学号：2022141410193　**姓名**：王枭越 **学院**：计算机学院

Assignment7

Ingredient

`ingredient.h`

```
#ifndef INGREDIENT_H
#define INGREDIENT_H

class Ingredient{
public:
    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;}

    virtual Ingredient* newtype()=0;

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 // INGREDIENT_H
```

Based on the original requirements, I add the member-method `virtual Ingredient* newtype()=0;`. As you can see it is a pure virtual function, being used to behave as the derived classes' interface.

> ***Question.*** why do you think the the constructor and the variables are defined as `protected` and not `private`? answer the question in your report after you completed your code.
>
> ***Answer.*** The reason is that the derived classes' constructors also need call the based class's constructor to initialize its member-data. Once if we make it private, we can't realize the function as I just said.

The detailed information about the method I add is as follows.

Sub_ingredients

`sub_ingredients.h`

```
#ifndef SUB_INGREDIENTS_H
#define SUB_INGREDIENTS_H

#include "ingredient.h"
#define DEFCLASS(NAME,PRICE)                                \
class NAME:public Ingredient{                               \
public:                                                     \
    NAME(size_t units):Ingredient{PRICE,units}{             \
        this->name = #NAME;                                 \
    }                                                       \
                                                            \
    std::string get_name()override{return this->name;}      \
    Ingredient* newtype()override{                          \
        return new NAME(this->units);                       \
    }                                                       \
};                                                          \

DEFCLASS(Cinnamon,5)
DEFCLASS(Chocolate,5)
DEFCLASS(Sugar,1)
DEFCLASS(Cookie,10)
DEFCLASS(Espresso,15)
DEFCLASS(Milk,10)
DEFCLASS(MilkFoam,5)
DEFCLASS(Water,1)


// class Cinnamon:public Ingredient{
// public:
//     Cinnamon(size_t units):Ingredient{5,units}{
//         this->name = "Cinnamon";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new Cinnamon(this->units);
//     }
// };

// class Chocolate:public Ingredient{
// public:
//     Chocolate(size_t units):Ingredient{5,units}{
//         this->name = "Chocolate";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new Chocolate(this->units);
```

```
//     }
// };

// class Sugar:public Ingredient{
// public:
//     Sugar(size_t units):Ingredient{1,units}{
//         this->name = "Sugar";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new Sugar(this->units);
//     }
// };

// class Cookie:public Ingredient{
// public:
//     Cookie(size_t units):Ingredient{10,units}{
//         this->name = "Cookie";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new Cookie(this->units);
//     }
// };

// class Espresso:public Ingredient{
// public:
//     Espresso(size_t units):Ingredient{15,units}{
//         this->name = "Espresso";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new Espresso(this->units);
//     }
// };

// class Milk:public Ingredient{
// public:
//     Milk(size_t units):Ingredient{10,units}{
//         this->name = "Milk";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new Milk(this->units);
//     }
// };

// class MilkFoam:public Ingredient{
```

```
// public:
//     MilkFoam(size_t units):Ingredient{5,units}{
//         this->name = "MilkFoam";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new MilkFoam(this->units);
//     }
// };

// class Water:public Ingredient{
// public:
//     Water(size_t units):Ingredient{1,units}{
//         this->name = "Water";
//     }

//     std::string get_name()override{return this->name;}
//     Ingredient* newtype()override{
//         return new Water(this->units);
//     }
// };

#endif // SUB_INGREDIENTS_H
```

The member-method `virtual Ingredient* newtype()=0;` can new an object with the same type and value of member-data. Thus we can use this method to copy itself, or we can not find out which type it is outside either.

