**Рынк Даниил Витальевич группа 931921**

**Часть 1.**

1. Постановка задачи

Необходимо запрограммировать шаблонный класс, реализующий стек. Класс должен поддерживать следующие операции:

1. Помещение объекта в стек;

2. Извлечение объекта из стека;

3. Получение размерности стека.

В случае попытки вызова операции извлечение объекта из стека при условии, что стек пуст, должно генерироваться исключение класса EStackEmpty(наследник класса EStackException , пример приведен в документе ForTask\_1 ). Данный класс должен содержать публичный метод char\* what(), возвращающий диагностическое сообщение.

1. Предполагаемое решение

Стек это структура хранения данных, представляющий собой упорядоченный (по времени помещения в стек) набор элементов .

Связаны элементы так: каждый элемент указывает на тот, который нужно использовать следующим. Это линейная связь — данные идут друг за другом и нужно брать их по очереди. Из середины стека брать нельзя.

Главный принцип работы стека — данные, которые попали в стек недавно, используются первыми. Чем раньше попал — тем позже используется. После использования элемент стека исчезает, и верхним становится следующий элемент.

В качестве внутренний реализации стека я выбрал односвязный список состоящий из узлов (Node). Также в нем у меня присутствует 2 поля: Значение и указатель на следующий узел.

Класс Stack хранит список как указатель на верхний элемент списка(который является верхним элементом стека) и размер самого стека.

Рассмотрим реализации функций Stack:

1. Два конструктора. Один по умолчанию. Второй конструктор копирования
2. Функция помещения объекта в Стек – push.
3. Функция удаления верхнего объекта из Стека – pop. \*
4. Функция извлечения верхнего объекта из Стека без удаления – topFunc. \*
5. Функция для проверки на пустоту Стека – empty.
6. Функция получения размерности Стека – getSize.

\* - на этих функциях есть проверка, если стек будет пустой, то тогда вызовется исключение.

1. Коды программ

Все файлы находятся в репозитории: <https://github.com/rynkdesign/TRPO_lab1>

А именно:

Node.h  
Stack.h

Exception.cpp

1. Инструкция пользователя

При запуске программы появляется диалоговое окно, в которое предлагается ввести цифру от 1 до 5 которая обозначает Номер тестирования класса Стек.

В качестве простого примера в программе реализован Стек целочисленных переменных типа int.

Виды тестирования:

1 – Получение размера пустого стека.

2 – Вызов функции Top в пустом стеке.

3 – Вызов функции Pop в пустом стеке.

4 – Добавление элемента “3” в стек, отображение верхнего элемента и размера стека.

5 – Добавление 2ух элементов “3” и “5” в стек и отображение верхнего элемента и размера стека.

После вызова какого-то из Тестирований на экран будет выведена соответствующая информация по отработке класса Стек в конкретном случае.

1. Тестирование

Для тестирования в функции main реализован простой выбор типа теста.  
Задаём наш Стек типа целочисленных переменных и выбираем тип тестирования.

Stack<int> massiv;

Виды тестирований:

// Тестирование 1 – получаем Размер стека

std::cout<<"Test 1"<< std::endl;

std::cout<<"--------------------"<< std::endl;

std::cout<<"Razmer: "<<massiv.getSize()<< std::endl;

break;

// Т.к. в стек мы ничего не записывали, то получаем размер который задаётся по умолчанию – 0. (Рабочий вариант)

// Тестирование 2 - вызовем функцию Top в пустом стеке

std::cout<<"Test 2"<< std::endl;  
 std::cout<<"--------------------"<< std::endl;

std::cout<<"Top: "<<massiv.topFunc()<< std::endl;

break;

// При вызове функции Top в пустом стеке мы получим ошибку, которая сообщает, что в пустом стеке нельзя вызывать функцию Top, т.к. у нас нет элемента для вывода. (Ошибка)

// Тестирование 3 - вызовем функцию pop в пустом стеке

std::cout<<"Test 3"<< std::endl;

std::cout<<"--------------------"<< std::endl;

massiv.pop();

break;

// При вызове функции Pop в пустом стеке мы получим ошибку, которая сообщает, что в пустом стеке нельзя вызывать функцию Pop, т.к. у нас нет элемента для удаления. (Ошибка)

