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 № 3

По дисциплине «Языки программирования»

Выполнил: ст. гр. ТКИ – 241

Захаров Дмитрий Константинович

Проверил: к.т.н., доц.

Васильева М. А.

Москва 2023

**Задание:** написать линейный односвязный список с использованием умных указателей

**Код задания:**

#pragma once

#include <iostream>

#include <memory>

#include <sstream>

#include <vector>

template<typename T>

class Node;

template<typename T>

class List;

/\*\*

\* @brief Оператор вывода

\* @param os Поток вывода

\* @param list Список для вывода

\* @return Поток вывода с информацией списка

\*/

template<typename T>

std::ostream& operator<< (std::ostream& os, const List<T>& list);

/\*\*

@brief Оператор равенства

@param rha Первый список для сравнения

@param lha Второй список для сравнения

@return true - если равны, иначе false

\*/

template<typename T>

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

/\*\*

@brief Оператор неравенства

@param rha Первый список для сравнения

@param lha Второй список для сравнения

@return true - если неравны, иначе false

\*/

template<typename T>

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

/\*

@brief Класс, описывающий линейный односвязный список

\*/

template<typename T>

class List

{

public:

/\*

\* @brief Создает пустой объект лжносвязного списка

\*/

List();

/\*

\* @brief Создает список по элементам пользователя

\* @param list Элементы пользователя

\*/

List(const std::initializer\_list<T> list);

/\*

\* @brief Деструктор, чистит память при удалении объекта

\*/

~List();

/\*

\* @brief Очищает список

\*/

void clear();

/\*

\* @brief Преобразует список в строку

\* @return Строка, созданная по списку

\*/

std::string to\_string() const noexcept;

/\*

\* @brief Добавления элемента в список

\* @param value Элемент, который надо добавить

\*/

void push(const T& value);

/\*

\* @brief Удаление элемента из списка

\*/

void pop();

/\*

\* @brief Найти элемент в списке

\* @return true - если есть, иначе false

\*/

bool find(const T& item);

/\*

\* @brief Проверка наличия элементов в списке

\* @return true - есть элементы, иначе false

\*/

bool has\_elements() const noexcept;

/\*

\* @brief Проверка отсутствия элементов в списке

\* @return true - нет элементов, иначе false

\*/

bool is\_empty() const noexcept;

/\*

\* @brief Оператор копирования

\* @param list Список для копирования

\* @return Скопированный List

\*/

List<T>& operator=(const List<T>& list);

/\*

\* @brief Оператор перемещения

\* @param list Список для перемещения

\* @return Перемещенный List

\*/

List<T>& operator=(List<T>&& list) noexcept;

/\*

\* @brief Конструктор копирования

\* @param list Список для копирования

\*/

List(const List<T>& list) =default;

/\*

\* @brief Конструктор перемещения

\* @param list Список для перемещения

\*/

List(List<T>&& list) =default;

private:

/\*

\* @brief Класс, описывающий элемент списка

\*/

template<typename T>

class Node

{

public:

/\*

@brief Создает элемент списка

@param value Значение элемента

@param next\_node Следующий элемент списка

\*/

Node(const T& value)

:value{ value }, next\_node{ nullptr }{};

std::unique\_ptr<Node<T>> next\_node;

T value;

};

std::unique\_ptr<Node<T>> head;

};

template<typename T>

inline List<T>::List()

:head{ nullptr }

{

}

template<typename T>

inline List<T>::List(const std::initializer\_list<T> list)

: head{ nullptr }

{

std::vector<int> temp{};

for (auto item : list)

{

temp.insert(temp.begin(), item);

}

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

{

this->push(temp[i]);

}

}

template<typename T>

inline List<T>::~List()

{

while (head)

{

auto next = std::move(head->next\_node);

head = std::move(next);

}

}

template<typename T>

inline void List<T>::clear()

{

while (head)

{

auto next = std::move(head->next\_node);

head = std::move(next);

}

}

template<typename T>

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

{

std::stringstream result\_string;

result\_string << "[ ";

auto curr = this->head.get();

while (curr != nullptr)

{

result\_string << curr->value << " ";

curr = curr->next\_node.get();

}

result\_string << "]";

return result\_string.str();

}

template<typename T>

inline void List<T>::push(const T& value)

{

std::unique\_ptr<Node<T>> element(new Node<T>(value));

if (this->has\_elements())

{

element->next\_node = std::move(this->head);

this->head = std::move(element);

}

else

{

this->head = std::move(element);

}

}

template<typename T>

inline void List<T>::pop()

{

if (this->has\_elements())

{

std::unique\_ptr<Node<T>> temp = std::move(head);

this->head = std::move(temp->next\_node);

}

}

template<typename T>

inline bool List<T>::find(const T& item)

{

auto curr = this->head.get();

while (curr != nullptr)

