Умное хранилище

В очередной раз напишите класс хранилища на N штук int-ов. (Да, это опять обёртка над массивом или аналогом.) Теперь хранилище при некорректных запросах должно не падать, а осмысленно ругаться, выбрасывая исключения. Прототип класса хранилища:

```
class Storage
{
public:
    // Конструктор хранилища размерности n
    Storage(unsigned int n);

    // Добавьте деструктор, если нужно

    // Получение размерности хранилища
    unsigned int getSize() const;

    // Получение значения i-го элемента из хранилища
    // Если индекс некорректен, нужно выбросить IncorrectIndexException
    int getValue(unsigned int i) const;

    // Задание значения i-го элемента из хранилища равным value
    // Если индекс некорректен, нужно выбросить IncorrectIndexException
    void setValue(unsigned int i, int value);
};
```

Важно: при обращении по некорректному индексу нужно выбрасывать именно IncorrectIndexException. Это нестандартный класс exception-а, он описывает для окружающего кода суть возникшей проблемы. Этот класс exception-а предстоит определить.

Приблизительный код для тестирования реализованного класса:

```
int main() {
    unsigned int index;
    std::cin >> index;
    Storage s(42);
    s.setValue(index, 0);
    std::cout << s.getValue(index) << std::endl;
    return 0;
}
```

Важно: при локальной проверке стоит прогнать тестовый код под valgrind-ом. И ещё стоит накидать разных exception-ов в уместные проблемные моменты, чтобы убедиться, что они обработаны.

Нестабильное подключение

У вас есть готовый компонент, который позволяет подключаться к удалённым серверам и что-нибудь им отправлять. (На самом деле нет. На самом деле это заглушка. Но это неважно.) Интерфейс класса вот такой:

```
class Connector {
public:
    // Конструктор. Получает параметром адрес, куда требуется подключиться.
    // Выполняет необходимую инициализацию соединения.
    // После завершения работы конструктора соединение установлено и готово к работе.
    // Если возникают какие-либо проблемы - выбрасывает exception с описанием про-
блемы.
    Connector(const string& address) {
    }

    // Отправить по установленному соединению сообщение, текст сообщения передаётся в
параметре data.
    // Если возникают какие-либо проблемы - выбрасывает exception с описанием про-
блемы.
    void sendRequest(const string& data) {
    }
};
```

Напишите программу, которая:

- Читает с клавиатуры список адресов, к которым нужно подключиться.
- Отправляет каждому из адресов сообщение HELLO, используя для отправки класс Connector.
- Корректно обрабатывает exception-ы, которые может выбросить Connector. Если упало одно подключение, это не должно мешать последующим обращениям к другим адресам.
- Если отправка сообщения прошла до конца успешно (то есть строго после завершения sendRequest), программа выводит на экран очередной адрес, после него двоеточие и пробел, после чего ok.
- Если отправка сообщения не удалась по той или иной причине, программа выводит на экран очередной адрес, после него двоеточие и пробел, после чего текст ошибки, полученный из exception-а.

На время тестирования можете произвольным образом реализовать методы класса Connector, выбрасывая какие-нибудь исключения.

Формат входных данных

На первой строке вводится целое число N - количество адресов. Далее вводятся N строк с адресами.

Формат выходных данных

N строк, на каждой либо "очередной_адрес: ok", либо "очередной_адрес: сообщение_об_ошибке".

При тестировании между вашими строками будут видны дополнительные строки с логами класса Connector. Это нормально.

Примеры

Ввод	Вывод
3 good.com bad.com ugly.com	good.com: ok bad.com: connection refused <-- текст "connection refused" получен из пойманного exception-а ugly.com: ok

Общий мозг

Легенда

Вы с друзьями продолжаете писать свою игру. И вот настал момент, когда в игре должны появиться орды монстров. Начать решили с гоблинов. Гоблины - существа недолговечные, поэтому создавать их придётся тысячами. Но гоблины будут умные и хитрые - мозг для них уже написан, и он использует все последние достижения в области искусственного интеллекта.

