협성대학교 소프트웨어공학과

고급프로그래밍 2 기말고사 정리

9 ~ 14 주차 강의 내용 정리

이시헌 제작

2023-12-11

9주차 수업 정리

1. 프로그램이 기억 공간을 참조하는 방법
   A. 변수 이름 / 포인터(주소)

2. 객체 포인터에서 멤버에 접근할 때.
   A. 간접참조연산자 '->'로 접근함.
   B. 특이한 점: 아래의 두 코드는 같다.

```
d = p->getArea();
d = (*p).getArea();
```

   C. 주의사항:
      i. 객체 이름은 주소가 아니다. (인스턴스의 이름에도 '&'를 붙여야 주소임. 배열과 다르다는 것을 명심하라!)
      ii. 아래의 코드를 보라. donut객체의 이름은 주소가 아니다.

```
Circle donut;
Circle* p = &donut;
```

3. 포인터 변수의 크기는 항상 8Byte다. (시스템 환경에 따라 4Byte일 수 있음.)

4. 객체 배열은 일반 배열 만들듯이 '[ ]'기호에 크기 넣어서 선언.
   A. 이때 각각의 원소마다 기본 생성자가 호출된다. (생성자의 선택이 불가능하다.) -> 디폴트 생성자 없이 다른 생성자를 가진 클래스라면, 오류가 발생한다.
   B. 단, 초기화 리스트의 형태로 각 원소를 초기화 할 수 있다. (기본 생성자로 만들어진 각 객체를, 새로 호출한 생성자로 만들어진 새 객체로 대체)

```
Circle circleArray[3];
//초기화 없이 객체 배열 생성시에는, 생성자 지정 불가.

Circle circleArray[3] = {Circle(10), Circle(20), Circle()};
//객체 배열 생성과 동시에 초기화.
```

5. 객체 배열의 이름은 주소다.

6. 다차원 객체 배열도, 일반적인 다차원 배열과 다를 것이 없다.

```
Circle circles[2][3] = {{Circle(20), Circle(2), Circle(44)}, {......}};
//행 단위로 잘라서 위의 예시처럼 해도 되고, 한번에 이어서 초기화 해도 됨.
```

7. 동적 메모리 할당: 프로그램 실행 도중, 힙 영역에 메모리를 할당하는 것.

   A. 필요한 공간의 양이 예측되지 않는 경우 사용한다.

   B. 힙 영역은 동적 메모리 할당을 위하여 예비되어 있는 공간이다. 정적 메모리 할당시에는 스택 공간을 사용함.

   C. new연산자: 〈기본 타입/배열/객체/객체 배열〉을 위한 동적 할당 가능.

   D. delete연산자: 객체 소멸 + 메모리 반환 수행.

   E. 객체 생성과 소멸 예시들:

```
//힙 공간 부족시, new가 NULL을 리턴한다는 점을 주의할 것.
int* p = new int;
if(!p) return 0;  //main에 반환하는 경우.

Cirlce* pCircle = new Circle();  //생성자를 호출하여 동적 생성.
delete pCircle;  //객체 동적 소멸.

int *pArr = new int[5];  //동적 배열 생성.
delete []pArr;  //동적 배열 소멸.

int n;
int* p2 = &n;
delete p2;  //런타임 에러 발생. (동적 생성되지 않은 포인터는 delete 불가.)

delete p;  //동적 소멸.
delete p;  //런타임 에러 발생. (이미 반환한 공간을 다시 반환 불가.)

int* p3 = new int(20);  //기본 타입도 생성자처럼 초기화 가능.
int* pA = new int[]{1, 2, 3, 4};  //배열의 동적 할당시 초기화 가능.

Circle* pp = new Cirlce[3];  //객체 배열 동적 생성.
pp[0].setRadius(10);  // pp[0]. == *(pp+0).  == (pp+0)->
delete p[];  //객체 배열 동적 소멸.
//0 - 1 - 2 순서 생성 … 2 - 1 - 0 순서 소멸.
```

9주차 수업 정리 끝.

10주차 수업 정리.

1. this 포인터: 클래스의 멤버 함수 내부에서, 함수가 속한 클래스의 주소를 담고 있는 포인터. 사용법은 다 알고 있으니 이 부분은 서술을 하지 않겠다.

