Object Oriented Programming

프로그래밍 입문(2)

Topics

- Midterm Exam Review
- **OOP Basics**
- Encapsulation

Object Oriented Programming (OOP)

- 객체지향 프로그래밍(OOP)이란 모든 것을 프로그램 내에서 객체(Object)로 표현하는 프로그래밍 패러다임.

- 프로그래밍 패러다임은 어떤 프로그래밍 언어가 고려하는 원칙들이나 전략들을 의미함.

- 객체는 멤버 변수(member variable)와 멤버 함수(member function)을 가질 수 있음.

    - 멤버 변수는 객체의 데이터 또는 특성을 나타내고, 멤버 함수는 객체의 연산 또는 동작(operation)을 나타낸다고 이해하면 좋음.

Why OOP?

- OOP는 데이터 추상화(Data Abstraction)을 구현하기 위한 전략.
- 단순한 수학적 계산 등을 대신하던 프로그램의 활용범위가 넓어짐.
    - 기본적인 데이터형들은 현실의 다양한 데이터를 표현하기에 부족한 점이 많아졌음.
- 프로그램 내에서 복잡한 데이터들을 효율적으로 다루면서도 개발하기에 적합한 방법이 필요하였음.

객체(Object)

- 객체 = 만물 + 사상
- 단순히 물리적인 실체 뿐만 아니라 여러가지 실체가 없는 개념들 또한 객체로 표현할 수 있음.
    - 물리적 실체가 있는 것: Cat, Car, VendingMachine
    - 개념들: SortingAlgorithm, DisplayStrategy

Student Class

- 학생의 정보를 나타내기 위해 학번, 이름, 학점의 멤버 변수를 가진 클래스를 정의.

- 멤버 함수로 이런 데이터를 적절하게 다룰 수 있는 방법을 제공.

- 데이터 + 연산을 하나로 묶어 새로운 데이터형인 Student형을 새로 정의하게 됨.

Student.hh

```
#ifndef STUDENT_H_
#define STUDENT_H_
#include <string>
#include <vector>

class Student {
    private:
        int studentId;
        std::string name;
        std::vector<double> grades;
    public:
        Student(int, std::string);
        double getGPA();
        void addGrade(double point);
        void addGrade(char grade);
};
#endif
```

std::vector

- 가변길이의 배열처럼 사용할 수 있는 유용한 자료구조.
- #include <vector>
- std::vector<사용할 데이터형> vec;
- std::vector<사용할 데이터형> vec(벡터 크기);
- vec.push_back(넣고 싶은 값)으로 원소를 제일 뒤에 추가.
- vec.size()로 현재 들어있는 원소의 개수를 알 수 있음.

Student Class

- 선언과 구현의 분리
  - 클래스 선언은 <클래스이름>.hh 헤더 파일에 저장.
  - 실제 구현은 <클래스이름>.cpp 파일에 저장.
  - 다른 코드에서 이 클래스를 사용하고 싶으면 헤더파일(.hh)만 #include 하여 쓰면 됨.

Student.cpp

```cpp
#include "Student.hh"

Student::Student(int studentId,
    std::string name) {
    this->studentId = studentId;
    this->name = name;
}

void Student::addGrade(char grade) {
    this->grades.push_back(69-grade);
}

void Student::addGrade(double point) {
    this->grades.push_back(point);
}
```

Student Class

- 사용자는 어떻게 학점이 추가되고, 평점이 계산되는 지 자세히 알 필요가 없음.

- 단지 평점을 얻고 싶으면 getGPA()를 호출한다는 것만 알면 됨.
  - 마치 우리가 두 숫자를 더하고 싶으면 +를 쓰면 되는 것을 아는 것처럼.
  - 내부적으로 +가 어떻게 구현되는지 몰라도 사용에 문제 없음.

- 복잡한 데이터형을 추상화하여 단순하게 다룰 수 있게 됨 → 복잡한 프로그램 설계가 훨씬 쉬워짐.

Student.cpp

```
void Student::addGrade(char grade) {
    this->grades.push_back(69-grade);
}

void Student::addGrade(double point) {
    this->grades.push_back(point);
}

double Student::getGPA() {
    double sum = 0;
    int size = this->grades.size();
    for(int i=0; i<size; i++) {
        sum += this->grades[i];
    }
    return size > 0 ? sum/size : 0.0;
}
```

OOP방식으로 코드 작성

- 클래스를 정의하는 것은 복잡한 데이터를 하나의 개념으로 묶는 새로운 데이터형을 추가하는 것.

