Student Name: 戈天齐
Student ID: 2022141460202

C++

Assignment 7

头文件

ingredient.h

```
#ifndef INGREDIENTS
#define INGREDIENTS
#include <iostream>
//不需要写析构函数吗？
class Ingredient
{
public:
    virtual Ingredient* clone() = 0;
    double get_price_unit(){
        return price_unit;
    }
    size_t get_units(){
        return units;
    }
    virtual std::string get_name() = 0;

    double price(){
        return price_unit * units;
    }

protected:
    Ingredient(double price_unit, size_t units){
        this->price_unit = price_unit;
        this->units = units;
    }

    double price_unit;
    size_t units;
    std::string name;
};
#endif //INGREDIENTS
```

不需要额外的操作所以可以不显式声明定义析构函数，也可以 Ingredient() = default 同时，其派生类也没有进行手动管理内存空间，这样做在析构时也是安全的，不会造成内存泄漏

sub_ingredients.h

```
#ifndef SUB_INGREDIENTS_H
#define SUB_INGREDIENTS_H
#include "ingredient.h"
//宏类怎么用的
//构造函数需要包含哪些变量
class Cinnamon : public Ingredient
{
public:
    Cinnamon(size_t units) : Ingredient{5, units}
    {
        this->name = "Cinnamon";
    }
    virtual Cinnamon* clone() {return new Cinnamon(*this);}

    virtual std::string get_name() {return this->name;}
};
class Chocolate : public Ingredient
{
public:
    Chocolate(size_t units) : Ingredient{5, units}
    {
        this->name = "Chocolate";
    }
    virtual Chocolate* clone() {return new Chocolate(*this);}

    virtual std::string get_name() {return this->name;}
};
class Sugar : public Ingredient
{
public:
    Sugar(size_t units) : Ingredient{1, units}
    {
        this->name = "Sugar";
    }

    virtual Sugar* clone() {return new Sugar(*this);}

    virtual std::string get_name() {return this->name;}
};
class Cookie : public Ingredient
{
public:
    Cookie(size_t units) : Ingredient{10, units}
    {
        this->name = "Cookie";
```

```
    }

    virtual Cookie* clone() {return new Cookie(*this);}

    virtual std::string get_name() {return this->name;}
};
class Espresso : public Ingredient
{
public:
    Espresso(size_t units) : Ingredient{15, units}
    {
        this->name = "Espresso";
    }

    virtual Espresso* clone() {return new Espresso(*this);}

    virtual std::string get_name() {return this->name;}
};
class Milk : public Ingredient
{
public:
    Milk(size_t units) : Ingredient{10, units}
    {
        this->name = "Milk";
    }

    virtual Milk* clone() {return new Milk(*this);}

    virtual std::string get_name() {return this->name;}
};
class MilkFoam : public Ingredient
{
public:
    MilkFoam(size_t units) : Ingredient{5, units}
    {
        this->name = "MilkFoam";
    }

    virtual MilkFoam* clone() {return new MilkFoam(*this);}

    virtual std::string get_name() {return this->name;}
};
class Water : public Ingredient
{
public:
    Water(size_t units) : Ingredient{1, units}
```

```
92      {
93          this->name = "Water";
94      }
95
96      virtual Water* clone() {return new Water(*this);}
97
98      virtual std::string get_name() {return this->name;}
99  };
100 #endif // SUB_INGREDIENTS_H
```

派生类的构造函数中，基类部分调用基类的构造函数就好了

espresso_based.h

```
1  #ifndef ESPRESSO_BASED_H
2  #define ESPRESSO_BASED_H
3  #include <iostream>
4  #include "ingredient.h"
5  #include <vector>
6  class EspressoBased
7  {
8  public:
9      virtual std::string get_name() = 0;
10     virtual double price() = 0;
11
12     void brew();
13     std::vector<Ingredient*>& get_ingredients();
14     //更改为定义为虚析构函数
15     virtual ~EspressoBased();
16
17 protected:
18     EspressoBased();
19     EspressoBased(const EspressoBased& esp);
20     void operator=(const EspressoBased& esp);
21
22     std::vector<Ingredient*> ingredients;
23     std::string name;
24
25 };
26 #endif // ESPRESSO_BASED_H
```

