

다형성과 가상 함수

(Chapter 15. Polymorphism and Virtual Functions)

송실대학교
김강희 교수
(khkim@ssu.ac.kr)

가상 함수의 기초

❖ 다형성

- 하나의 (가상) 함수에 많은 의미들을 부여할 수 있음
- 가상 함수는 이러한 능력을 가지고 있음
- 객체 지향 프로그래밍의 근간을 이루는 원칙임

❖ 가상 함수

- 그 함수(의 body)가 정의되기 전부터 사용 가능함
 - ❖ 컴파일러의 "late binding"에 의해서 가능함
- 자식 클래스는 가상 함수를 정의하지 않고서는 객체를 만들 수 없음
 - ❖ 가상 함수는 자식 클래스 안에서 override 된다고 말함
 - ❖ 멤버 함수는 자식 클래스 안에서 redefine 된다고 말함
- 예: ActionHandler 의 run() 함수는 정의되기 전부터 Tetris::accept() 함수에서 사용되고 있음

Tetris. cpp

```
213 // accessors
214 Matrix *Tetris::overlap_currBlk(void) {
215     Matrix *tBlk1, *tBlk2;
216     tBlk1 = iScreen->clip(top, left, top + currBlk->get_dy(), left + currBlk->get_dx());
217     tBlk2 = tBlk1->add(currBlk);
218     delete tBlk1;
219     return tBlk2;
220 }
221
222 // mutators
223 void Tetris::update_oScreen(Matrix *tempBlk, int y, int x) {
224     oScreen->paste(iScreen, 0, 0);
225     oScreen->paste(tempBlk, y, x);
226 }
227
228 // TetrisOperation 들을 기반으로 동작하는 accept 함수
229 TetrisState Tetris::accept(char key) {
230     int idx = findOpIdxByKey(key);
231     if (idx == -1) {
232         cout << "unknown key! (int=" << (int) key << ")" << endl;
233         return state;
234     }
235     TetrisOperation *op = operations[idx];
236     if (state != op->preState) {
237         cout << "wrong preState for the current key!" << endl;
238         return state;
239     }
240     op->hAction->run(this, key);
241     Matrix *tempBlk = overlap_currBlk();
242     if (anyConflict(tempBlk) == false) {
243         state = op->postAState;
244     }
245     else {
246         op->hCounterAction->run(this, key);
247         delete tempBlk;
248         tempBlk = overlap_currBlk();
249         state = op->postCState;
250     }
251     update_oScreen(tempBlk, top, left);
252     delete tempBlk;
253     return state;
254 }
```

C++11 override keyword

- ❖ C++11 의 **override** 키워드는 어떤 함수가 override 되는지 redefine 되는지를 명확하게 해 줌 (이 키워드는 없어도 상관 없음)
- ❖ 예시 코드:

```
class MyOnLeft : public ActionHandler {  
public: void run(Tetris *t, char key) override { t->left-- }  
};  
class MyOnRight : public ActionHandler {  
public: void run(Tetris *t, char key) override { t->left++ }  
};
```

C++11 final keyword

- ❖ C++11의 **final** 키워드는 어떤 함수가 override 되는 것을 방지함
- ❖ 부모 클래스의 어떤 함수가 자식 클래스에서 한 번 override 되고나서, 다시 손주 클래스에서 override 되는 것을 **자식 클래스에서** 방지하고자 할 때 유용함
- ❖ 예시 코드:

```
class MyOnLeft : public ActionHandler {  
public: void run(Tetris *t, char key) override final { t->left-- }  
};  
  
class MyOnRight : public ActionHandler {  
public: void run(Tetris *t, char key) override final { t->left++ }  
};
```

가상 함수는 왜 항상 사용하지 않나?

- ❖ 한 가지 단점은 성능 오버헤드임
 - 결과적으로 프로그램은 저장 장치 공간을 더 많이 사용함
 - 'Late binding'은 프로그램 실행을 느리게 만듦
- ❖ 꼭 필요한 곳이 아니면, 가상 함수를 사용하지 말 것

순수 가상 함수

- ❖ 부모 클래스의 가상 함수는 body 정의를 가지고 있을 수도 있음
 - 자식 클래스에서 override 하기 전에는 부모 클래스 안에서는 기존 body 정의가 사용됨
- ❖ 부모 클래스의 가상 함수가 body 정의를 갖고 있지 않으면, 순수 가상 함수라고 말함
 - 다음 예시처럼 "=0"으로 body 를 대체함
 - ❖ `virtual void run() = 0;`

비가상 vs 가상 함수의 차이점

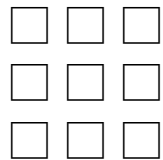
```
int main5(int argc, char *argv[]) {  
    Matrix* m1 = new Matrix(3,3);  
    m1->print(); cout << endl;  
    int A[] = { 0, 1, 0, 1, 1, 1, 0, 0, 0 }; // int A[][]  
    Matrix* m2 = new Matrix(A, 3, 3); // new Ma  
    m2->print(); cout << endl;  
    MyMatrix* m3 = new MyMatrix(3,3);  
    m3->print(); cout << endl;  
    MyMatrix* m4 = new MyMatrix(A, 3, 3); // n  
    m4->print(); cout << endl;  
    m2 = m4; // polymorphism: Matrix covers M  
    m2->print(); cout << endl;  
    return 0;  
}
```