- 데이터형은 그것이 표현하는 값의 종류와 이 값에 대한 연산으로 정의될 수 있음.

  - 멤버 변수 + 멤버 함수

- 어떤 값들이 객체를 나타내기 위해 필요한 지 생각 → 멤버 변수 작성.

- 이 값들을 초기화, 수정, 사용하기 위해 어떤 동작이 필요한지 생각 → 멤버 함수 작성.

클래스 = 멤버 변수 + 멤버 함수?

- 항상 이 등식이 성립하는 것은 아님.
- 단순히 여러 데이터형의 변수들을 하나로 묶어 처리하고 싶은 경우.
  - 멤버 함수 없이 멤버 변수들만 나열한 클래스.
  - e.g.) 함수의 반환형으로 사용 → 실제 연산은 각각의 멤버 변수를 가지고 수행.

클래스 = 멤버 변수 + 멤버 함수?

- 유사하지만 내부적으로 구현이 조금씩 다른 다수의 함수들을 묶어 제공.

- 멤버 변수 없이 함수들만 나열한 클래스.

- e.g.) SortingAlgorithm → 단순히 배열을 받아 정렬해주는 함수들만을 제공.

- 배열은 인자로 제공되므로 멤버 변수로 가지고 있을 필요가 없음.

OOP의 핵심 개념들

- 캡슐화 (Encapsulation)
- 다형성 (Polymorphism)
- 상속 (Inheritance)

캡슐화

- Information Hiding을 통해 추상화를 구현하는 핵심.
- 사용자는 내부적으로 어떤 정보가 어떻게 다루어지는지 몰라도 클래스 사용 가능.
- 데이터를 원하는 방식, 또는 정의된 방식으로만 다루도록 강제함.
  - 복잡한 프로그램에서 예상치 못한 오류를 줄일 수 있음.
  - 데이터의 가공을 통제하여 프로그램 설계의 단순화.
- private, public 접근 지시자를 이용하여 외부에서 접근이 가능한 부분과 접근이 불가능한 부분을 분리.

다형성

- 우리가 언어를 사용할 때는 동일한 단어가 문맥에 따라 다른 의미를 갖게 되는 경우가 있음.
- e.g.) 먹다: "나 떡볶이 먹었어!^_^", "나 커피 먹었어!>_<", "나 나이 먹었어..T_T"
    - 실제로는 음식을 먹고, 음료를 마시고, 나이가 증가한 것을 의미.
    - 모두 "먹었다"는 단어로 표현됨.

다형성

- 이처럼 동일한 형태의 코드를 사용했더라도 실제 실행되는 상황에 따라 다른 동작이 필요할 수 있음.
    - 이미 함수 오버로딩(Overloading)으로 일부 경험해 보았음.
        - 상속이라는 개념을 배우고 나면 오버라이딩(Overriding)을 배우게 됨.
- 연관된 데이터형에 따라 같은 이름의 다른 동작을 하는 함수가 수행되기도 하고,
- 다른 데이터형의 값들이 동일한 함수에 의해 처리되기도 함.

상속

- 실제 세계에서의 복잡한 데이터형들은 서로 깊은 연관성을 지닌 것들이 많음.
- 이들을 계층구조를 이용하여 하나로 묶는다면?
  - 부모-자식 관계를 이용하여 부모의 특성을 자식이 물려받을 수 있게 됨.
  - 공통된 특성을 하나의 부모 클래스에 몰아넣은 상태에서 자식 클래스들에는 각자의 특성을 살린 부분을 구현할 수 있음.
- 코드의 재사용성이 크게 증가.
- 계층 구조를 통해 복잡한 프로그램의 유지보수가 쉬워짐.

Summary

- OOP는 데이터 추상화를 구현하는 프로그래밍 패러다임.

- 클래스는 새로운 데이터형을 정의하게 되며, 값을 표현하기 위한 멤버 변수와 이 값에 대한 연산인 멤버 함수로 구성.

- 주요 개념으로 캡슐화, 다형성, 상속이 있음.