Шарко Полина Алексеевна. Группа 931920.

Лабораторная №1 состоит из нескольких частей

# **Часть 1. Реализация шаблонного класса стек.**

## **1. Постановка задачи.**

Запрограммируйте 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.

## **2. Предлагаемое решение.**

Теория:

Стек - структура данных, являющаяся упорядоченным набором элементов, в которой удаление и добавление элементов производится с одного конца (вершины стека). Из определения следует, что обрабатывать при работе со стеком можно только элемент в вершине стека.

Определим способ реализации стека. Данные в стеке связаны линейно, из середины элемент брать нельзя. Учитывая эту особенность, для задачи мы используем односвязный список, т.к.

1. односвязный список занимает меньший объем памяти чем, например, динамический массив

2. односвязный список больше других структур данных отвечает принципу LIFO,

в соответствии с которым организован стек.

Реализуем шаблонный класс Stack. Шаблонный класс - параметризованный класс, в качестве параметров которому передаются типы данных. Шаблонные классы позволяют уменьшить повторяемость кода. Определения элементов шаблона класса должны быть видны компилятору, когда они будут нужны. Поэтому оставляем и объявления, и определения в файле заголовка.

Реализация:

Реализованы класс Stack и структура Node.

Структура Node была добавлена для реализации односвязного списка. Структура используется только внутри класса Stack и только как privatе-член, поэтому рационально объявить структуру в определении класса. В нашем случае стек будет состоять из узлов типа Node. Node имеет два поля: значение и указатель на следующий узел.

Объект класса Stack хранит размер стека и указатель на элемент в вершине стека.

Функции, реализованные в классе Stack:

Конструкторы :

**Stack**(const T\* array = nullptr, int n = 0) - конструктор из массива - создает новые узлы стека на основе данных массива

**Stack**(const Stack <T>&copy) - конструктор копирования - копирует узлы переданного стека. const - чтобы не повредить данные, передача ссылки - для оптимизации работы с памятью.

~**Stack**() - деструктор - удаляет головной узел, пока узлы есть.

void **Push**(const T &val) - метод помещения объекта в стек - создаем новый узел и ставим его в голову списка. const - чтобы не повредить данные, передача ссылки - для оптимизации работы с памятью.

const T **Pop**() - функция, позволяет удалять элемент из стека (из головы стека). При применении к пустому стеку вызывается исключение.

int **Size**() - метод получение размера стека. Возвращает число узлов стека.

void **Print**() - метод вывода стека. Выводит узлы стека начиная с (...) в консоль.

bool **Empty**() - функция, проверка на пустоту стека. Возвращает 1 если в стеке нет узлов, 0 - если узлы есть.

void **Clear**() - функция для освобождения стека. Удаляет все узлы начиная с головного пока стек не пуст.

Функции, реализованные в классе EStackException - обработка исключений при использовании стека, сообщают об ошибке, выводя строку с описанием исключения.

**EStackException**(const char \*msg) - конструктор, принимает сообщение, записывает его в соответствующее поле класса. const - чтобы не повредить передаваемые данные

**EStackException**(const EStackException &obj) - конструктор копирования, переписывает сообщение из переданного объекта класса. const - чтобы не повредить передаваемые данные

~**EStackException**() - деструктор - удаляет сообщение, освобождает память

const char\* **what**() const - метод, возвращающий диагностическое сообщение

Функции, реализованные в классе EStackEmpty - обработка исключений при пустом стеке, сообщают об ошибке, выводя строку с описанием исключения. Класс-наследник EStackException.

**EStackEmpty**():EStackException("This stack is empty. ") - конструктор

**3. Коды программ**

В репозитории представлены коды программ. Файлы предложенной реализации:

Stack.h - содержит описание интерфейса класса стека. Stack.cpp - содержит реализацию методов класса стек.

EStackException.h - описание и реализация методов класса EStackException.

EStackEmpty.h - описание и реализация методов класса EStackEmpty.

main.cpp - содержит тесты класса стек.

