Хищник

Задача на получение базовых навыков общения с reference-ами и const-ами.

Дан следующий базовый класс, описывающий зверушку:

```
class Animal
{
protected:
    double mass;
    double speed;

public:
    Animal() {}
    virtual ~Animal() {}

    // Создаём зверушку с заданной массой и скоростью
    Animal(double mass, double speed) {
        this->mass = mass;
        this->speed = speed;
    }

    // Получить массу зверя
    virtual double getMass() const {
        return mass;
    }

    // Получить скорость зверя
    virtual double getSpeed() const {
        return speed;
    }

    // Может ли зверушка нападать на других зверей
    // (не может, зверь мирный)
    virtual bool canAtack() const {
        return false;
    }

    // Съесть другого зверя, не делает ничего, мы же мирные
    virtual void eat(const Animal& a) {
    }
};
```

Напишите класс Predator, унаследованный от базовой зверушки и описывающий хищника. Класс должен:

- Технически корректно унаследоваться от зверушки.
- Перегрузить метод canAtack для возврата true (хищник атаковать других зверей может).
- Перегрузить метод eat, реализовав в нём следующую логику: (1) если переданный зверь больше или быстрее хищника, то ничего не происходит, (2) если переданный зверь не больше и не быстрее хищника, то масса хищника увеличивается на массу переданного зверя.

Код для базового тестирования реализации класса:

```
    Animal* predator = new Predator(1, 10);
    cout << "Current mass: " << predator->getMass() << endl;
```

```
    if(predator->canAtack()) {
        predator->eat(Animal(2, 1)); // Этого не съедим - слишком большой
        cout << "Current mass: " << predator->getMass() << endl;
        predator->eat(Animal(1, 1)); // А вот этого вполне
        cout << "Current mass: " << predator->getMass() << endl;
        predator->eat(Animal(1, 20)); // Этого не выйдет - слишком быстрый
        cout << "Current mass: " << predator->getMass() << endl;
        predator->eat(Animal(2, 2)); // А вот этого догоним
        cout << "Current mass: " << predator->getMass() << endl;
    }

    delete predator;
```

Базовый тест должен вывести:

```
Current mass: 1
Current mass: 1
Current mass: 2
Current mass: 2
Current mass: 4
```

Конвой

Задача на получение базовых навыков общения с reference-ами и const-ами.

Ваша задача - написать класс, формирующий колонну автотранспорта. Вам неизвестно, какой транспорт бывает, но известно, что к любой машине можно обращаться через следующий интерфейс:

```
class Car
{
public:
    // Разрешена ли посадка пассажиров
    bool isBoardingAllowed() const;

    // Разрешена ли погрузка грузов
    bool isLoadingAllowed() const;

    // Сколько в машине мест для пассажиров
    unsigned int getNumberOfSeats() const;

    // Сколько в машине мест для грузовых контейнеров
    unsigned int getNumberOfContainers() const;
};
```

Напишите класс ConvoyManager, который управляет формированием автоколонны. Прототип класса:

```
class ConvoyManager
{
public:
    // Зарегистрировать новую машину в колонне
    void registerCar(const Car& c);

    // Сообщить, сколько всего пассажиров может принять колонна
    unsigned int getTotalSeats() const;

    // Сообщить, сколько всего грузовых контейнеров может взять колонна
    unsigned int getTotalContainers() const;
};
```

При регистрации машины необходимо учитывать не только её ёмкость, но и готова ли она вообще брать пассажиров и грузы.

Для тестирования можете использовать следующую реализацию класса Car:

```
class Car
{
protected:
    bool allowesBoarding;
    bool allowesLoading;
    unsigned int numberOfSeats;
    unsigned int numberOfContainers;

public:
    Car(bool allowesBoarding, bool allowesLoading, unsigned int numberOfSeats,
unsigned int numberOfContainers) {
        this->allowesBoarding = allowesBoarding;
        this->allowesLoading = allowesLoading;
        this->numberOfSeats = numberOfSeats;
        this->numberOfContainers = numberOfContainers;
    }
    ~Car() {}
```

```
    bool isBoardingAllowed() const {
        return allowesBoarding;
    }

    bool isLoadingAllowed() const {
        return allowesLoading;
    }

    unsigned int getNumberOfSeats() const {
        return numberOfSeats;
    }

    unsigned int getNumberOfContainers() const {
        return numberOfContainers;
    }
};
```

Код для базового тестирования реализации класса:

```
    ConvoyManager cm;

    Car c1(true, false, 12, 3);
    cm.registerCar(c1);
    Car c2(false, true, 12, 3);
    cm.registerCar(c2);
    Car c3(true, true, 12, 3);
    cm.registerCar(c3);
    Car c4(false, false, 12, 3);
    cm.registerCar(c4);

    cout << "Total available seats: " << cm.getTotalSeats() << endl;
    cout << "Total available containers: " << cm.getTotalContainers() << endl;
```

Базовый тест должен вывести:

```
Total available seats: 24
Total available containers: 6
```

Приключенцы v0.3

Задача на получение базовых навыков общения с reference-ами и const-ами.

