CSED232: Object-Oriented Programming Midterm Exam

권민재, 20190084

May 7, 2020

1. 다음은 객체지향 프로그래밍의 중요한 개념들이다. 각각의 개념들이 무엇을 뜻하는지 자세히 기술하시오.

   a. Encapsulation

   Encapsulation은 기본적으로 데이터와, 그 데이터에 연관된 메서드들을 하나로 묶는 것을 의미한다. 그렇게 묶은 데이터와 메소드들의 접근을 제한함으로써 데이터와 상관 없는 외부 코드가 데이터를 함부로 수정하는 것을 막아서 데이터의 안정성을 확보하고, 데이터와 유관한 메서드들을 한 곳에 묶어둠으로써 코드의 가독성을 늘린다.

   b. Information hiding (or data hiding)

   Information hiding이란, 객체의 구체적인 정보를 노출하지 않는 것을 일컫는다. Information hiding은 Encapsulation을 위한 한 가지 방법으로도 생각할 수도 있다. 왜냐하면 객체가 서로에 대한 정보가 없는 상태를 만듬으로써 각 객체에 대한 독립적인 코드를 작성할 수 있기 때문이다. 물론, information hiding이 객체 간의 그 어떤 정보 교환도 금지하겠다는 규칙은 아니다. 이를 모두 금지시키면 객체들이 모여서 프로그램을 구성하고 실행시킨다는 객체지향 프로그래밍의 근본적인 개념을 해칠 수 있기 때문이다. 그렇기 때문에 information hiding은, 정보를 무조건적으로 숨기는 것이 아닌, 메소드의 노출을 최소화하는 등의 방법으로 정보 노출을 최소화한다는 개념이라는 것을 알 수 있다. 즉, 객체 간의 정보 노출을 최소화 시킴으로서 객체 간의 독립성을 확보해서 기능을 자유롭게 추가, 수정, 삭제할 수 있도록 코드의 유연성을 향상시키는 것이 information hiding의 목적이라고 말할 수 있다.

2. 아래와 같은 코드가 있다. MyNameSpace 안의 func2 에서 MyNameSpace 밖의 func 함수를 호출하기 위해서는 어떤 식으로 호출해야 하는지 빈 칸 (a)에 들어갈 코드를 작성하시오.

```
void func() {
    std::cout << "Hello" << std::endl;
}

namespace MyNameSpace {
    void func()
    {
```

```
        std::cout << "Bye~" << std::endl;
    }

    void func2()
    {
        ::func(); // (a)
    }
}
```

3. 아래와 같은 코드가 있다. 아래 코드에서 Namespace1 에 정의된 함수인 func()를 Namespace2 에 포함시켜서 Namespace2::func() 와 같이 사용하고 싶다. 이때 필요한 코드를 빈칸 (a)에 작성하시오.

```
namespace Namespace1 {
    void func()
    {
        std::cout <<"func()" << std::endl;
    }
    void func(int a)
    {
        std::cout << "func(int)" << std::endl;
    }
    void func(double a)
    {
        std::cout << "func(double)" << std::endl;
    }
}
namespace Namespace2 {
    void func2(int a, int b) { std::cout << "func2(int,int)" << std::endl; }

    using namespace Namespace1; // (a)
}

int main() {
    Namespace2::func();
    Namespace2::func(3);
    Namespace2::func(3.);
    return 0;
}
```

4. 아래의 코드는 main.cpp 를 컴파일할 때 에러가 발생한다. 이 원인은 무엇이고 이를 해결하기 위해서는 어떻게 해야 하는지 기술하시오.

```
// polar.h
struct polar { double distance, angle; };
struct rect { double x, y; };
```

```
// polar_ops.h
#include "polar.h"
rect polar_to_rect(polar p);
polar rect_to_polar(rect r);
```

```
// main.cpp
#include <iostream>
#include "polar.h"
#include "polar_ops.h"
using namespace std;
int main() {
    rect rplace;
    polar pplace;
    cout << "Enter the x and y values: ";
    while (cin >> rplace.x >> rplace.y) {
        pplace = rect_to_polar(rplace);
        cout << "polar: " << pplace.distance << ", " << pplace.angle << endl;
        cout << "Next two numbers (q to quit): ";
    }
    cout << "Done.\n"; return 0;
}
```

주어진 코드는 크게 두 가지 문제를 가지고 있다.

