Student Name: 杨卓
Student ID: 2022141450295

高级语言程序设计-II
Assignment 7

1 Abstract

In this assignment, we need to implement the class Cappuccino and Mocha with ingredients. The structures of these classes are below:

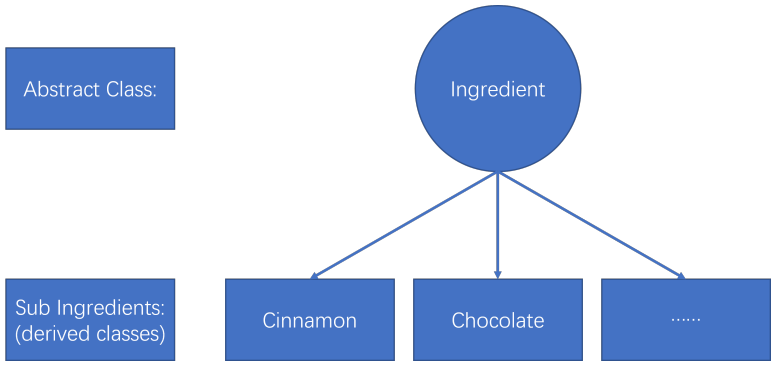

图 1: structure 1

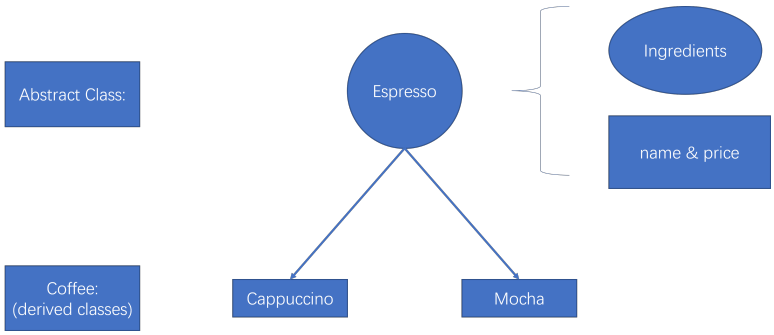

图 2: structure 2

2 Ingredient

`class Ingredient` is an abstract class with pure virtual function `std::string get_name()`. To better use the class derived from it, I also define another virtual function `Ingredient* Newself()` in order to get a new pointer with the same value.

```cpp
#ifndef INGREDIENT_H
#define INGREDIENT_H

#include <iostream>
#include <cstring>

class Ingredient {
public:
    double get_price_unit() { return price_unit; }
    std::size_t get_units() { return units; }
    virtual std::string get_name() = 0; // pure virtual function
    virtual Ingredient* Newself() = 0;

    double price() { return price_unit*units; }
protected:
    Ingredient(double price_unit, std::size_t units) :
        price_unit(price_unit), units(units), name("") {}

    double price_unit;
    std::size_t units;
    std::string name;
};

#endif // INGREDIENT_H
```

For example, in the copy constructor of `class EspressoBased`, we can't new an Ingredient for it is an abstract class, and also we can't push the pointer of ingredients directly because they should be different. In this situation, we can use `Newself()` to get a new pointer. Of course, it should be different in different sub class.

Answer of the question is that putting the constructor and variables in protected can let the sub ingredients access it.

3 Sub ingredients

Normally, we need to write 8 classes to define these sub ingredients. But using macro can simplify this step.

```cpp
#ifndef SUB_INGREDIENTS_H
#define SUB_INGREDIENTS_H

#include "ingredient.h"

#include <iostream>
#include <cstring>
```

```
8
9  // macro define
10 #define DEFCLASS(NAME, PRICE)                                    \
11 class NAME : public Ingredient {                                  \
12 public:                                                          \
13     NAME(std::size_t units) : Ingredient(PRICE, units) {         \
14         this->name = #NAME;                                      \
15     }                                                            \
16     std::string get_name() override { return name; }             \
17     NAME* Newself() override { return new NAME(this->units); }   \
18 };                                                               \
19
20 // class define
21 DEFCLASS(Cinnamon, 5);
22 DEFCLASS(Chocolate, 5);
23 DEFCLASS(Sugar, 1);
24 DEFCLASS(Cookie, 10);
25 DEFCLASS(Espresso, 15);
26 DEFCLASS(Milk, 10);
27 DEFCLASS(MilkFoam, 5);
28 DEFCLASS(Water, 1);
29
30 #endif // SUB_INGREDIENTS_H
```

4 EspressoBased

EspressoBased.h is below.

```
1  #ifndef ESPRESSO_BASED_H
2  #define ESPRESSO_BASED_H
3
4  #include "ingredient.h"
5
6  #include <iostream>
7  #include <cstring>
8  #include <vector>
9
10 class EspressoBased {
11 public:
12     virtual std::string get_name() const = 0;
13     virtual double price() = 0;
14
15     void brew();
16     std::vector<Ingredient*>& get_ingredients();
17
```

```
    virtual ~EspressoBased();
protected:
    EspressoBased() = default;
    EspressoBased(const EspressoBased& esp);
    void operator=(const EspressoBased& esp);

    std::vector<Ingredient*> ingredients;
    std::string name;

};

#endif // ESPRESSO_BASED_H
```

And `EspressoBased.cpp` is below.

```
#include "espresso_based.h"
#include "ingredient.h"

#include <vector>

// constructors

EspressoBased::EspressoBased(const EspressoBased& esp) {
    this->name = esp.get_name();
    for (auto e : esp.ingredients) {
        this->ingredients.push_back(e->Newself());
    }
}

// destructor

EspressoBased::~EspressoBased() {
    for (const auto& v : ingredients) {
        delete v;
    }
    ingredients.clear();
}

// copy constructor
void EspressoBased::operator=(const EspressoBased& esp) {
    this->name = esp.get_name();
    for (auto e : esp.ingredients) {
        this->ingredients.push_back(e->Newself());
    }
}

// member function
```

```cpp
std::vector<Ingredient*>& EspressoBased::get_ingredients() {
    return this->ingredients;
}
```

`Newself()` is used in the copy constructor.

