Зоопарк v1

Пишем класс, который умеет взаимодействовать с другими классами. Причём другие классы могут быть очень разные - иерархия наследования у них может оказаться очень развесистой. Но наш класс общается с ними через единый интерфейс, благодаря чему отдельной обработки всех случаев писать не требуется.

У вас есть зоопарк разных классов. У них есть общий интерфейс. Вот такой:

```
class Animal {
public:
    virtual string getType() = 0;
    virtual bool isDangerous() = 0;
};
```

Сколько и каких конкретных зверей будет потом создано - вы не знаете. Но для вас интерфейс к ним всем будет таким.

Нужно написать класс смотрителя зоопарка. Классу передают на вход зверей, он их всех последовательно осматривает и пересчитывает, сколько из них было опасных. Прототип класса вот такой:

```
class ZooKeeper {
public:
    // Создаём смотрителя зоопарка
    ZooKeeper() {}

    // Смотрителя попросили обработать очередного зверя.
    // Если зверь был опасный, смотритель фиксирует у себя этот факт.
    void handleAnimal(Animal* a);

    // Возвращает, сколько опасных зверей было обработано на данный момент.
    int getDangerousCount();
};
```

Публичные методы должны быть ровно такие. Необходимую protected / private часть класса можете создать себе любую.

Пример ожидаемого сценария работы всей конструкции:

```
ZooKeeper z;
Monkey *m = new Monkey();
z.handleAnimal(m);
delete m;
m = new Monkey();
z.handleAnimal(m);
delete m;
Lion *l = new Lion();
z.handleAnimal(l);
delete l;
cout << z.getDangerousCount() << endl;
```

Должно напечатать 1, так как зверей было 3, но опасный среди них только лев.

Приключенцы v0.1

Учимся писать свои классы, которые унаследованы от предоставленных базовых классов. Наши классы часть своего поведения наследуют от базовых (используют готовые поля и методы из них), а часть поведения приносят свою (содержат реализацию методов, объявленных в родительском классе).

Легенда

Вы с друзьями решили написать игру (очередную, зато свою!), в которой партия героев ходит по волшебному миру и ищет приключений на свою голову. Писать придётся много, так что задачи сразу поделили между участниками разработки. Вам досталось писать героев-приключенцев - классы, расы и всё такое. Как это принято во "взрослой" разработке, вы сразу договорились между собой про интерфейсы между компонентами (детали ниже, в постановке задачи). Теперь нужно реализовывать свою часть, аккуратно соблюдая интерфейсы, которые от вас ожидают ваши друзья. *(Очень аккуратно! Не забывайте - они знают, где вы живёте!)*

На данный момент все сущности у вас в игре ну очень базовые - разработка у вас устроена по agile-у, так что сейчас нужно зарелизить версию 0.1, которая ещё ничего не будет уметь, но уже будет запускаться.

Постановка задачи

Вы договорились, что для версии 0.1 предметы в инвентаре описываются вот так:

```
class Item {
public:
    // Так можно создать предмет, указав его название, вес, уровень и магичность
    Item(string title, int weight, int level, bool magical);
    // Получить вес предмета
    int getWeight();
    // Получить уровень предмета
    int getLevel();
    // Получить, является ли предмет магичным
    int isMagical();
};
```

Сами предметы пишет ваш коллега. Вы ничего не знаете, о том, как они будут устроены внутри. Но точно знаете, что ваш код должен с ними общаться через ровно эти методы.

Базовый класс героя-игрока у вас уже есть. Он вот такой:

```
class Player {
protected:
    // Сила и уровень героя
    int strength;
    int level;
public:
    // Создать героя, все подробности будут указаны позже
    Player() { }
    // Удалить героя, ничего умного эта процедура пока что не требует
    virtual ~Player() { }

    // Базовые методы, пока что очень простые.
    // На данном этапе можно считать, что для всех героев они ведут себя одинаково,
    // так что пусть будут в базовом классе.

    // Задать силу
    void setStrength(int strength) {
        this->strength = strength;
    }
    // Задать уровень
    void setLevel(int level) {
```

```
        this->level = level;
    }

    // Получить силу
    int getStrength() {
        return this->strength;
    }
    // Получить уровень
    int getLevel() {
        return this->level;
    }

    // Проверка, может ли игрок использовать предмет
    virtual bool canUse(Item* item) = 0;
};
```

Этот интерфейс нужно аккуратно соблюдать - весь остальной движок полагается на то, что герои описаны именно так. Базовые методы вообще-то уже готовы. Нужно только аккуратно реализовать метод canUse(...) для разных героев.

Для версии 0.1 от вас ожидается два класса героев - Knight и Wizard. Оба должны быть унаследованы от Player (движок будет обращаться к ним через указатель на Player). В каждом нужно реализовать свой canUse(...). Рыцарь может использовать только немагические предметы и только если (а) сила героя не меньше веса предмета, (б) уровень героя не меньше уровня предмета. Волшебник устроен в целом так же, но магические предметы использовать тоже может.

Движок будет обращаться с вашей реализацией примерно вот так:

```
    Item* items[3];
    items[0] = new Item("Small sword", 1, 1, false);
    items[1] = new Item("Big sword", 5, 3, false);
    items[2] = new Item("Ward", 1, 3, true);

    Player* players[2];
    players[0] = new Wizard();
    players[0]->setStrength(3);
    players[0]->setLevel(5);
    players[1] = new Knight();
    players[1]->setStrength(6);
    players[1]->setLevel(5);

    for(int i = 0; i < 2; i++) {
        for(int j = 0; j < 3; j++) {
            cout << "Can use: " << players[i]->canUse(items[j]) << endl;
        }
    }
```

Итак, задача - реализуйте классы Knight и Wizard под эти требования.

Сопротивление материалов

Дан следующий интерфейс, описывающий поведение материала в терминах "напряжение-деформация".

```
class Material
{
public:
    // Принимает на вход величину деформации.
    // Возвращает величину напряжения, посчитанную с учётом реологии материала.
    virtual float getStress(float strain) = 0;
};
```

Реализуйте данный интерфейс для идеально-упругого материала и для упруго-пластического.
Идеально упругий материал должен быть описан в классе `ElasticMaterial`. Данный класс должен:

- технически корректно реализовать интерфейс `Material`;
- иметь конструктор `ElasticMaterial(float elasticModulus)`, где `elasticModulus` - модуль Юнга;
- реализовать метод `float getStress(float strain)`, где напряжение считается прямо пропорциональным деформации во всём диапазоне значений.

Упруго-пластический материал должен быть описан в классе `PlasticMaterial`. Данный класс должен:

- технически корректно реализовать интерфейс `Material`;
- иметь конструктор `PlasticMaterial(float elasticModulus, float strainLimit)`, где `elasticModulus` - модуль Юнга, а `strainLimit` - предельное значение упругой деформации;
- реализовать метод `float getStress(float strain)`, где напряжение считается прямо пропорциональным деформации в диапазоне деформаций от нуля до `strainLimit`, а при дальнейшем возрастании деформации напряжение более не меняется (материал "течёт").

Код для базового тестирования реализации класса:

```
Material* m;
m = new ElasticMaterial(100);
cout << "Material stress is: " << m->getStress(0.1) << endl;
delete m;
m = new PlasticMaterial(100, 0.01);
cout << "Material stress is: " << m->getStress(0.1) << endl;
delete m;
```

Базовый тест должен вывести:

```
Material stress is: 10
Material stress is: 1
```