Object Oriented Programming

프로그래밍 입문(2)

Topics

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

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.

캡슐화의 핵심

- **"클래스 외부에서는 오로지 내가 정해놓은 방식대로만 클래스를 사용할 수 있다."**
    - 내부의 데이터도 내가 원하는대로만 보여줌.
    - 데이터를 변경하는 것도 내가 원하는대로만 변경할 수 있음.
- 단단한 캡슐에 싸인 것처럼 외부에서는 내부를 볼 수 없음.
- 하지만 방법만 알면 사용은 가능함.

캡슐화의 핵심

- 내부에서 어떻게 돌아가는지는 모르지만, 사용은 할 수 있음 → 이런 것을 **블랙박스**라고 합니다.

- 구체적인 내용을 신경쓰지 않도록 하여 추상화를 구현.

- 외부에서 내가 정한 방식으로만 사용할 수 있음.

    - 내부 데이터의 직접적인 접근을 막으므로, 데이터 변경을 통제할 수 있게 됨.

    - 데이터의 변경 방식을 통제할 수 있으므로, 그 외 다른 경우의 수까지 고려할 필요가 없음.

- 클래스 내부 구현을 변경할 때, 사용방법(인터페이스)만 유지하면 외부의 코드와 독립적으로 수정이 가능함.

캡슐화의 핵심

- Student 클래스에서 학번과 이름을 무조건 입력받도록 생성자를 구현.

- 다른 동작 구현시 학번과 이름이 없는 경우를 일일이 확인할 필요가 없음.

- 평점을 어떤 방식으로 구하든, double형으로 반환되는 이상 외부의 코드는 계속해서 getGPA()를 사용할 수 있음.

```
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
```

For Encapsulation

- 캡슐화의 기본은 다음과 같습니다.
  - 외부에 노출되면 안 되는 것들은 모두 private에 넣는다.
  - 외부와 소통하기 위한 부분들은 모두 public에 넣는다.
- 물론 캡슐화의 장점을 제대로 얻기 위해서는 고려해야 할 점이 더욱 많습니다.

private에 들어 갈 것

- 모든 멤버 변수들.
    - 외부에 노출되지 않도록 보호하기 위함.
- 클래스 내부적으로만 쓰이는 멤버 함수들.
    - 주로 다른 멤버 함수들에서 호출되는 용도.
- 클래스 내부에서 쓰기 위해 정의한 상수들.

public에 들어 갈 것

- Getter와 Setter.
    - Getter는 멤버 변수의 값을 반환해주는 멤버 함수.
    - Setter는 멤버 변수의 값을 변경해주는 멤버 함수.
- 외부에서 클래스를 사용할 수 있도록 제공되는 멤버 함수들.
- 외부에서도 접근해야하는 상수들.

OnlineStore Class

- private에 있는 것은 내부에서만 접근 가능.
- Product 상품들을 담고 있는 products.
- DeliverService 클래스는 배송관련 내용을 처리하는 함수를 가짐.
- OnlineStore의 각종 상태를 나타낼 수 있는 멤버 변수들.
- 상품을 제거하기 위한 멤버 함수.

```
#include <iostream>
#include "Product.hh"
#include "DeliveryService.hh"
```

```
class OnlineStore{
    private:
        Product* products;
        DeliveryService* service;
        long deposit;   //상품 판매로 번 돈.
        int stockSize;  //상품 저장공간 크기.
        int numOfProducts;  //현재 들어있는 상품 수.
        void removeProduct(int pNum); //상품을 제거.
```

OnlineStore Class

```
public:
    OnlineStore(const OnlineStore& store);
    OnlineStore(int stockSize, double freeWeight=0.0, double extraWeight=0.0);
    bool addProduct(const Product& p);  //상품을 추가.
    void displayProduct();  //상품 정보 표시.
    double purchase(int pNum, double distance); //상품 구매 처리.
    long getDeposit();
    int getSize();
    int getNumOfProducts();
    ~OnlineStore();
```

- public으로 외부에서 접근 가능한 것들.
- 생성자, 소멸자 및 getter.
- Product를 추가할 수 있는 함수들.
- purchase()로 상품을 구매하는 동작을 구현.

static 멤버 변수, 함수

- 클래스를 정의하고, 클래스의 변수를 선언하면 객체가 생성됨.
- 멤버 변수의 값들은 각각의 객체별로 다 다름.
- 만약 클래스 자체에 대한 변수, 함수를 사용하고 싶다면?
- 전역 변수처럼 동일 클래스의 모든 객체가 공유하는 클래스에 유일한 멤버를 선언.
    - static 키워드를 사용.

static 멤버 변수, 함수

```
class DeliveryService {
    private:
        static constexpr double UNIT_PRICE = 100.0;
        double freeWeight;
        double extraWeight;
```

- 단가는 모든 DeliveryService에서 동일하게 적용하고 싶고, 변경하고 싶지 않음.

- static 키워드를 추가하고, 상수임을 표시하기 위해 constexpr을 추가.

    - const가 아님에 주의. static 멤버의 경우 constexpr을 사용(C++11부터 적용).