**4. Инструкция пользователя**

Класс Stack.

1. Класс шаблонный, поэтому при использовании необходимо указывать используемый тип данных, хранящихся в узлах.

Н-р: Stack<int>stack;

1. Примеры использования методов:

В этой реализации стека пользователь может

Создать стек:

Конструкторы:

конструктор из массива -

при,например,

int ar[] = {3, 2, 1};

int n = 3;

функция:

Stack<int> stack(array, n);

конструктор копирования - Stack<int>stack(stack1);

void Push(const T &val) - stack.Push(value\_of\_node);

T Pop() - stack.Pop(); - сработает исключение, если stack пуст

int Size() - stack.Size();

void Print() - stack.Print();

bool Empty() - stack.Empty();

void Clear() - stack.Clear();

**5. Тестирование**

Протестируем функции, реализованные в Stack, для случаев пустого и непустого стека.

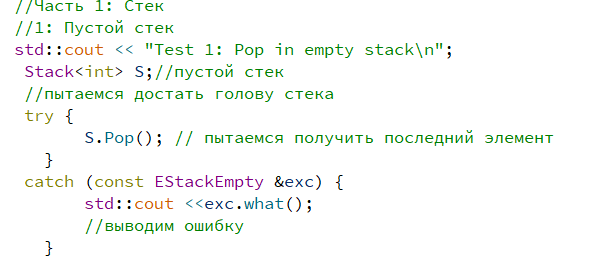
Сценарий тестирования:

| Метод | Состояние стека | |
| --- | --- | --- |
| стек пуст | стек не пуст |
| void **Push**(const T &val) | узел помещен в стек | узел помещен в стек |
| const T **Pop**() | error | верхний узел удален |
| int **Size**() | 0, возвращен размер стека | возвращен размер стека |
| void **Print**() | выводит пустую строку, выведен стек | выведен стек |
| bool **Empty**() | выведено: 1 | выведено: 0 |
| void **Clear**() | из стека ничего не удалено, стек пуст. стек освобожден | стек освобожден |

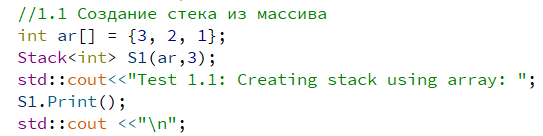
\*error - выводится сообщение об ошибке

В таблице представлено ожидаемое поведение методов. Результаты тестирования совпадают с предполагаемыми.

Тестирование:



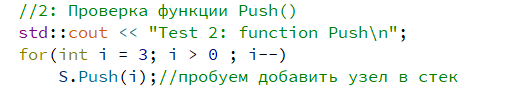
**Вывод:** - ошибка, показывает, что в пустом стеке нельзя взять начальный узел



**Вывод:**

****

-стек создан



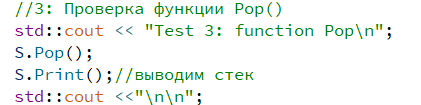
Продемонстрируем заполнение стека выводом значений узлов стека с использованием функции Print() класса Stack.



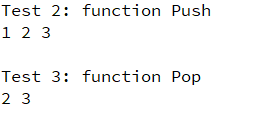
**Вывод:**



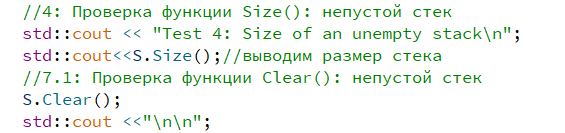
- рабочий вариант, показывает значения добавленных в стек узлов



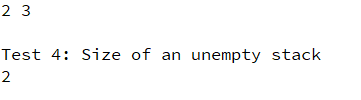
**Вывод:**Для стека из теста 2:



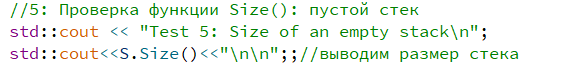
- рабочий вариант, узел был удален



**Вывод:** Для стека из теста 3:



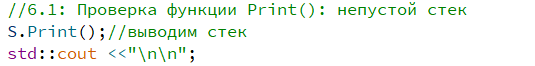
- рабочий вариант, показывает размер стека



**Вывод:**



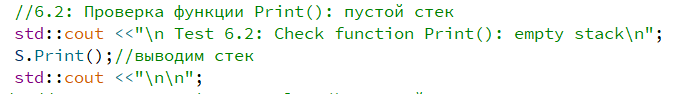
- рабочий вариант, размер пустого стека действительно равен нулю



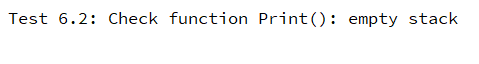
**Вывод:**



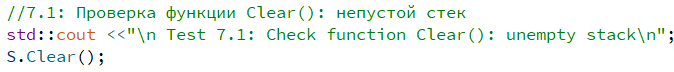
* выведены значения стека



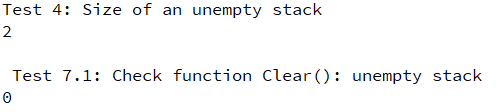
**Вывод:**



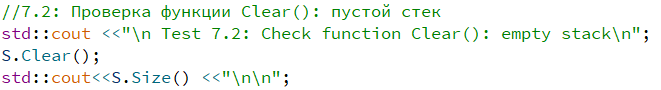
* ничего не выведено, как и должно быть



**Вывод:** Начальный стек - стек из пункта 4



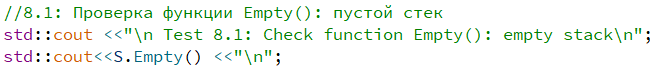
* стек стал пустым



**Вывод:**



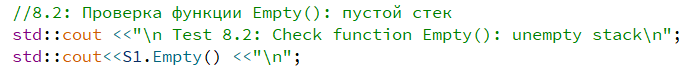
* при очистке пустого стека получаем пустой стек



**Вывод:**

****

* 1-стек пуст. Результат верен



**Вывод:**

****

* 0-стек не пуст. Результат верен

# **Часть 2. Реализация класса PersonKeeper.**

## **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.

## **2. Предлагаемое решение.**

Класс PersonKeeper должен быть реализован в соответствии с шаблоном Singleton. Рассмотрим шаблон подробнее.

Паттерн Singleton нужен для обеспечения доступа к уникальному объекту из любой точки приложения. Объект уникален в смысле существования единственного экземпляра объекта, новые экземпляры быть созданы не могут. Требование уникальности значимо, так как без него объект класса паттерна Singleton будет аналогичен статической переменной (переменной, видимой в ограниченном блоке или в файле и существующей в течение работы программы). Паттерн позволяет снизить потребление ресурсов.

При реализации необходимы методы

-instance: Singleton - статический открытый метод. Создается ссылка на объект класса.

-Singleton() - Конструктор класса Singleton закрытый - это необходимо для возможности гарантировать единственность экземпляра, иначе пользователи могли бы создать новый экземпляр, что не позволило бы шаблону выполнять основную свою цель.

Для реализации PersonKeeper необходим изменяемый Singleton.

Чаще всего шаблон Singleton применяется, когда необходимо предоставить доступ к общему ресурсу - например, к стеку ФИО в случае приложения, в котором класс PersonKeeper реализуется в соответствии с шаблоном Singleton.

Для решения задачи были реализованы классы:

* Person - класс, объект которого хранит ФИО человека из входного файла.
* PersonKeeper - класс-стек, в узлах которого записаны ФИО людей из входного файла.