   A. 사용처:

      i. 변수 이름 중복 사용 필요시에 사용.

      ii. 객체가 객체 자신을 리턴해야 할 때 사용.

   B. 주의사항: 정적 멤버 함수에서는 this 사용 불가. (객체 생성 이전에 호출될 수 있기 때문에.)

2. String class: 문자열을 객체로서 다룰 수 있도록 설계된, C++표준 라이브러리에 속한 클래스.

   A. 주요 멤버 함수

      i. .append("이어붙일 문자열")

      ii. string s("hello"); //생성자로 값 지정 가능.

      iii. cin >> s //cin으로 객체에 값 대입 가능.

      iv. getline(cin, s, '₩n'); //getline: 특정 문자를 만날 때 까지 입력 가능.

      v. .stoi(*string 객체*); //문자열을 정수로 바꾸어 리턴.

   B. 동적 생성과 소멸 방법.

```
string* s = new string("생성자에 넘길 값");
s->append("s 객체에 추가할 문장");
delete s;  //객체 소멸.
```

3. 클래스 연산.

A. 다른 타입의 객체와 연산을 수행하는 방법이 두 가지 있다.

i. 적절한 생성자 선언.

```
Circle c1 = 1+2;
//정수 하나를 받는 생성자가 있다면, 위의 문장은 아래와 동일.
Circle c2(3);
```

ii. 연산자 중복(오버로딩) <- 다음 번호에 자세히 설명 하겠음.

4. 연산자 오버로딩.

A. C++의 기본 지원 연산자만 중복 정의 가능.

B. 객체가 포함되지 않는 연산을 재정의하면 안됨. (객체의 함수로 정의되기에, 객체가 관여하지 않는 연산을 재정의 불가.)

C. 함수 형태로 재정의 해야 함.

D. 클래스의 멤버 함수나, 클래스의 프렌드 함수로서 전역 선언해야 함.

E. 피연산자의 개수 변경 불가.

F. 연산자 우선순위 변경 불가.

G. ‘ . ’, ‘ .* ’, ‘ :: ’, ‘ ? : ( ) ’의 4가지 연산자는 중복 정의 불가.

H. 정의 형태: 리턴타입 operator연산자(매개변수);

I. 예시: 이항 연산자 오버로딩.

```
class Power{
    Power operator+ (Power op2){
        //op2 를 받아서 모종의 연산을 하고, 리턴할 객체 만들기...
        return result;
    }
}
```

J. 단항 연산자 오버로딩시, 매개변수가 없음.

K. 후위 단항 연산자 오버로딩시, 사용하지 않는 매개변수 선언 필요. (정말로 의미 없는 값이 전달되는 변수 하나를 선언하라. 교재 p.346)

L. friend 함수 선언으로, 외부에서 정의된 연산자 오버로딩 함수를 사용 가능. (교재 p.350)

i. 기본 타입이 먼저 오는 클래스와의 연산식에서는, 클래스 내부에 연산자 함수 정의 불가.

10주차 수업 정리 끝.

11주차 수업 정리.

1. 상속.

A. 클래스 상속은 언어적으로는 자식에게 내려주는 것 이지만, 프로그래밍에서는 자식 클래스가 부모 클래스에게 요청하는 구조임을 명심하라. 자식이 부모에게 요청하는 것이다!

B. C++에서는 오로지 클래스 사이에서만 상속관계가 존재한다.

C. 클래스 객체간의 상속은 없다. 오로지 클래스간 상속만 존재.

D. 상속의 목적과 장점

i. 파생 클래스를 간결하게 작성 가능.

ii. 클래스간의 계층적 분류를 통해서 관리가 용이해짐. (상속된 클래스간의 관계 파악 용이)

iii. 클래스 재사용과 확장 용이 -〉 소프트웨어 생산성 향상.

E. 부모 클래스 설계: 만들어야 하는 객체들에 구현해야 하는 공통된 속성이나 기능을 모아서 부모 클래스에 구현한다.

F. 상속 문법: class *파생클래스명* : *접근지정자 부모클래스명* { *정의…* };

i. 상속을 수행하면, 파생클래스 생성시에 부모 클래스의 멤버를 포함하여 객체가 생성된다. 이는 메모리 공간에 부모와 자식의 모든 멤버를 위한 공간이 예비된다는 뜻이다.

ii. 즉, 부모 클래스의 모든 코드를 자식 클래스가 갖고 있는 것과 동일하게 동작함.