Answer of the question is that it will compile error because sub class, Cappuccino and Mocha, can't override the destructor, so it can't be destructed.

5 Coffee

The final coffee, Cappuccino and Mocha, has similar program.

```cpp
// cappuccino.h
#ifndef CAPPUCCINO
#define CAPPUCCINO

#include "ingredient.h"
#include "espresso_based.h"

#include <iostream>
#include <cstring>
#include <vector>

class Cappuccino : public EspressoBased {
public:
    Cappuccino();
    Cappuccino(const Cappuccino& cap);
    ~Cappuccino() override;
    void operator=(const Cappuccino& cap);

    std::string get_name() const override;
    double price() override;

    void add_side_item(Ingredient* side);
    std::vector<Ingredient*>& get_side_items();

private:
    std::vector<Ingredient*> side_items;
};

#endif // CAPPUCCINO
```

```cpp
// mocha.h
#ifndef MOCHA_H
#define MOCHA_H

```

```cpp
#include "ingredient.h"
#include "espresso_based.h"

#include <iostream>
#include <cstring>
#include <vector>

class Mocha : public EspressoBased {
public:
    Mocha();
    Mocha(const Mocha& moc);
    ~Mocha() override;
    void operator=(const Mocha& moc);

    std::string get_name() const override;
    double price() override;

    void add_side_item(Ingredient* side);
    std::vector<Ingredient*>& get_side_item();

private:
    std::vector<Ingredient*> side_item;
};

#endif // MOCHA_H
```

And these are their implements.

```cpp
// cappuccino.cpp
#include "cappuccino.h"
#include "ingredient.h"
#include "sub_ingredients.h"

Cappuccino::Cappuccino() : side_items() {
    name = "Cappuccino";
    ingredients.push_back(new Espresso(2));
    ingredients.push_back(new Milk(2));
    ingredients.push_back(new MilkFoam(1));
}

Cappuccino::Cappuccino(const Cappuccino& cap) {
    this->name = cap.get_name();
    for (const auto& c : cap.ingredients) {
        this->ingredients.push_back(c->Newself());
    }
    for (const auto& c : cap.side_items) {
        add_side_item(c);
```

```
    }
}

Cappuccino::~Cappuccino() {
    for (const auto& item : side_items) {
        delete item;
    }
    side_items.clear();
}

void Cappuccino::operator=(const Cappuccino& cap) {
    Cappuccino _cap = cap;
    this->name = cap.get_name();
    this->ingredients.clear();
    this->side_items.clear();
    for (const auto& c : _cap.ingredients) {
        this->ingredients.push_back(c->Newself());
    }
    for (const auto& c : _cap.side_items) {
        add_side_item(c);
    }
}

std::string Cappuccino::get_name() const { return name; }

double Cappuccino::price() {
    double _price = 0;
    for (auto v : ingredients)
        _price += v->price();
    for (auto v : side_items)
        _price += v->price();
    return _price;
}

void Cappuccino::add_side_item(Ingredient* side) {
    side_items.push_back(side->Newself());
}

std::vector<Ingredient*>& Cappuccino::get_side_items() {
    return side_items;
}
```

```
// mocha.cpp
#include "mocha.h"
#include "ingredient.h"
#include "sub_ingredients.h"
```

```
5
6  Mocha::Mocha() : side_item() {
7      name = "Mocha";
8      ingredients.push_back(new Espresso(2));
9      ingredients.push_back(new Milk(2));
10     ingredients.push_back(new MilkFoam(1));
11     ingredients.push_back(new Chocolate(1));
12 }
13
14 Mocha::Mocha(const Mocha& moc) {
15     this->name = moc.get_name();
16     for (const auto& c : moc.ingredients) {
17         this->ingredients.push_back(c->Newself());
18     }
19     for (const auto& c : moc.side_item) {
20         add_side_item(c);
21     }
22 }
23
24 Mocha::~Mocha() {
25     for (const auto& item : side_item) {
26         delete item;
27     }
28     side_item.clear();
29 }
30
31 void Mocha::operator=(const Mocha& moc) {
32     Mocha _moc = moc;
33     this->name = moc.get_name();
34     this->ingredients.clear();
35     this->side_item.clear();
36     for (const auto& c : _moc.ingredients) {
37         this->ingredients.push_back(c->Newself());
38     }
39     for (const auto& c : _moc.side_item) {
40         add_side_item(c);
41     }
42 }
43
44 std::string Mocha::get_name() const { return name; }
45
46 double Mocha::price() {
47     double _price;
48     for (auto v : ingredients)
49         _price += v->price();
50     for (auto v : side_item)
```

```
51          _price += v->price();
52      return _price;
53  }
54
55  void Mocha::add_side_item(Ingredient* side) {
56      side_item.push_back(side);
57  }
58
59  std::vector<Ingredient*>& Mocha::get_side_item() {
60      return side_item;
61  }
```

For this assignment expression `a = a`, during their overloading of assignment operator, we should make a copy of the cap or moc.