ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«РОССИЙСКИЙ УНИВЕРСИТЕТ ТРАНСПОРТА»  
(РУТ (МИИТ))

Институт транспортной техники и систем управления

Кафедра «Управление и защита информации»

ОТЧЁТ  
О ПРАКТИЧЕСКОЙ РАБОТЕ № 1

По дисциплине «Языки программирования»

Выполнил: ст. гр. ТКИ – 241

Райхман И. А.

Проверил: к.т.н., доц.

Васильева М. А.

Москва 2023

**Содержание**

Задание 13

Задание 1

## Формулировка задачи

Создать решение (Solution), которое минимально содержит три проекта (Projects): исполняемый (Console Application), библиотеку классов (Library), и модульные тесты (Tests). Разработать библиотеку классов по заданному варианту. Важно! Библиотека классов не должна зависеть от потоков ввода-вывода. Каждый класс необходимо размещать в отдельных *двух* файлах, снабжённых «говорящим именем» и специальными расширениями: \*.h для заголовочных файлов (Header), содержащих API класса, и \*.cpp для компилируемых (Source), содержащих реализацию класса. В запускаемом проекте требуется создать файл main.cpp, содержащий точку входа в демонстрационную программу – главную функцию (main). В рамках данной функции показать работу с коллекцией.

Реализовать библиотеку заданной (по вариантам) структуры данных целых чисел. Реализовать все конструкторы, создаваемые компилятором по умолчанию, реализовать деструктор. Предусмотреть методы вывода в строку содержимого коллекции. Переопределить операторы присваивания, сдвига влево и сдвига вправо. Предусмотреть метод определения наличия элементов коллекции (пустая ли коллекция?).

**Вариант 2. Линейный односвязный список**

Коллекция, реализующая вставку и удаление элемента, поиск элемента по индексу и его модификация.

## UML - диаграмма алгоритма

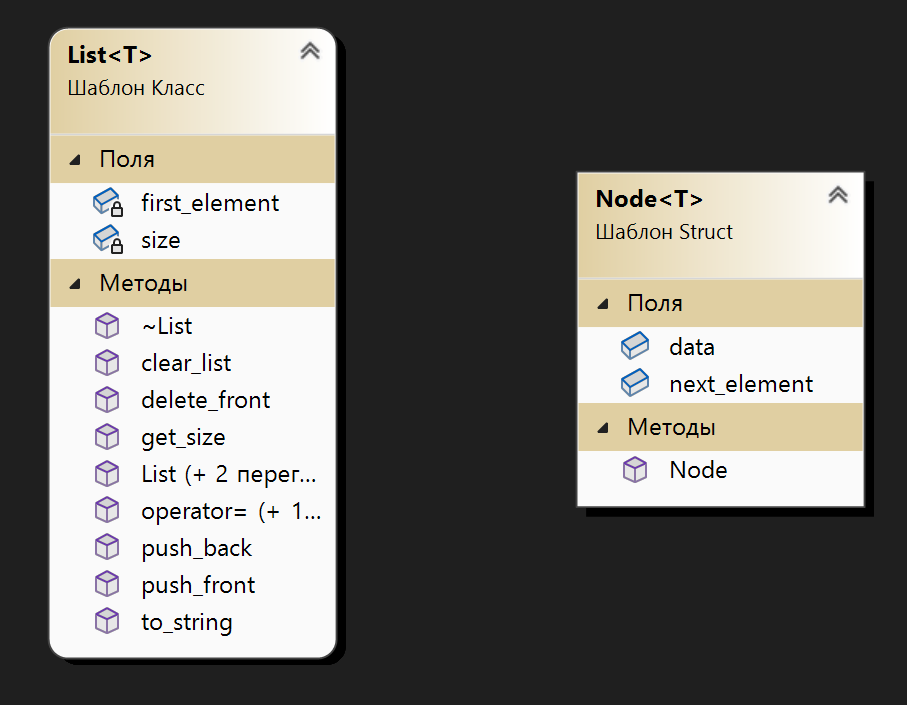




Рисунок 1 – UML диаграмма класса

## Решение задачи на языке программирования C++

main.cpp:

#include "..//list/List.h"

int main()

{

List<int> first\_list;

first\_list.push\_front(1);

first\_list.push\_front(2);

first\_list.push\_back(3);

first\_list.push\_front(4);

std::cout << first\_list.get\_size() << "\n";

std::cout << first\_list.to\_string() << "\n";

List<int> second\_list;

second\_list = first\_list;

std::cout << second\_list.get\_size() << "\n";

std::cout << second\_list.to\_string();

return 0;

};

Node.h:

#pragma once

template <typename T>

/\*\*

\* @brief Структура "Узел"

