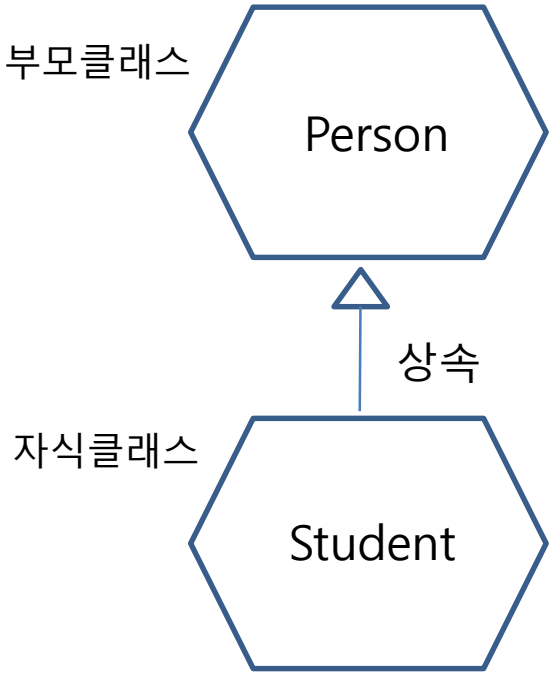
C++_상속, 다형성

Visual Studio 2022

상속(Inheritance)

- 상속이란?

  이미 구현된 클래스를 재사용해서 속성이나 기능을 확장하는 객체 지향 언어의 특성을 말한다.




```
class 클래스이름 : 부모클래스 이름{
    멤버 변수
}
```

```
class Person{
    멤버 변수
};
class Student : public Person{
    멤버 변수
};
```

콜론(:) 1개 사용
public 사용

상속의 선언과 활용

- 부모 클래스 정의

```
//부모 클래스
class Person {
private:
    string name;

public:
    void setName(string name) {
        this->name = name;
    }

    string getName() { return name; }
};
```

상속의 선언과 활용

- 자식 클래스 정의 : 상속

```
//자식 클래스
class Student : public Person {
private:
    int studentId;  //학생 아이디

public:
    void setStudentId(int studentId) {
        this->studentId = studentId;
    }

    int getStduentId() { return studentId; }
};
```

상속의 선언과 활용

- 상속 테스트

```
int main()
{
    //부모 객체 생성
    Person p1;
    p1.setName("이종범");
    cout << "부모의 이름: " << p1.getName() << endl;

    //자식 객체 생성
    Student st1;
    st1.setName("이정우");
    st1.setStudentId(101);

    cout << "학생의 이름: " << st1.getName() <<
        ", 학번: " << st1.getStduentId() << endl;

    return 0;
}
```

```
부모의 이름: 이종범
학생의 이름: 이정우, 학번: 101
```

생성자 상속 및 protected

- 생성자 상속

```
자식 클래스(부모 멤버 변수, 자식 멤버 변수) :
        부모 클래스(부모 멤버), 자식 클래스(자식 멤버) {
}
```

- 접근 지정자

접근 지정자	설 명
public	외부 클래스 어디에서나 접근 할수 있다.
protected	클래스 내부와 상속관계의 모든 자식 클레스에서 접근 가능
private	같은 클래스 내부 가능, 그 외 접근 불가

멤버 함수 재정의(Override)

- 메서드 오버라이드(Override)

  부모 클래스의 멤버 함수를 자식 클래스에서 다시 정의하는 것으로 함수 재정의(Override)라 한다.

Person 클래스

```
void greet() {
    cout << "안녕하세요. 성명: " << name << endl;
}
```

Student 클래스

```
void greet() {
    cout << "안녕하세요. 성명: " << name <<
            ", 학번: " << studentId << endl;
}
```

생성자 상속 및 함수 재정의

- 부모 클래스 정의

```
class Person {
protected: //자식 클래스에서만 접근 가능
    string name;

public:
    Person(string name) : name(name) {}

    void greet() {
        cout << "안녕하세요. 성명: " << name << endl;
    }

    void displayInfo() {
        cout << "Person name: " << name << endl;
    }
};
```