Класс Person

Поля:

std::string lastName\_ - имя, закрытое

std::string firstName\_ - фамилия, закрытое

std::string patronymic\_ - отчество, закрытое

Методы:

**Person**(){lastName\_=""; firstName\_=""; patronymic\_="";} - конструктор по умолчанию

**Person**(const std::string& full\_name) - конструктор из строки

const std::string &**getLastName**() - получение имени

void **setLastName**(const std::string lastName) - изменение имени

const std::string &**getFirstName**() - получение фамилии

void **setFirstName**(const std::string &firstName) - изменение фамилии

const std::string &**getPatronymic**() - получение отчества

void **setPatronymic**(const std::string & patronymic) - изменение отчества

Класс PersonKeeper

Методы:

**PersonKeeper**() - конструктор по умолчанию

~**PersonKeeper**()-деструктор

static PersonKeeper& **instance**(); - статический открытый метод. Создается ссылка на объект класса

Stack<Person> **readPersons**(std::ifstream& file); - запись ФИО из файла в стек

void **writePersons**(Stack<Person> s, std::fstream& file); - запись ФИО из стека в файл

**3. Коды программ**

В репозитории https://github.com/PSh-2022/Labs.git представлены коды программ. Файлы предложенной реализации:

person.h

person.cpp

person\_keeper.h

person\_keeper.cpp

main.cpp - файл для тестирования

Дополнительные файлы для тестирования:

file.txt - исходный файл, содержит ФИО

incor.txt - файл с некорректными данными

file1.txt - пустой файл

**4. Инструкция пользователя**

Примеры использования методов класса PersonKeeper:

Чтение из файла в стек:

Для открытого на чтение файла:

File.open("file.txt", std::ios\_base::in);

Применяем функцию:

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

Запись из стека в файл:

Для открытого на запись файла:

File1.open("file.txt",std::ios\_base::out);

И стека:

Stack<Person> stack(array,n)

(стек может быть и пуст, тогда файл будет также пуст)

Применяем функцию:

PersonKeeper::instance().writePersons(stack, File1);

**5. Тестирование**

**Сценарий тестирования:**

|  | Особенности файла | | | |
| --- | --- | --- | --- | --- |
| файл не существует | файл пуст | данные введены некорректно | файл существует, данные корректны |
| readPersons | error | записывает в стек пустой файл | error | читает данные из файла |
| writePersons | файл будет создан и заполнен | записывает данные из стека в пустой файл | записывает данные из стека, удаляя данные файла | записывает данные из стека, удаляя данные файла |

\*error - выводится сообщение об ошибке

Тестирование:

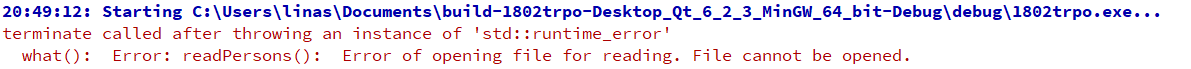
Для всех тестов: 

1. Файл не существует

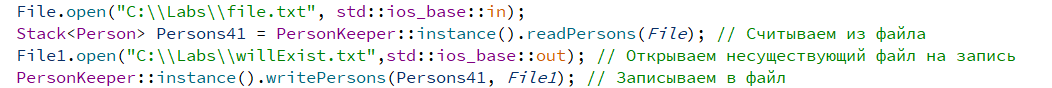
readPersons:



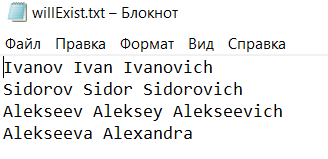
**Вывод**:



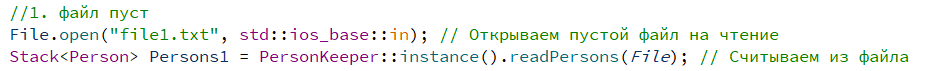
writePersons:



**Вывод**:

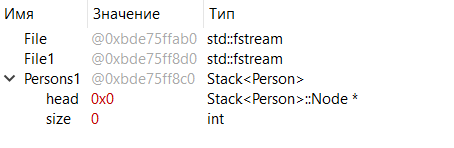


1. Файл пуст

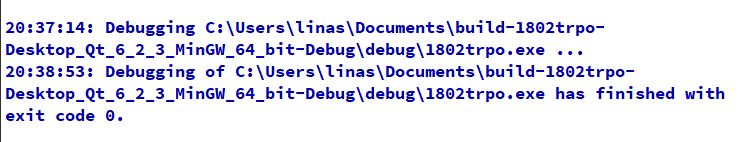
readPersons:

**Вывод**:

Стек с записанными из пустого файла значениями:



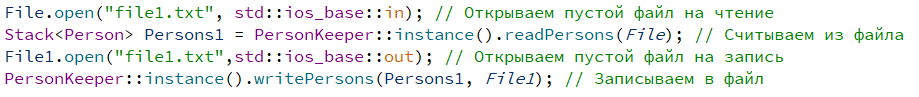
Вывод программы:



-стек остался пустым

writePersons:

Тест при пустом стеке. Аналогичный тест для непустого стека приведен в п.4



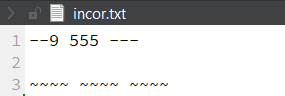
**Вывод**:



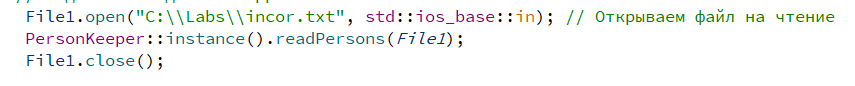
1. Данные введены некорректно

readPersons:

Файл:



Тест:

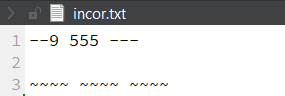


**Вывод**:

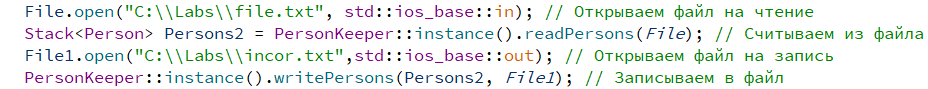


writePersons:

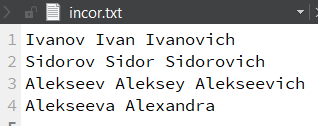
Файл:



Тест:



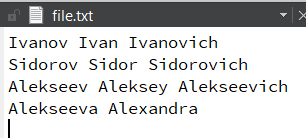
**Вывод**:



1. Файл существует, данные корректны

readPersons:

Файл:

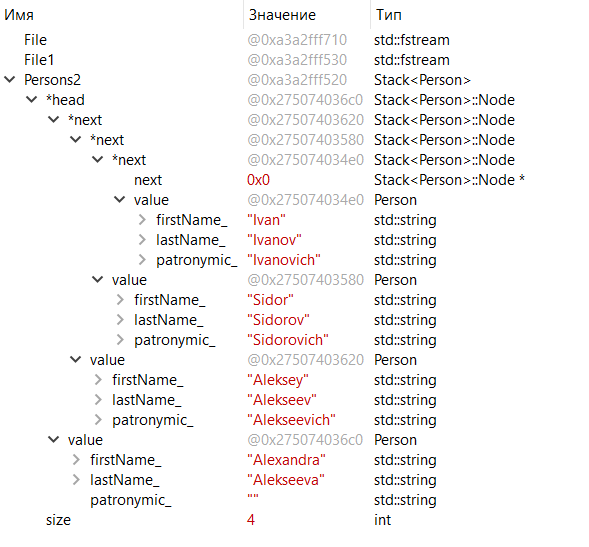


Тест:



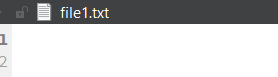
**Вывод**:

Стек:

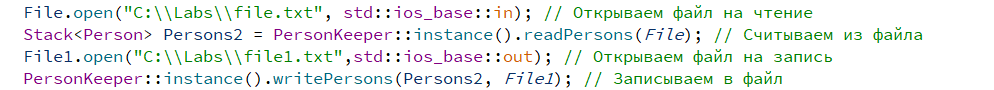


writePersons:

Файл:



Тест:



**Вывод**:

