VE280 (23FA) Exercise 6

This exercise focuses on *Dynamic Memory Allocation* and *Invariant*. The deadline is **Dec 3, 2023, 23:59p.m.**

Description

Last week, Kurumi has completed her implementation of the vending machine and her coffee vendor. However, she is tired of throwing an exception when her vending machine goes wrong. This week, she decides to improve the performance on the vending machine by facilitating the dynamic memory allocation and invariant mechanism. She created the following plan of the machine improvement:

Here are some modifications based on the previous exercise:

1. `price`: An dynamic array of positive integer representing the price of each type of food.
2. `empty`: True if there is no food exists in the current vending machine.

To avoid the mistakes and bugs that might occur, Kurumi has set the standard regulation of invariants:

1. The vending machine is empty if and only if all the food has the price of 0.
2. Any type of food that is more delicious than the most delicious food that currently exists must have the price of 0.

Kurumi believes that if the above invariants always hold, she would have no need to throw the exception any more. However, she is too lazy to complete her improvement plan, she would like to ask you for help.

Task

Here are some requirements that you should carefully read to design the new vending machine.

- Vending Machine

  The declaration of `VendingMachine` is provided in `vendingmachine.h`. You need to implement the `VendingMachine` class in your `vendingmachine.cpp`. Do not modify anything in `vendingmachine.h`.

  You can implement the functions based on what you have written in Exercise 5, but be careful to deal with the newly added member `empty` and `repOK()`.

```
#ifndef _VENDINGMACHINE_H
#define _VENDINGMACHINE_H
#include <string>

#define MAX_TYPE 10

static std::string food[10] = {"Latte", "Cappuccino", "Americano", "Milk",
"Chocolate"
                                , "Bread", "Chips", "InstantNoodle",
"Biscuit", "Sandwich"};
```

```
enum FoodType{
    Latte,
    Cappuccino,
    Americano,
    Milk,
    Chocolate,
    Bread,
    Chips,
    InstantNoodle,
    Biscuit,
    Sandwich,
    Exceed,
};

class Exception{

    std::string message="";

public:
    Exception(const std::string & message): message(message) {}
    std::string what() {return message;}
};

class VendingMachine {

protected:
    // dynamic array of the price
    int *price;
    // true if no food exists
    bool empty;
    // the most delicious type of food existing
    FoodType type;

public:

    /**
     * @brief Default Constructer. Initialize all the price to zero,
     * and set this->type to be the least delicious food.
     */
    VendingMachine();

    /**
    * @brief Overloaded Constructer. Initialize a vending machine with one
type of food.
    * @param type
    * @param price
    * @throw If the given degree is greater or equal to MAX_TYPE, throw an
    * Exception object with error message "Exceed maximum type!".
    */
    VendingMachine(FoodType type, int price);

    /**
     * @brief Set the price of the food for the type given,
     * and update the most delicious type.
     * @param type
     * @param price
     * @throw If the given type is greater or equal to MAX_TYPE, throw an
Exception
     * object with error message "Exceed maximum type!".
     */
    virtual void setPrice(FoodType type,int price);

    /**
     * @brief Get the price of the food for the type given.
     *
     * @param type
     * @throw If the given type is more delicious than this->type
     * , throw an Exception object with error message "No food of such
type!".
     */
    int getPrice(FoodType type) const;

    /**
     * @brief Print the menu with format "<food type> $<price>\n" to stdout,
     * from the most delicious type to least one, skip all the food with
price of zero.
     * Each line represents one type of food.
     * Add a new line after printing the whole menu.
     * @example Sandwich $12
     * @example Chocolate $5
     * @example Latte $1
     */
    virtual void print() const;

    /**
     * @return FoodType, the most delicious food existing
    */
    FoodType getType() const;

    bool isEmpty() const;

    void setEmpty(bool empty) { this->empty = empty; };

    void setType(FoodType type) { this->type = type; };
    /**
    * @brief Check the invariants of the class
    * return whether the invariants of class are correct.
    */
    bool repOK();

    /**
    * @brief Default Destructer.
    * Release memory of price.
    */
    ~VendingMachine();

};


#endif // _VENDINGMACHINE_H
```

Testing

To avoid any unexpected results of your program when encountering hidden cases, you can write the following driver program to test your code.

```
int main() {
  int test_num;
  cin >> test_num;
  try {
    switch (test_num) {
    case 1: {
      VendingMachine p;
      if (p.repOK())
        cout << "success";
    } break;
    case 2: {
      VendingMachine p(Latte, 0);
      if (p.repOK())
        cout << "success";
   } break;
    case 3: {
        ...
    } break;
    ...
}
```

Submission

Please compress your `vendingmachine.cpp` into `.zip` file and submit on JOJ.