생성자 상속 및 함수 재정의

- 자식 클래스 정의 : 상속

```
class Student : public Person {
private:
    int studentId;  //학생 아이디

public:
    Student(string name, int studentId) :
        Person(name), studentId(studentId) {
    }

    void greet(){
        cout << "안녕하세요. 성명: " << name <<
            ", 학번: " << studentId << endl;
    }

    void displayInfo(){
        cout << "Student name: " << name << endl;
    }
};
```

생성자 상속 및 함수 재정의

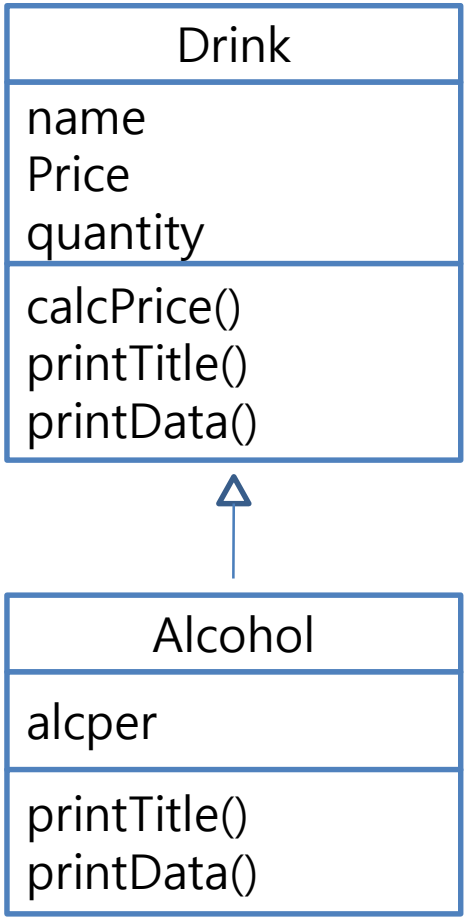
- 상속 테스트

```
int main()
{
    //부모 객체 생성
    Person p1("이종범");
    p1.greet();
    p1.displayInfo();

    //자식 객체 생성
    Student st1("이정후", 101);
    st1.greet();
    st1.displayInfo();

    return 0;
}
```

```
안녕하세요. 성명: 이종범
Person name: 이종범
안녕하세요. 성명: 이정후, 학번: 101
Student name: 이정후
```

매출 전표(statement)

- 매출 전표 작성하기




```
============= 매출 전표 =============
상품명   가격     수량     금액
커피     2500     4        10000
녹차     3000     3        9000

상품명(도수[%]) 가격     수량     금액
soju(15.1)      4000     2        8000

*** 합계 금액: 27000원 ***
```

매출 전표(statement)

- 음료(Drink) 클래스

```
class Drink {
protected:
    string name;     //상품명
    int price;       //가격
    int quantity;    //수량

public:
    Drink(string name, int price, int quantity) :
        name(name), price(price), quantity(quantity){ }

    int calcPrice() { return price * quantity; }
    static void printTitle() {
        cout << "상품명\t가격\t수량\t금액\n";
    }
    void printData() {
        cout << name << "\t" << price << "\t" <<
                quantity << "\t" << calcPrice() << endl;
    }
};
```

매출 전표(statement)

- 알코올(Alcohol) 클래스

```
class Alcohol : public Drink {
private:
    float alcper;

public:
    Alcohol(string name, int price, int quantity, float alcper) :
        Drink(name, price, quantity), alcper(alcper){ }

    static void printTitle() {
        cout << "상품명(도수[%])\t가격\t수량\t금액\n";
    }
    void printData() {
        cout << name << "(" << alcper << ")\t" << price << "\t" <<
            quantity << "\t" << calcPrice() << endl;
    }
};
```

매출 전표(statement)