声明了虚函数的基类的析构函数应该声明为虚析构函数，否则析构基类指针或者引用时，编译器只能默认的调用基类的析构函数，所以我们最好把析构函数声明为虚析构函数，在派生类里面把派生部分的手动管理的指针给 delete，比如 cappuccino 里面的 side_items 就需要手动删除，所以，将这一部分在 cappuccino 的析构函数中完成，在析构一个 cappuccino 的对象时，编译器会自动调用基类的析构函数来完成基类部分的析构

cappuccino.h

对于重载操作符 =，如果函数的返回类型为 cappuccino &，则可以实现 a = b = c 的操作实现 b = c a = b

其次，对于函数 get_name 完全没必要写成虚函数，无论在基类还是派生类，它的定义都是一样的

```
#ifndef CAPPUCCINO
#define CAPPUCCINO
#include "espresso_based.h"
#include "sub_ingredients.h"
class Cappuccino : public EspressoBased
{
public:
    Cappuccino();
    Cappuccino(const Cappuccino& cap);
    virtual ~Cappuccino();
    //为什么是void
    void operator=(const Cappuccino& cap);
    //这个为什么要声明为虚函数
    virtual std::string get_name();
    virtual double price();

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

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

};
#endif // CAPPUCCINO
```

mocha.h

```
#ifndef MCOHA_H
#define MCOHA_H
#include "espresso_based.h"
#include "sub_ingredients.h"
class Mocha : public EspressoBased
{
public:

    Mocha();
    Mocha(const Mocha& cap);
    ~Mocha();
    //为什么是void
    void operator=(const Mocha& cap);
    //这个为什么要声明为虚函数
    virtual std::string get_name();
    virtual double price();
```

```
17
18      void add_side_item(Ingredient* side);
19      std::vector<Ingredient*>& get_side_items();
20
21  private:
22      std::vector<Ingredient*> side_items;
23
24  };
25  #endif // MCOHA_H
```

源文件

espresso_based.cpp

这里涉及一个知识，浅拷贝和深拷贝，对于指针和引用的直接赋值，都是浅拷贝，也就是两个变量指向同一指针，如果对其中一个 delete，则另一个也会销毁。所以如果我们想实现，让两个相同的指针和引用相互独立这时候就需要使用深拷贝，操作是我们手动分配一个新的空间。对于 EspresspBased(const EspressoBased&) 的构造函数，我们想要实现的是创建两个相互独立的对象，所以，这里需要使用深拷贝操作

```
1   #include <cstring>
2   #include "sub_ingredients.h"
3   #include "espresso_based.h"
4   std::vector<Ingredient*>& EspressoBased::get_ingredients() {
5       return ingredients;
6   }
7
8   EspressoBased::EspressoBased()  = default;
9
10  EspressoBased::~EspressoBased() {
11      for(const auto& i : ingredients)
12          delete i;
13      ingredients.clear();
14      std::cout << "EspressoBased destructor called" << std::endl;
15  }
16
17  EspressoBased::EspressoBased(const EspressoBased &esp) {
18      name = esp.name;
19      //ingredients = esp.ingredients;
20      //上述实现是一个浅拷贝实现，会导致两个对象的ingredients里面的元素指针指向同一个地址
21  //    for(const auto i : ingredients){
22  //        Ingredient *temp = new Ingredient(*i); 抽象类无法作为类型
23  //        ingredients.push_back(temp);
24  //    }
25      if(name == std::string("Cappuccino")){
```

```
        Ingredient* espresso = new Espresso(2);
        this->ingredients.push_back(espresso);
        Ingredient* milk = new Milk(2);
        this->ingredients.push_back(milk);
        Ingredient* milkfoam = new MilkFoam(1);
        this->ingredients.push_back(milkfoam);
    }
    else{
        Ingredient* espresso = new Espresso(2);
        this->ingredients.push_back(espresso);
        Ingredient* milk = new Milk(2);
        this->ingredients.push_back(milk);
        Ingredient* milkfoam = new MilkFoam(1);
        this->ingredients.push_back(milkfoam);
        Ingredient* chocolate = new Chocolate(1);
        this->ingredients.push_back(chocolate);
    }
}
//左值呢
void EspressoBased::operator=(const EspressoBased& esp){
    this->ingredients = esp.ingredients;
    //this->ingredients = esp.get_ ?哪一种
    name = esp.name;
}
```

在 cappuccino 类中，我们还遇到一个问题，在深拷贝 side_items 时，我们并不知道它里面的元素时 ingredient 的具体哪一种派生类，但是由于基类是一个抽象类，我们还不可以创造基类的指针或者引用，这里采用的时虚拟拷贝构造函数，在 ingredient 基类声明一个拷贝函数，在派生类里面分别实现，这样我们即使不知道他具体是哪一种派生类也可以实现深拷贝