- 객체 외부에서 사용시에는 DeliveryService::UNIT_PRICE처럼 클래스 이름 namespace를 사용하여 접근 가능 - 만약 public이었다면.

static 멤버 변수, 함수

- static 멤버 변수 또는 함수가 유용한 경우.
- 특정 클래스의 모든 객체에 대한 정보를 모으고 싶을 때.
- e.g.) DeliveryService에서 총 배달 건수를 기록하고 싶다면,
    - 일반 멤버 변수로 numOfDelivery 추가하고, 객체에서 배달 건수마다 이 변수를 증가.
    - 객체가 모두 100개라면? - 100개의 객체별로 배달 건수를 읽어 합을 구해야 함.
    - 이를 static 멤버 변수로 선언하면, 모든 객체가 클래스에서 유일한 numOfDelivery 변수에 같이 접근할 수 있음.
- static으로 상수, 함수를 정의하면 객체를 생성하지 않고도 상수, 함수를 사용 가능.

private, public으로 캡슐화 완료?

- 핵심은 내부의 데이터를 외부에서 예상치 못하게, 정하지 않은 방식으로 건드리지 못하게 하는 것.
- 단순히 private에 넣는다고 이 문제가 완벽하게 해결되지 않음.
  - 포인터, 참조 문제.
  - 복사 생성자.
  - 객체 대입 연산자.

포인터, 참조 문제

- C++에서는 포인터와 참조를 사용할 수 있음.

- 만약 public에서 private에 있는 데이터에 대한 포인터나 참조를 반환하게 된다면?

- 또는 매개변수로 포인터를 받아 멤버 변수에 저장한다면?

- 메모리에 직접 접근이 가능해지므로, 클래스 정의와 상관없이 내부데이터를 마음대로 바꾸는 것이 가능.

- 이런 경우를 방지하도록 주의하여 public 부분의 코드를 작성해야 함.

복사 생성자

- 클래스이름(const 클래스& 변수명);
- 복사 생성자가 호출 시기.
  - 다른 객체로 초기화될 때.
    - Call by Value로 객체 매개변수 사용.
    - 함수 반환시 객체를 그대로 반환.
- 제시하지 않아도 기본적으로 멤버변수들을 그대로 복사하게 됨.

```
OnlineStore store(2);
OnlineStore secondStore = store;
Product p("P0", 100, 0.5);
store.addProduct(p);
secondStore.addProduct(p);
secondStore.displayProduct();
```

```
private:
    Product* products;
    DeliveryService* service;
    long deposit;    //상품 판매로 번 돈.
    int stockSize;   //상품 저장공간 크기.
    int numOfProducts;  //현재 들어있는 상품 수.
    void removeProduct(int pNum); //상품을 제거.
```

```
OnlineStore(const OnlineStore& store);
```

복사 생성자

- 제시하지 않아도 기본적으로 멤버변수들을 그대로 복사하게 됨.
  - 포인터의 메모리 값도 그대로 복사한다는 의미.
- 이 경우 secondStore의 products는 store와 동일.
- 예제 코드에서 store, secondStore 모두 2개의 상품 정보를 갖게됨.
- 소멸자에서 delete[] 호출시 문제 발생.

```
OnlineStore store(2);
OnlineStore secondStore = store;
Product p("P0", 100, 0.5);
store.addProduct(p);
secondStore.addProduct(p);
secondStore.displayProduct();
```

```
private:
    Product* products;
    DeliveryService* service;
    long deposit;   //상품 판매로 번 돈.
    int stockSize;  //상품 저장공간 크기.
    int numOfProducts;  //현재 들어있는 상품 수.
    void removeProduct(int pNum); //상품을 제거.
```

```
OnlineStore(const OnlineStore& store);
```

복사 생성자

```
OnlineStore::OnlineStore (const OnlineStore& store) {
    this->products = new Product[store.stockSize];
    for(int i=0; i<store.stockSize; i++)
        this->products[i] = store.products[i];
    this->stockSize = store.stockSize;
    this->service = store.service;
    this->numOfProducts = 0;
    this->deposit = 0;
}
```

- 이런 문제를 해결할 수 있도록 복사 생성자를 구현해야 함.
- 생성자 내에서 매개변수의 private 멤버 접근 가능.
- 데이터 형태를 주의하여 하나씩 복사 → **깊은 복사(Deep Copy)**.
- 반대로 포인터, 참조 등을 그대로 복사하는 것을 **얕은 복사(Shallow Copy)**라고 함.

객체 대입 연산자

- products는 Product형의 배열임.
- 각각의 원소는 Product 객체 → 객체에 다른 객체를 대입하고 있음.
- 초기화와 마찬가지로 얕은 복사가 일어나게 됨.
- 깊은 복사가 되도록 두 가지 방법을 사용.
  - 대입 연산자 오버로딩.
  - 복사를 위한 함수 사용.

```
for(int i=0; i<store.stockSize; i++)
    this->products[i] = store.products[i];
```

```
Product& Product::operator=(const Product& p) {
    this->name = p.name;
    this->price = p.price;
    this->weight = p.weight;
    return *this;
}
```

Summary

- 캡슐화의 핵심 개념
- 캡슐화를 구현하기 위한 방식.
- 캡슐화 달성을 위해 주의해야 할 점들.
  - 포인터, 참조.
  - 복사 생성자와 객체 대입 연산자 문제.