Шарко Полина Алексеевна. Группа 931921.

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.

Класс Stack - шаблонный. Определения шаблонных классов должны быть видны из заголовков. Поэтому оставляем и заголовок с объявлениями, и определения в файле заголовка.

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

Для реализации односвязного списка также была добавлена структура Node. В нашем случае стек будет состоять из узлов типа Node. Node имеет два поля: значение и указатель на следующий узел.

Функции, реализованные в Stack:

Конструкторы: конструктор по умолчанию, конструктор копирования

Деструктор

void Push(const T &val) - метод помещения объекта в стек

const T Pop()- функция удаления верхнего объекта

int Size() - получение размера стека

void Print() - вывод стека

bool Empty() - проверка на пустоту стека

void Clear() - функция для освобождения стека

3. Коды программ

В репозитории представлены коды программ. Файлы предложенной реализации:

Stack.h

EStackEmpty.h

EStackException.h

4. Инструкция пользователя

В файле main.cpp приведен стек переменных типа int. Диалоговых окон не представлено, результат работы программы покажет все подписанные результаты тестирования.

5. Тестирование

Поведение функций может меняться в зависимости от некоторых характеристик стека. Тестирование стоит проводить для общих случаев - тех, в которых функции демонстрируют типичное для них поведение, рабочий вариант, и случаи, в которых из-за особенностей стека может возникнуть необходимость сообщить об ошибке.

Стек состоит из узлов, хранящих значение, тип данных которого может быть различным, и указатель на потомка в стеке. Также нам известен размер стека и указатель на первый узел стека.

Из перечисленных характеристик поведение функций может отличаться только при изменении размера стека, в частности нас будет интересовать случай пустого стека. Отметим, что от типа данных значений узлов поведение методов шаблонного класса зависеть не должно.

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

| 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() | выведено: 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 |

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

\*error - выводится сообщение об ошибке

//Часть 1: Стек

//1: Пустой стек

std::cout << "Test 1: Empty stack\n";

Stack <int> S;//пустой стек

try{S.Pop();//пытаемся достать голову стека

}

catch(EStackEmpty& e){

std::cout << e.what() <<"\n";

}

S.Clear();

std::cout <<"\n";

**Вывод:** This stack is empty. - ошибка, показывает, что в пустом стеке нельзя взять начальный узел

//2: Проверка функции Push()

std::cout << "Test 2: function Push\n";

for(int i = 3; i > 0 ; i--)

S.Push(i);//пробуем добавить узел в стек

//6.1: Проверка функции Print(): непустой стек

S.Print();//выводим стек

std::cout <<"\n\n";

**Вывод:** 1 2 3 - рабочий вариант, показывает значения добавленных в стек узлов

//3: Проверка функции Pop()

std::cout << "Test 3: function Pop\n";

S.Pop();

S.Print();//выводим стек

std::cout <<"\n\n";

**Вывод:** 2 3 - рабочий вариант, узел был удален

//4: Проверка функции Size(): непустой стек

std::cout << "Test 4: Size of an unempty stack\n";

std::cout<<S.Size();//выводим размер стека

//7.1: Проверка функции Clear(): непустой стек

S.Clear();

std::cout <<"\n\n";

**Вывод:** 2 - рабочий вариант, показывает размер стека

//5: Проверка функции Size(): пустой стек

std::cout << "Test 5: Size of an empty stack\n";

std::cout<<S.Size();//выводим размер стека

std::cout <<"\n\n";

**Вывод:** 0 - рабочий вариант, размер пустого стека действительно равен нулю

//6.2: Проверка функции Print(): пустой стек

std::cout <<"\n6.2: Check function Print(): empty stack\n";

S.Print();//выводим стек

//7.2: Проверка функции Clear(): пустой стек

std::cout <<"\n6.2: Check function Clear(): empty stack\n";

S.Clear();

std::cout <<"end of 1 part of testing\n\n";

# 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.

Для решения задачи были реализованы классы:

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

У класса PersonKeeper должен быть только один экземпляр, один статический объект, являющийся глобальной точкой доступа к классу. Эти особенности отражает шаблон Singleton, который и был использован в разработке. В нашем случае для данной задачи служит метод instance. возвращающий ссылку на единственный статический объект. Он не позволяет создать объект класса другим способом, например с помощью конструктора. В итоге у нас есть один экземпляр класса PersonKeeper, создаваемый единожды в методе instance.

Методы класса PersonKeeper:

* **PersonKeeper**() - конструктор по умолчанию
* ~**PersonKeeper**(){}-деструктор
* Stack<Person> stack\_ - стэк ФИО
* static PersonKeeper& **instance**(); - вернет ссылку на объект класса. static позволяет не создавать новый объект при вызове функции
* Stack<Person> **readPersons**(std::ifstream& file); - запись ФИО из файла в стек
* void **writePersons**(Stack<Person> s)const; - 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 - файл, пустой изначально, используется для проверки функции

**writePersons.**

4. Инструкция пользователя

Диалоговых окон не представлено, результат работы программы покажет все подписанные результаты тестирования.

5. Тестирование

Протестируем класс PersonKeeper.Поведение методов класса может зависеть от вида файла - от того, существует ли он, является ли пустым, корректно ли введены данные

| Особенности файла | | | |
| --- | --- | --- | --- |
| файл не существует | файл пуст | данные введены некорректно | файл существует, данные корректны |
| error | записывает пустой файл | error | читает данные из файла |

\*error - выводится сообщение об ошибке

//1. файл пуст

std::fstream File, File1;

File.open("file1.txt", std::ios\_base::in); // Открываем пустой файл на чтение

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

File1.open("file1.txt",std::ios\_base::out); // Открываем пустой файл на запись

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

**Вывод**: пустой файл file1.txt, рабочий вариант

//2. файл существует, данные корректны

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

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

File1.open("file1.txt",std::ios\_base::out); // Открываем файл на запись

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

**Вывод**: файл file1.txt такой же, как и file.txt, рабочий вариант

//3. данные введены некорректно

try {

PersonKeeper::CreateInstance().readPersons(invalid\_file);

}

catch( std::exception& e) {

std::cerr << e.what() << std::endl;

}

Вывод: ошибка, получен некорректный символ

//4. файл не существует

File.open("ex4.txt", std::ios\_base::in); // Открываем несуществующий файл на чтение

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

File1.open("ex4.txt",std::ios\_base::out); // Открываем несуществующий файл на запись

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

**Вывод**: ошибка, сообщает, что такого файла нет