PUSAN NATIONAL UNIVERSITY

REPORT

프로그래밍언어론 과제6

과 목 명 :	프로그래밍언어론
지도교수 :	
학 과 :	전기컴퓨터공학부 정보컴퓨터공학전공
학 번 :	
이 름 :	장 수현
제 출 일 :	2020년 5월 28일

부산대학교

1. Why do we need null virtual classes in C++? We could just refrain from allocating objects of this class and simply use subclasses of the superclass.
–구체적인 C++ 예를 통해서 설명하시오.

sol)

1.0 문제에 대한 답을 하기 위해 사전에 알아야 할 개념들이 몇 가지 있다.

1.1 Inheritance (상속)

구조적 프로그래밍에서 객체지향으로 넘어오는 과정에서 여러 객체들을 묶는 캡슐화가 많이 장려되었다. 캡슐화의 결과로 너무 많은 그룹들이 생성 되었는데, 그 중 어떤 그룹들 간에는 공통점이 있었다. 그 공통점을 기준으로 또 그룹화를 했고, 상속이라는 개념이 생성되었다. 그래서 공통된 부분을 base class로 만들고, base class를 기준으로 복잡한 내용을 단순하게 인지하는 것이 가능해 객체지향을 지원할 수 있게 되었다. 아래 코드는 상속의 예시 코드이다.

```
#include <iostream>

using namespace std;

struct people {             // base
    int id;
    char name[20];
};

struct student: people {    // derived
    int s_id;
    int grade;
};

struct professor: people {  // derived
    int p_id;
};


int main(void)
{
    people p;
    student s;
    professor f;

    return 0;
}
```

1.2 Overriding

base class와 이름과 signature가 같은 member를 derived class에 재정의하는 것으로, 인자 타입이 다른 overloading과는 차이를 가진다. Derived class의 객체를 통해서 overriding된

member를 참조할 경우 base class의 member가 아닌 derived class에서 정의된 member가 참조된다. 아래 코드는 overriding이 적용된 예시 코드이다.

```
#include <iostream>

using namespace std;

class vec1 {                                    // base
    int id, x;
public:
    vec1(int _id, int _x): id(_id), x(_x) {}
    int vid(void) const {
        return id;
    }
};

class vec2: vec1 {                              // derived
    int id, y;
public:
    vec2(int bid, int _id, int _x, int _y)
        : vec1(bid,_x), id(_id), y(_y) {}
    int vid(void) const {           // func. overriding
        if(id > 0)
            return id;
        return vec1::vid();         // access base func.
    }
};

int main(void)
{
    vec2 v2(1,22,333,4444);
    cout << v2.vid() << endl;       // derived class 에서 정의된 member 가 참조
    return 0;
}
```

1.3 Type Conversion

base class와 derived class 간의 타입 변환으로 up-casting과 down-casting 두 가지가 있다. 타입 변환 시 참조 타입이 일치하지 않을 경우에는 명시적으로 타입 변환을 하는 것이 원칙이지만, derived class*에서 base class*로의 타입 변환을 하는 up-casting의 경우 헤석상의 문제도 없고, dangling pointer도 발생하지 않으므로 안전하기 때문에 예외적으로 묵시적 타입 변환을 허용한다. 아래 코드는 up-casting을 사용하는 예시 코드이다.

```
#include <iostream>

using namespace std;

class vec2{
    int x,y;
public:
    vec2(int _x, int _y): x(_x), y(_y){}
    int sum(void) const {return x+y;}
};

class vec3:public vec2{
    int z;
public:
    vec3(int  x, int  y, int  z)
        : vec2(_x,_y), z(_z) {}
    int sum(void) const {return vec2::sum() + z;}
};
```

```
int main(void)
{
    vec2 v2(11,22);
    vec3 v3(111, 222,333);
    vec2* p1 = &v2;
    vec2* p2 = &v3;             // implicit type casting (up-casting)
    vec2* p3 = &v3;
    cout << p1->sum() << endl;  // vec2::sum()
    cout << p2->sum() << endl;  // vec2::sum()
    cout << p3->sum() << endl;  // vec3::sum()
    return 0;
}
```