첫째, 함수 `rect polar_to_rect(polar p)` 와 `polar rect_to_polar(rect r)` 가 프로토타입만 선언되어 있다. 하지만 main.cpp에서 함수 `polar_to_rect(polar p)` 를 사용하고 있기 때문에, 각 함수에 대한 내용을 작성하여 문제를 해결해야한다.

둘째, `polar.h` 가 중복되어 include 되고 있다. 이는 `polar` 와 `rect` 가 중복되어 선언되는 문제를 야기할 수 있기 때문에

- 각 헤더에 **`#pragma once`** 를 추가하거나,
- 각 헤더에 **`#ifndef`** UNIQUE_NAME
  `#define` UNIQUE_NAME
  ... *// Content of existing header*
  `#endif` 를 추가하거나,
- main.cpp에서 **`#include`** "polar_ops.h" 만 include해서

해서 문제를 해결할 수 있다. `UNIQUE_NAME` 은 각 헤더 별로 들어갈 unique한 이름을 의미하며, `...` 은 각 헤더의 기존 내용이 들어갈 자리이다. 그리고, **`#pragma once`** 는 이를 지원하는 컴파일러에서만 작동함을 유의해야 한다.

5. 다음과 같이 class 와 static method 를 이용하면 namespace 와 비슷한 기능을 흉내 낼 수 있다. 그렇다면 class 와 static method 를 사용하는 것과 비교하였을 때 namespace 를 사용하는 장점은 무엇인가?

Namespace 의 장점들을 최대한 많이 기술하시오.

```
class FakeNamespace
{
public:
    static void func();
    static void func2();
    static void func3();
};

int main() {
    FakeNamespace::func();
    return 0;
}
```

class는 객체지향 프로그래밍에서 Encapsulation을 달성하기 위한 오브젝트이고, namespace는 코드를 논리적으로 분리하는 단위라는 점에서 근본적인 차이가 있다. class와 비교하였을 때 namespace가 가지는 장점은 크게 3가지가 있다.

우선 using을 사용하여 코드를 줄일 수 있다. 예를 들어, 우리가 std의 namespace에 있는 코드들과 이름이 겹치는 코드가 없다면, **using namespace** std; 와 같이 using을 사용하여 일일이 std:: 를 입력하지 않아도 된다. 물론, 여러 namespace의 코드를 이용하는데 이름이 겹치는 코드가 있다면 using의 사용을 자제해야 한다. 하지만, using이라는 keyword를 쓸 수 있으면서 조건부로 못쓰는 경우(namespace)와, 어떠한 경우에도 못쓰는 경우(class) 중에서는 분명히 전자가 장점이 될 수 있다고 생각한다.

둘째, 자유롭게 코드를 추가할 수 있다. namespace 처럼 쓰기 위한 class를 정의했다고 해보자. 이때, class에는 자유롭게 함수나 변수 등의 메서드를 추가할 수 없다. 같은 이름의 클래스를 다시 정의하는 것이 불가능하기 때문이다. 하지만 namespace는 이와 달리 자유롭게 열고 닫는 것이 가능하기 때문에 함수나 변수 등의 메서드를 비교적 자유롭게 추가할 수 있다.

셋째, 익명 namespace를 만들 수 있다. class와 달리, 익명으로 namespace를 만들 수 있는 옵션을 제공함으로써, 이를 특정 코드의 접근을 힘들게 만드는 등의 특수한 경우에 활용할 수 있다.

게다가 근본적으로, class는 어떠한 데이터와 그 데이터에 유관한 메서드들의 집합이지, 단순한 메서드의 세트가 아니므로 class를 namespace처럼 쓰는 것은 피해야 할 것이다.

