캡슐에 해당 기능을 물려주는 것입니다. 이는 코드 재사용성을 높이는데 기여를 합니다. 문법으로는 **부모와 자식 클래스**가 있습니다.

**다형성**은 같은 요청을 해도 상황에 따라 다른 결과를 내는 **유동성**을 의미합니다. 함수의 **오버로딩**과 **오버라이딩**을 예시로 들 수 있습니다.

Ⅱ. 객체지향 문법

1. **디폴트(default) 인수**는 객체지향의 특징인 캡슐화의 예시 중 하나입니다. 주의할 점은 디폴트 인수를 가진 매개변수는 가장 뒤에 위치해야 합니다.

2. **함수의 오버로딩(overloading)**은 다형성의 예시 중 하나입니다. 함수의 이름을 같게 한다는 것은 혼란으로 인해 위험할 수도 있기 때문에 오버로딩은 몇 가지 주의사항이 있습니다.

- 리턴 타입만 다른 경우는 오버로딩하지 않습니다. (형변환이 있기 때문)

- 디폴트 인수에 의해 같아지는 함수는 오버로딩하지 않습니다.

- const 지정자는 오버로딩해도 괜찮습니다. (자주 쓰이는 경우는 아닙니다)

3. **클래스(class)**는 C++의 핵심 문법 중 하나입니다. 클래스는 ‘사용자 지정 ADT’를 손쉽게 쓰기 위해서 만든 문법이라고 봐도 무관합니다. C언어에서의 구조체와는 달리 변수뿐만 아니라 함수도 가질 수 있는 것이 특징이며 상속이나 접근지정자라는 개념이 있는 게 특징입니다.

함수 커리큘럼에서 잠시 언급했던 분업화는 이제 클래스 단위로 생각해볼 수 있습니다. 어떤 프로그램의 테스트는 함수로 구성했다면 유지보수가 쉽다고 했었죠. C++에서는 클래스 단위로 프로그램을 평가할 수 있기 때문에 유지보수가 더욱 쉬워집니다. 또한 상속이라는 개념으로 코드의 재사용성이 더욱 높아졌습니다. 기능의 확장을 손쉽게 할 수 있는 것이죠.

클래스 단위 평가로 인한 유지보수 용이성과 상속으로 인한 코드 재사용성 증가, 이들이 바로 모듈화의 핵심적인 장점입니다.

4. **객체(object)**는 클래스를 실제로 구현한 것입니다. 인스턴스라고도 하며 클래스는 1개이지만 객체는 여러 개일 수 있습니다.

5. **내부 포인터 this**는 클래스 내부에서 쓰이는 문법입니다. 해당 문법을 쓰는 곳의 클래스 자기 자신을 가리키게 됩니다.

6. **생성자(constructor)**는 객체를 생성할 때 해당 객체의 값들을 초기화할 수 있는 함수입니다. 프로그래머가 명시하지 않아도 new 시 저절로 생성되어 호출됩니다.

7. **소멸자(Destructor)**는 객체가 파괴될 때 호출되는 함수입니다. 생성자와 마찬가지로 프로그래머가 명시하지 않아도 delete 시 저절로 생성되어 호출됩니다. 생성자와 소멸자는 오버로딩하여 사용할 수 있습니다.

8. **Getter, Setter 함수**는접근지정자와 캡슐화 때문에 나온 개념입니다. 만약 클래스에 private int age; 라는 변수가 있으면 int GetAge() {return age;} 가 getter함수, void SetAge(int age) {this->age = age;} 가 setter함수입니다. 즉, 따로 문법이 존재하는 것은 아니고 private 변수를 외부에서 다룰 수 있도록 한 함수들을 가리키는 함수들입니다.

그냥 public으로 선언하면 안 되는 걸까요? 사실 그렇게 해도 무관합니다. 특히 위처럼 아무런 의미없이 getter, setter 함수가 존재한다면 public으로 선언해도 됩니다. 하지만 OOP의 원칙 중 하나인 “노출시킬 필요 없는 데이터는 감춰라”라는 것에 따르면 모든 변수는 관용적으로 private을 사용하며 객체는 변수에 직접 접근하지 못하고 무조건 함수만으로 접근해야 합니다. 이는 **디미터의 법칙(Law of Demeter)**으로도 잘 알려져 있습니다.

그리고 실제로 getter, setter를 만드는 이유는 위처럼 의미없는 주고받음이 아니고 객체 내부의 값의 안전성을 보장하는 것입니다. 예를 들어, 위의 setter는 void SetAge(int age) { if (age > 0 && age < 100) this->age = age; } 로 바꿀 수 있습니다. 이와 같이 setter함수를 바꾼다면 age가 가질 수 있는 값에는 범위가 생기겠죠. 이를 변수값의 안전성을 보장한다고 합니다.

9. **상속(Inheritance)**은 OOP의 핵심 개념 중 하나입니다. 하지만 코드를 재사용할 수 있다는 장점과 C++에서는 여러 개의 클래스에서 상속받을 수 있다는 점 때문에 무분별하게 상속을 받을 수도 있기에 상속도 몇 가지 주의점이 있습니다.