down-casting은 Base class*에서 Derived class*로의 타입 변환으로 안전하지 못하다. 부분적으로 할당 안된 메모리 영역을 가리키기 때문에 부분적 dangling pointer를 발생시킨다. 이는 Run-time Error를 야기시킬 수 있기 때문에 매우 위험하고, 바람직하지 못하다. 그럼에도 불구하고, down-casting이라는 것을 명시적 타입 변환으로 수행할 수는 있는데, base class pointer가 가리키는 대상은 반드시 derived class의 객체이거나 그것을 상속받은 클래스의 객체여야 한다. 아래의 코드는 down-casting 사용의 예시 코드로서 바람직하지 못한 코드이다.

```
#include <iostream>

using namespace std;

class vec2{
    int x,y;
public:
    vec2(int  x, int  y): x( x), y( y){}
    int sum(void) const {
        return x+y;
    }
};

class vec3:public vec2{
    int z;
public:
    vec3(int _x, int _y, int _z)
        : vec2(_x,_y), z(_z) {}
    int sum(void) const {
        return vec2::sum() + z;
    }
};

int main(void)
{
    vec2 v2(1,2);
    vec3 v3(1,2,3);

    vec2* p1 = &v2;
    vec2* p2 = &v3;
    vec3* p3 = (vec3*)p1;       // RE 의 위험성이 존재함
    vec3* p4 = (vec3*)p2;       // explicit type casting (down-casting)
    cout << p3->sum() << endl;  // undesirable
    cout << p4->sum() << endl;
    return 0;
}
```

1.4 Virtual Function

가상 함수는 derived class에서 재정의할 것으로 기대하는 멤버 함수로, 호출 시 참조 포인터의 참조 타입이 중요한 것이 아니라 실제 가리키는 본질이 중요하다. 그래서 derived class에 대한 포인터가 base class에 대한 포인터로 묵시적 타입 변환이 일어나면, 가상 함수가 overriding 되었을 경우 derived class의 함수가 호출된다. 아래는 가상 함수를 사용하는 예시 코드이다.

```
#include <iostream>

using namespace std;

class vec2{
    int x,y;
public:
    vec2(int _x, int _y): x(_x), y(_y){}
    virtual int sum(void) const {
        return x+y;
    }
};

class vec3:public vec2{
    int z;
public:
    vec3(int _x, int _y, int _z)
        : vec2( x, y), z( z) {}
    int sum(void) const {
        return vec2::sum() + z;
    }
};

int main(void)
{
    vec2 v2(11,22);
    vec3 v3(111, 222,333);
    vec2* p1 = &v2;
    vec2* p2 = &v3;             // implicit type casting
    vec3* p3 = &v3;
    cout << p1->sum() << endl;  // vec2::sum()
    cout << p2->sum() << endl;  // vec3::sum()
    cout << p3->sum() << endl;  // vec3::sum()
    return 0;
}
```

위의 예제 코드에서 virtual을 안 쓰면 vec2를 가리키는 것이므로 vec2_sum이 호출되는데, virtual을 쓰면 실제 가리키는 대상이 vec3에 있는 vec2이기 때문에 참조타입이 vec2*이지만 vec3_sum이 호출된다. 추가적인 개념으로, 가상 함수는 실제로 함수 포인터이다. 그래서 vec2에는 x, y, 포인터까지 변수가 세 개가 있다. 이것은 sizeof로 확인해볼 수 있는데, virtual이 있을 땐 sizeof(v2) == 12, sizeof(v3) == 16이고, virtual을 제거하면 sizeof(v2) == 8, sizeof(v3) == 12가 되므로 가상 함수가 결국 함수 포인터인 것을 알 수 있다.

```
33
666
666

--------------------------------
Process exited after 0.07585 seconds with return value 0
계속하려면 아무 키나 누르십시오 . . .
```

위의 예제 코드의 실행 결과 화면.