6. 아래의 코드는 문자열을 다루기 위한 간단한 String 클래스이다. main 함수에 있는 예제와 같이 이 클래스의 객체를 기존의 C-style 문자열을 처리하기 위한 strcpy 와 같은 함수에 바로 사용하고 싶다. 그러나 현재에는 이런 것을 가능하게 하는 코드가 빠져 있어서 아래의 코드는 컴파일 에러가 난다. String 객체를 C-style 문자열처럼 다룰 수 있게 하기 위해서 String 클래스에 무엇을 추가해야 하는가? 추가해야 하는 코드를 작성하시오.

```
#include<iostream>
#include<cstring>
class String {
```

```
private:
    char *str;
    int len;
public:
    String(const char *s);
    String(const String &st); // copy constructor
    String(); // default constructor
    ~String(); // destructor
    int length() const { return len; }
    String &operator=(const String &st);
    operator char* () const; // 추가된 코드
};

String::String() { len = 0; str = new char[1]; str[0] = '\0'; }
String::String(const char *s)
{ len = std::strlen(s); str = new char[len+1]; std::strcpy(str, s); }
String::String(const String &s)
{ len = s.len; str = new char[len+1]; std::strcpy(str, s.str); }
String::~String() { delete[] str; }

String& String::operator=(const String &st) {
    delete[] str;
    len = st.len;
    str = new char[len+1];
    strcpy(str, st.str);
    return *this;
}

// 추가된 코드
String::operator char *() const {
    return str;
}

int main() {
    String pennywise("You'll float too.");
    char georgie[100];
    std::strcpy(georgie, pennywise);
    return 0;
}
```

7. 다음 코드에서 Derived 클래스의 constructor 의 파라미터 중 v_는 Base 클래스로부터 상속받은 v 를 초기화하기 위한 값이고, vv_는 Derived 클래스의 멤버인 vv 를 초기화하기 위한 값이다. 빈 칸 (a) 에 들어갈 코드를 작성하시오.

```
#include <iostream>

class Base {
private:
    int v;
```

```
public:
    Base(int v_) { v = v_; }
    ~Base() {}
};

class Derived : public Base {
private:
    int vv;
public:
    Derived(int v_, int vv_);
    ~Derived() {}
};
Derived::Derived(int v_, int vv_)
////////////// (a) //////////////
: Base(v_)
{
    vv = vv_;
}
////////////////////////////////
```

8. 아래의 코드는 빨간 색으로 표시된 부분에서 컴파일 에러가 발생한다. 이를 수정하기 위해서는 어떻게 해야 하는가? 빈칸 (a)에 들어갈 컴파일 에러를 피하기 위한 코드를 작성하시오.

```
class IntVector {
private:
    int *arr;
    int len;
public:
    IntVector(int len_) : len(len_), arr(new int[len]) { }
    IntVector(const IntVector &v) : len(v.len), arr(new int[len])
    { memcpy(arr, v.arr, sizeof(int) * len); }
    IntVector() : len(0), arr(nullptr) { }
    ~IntVector() { delete[] arr; }
    int length() const { return len; }

    int &operator[](int i)
    {
        return arr[i];
    }

    /////////// (a) ////////////
    int &operator[](int i) const
    {
        return arr[i];
    }
    ////////////////////////////
};

ostream& operator<<(ostream& os, const IntVector& v)
{
```

```
    for(int i = 0; i < v.length(); i++)
        os << v[i] << " ";
    return os;
}
```