2. 객체 포인터

A. 업 캐스팅: 자식 클래스의 객체를 부모 클래스의 포인터로 가리키는 것.

i. 이는 묵시적으로, 자동적으로 수행되기에, 명시적인 캐스팅 연산 문구가 필요치 않다.

```
ColorPoint cp;
ColorPoint *pDer = &cp;  //자식 클래스의 객체 주소를 자식 클래스 타입의 포인터에 대입.
Point* pBase = pDer;  //이 순간 업캐스팅이 일어남.
```

B. 다운 캐스팅: 부모 클래스의 포인터가 가리키는 객체를, 자식 클래스의 포인터로 가리키는 것.

i. 다운 캐스팅은 항상 명시적으로 타입 변환을 명령해야 함.

```
ColorPoint *pDer;  //자식클래스 타입 포인터 생성.
Point *pBase, po;
pBase = &po;  //부모클래스 타입 객체를 부모클래스 타입으로 가리킴.
pDer = (ColorPoint*)pBase;  //다운캐스팅 수행함.
//부모클래스 타입 포인터를, 자식클래스 타입 포인터에 대입.
//pDer 은 자식 타입이지만, 가리키는 객체는 Point 타입이므로, 자식 클래스의 멤버 함수들은 갖고 있지 않다.
```

ii. 이때 pDer은 자식클래스 타입이지만, 부모 클래스 객체를 가리키고 있으므로, 자식 클래스만 가지고 있는 멤버를 참조한다면 런타임 에러가 발생한다.

3. 접근 지정자

A. 종류:

i. 멤버 접근 지정자

1. private: 선언한 클래스 외부에서 접근 불가. (디폴트 값)
2. protected: 파생클래스와 선언한 클래스에서 접근 가능.
3. public: 어디서든 접근 가능.
4. 주의: 부모의 private멤버를 자식 클래스가 갖고 있기는 하지만, 자식 클래스의 멤버가 해당 멤버에 접근할 수는 없다.

ii. 상속 접근 지정자

1. public: 멤버의 접근 지정자를 유지하며 상속받음.
2. private: 부모의 모든 멤버를 private로 변환하여 상속받음.
3. protected: 부모의 public, protected멤버맘 protected로 변환하여 상속받음.
4. 즉, 제약을 약화할 수는 없고, 강화만 가능하다는 점을 기억하라.

4. 상속시의 생성자 호출

A. 생성자 호출은 상속관계의 최상위 클래스부터 순차적으로 실행된다. (상위 클래스의 생성자를 계속 호출 -> 최상위 생성자부터 순차 종료됨.)

B. 자식클래스 객체 생성시, 부모클래스의 기본 생성자를 호출함.

i. 이 때, 기본 생성자가 아닌 다른 생성자를 호출하도록 지정 가능.

ii. B(int x) : A(x+3) { 자식클래스 생성자 정의… }

1. 위의 코드는, 자식클래스 생성자인 B가 호출될 때, 부모클래스에 자신이 받은 인자 x에 3을 더하여 넘기는 것이다. 이때는 부모클래스의 생성자 중에서, 정수형 매개변수를 하나 가진 생성자를 호출한다.

5. 주의! 생성자와 소멸자는 상속이 안된다.

6. 주의! 대입 연산자의 오버로딩은 상속이 안된다.

7. 주의! 부모 클래스의 private 멤버는 상속이 안되는 것이 아니다! 자식 클래스의 멤버에서 접근할 수 없을 뿐이다.

8. 다중 상속

A. 하나의 클래스가 복수의 클래스로부터 동시에 상속받는 것.

B. 쉼표로 연결하면 된다.

C. class A : public B, public C, public D { 클래스 정의 };

D. 이때 동일 내용을 중복하여 상속받을 가능성을 해소하기 위해, 가상 상속을 할 수 있음.

i. class A : virtual public B { 클래스 정의 };

ii. 즉, virtual 키워드를 붙여서 상속받으면, 부모와 조상 클래스들이 지닌 코드의 중복이 억제된다.

11주차 수업 정리 끝.

12주차 수업 정리.