1.5 Pure Virtual Function

가상 함수이지만, base class에서 정의되지 않고 derived class에서 정의되어야 하는 함수로, 함수 선언에 "=0"을 붙임으로써 함수 정의를 하지 않는다. 그러니까 순수 가상 함수는 어떤 함수도 가리키고 있지 않은 함수 포인터이다. 즉, NULL로 초기화된 함수 포인터이기 때문에 당연히 초기화만 되어 있는 상태로 함수 호출하게 되면 Run-time Error가 발생한다.

1.6 Abstract Class

null virtual class라고도 불리며, 멤버 함수로 pure virtual function을 한 개 이상 소유하고 있는 클래스이다. Run-time Error를 예방하기 위해서 Abstract calss로 객체(변수)를 선언할 수 없다. Derived class 중에서도 base class의 pure virtual function을 모두 정의하지 않을 경우 이 클래스는 abstract class가 된다. 아래 코드는 Abstract Class의 예시 코드이다.

```
#include <iostream>

using namespace std;

class base {                    // abstract class
public:
    virtual int f(void) const =0;   // pure virtual function
    virtual int g(void) const =0;   // pure virtual function
};

class derived1: public base {   // abstract class
public:
    int f(void) const {
        return 0;
    }
    // g() is not overriding
};

class derived2: public base {   // concrete class
public:
    int f(void) const {
        return 0;
    }
    int g(void) const {
        return 0;
    }
};

int main(void)
{
    base a;     // error: abstract class
    derived1 b; // error: abstract class
    derived2 c; // ok : concrete class

    return 0;
}
```

위의 코드에서 a는 abstract base class로 그 자체로 객체를 만들 수 없기 때문에 error이다. 그리고 b는 모든 pure virtual function에 대해 정의하지 않았기 때문에 abstract class가 되어 error이다.

1.7 Polymorphism (다형성)

다형성은 동일한 인터페이스로 서로 다른 데이터 타입을 조작할 수 있게 지원되도록하는 프로그래밍 언어의 특성이다. 즉, base class 포인터가 가리키는 대상에 따라 다르게 동작하도록 하기 위한 개념이다. 객체지향은 복잡한 것을 단순하게 보이도록 하는 것이다. 상속을 통해 공통적인 것을 보이게 함으로써 단순하게 보이게 한다. 그런데 공통부분이 있다고 해도 각각은 결국 다 다르다. 이것에 대해 상세한 처리를 해주지 않으면 복잡한 것을 그저 단순하게 만든 것이다. 객체지향은 단순하게 보이도록 하는 것이지, 단순하게 만드는 것이 아니다. 그러므로 상세한 처리를 해주되, 이 상세한 부분을 감춰야 한다. 이것을 common processing이라 하고 다형성(Polymorphism)을 지원하도록 한다. 이 polymorphism을 구현하기 위해 사용되어야 할 것들이 Inheritace, Overriding, Virtual Function이고, 이 세 개를 포함하며 다형성을 지원하기 위해 등장한 개념이 Abstract class이다. 따라서 Abstract class는 다형성을 지원하여 객체지향을 실현하고, 그로 인해 소프트웨어의 기능을 추가하고 유지보수를 쉽게하도록 하기 위해 반드시 필요한 것이다. 아래는 다형성을 구현한 코드이다.

```
#include <iostream>

using namespace std;

class vec2 {
    int x, y;
public:
    vec2(int _x, int _y): x(_x), y(_y) {}
    virtual int sum(void) const {
        return x+y;
    }
};

class vec3: public vec2 {
    int z;
public:
    vec3(int _x, int _y, int _z): vec2(_x,_y), z(_z) {}
    int sum(void) const {
        return vec2::sum()+z;
    }
};

void print(vec2* a[],int num)
{
    for(int i=0;i<num;++i)
        cout << a[i]->sum() << endl;
}

int main(void)
{
    vec2 v1(1,2), v2(2,3), v5(3,4);
    vec3 v3(11,22,33), v4(22,33,44);
    vec2* a[] = {&v1, &v2, &v3, &v4, &v5};

    print(a,sizeof(a)/sizeof(*a));

    return 0;
}
```

sum에 대해서는 일괄적으로 처리하므로 common processing을, 실제 가리키는 대상에 따라 덧셈을 다르게 처리하므로 detail processing을 하는 polymorphism이 반영된 코드이다.