\*/

struct Node

{

/\*\*

\* @brief Данные, лежащие в узле

\*/

T data;

/\*\*

\* @brief Указатель на следующий элемент

\*/

Node\* next\_element;

/\*\*

\* @brief Функция, инициализирующая узел

\* @param data Данные узла

\* @param next Указатель на пустой следующий элемент

\*/

Node(T data, Node\* next = nullptr);

};

template <typename T>

Node<T>::Node(const T data, Node\* next\_element)

: data(data), next\_element(next\_element)

{

}

List.h:

#pragma once

#include "Node.h"

#include <iostream>

#include <string>

#include <sstream>

template <typename T>

class List;

template<typename T>

bool operator==(const List<T>& rha, const List<T>& lha) noexcept;

template<typename T>

bool operator!=(const List<T>& rha, const List<T>& lha) noexcept;

template <typename T>

class List

{

public:

/\*\*

\* @brief Функция, инициализирующая список

\*/

List(std::initializer<T> list);

/\*\*

\* @brief Функция, удаляющая список

\*/

~List();

/\*\*

\* @brief Функция, копирующая список

\* @param second\_list Список, который будет скопирован

\*/

List(const List& second\_list);

/\*\*

\* @brief Функция, перемещающая список

\* @param second\_list Список, который будет перемещен

\*/

List(List&& second\_list) noexcept;

/\*\*

\* @brief Функция, добавляющая элемент в начало списка

\* @param data Данные, которые будут добавлены

\*/

void push\_front(const T& data);

/\*\*

\* @brief Функция, добавляющая элемент в конец списка

\* @param data Данные, которые будут добавлены

\*/

void push\_back(const T& data);

/\*\*

\* @brief Функция, удаляющая узел в начале списка

\*/

void delete\_front();

/\*\*

\* @brief Функция, удаляющая все узлы списка

\*/

void clear\_list();

/\*\*

\* @brief Функция, возвращающая размер списка

\* @return Размер списка

\*/

size\_t get\_size();

/\*\*

\* @brief Функция, превращающая список в строку

\* @return Строка, состоящая из узлов

\*/

std::string to\_string() const;

/\*\*

\* @brief Функция, перегружающая оператор =

\* @param second\_list Список, который будет скопирован

\* @return Скопированный список

\*/

List<T>& operator=(const List<T>& second\_list);

/\*\*

\* @brief Функция, перегружающая оператор =

\* @param second\_list Список, который будет перемещен

\* @return Перемещенный список

\*/

List& operator=(List&& second\_list) noexcept;

private:

Node<T>\* first\_element;

size\_t size;

};

template <typename T>

List<T>::List(std::initializer<T> list)

:first\_element(nullptr), size(0)

{

}

template <typename T>

List<T>::~List()

{

clear\_list();

}

template <typename T>

List<T>::List(const List& second\_list):List()

{

if (this->first\_element == nullptr)

{

for (Node<T>\* node = second\_list.first\_element; node != nullptr; node = node->next\_element)

{

this->push\_back(node->data);

}

}

}

template <typename T>

List<T>::List(List&& second\_list) noexcept : List()

{

std::swap(this->first\_element, second\_list.first\_element);

}

template <typename T>

void List<T>::push\_front(const T& data)

{

first\_element = new Node<T>(data, first\_element);

size++;

}

template <typename T>

void List<T>::push\_back(const T& data)

{

if (first\_element == nullptr)

{

first\_element = new Node<T>(data);

}

else

{

auto this\_node = first\_element;

while (this\_node->next\_element != nullptr)

{

this\_node = this\_node->next\_element;

}

this\_node->next\_element = new Node<T>(data);

};

size++;

}

template <typename T>

void List<T>::delete\_front()

{

if (first\_element != nullptr)

{

Node<T>\* temp = first\_element;

first\_element = first\_element->next\_element;

size--;

delete temp;

}

}

template <typename T>

void List<T>::clear\_list()

{

while (size > 0)

{

del\_front();

}

}

template <typename T>

size\_t List<T>::get\_size()

{

return size;

}

template <typename T>

std::string List<T>::to\_string() const

{

std::stringstream buffer;

if (first\_element != nullptr)

{

Node<T>\* this\_node = first\_element;

for (size\_t i = 0; i < size; i++)

{

buffer << this\_node->data << ' ';

this\_node = this\_node->next\_element;

}

}

return buffer.str();

}

template<typename T>

inline List<T>& List<T>::operator=(const List<T>& second\_list)

{

if (\*this == second\_list)

{

List other\_list(second\_list);

std::swap(this->first\_element, other\_list.first\_element);

std::exchange(this->size, other\_list.size);

return \*this;