이 교재는 오버라이딩과 함수 재정의를 구분한다. 개인적으로는 이런 구분이 무의미하며, 괜히 이해를 어렵게 하는 허세에 가깝다고 생각하지만, 당장 시험 성적을 버릴 수는 없으니, 나도 교재의 저자처럼 구분해 보겠다.

교재의 저자는 다음과 같이 정의한다.

오버라이딩(overriding): 가상함수를 재정의 한 것.

함수 재정의(redefine): 일반 함수를 재정의 한 것.

어이가 없다. 함수를 재정의 한 것을 모두 오버라이딩이라 칭하고, 가상함수를 재정의 한 경우와, 일반 함수를 재정의 한 것으로 세분해 인식하지 않는 이유가 무엇인가?

1. 부모에게서 상속받은 함수를 다시 정의하면, 자식 클래스에서는 재정의된 함수가 우선 호출된다.
    A. 이때, 자식 클래스의 객체를 업캐스팅 하면, 자식 클래스에서 재정의 한 함수 대신, 부모가 지닌 본래의 함수가 호출된다.
    B. 함수를 재정의 하더라도, 부모 클래스가 지닌 본래 함수의 코드를 상속받는다는 반증이기도 하다.

2. 가상함수: virtual키워드로 선언된 함수.
    A. 내용이 없는, 상속만을 위한 함수를 말한다. JAVA의 인터페이스 개념과 유사하다. 즉, 자식클래스에게 구현할 함수의 인터페이스를 제공하는 것이다.
    B. 실행 시점까지 함수의 내용을 결정짓지 않는다. 이를 동적 바인딩이라 함. (virtual 키워드 자체도, '동적 바인딩 지시어'라고 할 수 있다.)
        i. 함수의 실행 시점에, 연결된 코드를 찾아가는 것이다. 실행 중에 코드를 찾는다는 점에서 '동적' 바인딩 인 것이다.
    C. 가상함수에 내용이 정의되어 있어도 된다. 하지만 자식 클래스에서 가상함수를 오버라이딩 했다면, 부모 클래스가 지닌 가상함수의 원본은 무시된다. 단, 스코프 연산자로 원본 코드를 실행할 수 있다.
    D. 가상함수를 상속받아서 오버라이딩을 할 때, virtual키워드를 생략해도 컴

파일러가 추가해 준다.

3. 동적 바인딩(업캐스팅)의 의미.

A. 부모 클래스 포인터에, 동적으로 생성한 자식 클래스를 할당하는 것이 핵심이다.

```
#include <iostream>
using namespace std;

class A{
    public:
        virtual void printName(){
            cout << "A" << endl;
        }
};
class B : public A{
    public:
        void printName(){
            cout << "B" << endl;
        }
};
class C : public A{
    public:
       void printName(){
            cout << "C" << endl;
        }
};
int main(){
    //B 클래스 객체를 동적 생성하여 부모 클래스 포인터에 할당.
    A* p_parent = new B();
    p_parent->printName();

    //C 클래스 객체를 동적 생성하여 부모 클래스 포인터에 할당.
    p_parent = new C();
    p_parent->printName();
}
```

B. 위 코드는 B와 C를 출력한다.

C. 즉, 동적 바인딩으로, 객체의 다형성을 실현하기 쉽다. 프로그램 실행 도중에 어떤 객체이든 새로 바인딩 하여, 달리 재정의 된 함수를 호출할 수 있다. 이때 가상함수를 사용하면 좋다!

4. 주의!

A. 반복된 상속과 오버라이딩을 수행하면 맨 마지막 자손의 메소드가 호출된다. (실행 순서와 동작 원리상 당연한 말이지만, 교수님이 상당히 긴 시간을 들여서 설명했기에 혹시나 하는 마음에 한 줄 더 적는다.)

B. 오버라이딩을 했어도, 부모 클래스의 본래 가상함수 호출 가능. ' :: '연산자로 호출하면 된다.

```
Shape *pS;  //부모클래스 포인터
Circle c;  //자식클래스 객체.
pS = &c;  //업캐스팅.
pS->Shape::draw();  //부모클래스를 특정하여 본래의 함수 실행 가능.
```