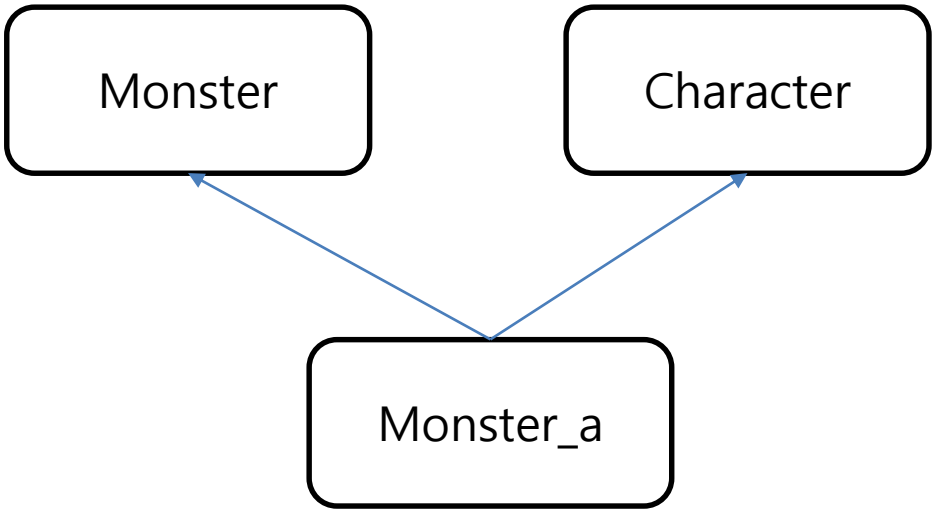
- Main 테스트

```
//Drink 인스턴스 생성
Drink coffee("커피", 2500, 4);
Drink tea("녹차", 3000, 3);

cout << "========== 매출 전표 ==========\n";
Drink::printTitle();
coffee.printData();
tea.printData();
cout << endl;

//Alcohol 인스턴스 생성
Alcohol soju("소주", 4000, 2, 15.1f);
Alcohol::printTitle();
soju.printData();

int total;
total = coffee.calcPrice() + tea.calcPrice() + soju.calcPrice();
cout << "******* 합계 금액: " << total << "원 *******\n";
```

다중 상속(Multiple Inheritance)

- 다중상속(multiple inheritance)

  하나의 파생 클래스가 여러 클래스를 동시에 상속받는 것이다.



다중 상속(Multiple Inheritance)

- Character, Monster 클래스

```
class Character {
public:
    Character() {
        cout << "Character 클래스 생성자" << endl;
    }
    ~Character() {
        cout << "Character 클래스 소멸자" << endl;
    }
};

class Monster {
public:
    Monster() {
        cout << "Monster 클래스 생성자" << endl;
    }
    ~Monster() {
        cout << "Monster 클래스 소멸자" << endl;
    }
};
```

다중 상속(Multiple Inheritance)

- Character, Monster 클래스를 상속받은 MonsterA 클래스

```
class MonsterA : public Monster, Character {
private:
    int location[2]; //좌표 저장

public:
    //기본생성자 : 초기화 목록
    MonsterA() : MonsterA(0, 0) {
        cout << "MonsterA 클래스 생성자" << endl;
        //MonsterA(0, 0); //초기화 되지 않음
    }

    MonsterA(int x, int y) : location{ x, y } {
        cout << "MonsterA 클래스 생성자(매개변수 추가)" << endl;
    }

    void showLocation() {
        cout << "위치(" << location[0] << ", " << location[1] << ")" << endl;
    }
};
```

다중 상속(Multiple Inheritance)

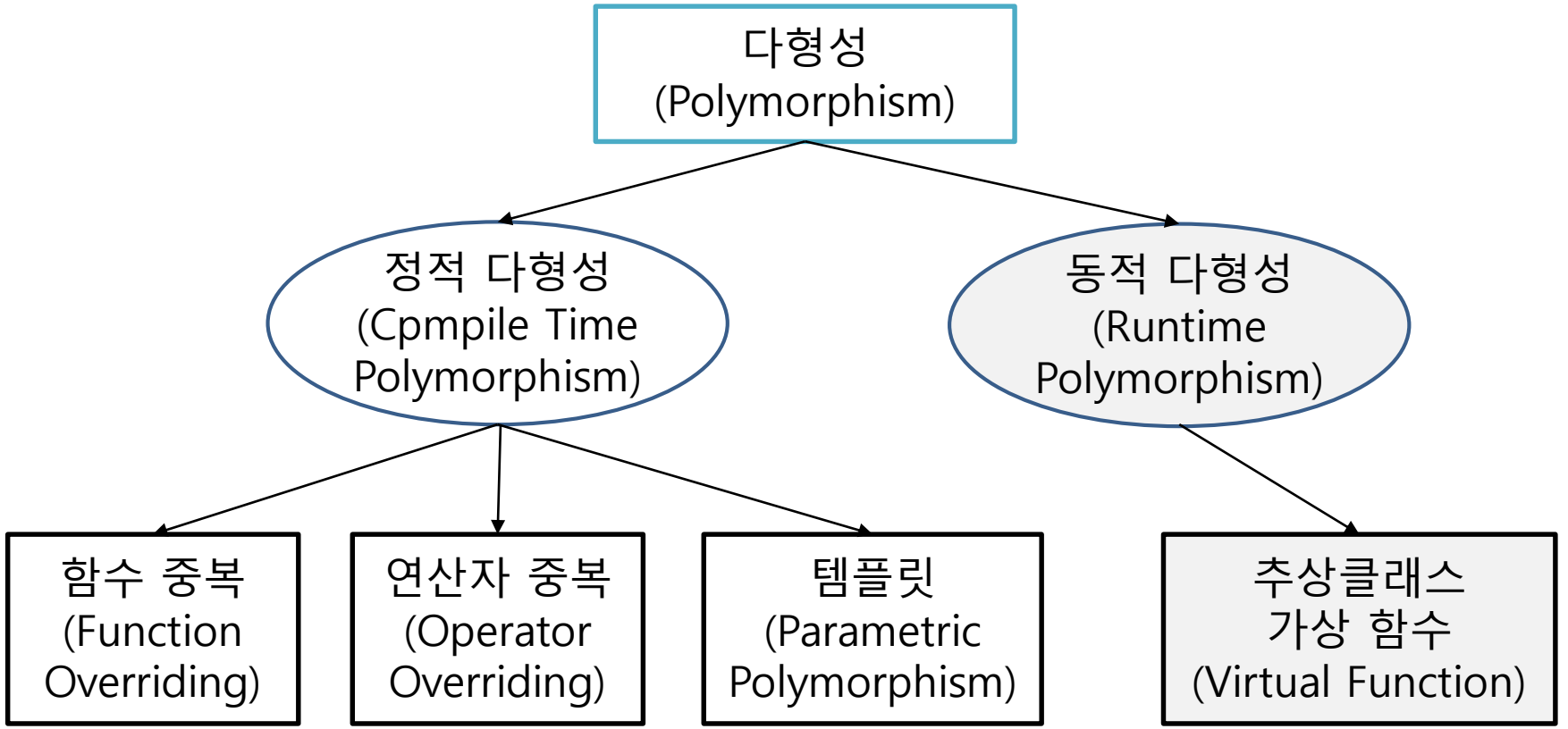
- Character, Monster 클래스를 상속받은 MonsterA 클래스

```
int main()
{
    MonsterA forestMonster;   //기본 생성자 호출
    forestMonster.showLocation();

    MonsterA woodMonster(10, 20); //매개변수가 있는 생성자 호출
    woodMonster.showLocation();

    return 0;
}
```

```
Monster 클래스 생성자
Character 클래스 생성자
MonsterA 클래스 생성자(매개변수 추가)
MonsterA 클래스 생성자
위치(0, 0)
Monster 클래스 생성자
Character 클래스 생성자
MonsterA 클래스 생성자(매개변수 추가)
위치(10, 20)
Character 클래스 소멸자
Monster 클래스 소멸자
Character 클래스 소멸자
Monster 클래스 소멸자
```

다형성(Polymorphism)

다형성이란? 다양한 종류의 객체에게 동일한 메시지를 보내더라도 각 객체들이 서로 다르게 동작하는 특성을 말한다.



다형성(Polymorphism)

- 다형성의 종류
  - **함수 중복(function overloading)** – 매개 변수의 개수나 타입이 서로 다른 동일한 이름의 함수들을 선언할 수 있게 한다.
  - **참조(reference)와 참조 변수** – 변수에 별명을 붙여 변수 공간을 같이 사용할 수 있다.
  - **new와 delete 연산자** – 동적 메모리 할당, 해제를 위한 new, delete연산자를 도입
  - **연산자 재정의(operator overloading)** – 기존의 연산자에 새로운 연산을 정의할 수 있게 한다.
  - **클래스와 제네릭 함수(generics)** – 함수나 클래스를 데이터 타입에 의존하지 않고 일반화 시킬수 있게 한다.

연산자 오버로딩(중복)

- 연산자 오버로딩
  - ✓ 연산자를 재정의하여 사용자 정의 클래스로 사용하는 것을 말한다.