Espresso_based

`espresso_based.h`

```
#ifndef ESPRESSO_BASED_H
#define ESPRESSO_BASED_H

#include <vector>
#include <string>
#include "sub_ingredients.h"

class EspressoBased{
public:
    virtual std::string get_name()=0;
    virtual double price()=0;

    void brew();
    std::vector<Ingredient*>& get_ingredients();

    virtual ~EspressoBased();

protected:
```

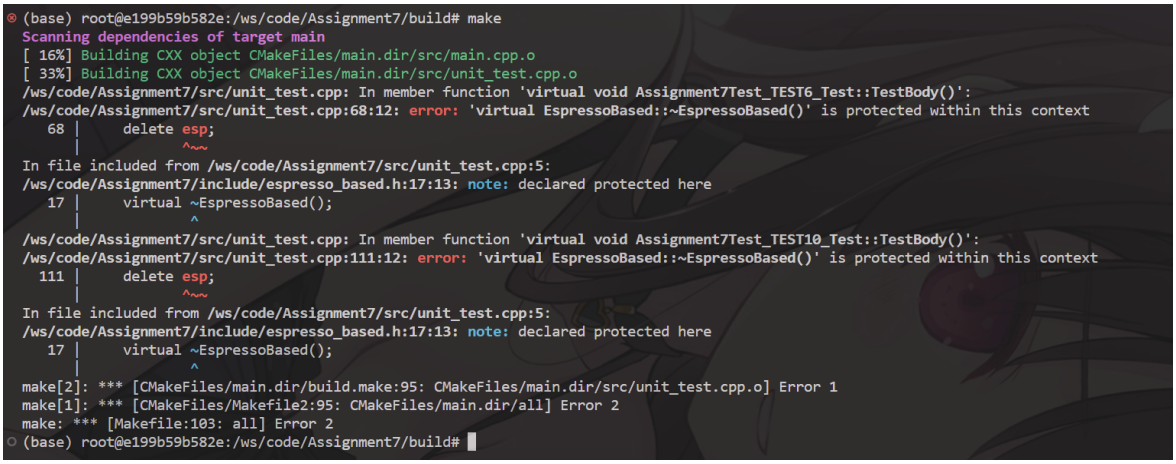
```
    EspressoBased()=default;
    EspressoBased(const EspressoBased& esp);
    void operator=(const EspressoBased& esp);

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

};

#endif // ESPRESSO_BASED_H
```

`espresso_based.cpp`

```
#include "../include/espresso_based.h"

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

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

EspressoBased::EspressoBased(const EspressoBased& esp){
    for(const auto& i : ingredients)
        delete i;
    ingredients.clear();

    for(auto it:esp.ingredients){
        ingredients.push_back(it->newtype());
    }
}

void EspressoBased::operator=(const EspressoBased& esp){
    for(const auto& i : ingredients)
        delete i;
    ingredients.clear();

    for(auto it:esp.ingredients){
        ingredients.push_back(it->newtype());
    }
    this->name=esp.name;
}
```

Cause this class is an abstract class, so we nearly or never call its `void EspressoBased::operator=(const EspressoBased& esp)`, as a consequence that we needn't realize it. And the other method's realization is as above.

> ***Question.*** what happens if you define the destructor i.e. `~EspressoBased()` in the protected section? explain your answer in your report.
>
> ***Answer.*** If we do so, we can't use specifier `delete` outside the derived class, since it is protected.



Cappuccino Class

`cappuccino.h`

```
#ifndef CAPPUCCINO
#define CAPPUCCINO

#include "espresso_based.h"

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

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

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

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

};

#endif // CAPPUCCINO
```

`cappuccino.cpp`

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

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

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

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

double Cappuccino::price(){
    double ret;
    for(auto it:ingredients){
        ret+=it->price();
    }
    for(auto it:side_items){
        ret+=it->price();
    }
    return ret;
}

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

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

Cappuccino::Cappuccino(const Cappuccino& cap):EspressoBased(cap){
    // Cappuccino::~Cappuccino();//Avoid using destructors in copying

    for(const auto& it:side_items)
        delete it;
    side_items.clear();

    name=cap.name;

    for(auto it:cap.side_items){
        side_items.push_back(it->newtype());
    }
}
```

