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

Байло А.В.

Проверил: к.т.н., доц.

Васильева М. А.

Москва 2023

**Задание:** Написать класс Single linked list

**Код программы:**

#include <iostream>

#include <sstream>

#include <string>

template <typename T> struct Node;

template <typename T> class List;

template <typename T>

bool operator==(const List<T> &first, const List<T> &second);

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

template <typename T>

/\*\*

\* @brief Struct node

\*

\*/

struct Node {

/\*\*

\* @brief Construct a new Node object

\*

\* @param data data which contains in node

\* @param pointerNext pointer to next node

\*/

Node(T data, Node<T> \*pointerNext = nullptr);

/\*\*

\* @brief override = operator

\*

\* @param otherData assigned value

\*/

void operator=(T otherData);

Node<T> \*pointerNext;

T data;

};

template <typename T>

inline Node<T>::Node(T data, Node<T> \*pointerNext)

: data(data), pointerNext(pointerNext) {}

template <typename T> void Node<T>::operator=(T otherData) {

this->data = otherData;

}

/\*\*

\* @brief Class list

\*

\* @tparam T dataType

\*/

template <typename T> class List {

public:

/\*\*

\* @brief Construct a new List object

\*

\*/

List();

/\*\*

\* @brief Destroy the List object

\*

\*/

~List();

/\*\*

\* @brief Construct a new List object by copy other object

\*

\* @param second list to be copied

\*/

List(const List &second);

/\*\*

\* @brief Construct a new List object by move other object

\*

\* @param second list to be movied

\*/

List(List &&second);

/\*\*

\* @brief function which delete element in front of list

\*

\*/

void popFront();

/\*\*

\* @brief function which clear the list

\*

\*/

void clear();

/\*\*

\* @brief function which add element to the back of the list

\*

\* @param data

\*/

void pushBack(T data);

/\*\*

\* @brief function which add element in front tof the list

\*

\* @param data

\*/

void pushFront(T data);

/\*\*

\* @brief Get the Size of the list

\*

\* @return size of the list

\*/

size\_t getSize() const noexcept;

/\*\*

\* @brief override [] operator

\*

\* @param index of necessary value

\* @return T& value of necessary object

\*/

T &operator[](int index);

/\*\*

\* @brief overload [] operator

\*

\* @param index of necassary value

\* @return T& CONST value of necessary element

\*/

T &operator[](int index) const;

/\*\*

\* @brief function which check is collection clear

\*

\* @return true if empty

\* @return false otherwise

\*/

bool isCollectionClear() const noexcept;

/\*\*

\* @brief funvtion which convert list into std::string

\*

\* @return std::string contains list

\*/

const std::string toString();

/\*\*

\* @brief overloading = operator

\*

\* @param second list which values will be copied

\* @return List<T>& list with the second list's values

\*/

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

/\*\*

\* @brief overloading = operator

\*

\* @param second list which values will be movied

\* @return List<T>& list with the second list's values

\*/

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

/\*\*

\* @brief override == operator

\*

\* @param first list to compare

\* @param second list to compare

\* @return true if list are equal

\* @return false otherwise

\*/

friend bool operator==<T>(const List<T> &first, const List<T> &second);

/\*\*

\* @brief override << operator

\*

\* @param os output stream

\* @param list to be put in output stream

\* @return std::ostream& which contains list as std::string

\*/

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

private:

size\_t size;

Node<T> \*head;

};

template <typename T> List<T>::List() : head(nullptr), size(0) {}

template <typename T> void List<T>::pushBack(T data) {

if (head == nullptr) {

head = new Node(data);

} else {

Node<T> \*current = head;

while (current->pointerNext != nullptr) {

current = current->pointerNext;

}

current->pointerNext = new Node(data);

}

size++;

}

template <typename T> void List<T>::pushFront(T data) {

head = new Node(data, head);

size++;

}

template <typename T> size\_t List<T>::getSize() const noexcept {

return this->size;

}

template <typename T> T &List<T>::operator[](int index) {

size\_t counter = 0;

Node<T> \*current = this->head;

while (current != nullptr) {

if (counter == index) {

return current->data;

}

current = current->pointerNext;

counter++;