5. 가상 소멸자

A. 업캐스팅을 사용하여 복수의 객체를 다룬다면, 가상 소멸자를 사용하는 것이 좋다.

B. virtual ~클래스이름( );

C. 부모 클래스 포인터에 자식 클래스 객체를 생성하여 대입하였기 때문에, 일반적인 경우에는 delete키워드로 포인터의 메모리 할당을 해제하면, 부모 클래스의 소멸자만 호출된다.

D. 이때, 부모와 자식의 소멸자를 virtual로 동적 바인딩 하도록 하면, 부모 클래스 포인터를 동적 소멸시킬 때, 자식 클래스 소멸자를 우선 실행시킬 수 있다.

6. 순수 가상 함수

A. 가상함수의 우측에 =0을 쓰면 선언 가능.

B. virtual void draw( )=0; //이런 방식으로 선언함.

C. 함수의 내용을 반드시 자식 클래스에서 오버라이딩 해야 하고, 부모 클래스에서 정의할 수 없다.

7. 추상 클래스

A. 순수 가상 함수를 하나 이상 지닌 클래스를 말한다.

B. 내용이 없는 메소드를 지녔기 때문에, 인스턴스를 만들 수 없다.

C. 오로지 상속만을 위해서 선언하는 클래스이다. 즉, 일종의 프로토타입 선언이고, 상속받은 클래스의 구조를 미리 제시하는 인터페이스의 역할을 하는 것이다.

12주차 수업 정리 끝.

13주차 수업 정리.

1. 템플릿 함수: 중복 함수들을 일반화 시킨 함수.
2. 템플릿 클래스: 다양한 자료형에 대응하여 일반화 된 클래스.
3. 위의 둘 모두 template <class T>의 선언문 뒤에 작성된다.
    A. 위의 선언문은 다음 구성 요소로 이루어졌다.
        i. template : 템플릿 선언 키워드.
        ii. <class T>: 제네릭 타입 변수 T 선언문.
    B. 즉, 위의 선언문 직후에, 변수 T를 사용한 함수나 클래스를 작성하면, 그것이 곧 템플릿 함수/클래스 인 것이다.
    C. 복수의 제네릭 타입 변수를 쉼표로 연결하여 선언 가능.
        i. template <class T, class T2>
    D. 이 제네릭 타입 변수에는 포인터와 클래스를 포함한 거의 모든 타입이 들어갈 수 있다.
4. 템플릿 함수의 경우는, 함수가 받은 인자의 타입으로 알아서 지정됨.
5. 템플릿 클래스의 경우는, 인스턴스 생성시 타입을 명시해야 함.
    A. 예: vector<int> v;

교재에 설명은 아주 많지만, 대부분 무의미하다. 위의 설명이 핵심인듯.

가장 좋은 방법은, STL을 사용하면서 템플릿을 이해하는 것 아닐까?

개인적인 생각으로는 이 부분을 조금 짧게 수업하고, 하나의 수업 시간에 바로 STL을 사용하게끔 했어야 한다고 생각한다.

13주차 수업 정리 끝.

14주차 수업 정리.

14주차에는 내가 코로나로 학교에 나오지 못하여, PPT를 보고 정리해 보겠다.

STL을 실제로 사용해 보는 실습 수업이었기에 딱히 어려운 점은 없을 것 같다.

중요한 것은, PPT에 등장하는 멤버 함수를 가볍게 암기하는 것?

1. STL: Standard Template Library <- C++의 표준 라이브러리 중 하나.
    A. STL은 세가지 구성 요소를 지닌다.
        i. 컨테이너: 자료 구조를 구현한 클래스들을 지칭.
        ii. iterator: 컨테이너 원소 순회를 위해 만들어진 포인터.
        iii. 알고리즘: 컨테이너 외부에 정의된, 다양한 기능의 함수들.
    B. 컨테이너는 각 클래스별로 독자적인 헤더파일에 정의되어 있다.
        i. 예1: vector클래스는 #include <vector>
        ii. 예2: list클래스는 #include <list>
    C. 알고리즘 함수들은 모두 하나의 헤더파일에 모아서 정의되어 있다.
        i. #include <algorithm>
    D. STL의 모든 예약어는 std 네임스페이스에 포함되어 있다.
        i. 예: std::vector<int> v;

2. vector
    A. 자주 사용하니, 특별히 알아야 할 것은 없다.
    B. push_back( ) / size( ) / at( ) / capacity( ) 정도만 알고 있으면 되겠다.

