МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
ИНЖЕНЕРНО-ТЕХНОЛОГИЧЕСКАЯ АКАДЕМИЯ
Институт компьютерных технологий и информационной безопасности

Кафедра математического обеспечения и применения ЭВМ

Лабораторная работа № 1
по курсу «Объектно-ориентированное программирование»
«Классы и объекты в C++»
9 ВАРИАНТ

Выполнили:
студенты гр. КТбо2-1
Яшенков А.В.

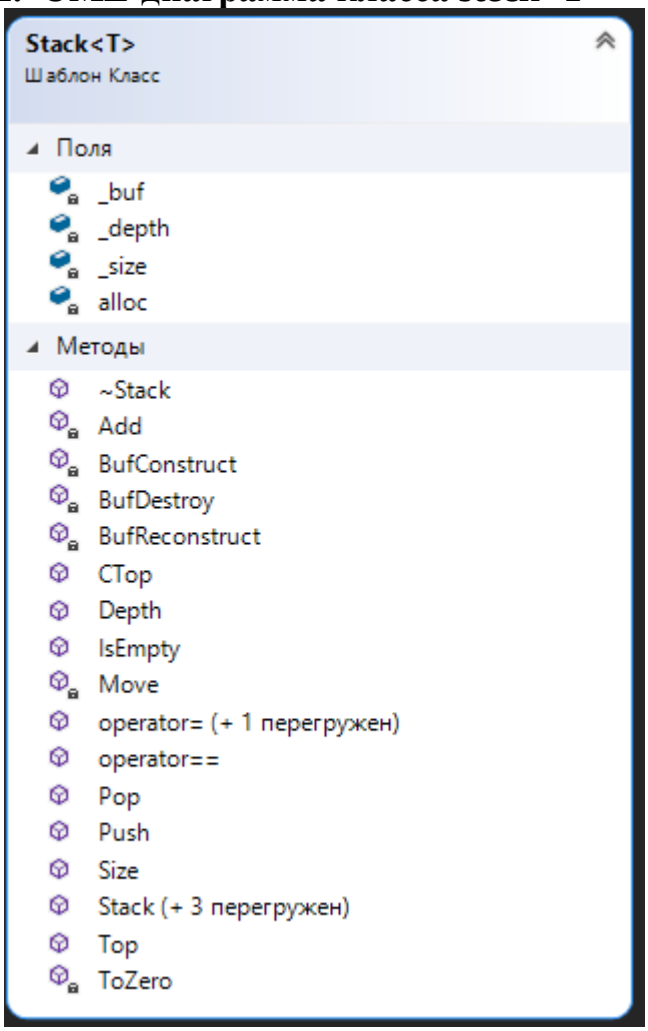
Проверил:
Тарасов С. А.

Таганрог 2020

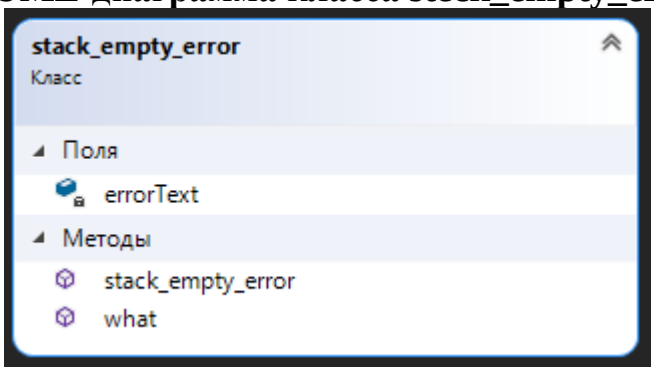
1. Вариант задания №9

Реализовать класс Стек (Stackдля какого-либо типа данных с методами push, pop, isEmptyи back(показывается последний элемент без его извлечения), работающими со-гласно соответствующей дисциплине обслуживания. Размер стека задается при его создании..