함수반환형 **Operator** 연산자 (연산대상){ … }

연산자 오버로딩(중복)

- 객체 더하기

```
//연산자 오버로딩
class Point {
private:
    int x, y;
public:
    Point(int x, int y) {
        this->x = x;
        this->y = y;
    }

    void print() {
        cout << "x=" << x << ", y=" << y << endl;
    }

    //더하기 연산 함수
    Point operator+(Point p) {
        x = x + p.x;
        y = y + p.y;
        return Point(x, y);
    }
};
```

연산자 오버로딩(중복)

- 객체 더하기

```
int main()
{
    //점 객체 생성
    Point p1(1, 2);
    Point p2(3, 4);

    p1.print();
    p2.print();

    //객체 더하기
    Point p3 = p1 + p2;

    p3.print();

    return 0;
}
```

```
x=1, y=2
x=3, y=4
x=4, y=6
```

연산자 오버로딩(중복)

- 객체의 크기 비교(비교 연산)

```
class Circle {
    double radius;
public:
    Circle(double radius) {
        this->radius = radius;
    }
    double getRadius() { return radius; }
    double getArea() { return PI * radius*radius; }
    bool operator >= (Circle c);
};

bool Circle::operator>=(Circle c) {
    if (this->radius >= c.radius)
        return true;
    else
        return false;
}
```

연산자 오버로딩(중복)

- 객체의 크기 비교(비교 연산)

```
int main()
{
    Circle c1(5.1), c2(12.3);
    cout << "원1의 반지름 : " << c1.getRadius() << endl;
    cout << "원1의 면적 : " << c1.getArea() << endl;
    cout << "원2의 반지름 : " << c2.getRadius() << endl;
    cout << "원2의 면적 : " << c2.getArea() << endl;

    if (c1 >= c2)
        cout << "객체 c1이 c2보다 크다." << endl;
    else
        cout << "객체 c2가 c1보다 크다." << endl;
    return 0;
}
```

가상함수와 동적 결합(Dynamic Binding)

- 다형성에 의해 함수 재정의시 요구 조건
  - 부모 클래스의 멤버 함수가 가상함수(추상함수)로 선언되어야 함
  - **virtual** 키워드를 사용한다.
  - 함수 구현부의 내용은 비워둔다.

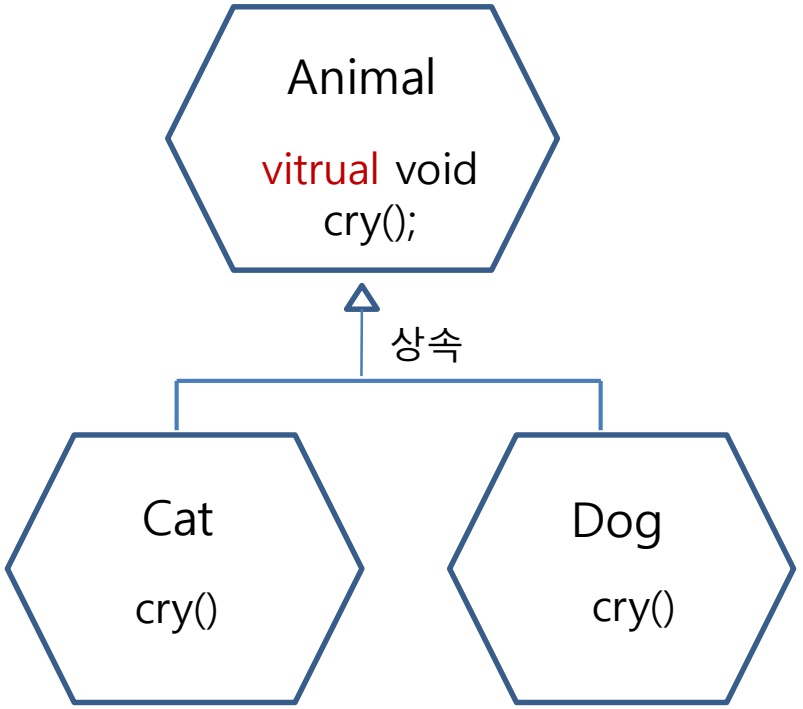
```
virtual cry() = 0
```

- 동적 결합 (Dynamic Binding)
  - 실행시 호출될 함수를 결정하는 것으로 이는 하나의 함수가 여러 클래스에서 오버라이딩 되었을 때 사용한다.
  - 객체 생성시 **new**, 해제 시 **delete** 사용

```
Animal* cat = new Cat     부모클래스 = new 자식클래스(자동 형변환 )
```