```
void Cappuccino::operator=(const Cappuccino& cap){
    //When overloading the equal sign, it is not easy to call the equal sign
overloaded      //by the parent class,
    //Because there is a situation where tmp=tmp, you must first copy it to an
instance        //and then clear yourself,
    //So it's not easy to call the equal sign of the overloaded parent class,
    //So the equal sign of the parent class actually doesn't need to be written.
    Cappuccino tmp(cap);
    // Cappuccino::~Cappuccino();//Avoid using destructors in copying

    for(const auto& it:side_items)
        delete it;
    side_items.clear();

    for(const auto& i : ingredients)
        delete i;
    ingredients.clear();

    name=tmp.name;
    // ingredients<-cap.ingredients;
    for(auto it:tmp.ingredients){
        ingredients.push_back(it->newtype());
    }
    // side_items<-cap.side_items;
    for(auto it:tmp.side_items){
        side_items.push_back(it->newtype());
    }
}
```

There are some notices put as explanatory note, so it is no need to tautology.

Other than this, there is still something need to be interpreted.

I realize the overload of `=` without calling the based class, since it is hard to realize in that way.

Honestly, it is of no use to do like that.

Mocha

`Mocha.h`

```
#ifndef MOCHA_H
#define MOCHA_H

#include "espresso_based.h"

class Mocha:public EspressoBased{
public:
    Mocha();
    Mocha(const Mocha& cap);
    ~Mocha();
    void operator=(const Mocha& cap);
```

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

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

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

};

#endif // MOCHA_H
```

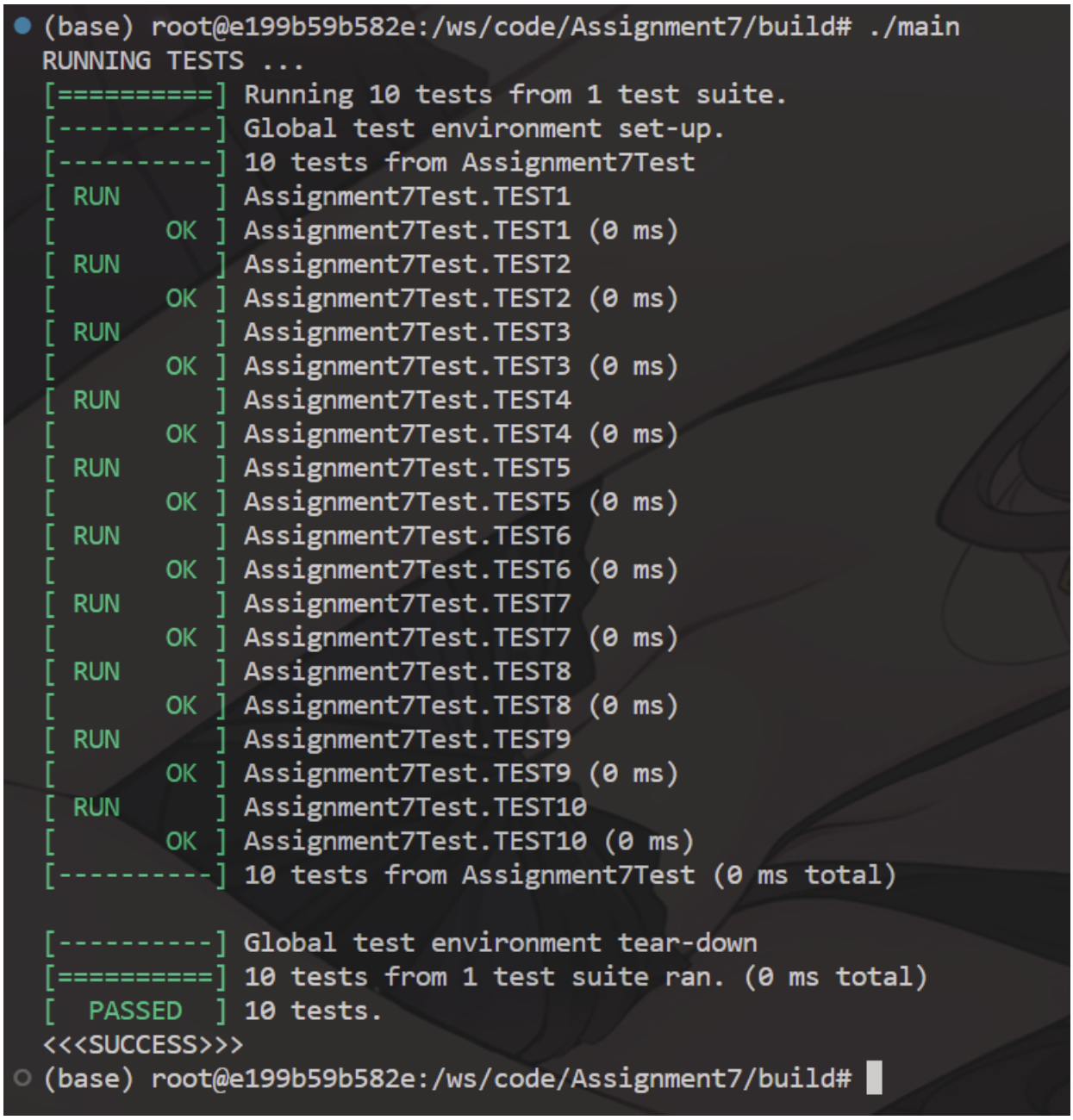
Mocha.cpp

```
#include "../include/mocha.h"

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

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

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

double Mocha::price(){
    double ret;
    for(auto it:ingredients){
        ret+=it->price();
    }
    for(auto it:side_items){
        ret+=it->price();
    }
    return ret;
}

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

Mocha::~Mocha(){
    for(const auto& it:side_items)
        delete it;
```

```
    side_items.clear();
}

Mocha::Mocha(const Mocha& cap):EspressoBased(cap){
    // Mocha::~Mocha();//Avoid using destructors in copying

    for(const auto& it:side_items)
        delete it;
    side_items.clear();

    name=cap.name;

    for(auto it:cap.side_items){
        side_items.push_back(it->newtype());
    }
}

void Mocha::operator=(const Mocha& cap){
    Mocha tmp(cap);
    // Mocha::~Mocha();//Avoid using destructors in copying

    for(const auto& it:side_items)
        delete it;
    side_items.clear();

    for(const auto& i : ingredients)
        delete i;
    ingredients.clear();

    name=tmp.name;
    // ingredients=cap.ingredients;
    for(auto it:tmp.ingredients){
        ingredients.push_back(it->newtype());
    }
    // side_items=cap.side_items;
    for(auto it:tmp.side_items){
        side_items.push_back(it->newtype());
    }
}
```