// Тестирование 4 - Добавим элемент в Стек и покажем, что он добавился и размер увеличился

std::cout<<"Test 4"<< std::endl;

std::cout<<"--------------------"<< std::endl;

massiv.push(3);

std::cout<<"Razmer: "<<massiv.getSize()<< std::endl;

std::cout<<"Top: "<<massiv.topFunc()<< std::endl;

break;

// На выходе получим Стек в который мы добавили число 3. В консоль выведется его размер – 1 и Его верхний (и единственный элемент) – 3. (Рабочий вариант)

// Тестирование 5 - Добавим ещё один элемент в Стек и покажем, что функция Top работает корректно

std::cout<<"Test 5"<< std::endl;

massiv.push(3);

massiv.push(5);

std::cout<<"Razmer: "<<massiv.getSize()<< std::endl;

std::cout<<"Top: "<<massiv.topFunc()<< std::endl;

std::cout<<"-----------------------------------"<< std::endl;

break;

// Мы сначала ввели 3, а затем 5. Поэтому на выходе получим Размер - 2 и верхний элемент (Последний записанный) – 5. (Рабочий вариант)

На этом все возможные варианты отработки класса Стек закончены.

**Часть 2.**

1. Постановка задачи

Необходимо реализовать класс *PersonKeeper* с методами *readPersons* и *writePersons.* Метод *readPersons* должен считывать информацию о людях из входного потока (файла), создавать на основе этой информации объекты класса *Person*, и помещать их в стек. Формат входного файла должен быть такой:  
  
**Фамилия Имя Отчество**

В качестве разделителей могут выступать пробелы, табуляции, переводы строки.

**Пример файла:**

**Иванов Василий Иванович**

**Сидоров Александр Михайлович**

…

Метод readPersons должен возвращать стек.

Метод writePersons должен записывать в поток из стека (стек передается аргументом) информацию о людях в соответствии с вышеописанным форматом. Передаваемый методу writePersons стек не должен изменяться.

Класс PersonKeeper должен быть реализован в соответствии с шаблоном Singleton.

1. Предполагаемое решение

В качестве решения я реализовал класс **PersonKeeper** , который отвечает за чтение либо запись Стека по информации о людях из входного файла, на основе этих данных он создаёт объекты класс **Person,** в которых хранятся ФИО людей и помещает их в стек.

Класс **PersonKeeper** я реализовал используя шаблон Singleton (Одиночка). **Singleton –** это порождающий паттерн, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа. То есть Одиночка скрывает все способы создания нового объекта, кроме **специального метода - Instance.** Она возвращает ссылку на этот единственный статический объект. Т.к. объект у нас единственный и он же создаётся в Instance, то привычным методом через Объект его вызвать не получится, поэтому Instance – статический метод и мы можем его реализовывать вне объекта.

В нашем случае этот паттерн реализован для того чтобы у нас был только единственный экземпляр класса PersonKeeper. Он создаётся только 1 раз в методе Instance.  
  
Класс **PersonKeeper** имеет следующие методы:

**Instance** – статический метод, возвращает ссылку на единственный экземпляр PersonKeeper;

**readPersons** – метод, получающий на вход ссылку на входной поток. Он считывает из потока информацию, и на ее основе создает и возвращает стек Person.

**writePersons** – метод, принимающий стек и выходной поток. В методе считывается информация из стека(стек не изменяется), которая записывается в выходной поток.

А также конструктор по умолчанию и конструктор копирования.

Также для удобства перемещения пришлось добавить в класс Stack – operator[], с помощью которого мы можем передвигаться по элементам Стека для чтения из файла.

1. Коды программ  
   Все файлы находятся в репозитории: https://github.com/rynkdesign/TRPO\_lab1

А именно:

person.h  
person.cpp

PersonKeeper.h  
PersonKeeper.cpp  
файлы для тестирования: main.cpp и текстовые файлы: input.txt и output.txt

1. Инструкция пользователя

При запуске программы появляется диалоговое окно, в которое предлагается ввести цифру от 1 до 5 которая обозначает Номер тестирования класса PersonKeeper.

Во время тестирования у нас будут читаться различные файлы формата “.txt”, в которых данные реализованы так, чтобы проверить весь функционал записи и вывода ФИО с одного файла в другой.