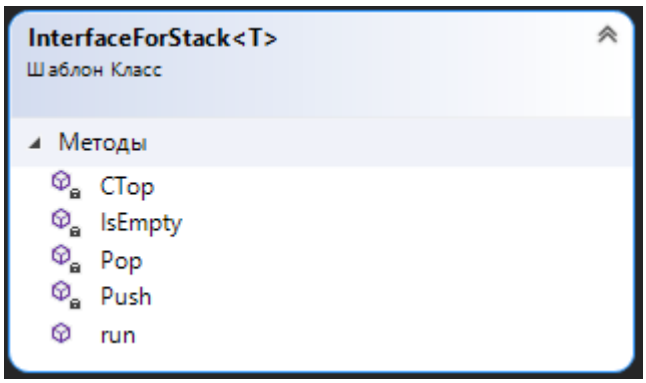
2. UML-диаграмма класса steck<T>



3. UML-диаграмма класса steck_empty_error



4. UML-диаграмма класса InterfaceForSteck



5. Листинг программы

main.cpp:

```
#include <iostream>
#include "stack.h"
#include "InterfaceForStack.h"


int main()
{
   InterfaceForStack<int> interface;

   interface.run();

   return 0;
}
```

steck.h:

```
#pragma once

class stack_empty_error
{
private:
    std::string errorText;

public:
    stack_empty_error(const std::string& err)
    {
        errorText = err;
    }

    std::string what() const
    {
        return errorText;
    }
```

```
};

template<typename T>
class Stack
{
private:
    void BufReconstruct()
    {
        T* new_buf = nullptr;
        int new_depth;

        if (!_depth)
        {
            new_depth = 1;
            new_buf = alloc.allocate(new_depth);
        }
        else
        {
            new_buf = alloc.allocate(_depth * 2);
            new_depth = _depth * 2;
        }

        for (int i = 0; i < _size; i++)
        {
            std::allocator_traits<std::allocator<T>>::construct(alloc, new_buf + i,
*(_buf + i));

            std::allocator_traits<std::allocator<T>>::destroy(alloc, _buf + i);
        }

        alloc.deallocate(_buf, _depth);

        _buf = new_buf;
        _depth = new_depth;
    }

    void BufConstruct(const T* other, int size)
    {
        for (int i = 0; i < size; i++)
        {
            std::allocator_traits<std::allocator<T>>::construct(alloc, _buf + i,
*(other + i));
        }
    }
```

```
    void BufDestroy()
    {
        for (int i = 0; i < _size; i++)
        {
            std::allocator_traits<std::allocator<T>>::destroy(alloc, _buf + i);
        }
    }

    void ToZero()
    {
        _depth = 0;
        _size = 0;
        _buf = nullptr;
    }

    void Move(Stack<T>& other)
    {
        _depth = other._depth;
        _size = other._size;
        _buf = other._buf;

        other.ToZero();
    }

    void Add(const T& value)
    {
        std::allocator_traits<std::allocator<T>>::construct(alloc, _buf + _size,
value);

        _size++;
    }

public:
    Stack() noexcept : _depth(4), _buf(alloc.allocate(_depth)) {};

    Stack(const int& depth) noexcept : _depth(depth), _buf(alloc.allocate(_depth)) {}

    Stack(const Stack<T>& other) noexcept : _size(other._size), _depth(other._depth),
_buf(alloc.allocate(other._depth))
    {
        this->BufConstruct(other._buf, other._size);
    }

    Stack(Stack<T>&& other) noexcept : _size(other._size), _depth(other._depth),
_buf(other._buf)
```

```
{
        other.ToZero();
}

~Stack() noexcept
{
        if (_buf)
        {
                this->BufDestroy();

                alloc.deallocate(_buf, _depth);
        }
}

Stack<T>& operator= (const Stack<T>& other)
{
        if (*this == other)
        {
                return *this;
        }

        if (!_buf)
        {
                _depth = other._depth;
                _size = other._size;
                this->BufConstruct(other._buf, other._size);

                return *this;
        }
        else
        {
                this->BufDestroy();

                this->BufConstruct(other._buf, other._size);

                _depth = other._depth;
                _size = other._size;

                return *this;
        }

}

Stack<T>& operator= (Stack<T>&& other)
{
```

```
        if (*this == other)
        {
                return *this;
        }

        if (!_buf)
        {
                this->Move(other);

                return *this;
        }
        else
        {
                this->BufDestroy();

                this->Move(other);

                return *this;
        }
}

bool operator==(const Stack<T>& other) const
{
        if (_depth == other._depth && _size == other._size)
        {
                for (int i = 0; i < _size; i++)
                {
                        if (_buf[i] != other._buf[i])
                        {
                                return false;
                        }
                }

                return true;
        }
        else
        {
                return false;
        }
}

bool IsEmpty() const
{
        return _size == 0;
}
```

```
    void Push(const T& value)
    {
        if (_size == _depth)
        {
            this->BufReconstruct();

            this->Add(value);
        }
        else
        {
            this->Add(value);
        }
    }

    void Pop()
    {
        if (!_size)
        {
            throw stack_empty_error("Stack is empty!\n");
        }
        else
        {
            std::allocator_traits<std::allocator<T>>::destroy(alloc, _buf + _size -
1);

            _size--;
        }
    }

    T& Top() const
    {
        if (!_size)
        {
            throw stack_empty_error("Stack is empty!\n");
        }
        else
        {
            return *(_buf + _size - 1);
        }
    }

    const T& CTop() const
    {
        if (!_size)
```

```
        {
            throw stack_empty_error("Stack is empty!\n");
        }
        else
        {
            return *(_buf + _size - 1);
        }
    }

    int Depth() const
    {
        return _depth;
    }

    int Size() const
    {
        return _size;
    }

private:
    int _depth;
    int _size = 0;
    T* _buf = nullptr;

    std::allocator<T> alloc;
};
```

InterfaceForSteck.h:

```
#pragma once
#include "Stack.h"

template<typename T>
class InterfaceForStack
{
private:

    void Push(Stack<T>& stack, const T& value)
    {
        stack.Push(value);
    }

    void Pop(Stack<T>& stack)
    {
```

```
            try
            {
                stack.Pop();
                std::cout << "Top element removed!\n\n";
            }
            catch (const stack_empty_error& err)
            {
                std::cerr << err.what() << std::endl;
            }
    }

    bool IsEmpty(Stack<T>& stack)
    {
            return stack.IsEmpty();
    }

    const T& CTop(Stack<T>& stack)
    {
            try
            {
                return stack.CTop();
            }
            catch (const stack_empty_error& err)
            {
                std::cerr << err.what() << std::endl;

                return T();
            }
    }

public:

    void run()
    {
            int command_number = INT_MAX;
            T value;

            int size;
            std::cout << "Enter a steck size: ";
            std::cin >> size;
            std::cout << std::endl;

            Stack<T> stack(size);

            std::cout << "Command list:\n";
```

```
	std::cout << "1. Add an item to the top.\n";
	std::cout << "2. Delete an item to the top.\n";
	std::cout << "3. Check Stack.\n";
	std::cout << "4. Print the top element.\n\n";
	std::cout << "0. Exit.\n\n";

	while (command_number)
	{
		std::cout << "Enter a command from 0 to 4: ";
		std::cin >> command_number;

		switch (command_number)
		{
		case 0:
			break;

		case 1:
			std::cout << "Enter a value: ";
			std::cin >> value;
			this->Push(stack, value);
			std::cout << std::endl;
			break;

		case 2:
			this->Pop(stack);

			break;

		case 3:
			if (this->IsEmpty(stack))
			{
				std::cout << "Your Stack is empty!\n\n";
			}
			else
			{
				std::cout << "Your Stack isn`t empty!\n\n";
			}
			break;

		case 4:

			if (!this->IsEmpty(stack))
			{
				std::cout << "Your the top element: " << this-
>CTop(stack) << std::endl << std::endl;
```

```
                }
                else
                {
                        std::cout << "Your Stack is empty!\n\n";
                }
                break;

        default:
                std::cout << "Please enter a valid value again!\n\n";
                break;
        }

    }
  }
};
```