9. C++에서 클래스의 상속을 통해 구현되는 다형성이 무엇인지 자세히 기술하시오. 그리고 이 때 virtual method 를 왜 사용해야 하는지 설명하시오.

   우선 다형성은 기본적으로 변수, 오브젝트, 함수 등이 다양한 자료형에 속할 수 있다는 특징을 말한다. 특히, C++에서는 어떤 클래스의 포인터 변수에 어떤 클래스의 자식 클래스들의 주소를 할당하는 upcasting을 통해 다형성을 이끌어냈다. 예를 들어, 클래스 Base가 있고, Base에서 derive된 클래스 ChildA와 ChildB가 있다고 해보자. 이때, Base에 `v_a()` 라는 virtual method가 있다면, Base 클래스 형의 포인터 변수 `Base* ptr_class` 에 Base, ChildA, ChildB 중 어떤 클래스의 인스턴스의 주소가 할당된 것에 상관 없이 `ptr_class->v_a()` 와 같이 사용할 수 있는 것이다. 이 경우, v_a()가 Base, ChildA, ChildB 등 다양한 클래스에서 사용될 수 있기 때문에, 다형성이 실현되었다고 볼 수 있다. 즉, C++에서 다형성이란, upcasting을 통해 자식 클래스들이 virtual 메서드들을 각각 수정하고 사용할 수 있게 함으로서 virtual 요소가 다양한 자료형에 속할 수 있게 만든 것이라고 말할 수 있다.

   이 때, virtual method를 사용하는 이유를 알기 위해서는 vtable에 대해 알아볼 필요가 있다. 모든 컴파일러에서 vtable을 이용하여 virtual method를 구현하는 것은 아니지만, 많은 컴파일러가 vtable을 활용하기 때문에 이를 이용하여 virtual method의 필요성을 설명하고자 한다. vtable은 오브젝트가 만들어질 때 생성되는 오브젝트 고유의 가상 메서드 테이블이다. 위 문단의 Base, ChildA, ChildB라는 클래스가 있는 상황에서, ChildB가 v_a()라는 메서드를 override해서 수정했다고 해보자. 이 때, Base와 ChildA 오브젝트의 vtable에서 v_a()가 가리키는 주소는 같고, ChildB 오브젝트의 vtable에서 v_a() 주소는 달라지게 된다. 즉, upcasting 하여 같은 v_a()를 호출해도, ChildB의 오브젝트인지 아닌지에 따라서 실제로 호출되는 함수가 달라지는 것이다. C++에서는 이 vtable을 통해 여러 상속된 클래스들 사이에서 같은 이름의 메서드가 호출되어도 다양한 활동을 할 수 있도록 다형성을 이끌어냈고, 이 vtable에 메서드를 추가하기 위해 키워드 **`virtual`** 를 사용하여 virtual method로 만들어야 하는 것이다. 즉, virtual method는 컴파일러에게 이 메서드가 여러 클래스에서 다형성을 가지고 실행될 수 있음을 알려주기 위해 필요한 것이고, 이를 많은 컴파일러에서 vtable을 이용해 구현했다는 것을 말하고자 하였다.

   하지만, 단순히 코딩의 편의를 위해 모든 메서드를 virtual로 선언하는 것을 권장하지는 않는다. 왜냐하면 virtual일 필요가 없는 메서드까지 vtable에 추가하게 되면 메모리와 실행 속도에 있어서 손해를 볼 수 있으며, vtable의 주소와 오프셋이 leak 되었을 때 virtual일 필요가 없었던 함수들까지 임의로 실행시킬 수 있기 때문이다. 이 때문에, 꼭 필요한 함수들만 virtual로 할당하는 것을 권장하는 편이다.

10. 다음 코드를 실행하면 코드의 아래에 있는 것과 같은 내용이 출력된다. 빈 칸 (a), (b), (c)에 들어갈 적절한

코드를 작성하시오.

```
#include <iostream>

class Vehicle {
public:
    Vehicle() {}
    virtual ~Vehicle() {} void show() const;
protected:
    virtual int num_wheels() const { return 0; } // (a)
};

void Vehicle::show() const
{
    const int n_wheels = num_wheels();
    std::cout << "Num wheels: " << n_wheels << std::endl;
}


class Sedan : public Vehicle
{
public:
    Sedan() {}
    ~Sedan() {} protected:
protected:
    int num_wheels() const override { return 4; } // (b)
};


class Motorcycle : public Vehicle
{
public:
    Motorcycle() {}
    ~Motorcycle() {}
protected:
    int num_wheels() const override { return 2; } // (c)
};

int main()
{
    Vehicle *v[2];
    v[0] = new Sedan();
    v[1] = new Motorcycle();
    for(int i = 0; i < 2; i++) v[i]->show();
    delete v[0];
    delete v[1];
    return 0;
}
```