И тут на этапе сведения компонентов и интеграционного тестирования возник нюанс. Мозг гоблина в оперативной памяти весит примерно 80 мегабайт. Гоблинов будут нужны тысячи. Ну ладно, хотя бы одна тысяча. Кажется, в системных требованиях придётся указать "не менее 96 гигабайт оперативной памяти". Что же делать? К счастью, ответ был быстро найден - пусть у гоблинов будет один мозг на всех. Хотя нет. Всё-таки один мозг на каждую армию гоблинов. Ну что ж, осталось это реализовать.

Постановка задачи

Мозг гоблина у вас уже есть. На этапе отладки гоблинов можете использовать вот такой макет мозга:

```
class Brain {
protected:
    // Здесь хранится что-то очень ценное.
    // Наверное, набор волшебных чисел для нейронной сети, управляющей гоблином.
    vector<double> data;

    // А здесь просто фраза, которой гоблин откликается по умолчанию
    string phrase;

public:
    // Конструктор мозга
    Brain() {
        // Мозг занимает в памяти много места, да
        data.resize(1000000);
        // Установим эталонную фразу
        phrase = "Booyahg Booyahg Booyahg";
    }

    // Мозг умеет подсказать гоблину, какую фразу выдать
    string speak() {
        return phrase;
    }
};
```

От вас требуется:

- написать класс гоблина, при этом каждый гоблин содержит мозг в том или ином виде (указатель, что-то иное - на ваш выбор);
- написать функцию create_goblin_army, которая создаёт армию гоблинов заданного размера, при этом каждая армия имеет свой экземпляр мозга (один на всю армию, к которому могут обратиться все гоблины армии - личный мозг для каждого гоблина считается непозволительной роскошью);

- обеспечить надёжное управление памятью - мозг армии существует, пока цел хотя бы один гоблин из этой армии, при исчезновении последнего гоблина армии мозг армии также удаляется.

Класс Goblin ожидается со следующим прототипом:

```
class Goblin {
public:
    // Подходящие конструкторы и деструкторы

    // Просто вернуть фразу, которую гоблину подсказывает мозг
    // (Метод используется для проверки, что голова гоблина содержит правильный мозг)
    string speak();
};
```

Функция create_goblin_army ожидается со следующим прототипом:

```
// Получает size - требуемый размер армии. Возвращает вектор гоблинов требуемого размера.
// На всю армию создаёт один мозг, к которому имеют доступ все гоблины армии.
// Мозг можно создать просто как Brain(), либо new Brain(), либо любым другим способом создания объекта.
vector<Goblin> create_goblin_army(unsigned int size);
```

Ожидается, что следующий пример корректно исполнится без утечек памяти:

```
int main()
{
    unsigned int size1 = 1;
    unsigned int size2 = 10;
    vector<Goblin> army1 = create_goblin_army(size1);
    vector<Goblin> army2 = create_goblin_army(size2);

    for(unsigned int i = 0; i < size1; i++) {
        cout << army1[i].speak() << endl;
    }
    for(unsigned int i = 0; i < size2; i++) {
        cout << army2[i].speak() << endl;
    }

    return 0;
}
```

Ремарки к задаче

1. Если вы ненароком создадите слишком много мозгов, переполнив память, то ошибка, скорее всего, выглядеть будет не очень внятно - начало коректного вывода, потом вывод молча обрывается, на тесте стоит красный крестик. Такое поведение означает, что программа превысила лимиты ресурсов и в какой-то момент была принудительно завершена.
2. Рекомендуется решать задачу с использованием материала лекции. Тем не менее, альтернативные решения в принципе возможны, и тоже будут засчитаны.
3. Если образ армии гоблинов с одним мозгом на всех вам не очень нравится, можете мысленно заменить их на Белых Ходоков под управлением Короля Ночи, на Рой зергов под управлением Сверхразума или даже на группу студентов в процессе сдачи.