Плюс ещё немного наследования - теперь мы хотим в методах базового класса использовать что-то, что будет доопределено в каждом унаследованном от него сабклассе.

Легенда

Версия 0.2 чудо-игры (с героями и котиками!) была успешно собрана и даже запущена. Пришло время для версии 0.3. В игро-механику решили добавить лекарей *(сражений и ран пока нет, но это неважно)*. После бурных дебатов решили, что делать отдельный класс лекаря не стоит - лечить может любой игрок при выполнении определённых условий.

Поэтому в **базовом** классе появился метод canHeal() - может ли данный конкретный герой кого-либо лечить. Внутрь этого метода в дальнейшем будет спрятана сложная логика *(например, в духе "вообще лечить может, но если устал, то уже не может, а если устал, но прямо очень нужно, то всё-таки может")*.

На сейчас внутри этого метода логика будет предельно проста - лечить могут волшебники (всегда, они же умные) или игроки высокого уровня (если вышло добраться до такого уровня, то уж как-нибудь жизнь должна была научить обращаться с бинтами).

Постановка задачи

Классы из прошлой задачи был немного улучшены: (а) расставили уместные const-ы, (б) заменили pointer-ы на reference-ы, (в) появились методы для реализации обсуждённой выше логики. После рефакторинга всё выглядит вот так (читайте внимательно, реализацию классов героев предстоит обновить, чтобы она собралась после рефакторинга):

```
// Класс предмета
class Item {
protected:
    string title;
    int weight;
    int level;
    bool magical;

public:
    // Так можно создать предмет, указав его название, вес, уровень и магичность
    Item(string title, int weight, int level, bool magical) {
        this->title = title;
        this->weight = weight;
        this->level = level;
        this->magical = magical;
    }

    // Получить вес предмета
    int getWeight() const {
        return weight;
```

```
    }

    // Получить уровень предмета
    int getLevel() const {
        return level;
    }

    // Получить, является ли предмет магичным
    int isMagical() const {
        return magical;
    }
};

// Базовый класс героя
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
    virtual void setStrength(int strength) final {
        this->strength = strength;
    }
    // Задать уровень
    virtual void setLevel(int level) final {
        this->level = level;
    }

    // Получить силу
    virtual int getStrength() const final {
        return strength;
    }
    // Получить уровень
    virtual int getLevel() const final {
        return level;
    }

    // Проверка, может ли игрок использовать предмет.
    // Рыцарь может использовать (а) только немагические предметы и только если
    // (б) сила героя не меньше веса предмета, (в) уровень героя не меньше уровня
предмета.
    // Волшебник устроен в целом так же, но магические предметы использовать тоже
может.
    virtual bool canUse(const Item& item) const = 0;

    // Может ли игрок колдовать, зависит от конкретного класса игрока
    virtual bool canCast() const = 0;

    // Проверка, может ли игрок лечить других игроков.
    // Лечить умеет или любой волшебник, или
    // игрок достаточно высокого уровня.
```

```
    virtual bool canHeal() const {
        return canCast() || getLevel() > 10;
    }
};
```

Новые методы вот такие:

```
    // Может ли игрок колдовать, зависит от конкретного класса игрока
    virtual bool canCast() const = 0;

    // Проверка, может ли игрок лечить других игроков.
    // Лечить умеет или любой волшебник, или
    // игрок достаточно высокого уровня.
    virtual bool canHeal() const {
        return canCast() || getLevel() > 10;
    }
```

В методе canHeal() уже написана ровно изложенная выше логика. Метод canCast() предполагается к реализации в сабклассах и к использованию в базовом классе - ровно в этом вся идея, ради этого всё затеяно.

Вам нужно всего лишь сделать следующее:

- Отрефакторить свои классы Knight и Wizard под новые требования. Метод canUse должен логически работать так же, как и раньше. Но технически его нужно обновить под новый движок.
- Реализовать canCast, аккуратно возвращая true для Wizard и false для Knight. После этого новая логика должна заработать.

Движок будет обращаться с вашей реализацией примерно вот так:

```
    Item items[3] = {
        Item("Small sword", 1, 1, false),
        Item("Big sword", 5, 3, false),
        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++) {
        cout << "Can heal: " << players[i]->canHeal() << endl;
    }

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

    for(int i = 0; i < 2; i++)
        delete players[i];
```

На выходе такого примера ожидается банальное:

```
Can heal: 1
Can heal: 0
Can use: 1
```

```
Can use: 0
Can use: 1
Can use: 1
Can use: 1
Can use: 0
```