가상(Virtual) 함수

- 가상 함수 사용



가상(Virtual) 함수

- 가상 함수 사용

```
class Animal {
public:
    void breathe() {
        cout << "숨을 쉽니다.\n";
    }
    virtual void cry() = 0; //순수 가상 함수
    //virtual void cry() {}
};

class Cat : public Animal {
public:
    void cry() {
        cout << "야~ 옹!\n";
    }
};
```

```
class Dog : public Animal {
public:
    void cry() {
        cout << "왈~ 왈~\n";
    }
};
```

가상(Virtual) 함수

- 가상 함수 사용

```
int main()
{
    //정적 객체 생성
    /*Cat cat;
    cat.breathe();
    cat.cry();*/

    //동적 객체 생성
    Animal* cat = new Cat;
    Animal* dog = new Dog;

    cat->breathe();
    cat->cry();

    dog->breathe();
    dog->cry();

    delete cat;  //메모리 해제
    delete dog;

    return 0;
}
```

```
숨을 쉽니다.
야옹~
숨을 쉽니다.
멍~ 멍~
```

auto 자료형 키워드

auto 자료형 키워드

- auto 키워드는 변수 선언 으로부터 변수의 타입을 결정하도록 지시한다.
- auto는 복잡한 형식의 변수 선언을 간소하게 해주어 타입 선언의 오타나 번거로움을 줄여준다.

```
/*int square(int x) {
    return x * x;
}*/
//inline 함수 - 함수 호출이 일어나지 않음
//프로그램의 실행 속도 저하를 막기 위한 기능
int square(int x) { return x * x; }

int main()
{
    auto ch = 'K'; //문자형
    auto num = 12; //정수형
    auto unit = 2.54; //실수형
    auto* ip = &num;  //정수형 포인터
```

auto 자료형 키워드

- **auto 자료형 키워드**

```
    //함수의 리턴 타입
    auto value = square(9);
    cout << value << endl;

    //벡터 자료구조
    vector<int> vec = { 1, 2, 3, 4 };

    /*for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }*/

    //범위 기반 for - int형 대신 auto 사용
    for (auto v : vec) {
        cout << v << " ";
    }
    return 0;
}
```

```
K, 12, 2.54
12
81
1 2 3 4
```

다형성 – 매출 전표

- Drink 클래스

```
class Drink {
protected:
    string name;
    int price;
    int quantity;

public:
    Drink(string name, int price, int quantity) :
        name(name), price(price), quantity(quantity) {
    }

    int calcPrice() { return price * quantity; }

    static void printTitle() {
        cout << "상품명\t가격\t수량\t금액\n";
    }

    virtual void printData() = 0;  // 순수 가상 함수
};
```

다형성 – 매출 전표

- NonAlcohol 클래스

```
class NonAlcohol : public Drink {
public:
    NonAlcohol(string name, int price, int quantity) :
        Drink(name, price, quantity) {
    }

    void printData() override {
        cout << name << "\t" << price << "\t"
            << quantity << "\t" << calcPrice() << endl;
    }
};
```

다형성 – 매출 전표

- 가상 함수 사용

```
class Alcohol : public Drink {
    float alcper;

public:
    Alcohol(string name, int price, int quantity, float alcper) :
        Drink(name, price, quantity), alcper(alcper) {
    }

    static void printTitle() {
        cout << "상품명(도수[%])\t가격\t수량\t금액\n";
    }

    void printData() override {
        cout << name << "(" << alcper << ") " << price << "\t"
            << quantity << "\t" << calcPrice() << endl;
    }
};
```

다형성 – 매출 전표

- 매출 전표 테스트

```
Drink* coffee = new NonAlcohol("커피", 2500, 4);
Drink* tea = new NonAlcohol("녹차", 3000, 3);
Drink* soju = new Alcohol("소주", 4000, 2, 15.1f);

Drink* drinks[3] = { coffee, tea, soju };

cout << "========== 매출 전표 ==========\n";
Drink::printTitle();
for (auto drink : drinks)
    drink->printData();

int total = 0;
for (auto d : drinks) {
    total += d->calcPrice();
}
cout << "******* 합계 금액: " << total << "원 *******\n";

delete coffee;
delete tea;
delete soju;
```