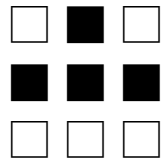
비가상 함수의 재정의
(static binding)

가상 함수의 재정의
(dynamic binding)

Matrix(3,3)
0 0 0
0 0 0
0 0 0

Matrix(3,3)
0 1 0
1 1 1
0 0 0

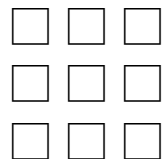


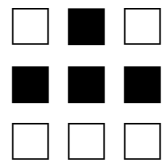


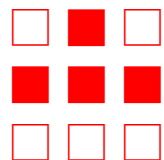
Matrix(3,3)
0 1 0
1 1 1
0 0 0

Matrix(3,3)
0 0 0
0 0 0
0 0 0

Matrix(3,3)
0 1 0
1 1 1
0 0 0







A)

비가상 vs 가상 함수의 차이점

```
7 class Matrix {
8     private:
9         int dy;
10        int dx;
11        int **array;
12        void alloc(int cy, int cx);
13    public:
14        int get_dy();
15        int get_dx();
16        int** get_array();
17        Matrix();
18        Matrix(int cy, int cx);
19        Matrix(const Matrix *obj);
20        Matrix(const Matrix &obj);
21        Matrix(int *arr, int col, int row);
22        ~Matrix();
23        Matrix *clip(int top, int left, int bottom, int right);
24        void paste(const Matrix *obj, int top, int left);
25        Matrix *add(const Matrix *obj);
26        int sum();
27        void mulc(int coef);
28        Matrix *int2bool();
29        bool anyGreaterThan(int val);
30        void print(); // virtual void print()
31        friend ostream& operator<<(ostream& out, const Matrix& obj);
32        Matrix& operator=(const Matrix& obj);
33    };
```

MyMatrix::print 재정의

```
#include "Matrix.h"
```

```
class MyMatrix : public Matrix {
```

```
public:
```

```
    MyMatrix() : Matrix() { }
```

```
    MyMatrix(int cy, int cx) : Matrix(cy, cx) { }
```

```
    MyMatrix(int *arr, int col, int row) : Matrix(arr, col, row) { }
```

```
    void print() {
```

```
        int dy = get_dy();
```

```
        int dx = get_dx();
```

```
        int **array = get_array();
```

```
        for (int y=0; y < dy; y++) {
```

```
            for (int x=0; x < dx; x++) {
```

```
                if (array[y][x] == 0) cout << "□ ";
```

```
                else if (array[y][x] == 1) cout << "■ ";
```

```
                else cout << "X ";
```

```
            }
```

```
            cout << endl;
```

```
        }
```

```
    }
```

추상 클래스

- ❖ 하나 이상의 순수 가상 함수를 가진 클래스를 "추상 클래스"라고 부른다
 - 추상 클래스는 객체를 생성할 수 없음
- ❖ 추상 클래스를 상속받아 자식 클래스가 모든 순수 가상 함수를 정의하지 않으면, 자식 클래스도 추상 클래스가 된다
 - 따라서, 자식 클래스도 객체를 생성할 수 없다.

Slicing Problem

❖ 예시:

```
Pet *ppet;  
Dog *pdog;  
pdog = new Dog;  
pdog->name = "누렁이";  
pdog->breed = "진도개";  
ppet = pdog;
```

❖ ppet 포인터로 breed 정보를 다음과 같이 출력할 수 없다

- `cout << ppet->breed;` // 컴파일 불가

❖ ppet 포인터로 breed 정보를 출력하려면?

- `ppet->print()` 함수를 가상 함수로 선언하고, Dog 클래스 안에서 그 가상 함수 body 를 정의했다면, `ppet->print()` 함수를 호출할 수 있다.

가상 소멸자

- ❖ 다음 코드는 문제가 있다:

```
Base *pBase = new Derived;
```

```
...
```

```
delete pBase;
```

- Base 타입으로 간주해서 소멸시키면, Derived 타입 안에서 추가로 생성된 포인터들이 가리키는 변수들이 소멸되지 않는다.

- ❖ 모든 소멸자를 virtual 로 정의하면, 문제가 해결된다.

Upcasting vs. Downcasting

❖ Upcasting 예시:

```
Pet vpet;  
Dog vdog;  
  
...  
vdog = vpet;    // 불법  
vpel = vdog;    // 합법  
vpel = static_cast<Pet>(vdog); // 합법
```

❖ Downcasting 예시 (dynamic_cast 사용):

```
Pet *ppet;  
ppet = new Dog;  
Dog *pdog = dynamic_cast<Dog*>(ppet); // 합법이나 조심  
필요
```

- downcasting 을 사용하려면 모든 멤버 함수들은 virtual 이
어야 한다

가상 함수들의 내부 동작

❖ 가상 함수 테이블

- 컴파일러가 테이블을 생성한다
- 각 가상 멤버 함수에 대한 포인터들을 가지고 있다
- 그 가상 함수를 위한 올바른 코드의 위치를 가리킨다

❖ 가상 함수를 가진 클래스들로부터 만들어진 객체들 또한 포인터를 가진다

- 그 포인터는 가상 함수 테이블을 가리킨다