3. map

A. ‘키’와 ‘값’을 쌍으로 묶어서 저장하는 컨테이너다.

B. 키로 값을 검색한다.

C. PPT의 예시를 그대로 긁어서 붙이겠다.

```
//맵생성
Map<string,string>dic;  //키는영어단어,값은한글단어

//원소저장
dic.insert(make_pair("love","사랑"));  //("love","사랑")저장
dic["love"]="사랑";  //("love","사랑")저장

//원소검색
stringkor=dic["love"];  //kor 은 "사랑"
stringkor=dic.at("love");  //kor 은 "사랑"
```

4. iterator

A. 반복자라고도 부른다. 컨테이너의 원소 순회를 편하게 한다.

B. 컨테이너마다 지원하는 반복자의 종류가 다르다. 단, 반복자를 선언할 때는, 원하는 컨테이너의 이름 뒤에 붙여서 선언하기에, 우리가 어떤 반복자를 사용해야 하는지를 매번 찾아서 지정해 줄 필요는 없다.

C. 아래 예제를 보면 금방 이해가 될 것이다.

```
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<int> v = {1, 2, 3, 4, 5};

    vector<int>::iterator it;
    for(it=v.begin(); it != v.end(); it++){
        *it = (*it) * 2;
    }

    for(it=v.begin(); it != v.end(); it++){
        cout << *it << endl;
    }
}
```

D. 위의 코드는 v의 모든 원소에 접근하여 값을 두 배로 만든다.

E. 이터레이터가 포인터임을 기억한다면, 어려울 것이 없다.

F. it = v.erase(it); <- it이 가리키는 현재 원소를 제거하고, 다음 포인터를 리턴한다. 즉, 현재 이터레이터가 가리키는 원소를 벡터에서 제거하고, 이후의 모든 원소를 한 칸 앞으로 옮긴다.

5. algorithm

A. 컨테이너 클래스 내부에 정의되어 있지 않음을 명심하라. algorithm 헤더에 정의되어 있다.

B. iterator와 함께 동작하는 경우가 많다.

C. sort(v.begin( ), v.end( )); //벡터 전체 정렬. (시작 포인터부터, 끝에서 하나 앞 포인터까지.)

i. sort(v.begin( ), v.begin( ) + 3); 의 경우, 3개 원소를 정렬함.)

D. for_each(v.begin( ), v.end( ), 컨테이너의 각 원소를 전달받을 함수);

i. 컨테이너의 원소를 순차적으로 검색하면서, 각 원소를 세번째 인자로 받은 함수로 전달하기를 반복하는 함수다.

6. auto: 자동 타입 추정 지시어.

A. 자료형 대신 auto 키워드를 쓰면, 초기화에 사용된 데이터의 자료형으로 알아서 변수를 선언해준다. (함수의 리턴값도 받는다)

B. 이터레이터의 명시적 선언 없이 auto로 대체 가능. (이게 올바른 행동인지는 잘 모르겠는데?)

i. vector<int>::iterator it;의 선언문을 생략하고 다음 for문 작성이 가능하다는 것.

ii. for(auto it = v.begin( ); it != end( ); it++){ 무언가 함… }

1. 위의 auto it 부분에서 이터레이터가 선언되는 것.

7. 람다식: 함수를 단순하게 표현하는 기법.

A. [캡처 리스트] (매개변수 리스트) -> 리턴타입 { 함수 내용 };

```
double pi = 3.14;
auto calc = [pi](int r) -> double { return pi*r*r; };
//캡처 리스트에 변수가 포함되어야 람다 함수가 참조 가능.
cout << calc(3) << endl;
//calc 이란 변수는 람다식으로 정의된 함수를 저장하고 있음.

int sum = 0;
[&sum](int x, int y) { sum = x + y; } (2, 3);
//함수가 지역변수를 레퍼런스타입으로 받아서 직접 접근 가능.
```

B. for_each( )함수의 세번째 인자로, 이름이 없는 함수를 람다식으로 작성하여 정의할 수 있다.

i. for_each(v.begin( ), v.end( ), [](int n){cout << n << endl;});

14주차 PPT 정리 끝.

이상으로 고급프로그래밍2 수업의 9주차 ~ 14주차 수업 내용의 정리가 끝났다.

배운 것은 거의 없지만, 혹시 몰라 가능한 자세히 예시들을 모아서 정리했다. (*)