{

if (curr->value == item)

{

return true;

}

curr = curr->next\_node.get();

}

return false;

}

template<typename T>

inline bool List<T>::has\_elements() const noexcept

{

return !(this->is\_empty());

}

template<typename T>

inline bool List<T>::is\_empty() const noexcept

{

return this->head == nullptr;

}

template<typename T>

inline List<T>& List<T>::operator=(const List<T>& list)

{

if (\*this != list)

{

List temp{ list };

std::swap(this->head, temp.head);

}

return \*this;

}

template<typename T>

inline List<T>& List<T>::operator=(List<T>&& list) noexcept

{

if (\*this != list)

{

std::swap(this->head, list.head);

}

return \*this;

}

template<typename T>

inline std::ostream& operator<<(std::ostream& os, const List<T>& list)

{

return os << list.to\_string();

}

template<typename T>

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

{

if (&lha == &rha)

{

return true;

}

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

}

template<typename T>

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

{

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

}

#include "List.h"

**Тесты:**

#include "pch.h"

#include "CppUnitTest.h"

#include "../LinkedListSmartPtr/List.h"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace Tests

{

TEST\_CLASS(ListTests)

{

public:

TEST\_METHOD(ToString\_ValidData\_Success)

{

//arrange

List list{ 1,2,3,4,5,6 };

//assert

Assert::IsTrue(list.to\_string() == "[ 1 2 3 4 5 6 ]");

}

TEST\_METHOD(EqualOperator\_ValidData\_Success)

{

//arrange

List list\_1{ 1,2,3,4,5,6 };

List list\_2{ 1,2,3,4,5,6 };

//assert

Assert::IsTrue(list\_1 == list\_2);

}

TEST\_METHOD(NotEqualOperator\_ValidData\_Success)

{

//arrange

List list\_1{ 1,2,3,4,5,6 };

List list\_2{ 3,5,6,75,6 };

//assert

Assert::IsTrue(list\_1 != list\_2);

}

TEST\_METHOD(Push\_ValidData\_Success)

{

//arrange

List list{ 1,2,3,4,5 };

List expected{ 6,1,2,3,4,5 };

//act

list.push(6);

//assert

Assert::IsTrue(list == expected);

}

TEST\_METHOD(Pop\_ValidData\_Success)

{

//arrange

List list{ 1,2,3,4,5 };

List expected{ 2,3,4,5 };

//act

list.pop();

//assert

Assert::IsTrue(list == expected);

}

TEST\_METHOD(HasElements\_ValidData\_Success)

{

//arrange

List list{ 1,2,3,4,5 };

//assert

Assert::IsTrue(list.has\_elements());

}

TEST\_METHOD(OutputOperator\_ValidData\_Success)

{

//arrange

std::stringstream sstring;

List list = List({ 1,2,3,4 });

//act

sstring << list;

//assert

Assert::AreEqual(sstring.str(), list.to\_string());

}

TEST\_METHOD(CopyOperator\_ValidData\_Success)

{

//arrange

List list\_1{ 1,2,3,4,5 };

List list\_2{ 1,5,10,15 };

//act

list\_1 = list\_2;

//assert

Assert::IsTrue(list\_1 == list\_2);

}

TEST\_METHOD(MoveOperator\_ValidData\_Success)

{

//arrange

List list\_1{ 1,2,3,4,5 };

List list\_2{ 1,5,10,15 };

List expected{ 1,5,10,15 };

//act

list\_1 = std::move(list\_2);

//assert

Assert::IsTrue(list\_1 == expected);

}

TEST\_METHOD(Find\_ValidData\_Success)

{

//arrange

List list\_1{ 1, 2,3,4,5 };

//assert

Assert::IsTrue(list\_1.find(3));

}

};

};

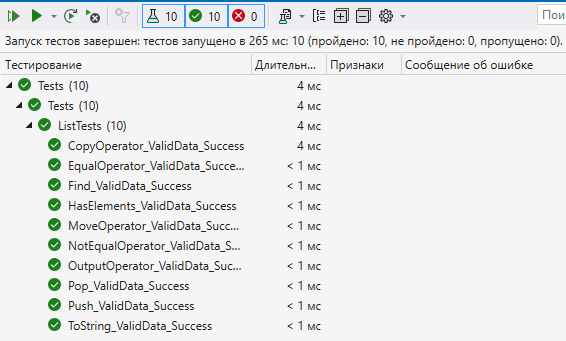
****

Рисунок 1 – прохождение тестов

**UML Диаграмма:**

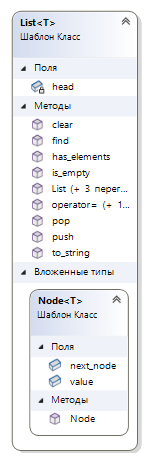


Рисунок 2 – UML диаграмма