Виды тестирования:

1 – Мы не можем открыть файл на чтение (Его нет) - "0input.txt"

2 – Файл существует, но файл пустой. - "empty.txt"

3 – Файл существует, данные введены корректно. - "normal.txt"

4 – Файл существует, но введено больше данных, чем 3 (ФИО состоит из 3) - "4item.txt"

5 – Файл существует, но некоторые данные введены через запятую - "commas.txt"

6 - Файл существует, но некоторые данные введены с некорректными символами - "num-sym.txt"

После вызова какого-то из Тестирований на экран будет выведена соответствующая информация по отработке класса PersonKeeper в конкретном случае.

Тестирование  
Тестирование реализовано в файле main.cpp  
Оно представляет собой попытки передать разные входные данные и посмотреть, как на это отреагирует наш класс.  
  
В файле для чтения в формате “.txt” у нас находятся ФИО и символы. Мы сначала считываем данные из файла, записываем их в Стек, а потом записываем их из Стека в файл “output.txt”  
И по ходу этого дела у нас возникают разные ситуации.  
  
Виды тестов:  
**// Тестирование 1**. Мы не можем открыть файл на чтение (Его нет)

std::fstream File; // Открываем файл

File.open("0input.txt", std::ios\_base::in); // Открываем его на чтение

Stack<Person> Persons = PersonKeeper::instance().readPersons(File); // Считываем

std::fstream File1; // Открываем файл1

File1.open("output.txt", std::ios\_base::out); // Открываем его на запись

PersonKeeper::instance().writePersons(Persons, File1); // Записываем

break;

// Появляется ошибка сообщающая о проблеме чтения, такого файла нет. (Ошибка)

**// Тестирование 2**. Файл существует, но файл пустой.

std::fstream File; // Открываем файл

File.open("empty.txt", std::ios\_base::in); // Открываем его на чтение

Stack<Person> Persons = PersonKeeper::instance().readPersons(File); // Считываем

std::fstream File1; // Открываем файл1

File1.open("output.txt", std::ios\_base::out); // Открываем его на запись

PersonKeeper::instance().writePersons(Persons, File1); // Записываем

break;

// На выходе получаем пустой файл "output.txt" (Рабочий вариант)  
  
**// Тестирование** **3**. Файл существует, данные введены корректно.

std::fstream File; // Открываем файл

File.open("normal.txt", std::ios\_base::in); // Открываем его на чтение

Stack<Person> Persons = PersonKeeper::instance().readPersons(File); // Считываем

std::fstream File1; // Открываем файл1

File1.open("output.txt", std::ios\_base::out); // Открываем его на запись

PersonKeeper::instance().writePersons(Persons, File1); // Записываем

break;

// На выходе получаем файл "output.txt" с корректно переписанными ФИО (Рабочий вариант)  
  
**// Тестирование 4**. Файл существует, но введено больше данных, чем 3 (ФИО состоит из 3)

std::fstream File; // Открываем файл

File.open("4item.txt", std::ios\_base::in); // Открываем его на чтение

Stack<Person> Persons = PersonKeeper::instance().readPersons(File); // Считываем

std::fstream File1; // Открываем файл1

File1.open("output.txt", std::ios\_base::out); // Открываем его на запись

PersonKeeper::instance().writePersons(Persons, File1); // Записываем

break;

// На выходе получаем файл "output.txt" с корректно переписанными ФИО, лишнее данные записываться не будут. (Рабочий вариант)  
  
**// Тестирование 5**. Файл существует, но некоторые данные введены через запятую

std::fstream File; // Открываем файл

File.open("commas.txt", std::ios\_base::in); // Открываем его на чтение

Stack<Person> Persons = PersonKeeper::instance().readPersons(File); // Считываем

std::fstream File1; // Открываем файл1

File1.open("output.txt", std::ios\_base::out); // Открываем его на запись

PersonKeeper::instance().writePersons(Persons, File1); // Записываем

break;

// На выходе получаем файл "output.txt" с корректно переписанными ФИО, данные записанные через запятую не будут записываться. (Рабочий вариант)  
  
**// Тестирование 6**. Файл существует, но некоторые данные введены с некорректными символами

std::fstream File; // Открываем файл

File.open("num-sym.txt", std::ios\_base::in); // Открываем его на чтение

Stack<Person> Persons = PersonKeeper::instance().readPersons(File); // Считываем

std::fstream File1; // Открываем файл1

File1.open("output.txt", std::ios\_base::out); // Открываем его на запись

PersonKeeper::instance().writePersons(Persons, File1); // Записываем

break;

// На выходе получаем файл "output.txt" с корректно переписанными ФИО, данные с некорректными символами записываться не будут. (Рабочий вариант)