其次，在重载等号运算符的时候，我们会把左值的内容删掉，然后将右值拷贝复制一边给左值，但是存在一个特殊情况是 a = a; 对于这种，如果直接把左值的内容删掉，右值也会跟着销毁，实现方法是，把左值浅拷贝一次，把 vector 给清空，但不是放内存，把右值深拷贝给左值，然后再把原来的左值给释放掉

cappuccino.cpp

```
#include "cappuccino.h"
Cappuccino::Cappuccino(){
    this->name = "Cappuccino";
    Ingredient* espresso = new Espresso(2);
    this->ingredients.push_back(espresso);
    Ingredient* milk = new Milk(2);
    this->ingredients.push_back(milk);
    Ingredient* milkfoam = new MilkFoam(1);
    this->ingredients.push_back(milkfoam);
}
```

```
11
12  Cappuccino::Cappuccino(const Cappuccino &cap) : EspressoBased(cap){
13      for(const auto i : cap.side_items){
14          Ingredient *temp = i->clone();
15          this->side_items.push_back(temp);
16      }
17  //    for( auto i : cap.ingredients){
18  //        Ingredient *temp;
19  //        temp = reinterpret_cast<Ingredient *>(new decltype(i));
20  //        *temp = *i;
21  //        this->ingredients.push_back(temp);
22  //    }
23
24  }
25
26  Cappuccino::~Cappuccino() {
27      //这里会调用基类的析构函数吗
28      for(const auto& i : side_items)
29          delete i;
30      side_items.clear();
31      std::cout << "Cappuccino destructor called" << std::endl;
32  }
33  //为什么void还能实现赋值
34  void Cappuccino::operator=(const Cappuccino& cap){
35      this->name = cap.name;
36      //存左值
37      std::vector<Ingredient*> temp_ing_l = this->ingredients;
38      std::vector<Ingredient*> temp_side_l = this->side_items;
39      //存右值
40      std::vector<Ingredient*> temp_ing_r = cap.ingredients;
41      std::vector<Ingredient*> temp_side_r = cap.side_items;
42
43      //清空左值,但没有释放内存
44      this->ingredients.clear();
45      this->side_items.clear();
46      //深拷贝右值
47      for(auto const& i : temp_ing_r){
48          Ingredient *temp = i->clone();
49          this->ingredients.push_back(temp);
50      }
51      for(auto const& i : temp_side_r){
52          Ingredient *temp = i->clone();
53          this->side_items.push_back(temp);
54      }
55      //释放左值内存
56      for(const auto& i : temp_ing_l)
```

```
        delete i;
    temp_ing_l.clear();
    for(const auto& i : temp_side_l)
        delete i;
    temp_side_l.clear();
}

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

double Cappuccino::price()
{
    double sum = 0;
    for(const auto i : ingredients){
        sum += i->price();
    }
    for(const auto i:side_items){
        sum += i->price();
    }
    return sum;
}

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

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

mocha.cpp

```
#include "mocha.h"
Mocha::Mocha(){
    this->name = "Mocha";
    Ingredient* espresso = new Espresso(2);
    this->ingredients.push_back(espresso);
    Ingredient* milk = new Milk(2);
    this->ingredients.push_back(milk);
    Ingredient* milkfoam = new MilkFoam(1);
    this->ingredients.push_back(milkfoam);
    Ingredient* chocolate = new Chocolate(1);
    this->ingredients.push_back(chocolate);
}

Mocha::Mocha(const Mocha &cap) : EspressoBased(cap){
```

```
    for(const auto i : cap.side_items){
        Ingredient *temp = i->clone();
        this->side_items.push_back(temp);
    }
}

Mocha::~Mocha() {
    //这里会调用基类的析构函数吗
    for(const auto& i : side_items)
        delete i;
    side_items.clear();
}

void Mocha::operator=(const Mocha& cap){
    this->ingredients = cap.ingredients;
    this->name = cap.name;
    this->side_items = cap.side_items;
}

std::string Mocha::get_name(){
    return name;
}

double Mocha::price()
{
    double sum = 0;
    for(const auto i : ingredients){
        sum += i->price();
    }
    for(const auto i:side_items){
        sum += i->price();
    }
    return sum;
}

void Mocha::add_side_item(Ingredient *side) {
    side_items.push_back(side);
}

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

1 Question

1.1 answer1

基类的变量和函数定义为 protected 和 private 的区别在于，在派生类里面是否可见，如果定义为 private，在派生类里面完全不可见

1.2 answer2

上面提到了一个问题，对于基类的指针和引用，编译器默认调用基类的析构函数来析构，就会可能导致内存泄露，解决的方法有两个，一个是虚析构函数，一个是将析构函数定义为 protected。析构函数定义为 protected，会让基类和派生类外使用自动对象和 delete 时报出错误（因为访问权限禁止调用析构函数），就不会导致以上问题