It is the same with `Cappuccino`, so there is no need to tautology.

So far, this assignment is completed.


(base) root@e199b59b582e:/ws/code/Assignment7/build# ./main
RUNNING TESTS ...
[==========] Running 10 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 10 tests from Assignment7Test
[RUN] Assignment7Test.TEST1
[OK] Assignment7Test.TEST1 (0 ms)
[RUN] Assignment7Test.TEST2
[OK] Assignment7Test.TEST2 (0 ms)
[RUN] Assignment7Test.TEST3
[OK] Assignment7Test.TEST3 (0 ms)
[RUN] Assignment7Test.TEST4
[OK] Assignment7Test.TEST4 (0 ms)
[RUN] Assignment7Test.TEST5
[OK] Assignment7Test.TEST5 (0 ms)
[RUN] Assignment7Test.TEST6
[OK] Assignment7Test.TEST6 (0 ms)
[RUN] Assignment7Test.TEST7
[OK] Assignment7Test.TEST7 (0 ms)
[RUN] Assignment7Test.TEST8
[OK] Assignment7Test.TEST8 (0 ms)
[RUN] Assignment7Test.TEST9
[OK] Assignment7Test.TEST9 (0 ms)
[RUN] Assignment7Test.TEST10
[OK] Assignment7Test.TEST10 (0 ms)
[----------] 10 tests from Assignment7Test (0 ms total)
[----------] Global test environment tear-down
[==========] 10 tests from 1 test suite ran. (0 ms total)
[PASSED] 10 tests.
<<<SUCCESS>>>
(base) root@e199b59b582e:/ws/code/Assignment7/build#