}

}

template <typename T>

List<T>& List<T>::operator=(List<T>&& second\_list) noexcept

{

std::swap(this->first\_element, second\_list.first\_element);

return \*this;

}

template<typename T>

inline bool operator==(const List<T>& rha, const List<T>& lha) noexcept

{

return(lha.to\_string() == rha.to\_string());

}

template<typename T>

inline bool operator!=(const List<T>& rha, const List<T>& lha) noexcept

{

return!(lha.to\_string() == rha.to\_string());

}

## Решение тестов

Код тестов:

#include "CppUnitTest.h"

#include "..//list/List.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace ListTests

{

TEST\_CLASS(ListTestsInt)

{

public:

TEST\_METHOD(push\_back\_test\_valid\_data\_success)

{

//Arrange

List<int> default\_list{};

std::string expected\_string = "1 2 ";

//Act

default\_list.push\_back(1);

default\_list.push\_back(2);

std::string actual = default\_list.to\_string();

//Assert

Assert::AreEqual(expected\_string, actual);

}

TEST\_METHOD(push\_front\_test\_valid\_data\_success)

{

//Arrange

List<int> default\_list{};

std::string expected\_string = "2 1 ";

//Act

default\_list.push\_back(1);

default\_list.push\_front(2);

std::string actual = default\_list.to\_string();

//Assert

Assert::AreEqual(expected\_string, actual);

}

TEST\_METHOD(get\_size\_test\_valid\_data\_success)

{

//Arrange

List<int> default\_list{};

int expected = 3;

//Act

default\_list.push\_back(1);

default\_list.push\_back(2);

default\_list.push\_back(3);

int actual = default\_list.get\_size();

//Assert

Assert::AreEqual(expected, actual);

}

TEST\_METHOD(comparison\_operator\_tets\_valid\_data\_success)

{

//Arrange

List<int> first\_list{};

List<int> second\_list{};

//Act

first\_list.push\_back(1);

first\_list.push\_back(2);

second\_list.push\_back(1);

second\_list.push\_back(2);

//Assert

Assert::IsTrue(first\_list == second\_list);

}

TEST\_METHOD(copy\_operator\_test\_valid\_data\_succeess)

{

//Arrange

List<int> first\_list{};

List<int> copied\_list{};

//Act

first\_list.push\_back(1);

first\_list.push\_back(2);

copied\_list = first\_list;

//Arrange

Assert::IsFalse(copied\_list == first\_list);

}

};

TEST\_CLASS(ListTestsString)

{

public:

TEST\_METHOD(push\_back\_test\_valid\_data\_success)

{

//Arrange

List<std::string> default\_list{};

std::string expected\_string = "1 2 ";

//Act

default\_list.push\_back("1");

default\_list.push\_back("2");

std::string actual = default\_list.to\_string();

//Assert

Assert::AreEqual(expected\_string, actual);

}

TEST\_METHOD(push\_front\_test\_valid\_data\_success)

{

//Arrange

List<std::string> default\_list{};

std::string expected\_string = "2 1 ";

//Act

default\_list.push\_back("1");

default\_list.push\_front("2");

std::string actual = default\_list.to\_string();

//Assert

Assert::AreEqual(expected\_string, actual);

}

TEST\_METHOD(get\_size\_test\_valid\_data\_success)

{

//Arrange

List<std::string> default\_list{};

int expected = 3;

//Act

default\_list.push\_back("1");

default\_list.push\_back("2");

default\_list.push\_back("3");

int actual = default\_list.get\_size();

//Assert

Assert::AreEqual(expected, actual);

}

TEST\_METHOD(comparison\_operator\_tets\_valid\_data\_success)

{

//Arrange

List<std::string> first\_list{};

List<std::string> second\_list{};

//Act

first\_list.push\_back("1");

first\_list.push\_back("2");

second\_list.push\_back("1");

second\_list.push\_back("2");

//Assert

Assert::IsTrue(first\_list == second\_list);

}

TEST\_METHOD(copy\_operator\_test\_valid\_data\_succeess)

{

//Arrange

List<std::string> first\_list{};

List<std::string> copied\_list{};

//Act

first\_list.push\_back("1");

first\_list.push\_back("2");

//Arrange

Assert::IsFalse(copied\_list == first\_list);

}

};

}

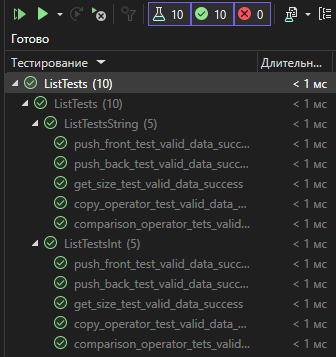


Рисунок 2 – Успешные тесты

## Зачет задания в GitHub



Рисунок 3 - Зачет задания в GitHub