- 의미적으로 상속관계인 것만 상속합니다. (동물 - 기린)

- 의미적으로 범주가 아닌 속성은 인터페이스 상속합니다. (네발동물 - 기린)

만약, 무분별하게 상속을 하다가 A가 B를 상속하고 B가 A를 상속하는 상황이 발생하면 이를 **Cyclic Reference** 라고 합니다. A를 생성하려면 B를 생성해야 하는데 B를 생성하려면 A를 생성해야 되고… 이것이 반복되어 프로그램은 순환상태로 들어가 정지하거나, 비효율적인 상태가 만들어지게 됩니다. 이런 실수를 할까? 싶지만 상속관계가 복잡해지면 발생할 수도 있는 실수입니다. 이 때문에 최신 프로그래밍 언어들의 상속은 단 1개만 가능한 경우가 많습니다.

10. **함수의 오버라이딩(Overriding)**은 오버로딩과 이름이 유사하기 때문에 주의해서 봐야 합니다. 오버라이딩은 상속과 관련된 개념이고 오버로딩과 달리 함수명, 변수의 개수, 반환형, 등 모든 게 원래의 함수와 똑같습니다. 다만, 내부 구현은 자식 클래스에 맞게 변경되겠죠.

11. **추상클래스(abstract class)**는 상속의 개념이 나오면서 같이 나온 개념입니다. 어느정도는 구현이 되어 있지만 100% 구현은 되어있지 않기 때문에 객체를 만들 수 없는 것이 특징입니다. 이런 일이 가능한 이유는 상속된 클래스에서 정의해야 하는 완전가상함수(pure virtual function)라는 문법이 있기 때문입니다.

12. **인터페이스(Interface)**도 상속의 개념이 나오면서 같이 나온 개념입니다. 사실 C++에는 인터페이스 문법이 별도로 존재하지 않습니다. 자바나 C#과 달리 다중상속을 허용하기 때문에 인터페이스는 평범한 class와 내부적으로 동일합니다. 추상클래스와 유사하여 헷갈릴 수도 있는데 추상클래스와 달리 인터페이스는 정말 틀만 제공되어 있고 구현은 0%입니다.

얼핏 보면 도대체 추상클래스와 인터페이스는 왜 쓰는 걸까 싶겠지만, 설계를 하다 보면 필요할 때도 있으며 더 구조적이고 모듈화된 설계를 하기 위해서 쓰이기도 합니다. 특히, 디자인 패턴(Design Pattern)에서 자주 쓰입니다. 디자인 패턴은 간단하게 설계의 방법론이라고 생각하시면 됩니다.

13. **템플릿(Template)**은 다형성의 예시 중 하나입니다. 어떨 때는 자료형만 다르고 같은 일을 하는 함수가 있을 때가 있습니다. 이럴 때마다 오버로딩하여 새로운 함수를 만드는 것은 굉장히 귀찮은 작업이겠죠. 따라서 C++은 템플릿이라는 것을 제공하기로 합니다. 마치 C언어의 void 포인터를 더욱 범용적이게 만든 것입니다. STL에서 사용한 것도 이 템플릿이고 자바와 C#에서는 C++의 템플릿과 유사한 기능을 제네릭(Generic)이라고 부릅니다.

Ⅲ. 추가 개념 – 프로그램의 확장성 판단

클래스의 상속이 얽히고 얽히면 상속이 많이 된 클래스를 무겁다, 또는 복잡하다고 합니다. 이와 관련되어 나온 개념이 2가지 있습니다. **응집도**(Cohesion)는 클래스 내부에서 내부 요소들이 긴밀히 상호관계를 맺고 있는 것을 의미합니다. 응집도가 높다면 외부에서 어떤 변경이 일어나도 해당 클래스가 받는 영향은 미미하여 수정이 많이 필요하지 않을 것입니다.

**결합도**(Coupling)는 클래스가 다른 클래스들과 얼마나 많은 의존을 하고 있는가를 나타냅니다. 만약 결합도가 높은 클래스는 한 클래스에서 수정을 하면 다른 클래스에서 대거 수정을 해야 할 수도 있습니다. 따라서 클래스는 응집도가 높고, 결합도가 낮게 만드는 것이 중요하고 이를 좋은 설계라고 합니다. 위와 같은 좋은 설계를 위해서 디자인 패턴을 쓰게 됩니다.

Ⅳ. 개념 확인

**-OOP 개념**

1. OOP가 프로그래밍 패러다임 중 하나란 것을 파악

2. 구조적 프로그래밍보다 좋은 점 파악

3. 모듈화가 무엇이고 장점이 무엇인지 파악

4. OOP의 4대 원칙이 무엇이고 그와 관련된 문법이 무엇인지 파악

**-소개된 문법 외의 개념**

1. 연산자도 오버로딩할 수 있음을 파악

2. 프렌드 함수 파악 후 OOP적 문제점 파악 (JAVA와 C#엔 존재하지 않음)

[문제]

시작은 2022.00.00() 입니다. 마감 기한은 **2022.00.00() 23:59** 입니다. 특별한 사정없이 마감 기한까지 제출을 못하면 경고를 부여받으므로 주의해주세요.

\*문제는 별도의 파일로 드리겠습니다.