### Обёртка

*Классы могут содержать внутри себя не только простые поля (int, double, указатели и т.д.), но и другие классы. Технически ничего нового для этого не требуется - просто поле объявляется как MyClass obj, и вот уже появилось хранимое значение obj (очевидно, оно же доступно как this -> obj), которое является экземпляром класса MyClass.*

Это была напоминалка. Задача вообще-то про namespace-ы.

### Легенда

В своём проекте вы используете несколько больших сторонних библиотек - A, B, C. Каждая из этих библиотек все свои сущности организует внутри своего namespace-а. Для библиотек A, B, C это namespaceA, namespaceB, namespaceC соответственно. Во всех библиотеках описан класс с названием Engine - так уж жизнь сложилась. У всех Engine-ов из всех библиотек есть метод run(). Вам в зависимости от ситуации нужно обращаться к Engine-ам из разных библиотек, не запутавшись во всех этих сущностях.

### Постановка задачи

Напишите класс MyEngine, который будет хранить в себе три Engine-а - по одному экземпляру для каждой из библиотек A, B, C. Ваш класс должен уметь по запросу вызвать нужный Engine.

Прототип публичной части вашего класса:

class MyEngine {

public:

// Если передан параметр 1 - должен быть вызван метод run и Engine-а из библиотеки A.

// Если передан параметр 2 - должен быть вызван метод run и Engine-а из библиотеки B.

// Если передан параметр 3 - должен быть вызван метод run и Engine-а из библиотеки C.

// Если передано что-то иное - должно ничего не произойти.

void run(unsigned int number);

};

На время тестирования можете использовать вот такую конструкцию для эмуляции этих самых сторонних библиотек:

namespace namespaceA {

class Engine {

public:

void run() {

cout << "EngineA run" << endl;

}

};

}

namespace namespaceB {

class Engine {

public:

void run() {

cout << "EngineB run" << endl;

}

};

}

namespace namespaceC {

class Engine {

public:

void run() {

cout << "EngineC run" << endl;

}

};

}

Пример ожидаемого сценария работы всей конструкции:

MyEngine e;

e.run(1); // вызов run из Engine-а из библиотеки A

e.run(2); // вызов run из Engine-а из библиотеки B

e.run(3); // вызов run из Engine-а из библиотеки C

e.run(10); // ничего не происходит

### Хранилище

Напишите класс хранилища на N штук int-ов. (Да, это пока что просто обёртка над массивом - никакого подвоха.)

**Важные условия:**

* Не нужно использовать в этой задаче STL (vector и прочие готовые контейнеры). Просто выделите память, используя malloc или new. Это условие не проверяется в тестах, но если воспользоваться готовым STL, то вы не получите нужного опыта на кончиках пальцев.
* От вашего хранилища будут унаследованы другие классы. И вот тут нужно не забыть подумать про срабатывание деструкторов. Это будет проверяться в тестах.

class Storage

{

public:

// Конструктор хранилища размерности n

Storage(unsigned int n);

// Добавьте нужный деструктор

// Получение размерности хранилища

unsigned getSize();

// Получение значения i-го элемента из хранилища,

// i находится в диапазоне от 0 до n-1,

// случаи некорректных i можно не обрабатывать.

int getValue(unsigned int i);

// Задание значения i-го элемента из хранилища равным value,

// i находится в диапазоне от 0 до n-1,

// случаи некорректных i можно не обрабатывать.

void setValue(unsigned int i, int value);

};

Приблизительный код для тестирования реализованного класса:

// Класс TestStorage, наследуется от вашей реализации Storage

class TestStorage : public Storage {

protected:

// Унаследованная реализация зачем-то хочет выделить ещё памяти. Имеет право.

int\* more\_data;

public:

// В конструкторе память выделяется,

TestStorage(unsigned int n) : Storage(n) {

more\_data = new int[n];

}

// ... а в деструкторе освобождается - всё честно.

~TestStorage() {

delete[] more\_data;

}

};

int main() {

Storage \*ts = new TestStorage(42);

delete ts;

return 0;

}

И проверить этот тестовый код под valgrind-ом.

### Приключенцы v0.2

*Знакомимся с множественным наследованием.*

### Легенда

Версия 0.1 чудо-игры была успешно собрана и даже запущена. Всё идёт неплохо. Но теперь маркетинг утверждает, что срочно нужно добавить в игру котиков, с которыми можно поговорить - пользователи просят. Никто пока не понял, что это за котики, и как конкретно они должны работать. Но у нас же agile - срочно добавляем котиков, а там разберёмся.

Было принято решение, что эти самые котики с точки зрения игро-движка будут одновременно и звери (потому что это логично), и NPC (потому что с ними можно поговорить).

### Постановка задачи

У вас есть интерфейс зверушки. Вот такой:

class Animal {

public:

// Погладить данную зверушку.

// Последствия зависят от реализации данного метода для класса конкретной зверушки.

virtual void pet() = 0;

virtual ~Animal() {};

};

У вас есть интерфейс NPC. Вот такой:

class NPC {

public:

// Поговорить с NPC.

// Что он скажет - зависит от реализации данного метода для конкретного NPC.

virtual void talk() = 0;

virtual ~NPC() {};

};

Нужно реализовать класс SmartCat, который реализует оба интерфейса. И на попытку заговорить, и на попытку погладить SmartCat должен выводить на экран строку "Meow!", завершённую символом конца строки. (Какое ТЗ - такая и реализация.)

Пример ожидаемого сценария работы всей конструкции:

// Примерно так с нашим классом будут обращаться части движка,

// которые вообще-то отвечают за взаимодействие со зверями.

Animal \*a = new SmartCat();

a->pet();

delete a;

// А так - части движка, которые работают с NPC.

NPC \*n = new SmartCat();

n->talk();

delete n;