}

}

template <typename T> T &List<T>::operator[](int index) const {

size\_t counter = 0;

Node<T> \*current = this->head;

while (current != nullptr) {

if (counter == index) {

return current->data;

}

current = current->pointerNext;

counter++;

}

}

template <typename T> bool List<T>::isCollectionClear() const noexcept {

return size == 0;

}

template <typename T> const std::string List<T>::toString() {

std::stringstream buffer;

if (head != nullptr) {

Node<T> \*current = head;

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

buffer << current->data << ' ';

current = current->pointerNext;

}

}

return buffer.str();

}

template <typename T> void List<T>::popFront() {

if (head != nullptr) {

Node<T> \*temp = head;

head = head->pointerNext;

size--;

delete temp;

}

}

template <typename T> void List<T>::clear() {

while (size) {

popFront();

}

}

template <typename T> List<T>::~List() { clear(); }

template <typename T> inline List<T>::List(List<T> &&list) : head(nullptr) {

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

}

template <typename T> List<T> &List<T>::operator=(List<T> &&second) noexcept {

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

return \*this;

}

template <typename T>

inline List<T>::List(const List<T> &second) : head(nullptr) {

if (this->head == nullptr) {

for (Node<T> \*node = second.head; node != nullptr;

node = node->pointerNext) {

this->pushBack(node->data);

}

}

}

template <typename T> List<T> &List<T>::operator=(const List<T> &second) {

if (this->head == nullptr) {

for (Node<T> \*node = second.head; node != nullptr;

node = node->pointerNext) {

this->pushBack(node->data);

}

}

return \*this;

}

template <typename T> bool operator==(List<T> &first, List<T> &second) {

return (first.toString() == second.toString());

}

template <typename T>

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

return os << list.toString();

}

**Тесты:**

#include <cstddef>

#include <gtest/gtest.h>

#include <string>

#include "SingleTemplatedList.cpp"

template <typename T> struct Node;

template <typename T> class List;

/\*\*

\* Tests consist of 3 parts

1) create a object and use some method

2) create actual and expected values

3) ASSERT

\*

\*/

TEST(ASSERT\_TRUE\_TESTS, stringListTest)

{

List<std::string> defaultList;

std::string str1 = "qwe";

defaultList.pushBack(str1);

std::string actual = defaultList.toString();

std::string expected = str1 + ' ';

ASSERT\_TRUE(actual == expected);

}

TEST(ASSERT\_TRUE\_TESTS, intListTest)

{

List<int> defaultList;

int a = 5;

defaultList.pushBack(a);

std::string actual = defaultList.toString();

std::string expected = std::to\_string(a) + ' ';

ASSERT\_TRUE(actual == expected);

}

TEST(ASSERT\_TRUE\_TESTS, isCollectionClearTets)

{

List<std::string> defaultList;

bool actual = defaultList.isCollectionClear();

bool expected = 1;

ASSERT\_TRUE(defaultList.isCollectionClear() == expected);

}

TEST(ASSERT\_TRUE\_TESTS, pushBackTest)

{

List <std::string> defaultList;

std::string str1 = "qwe";

std::string str2 = "zxc";

defaultList.pushBack(str1);

defaultList.pushBack(str2);

std::string actual = defaultList.toString();

std::string expected = str1 + ' ' + str2 + ' ';

ASSERT\_TRUE(actual == expected);

}

TEST(ASSERT\_TRUE\_TESTS, pushFrontTest)

{

List <std::string> defaultList;

std::string str1 = "qwe";

std::string str2 = "zxc";

defaultList.pushFront(str1);

defaultList.pushFront(str2);

std::string actual = defaultList.toString();

std::string expected = str2 + ' ' + str1 + ' ';

ASSERT\_TRUE(actual == expected);

}

TEST(ASSERT\_EQUAL\_TESTS, getSizeTest)

{

List<std::string> defaultList;

std::string str1 = "qwe";

std::string str2 = "zxc";

defaultList.pushFront(str1);

defaultList.pushFront(str2);

int actual = defaultList.getSize();

int expected = 2;

ASSERT\_EQ(actual, expected);

}

TEST(ASSERT\_TRUE\_TESTS, popFrontTest)

{

List<std::string> defaultList;

std::string str1 = "qwe";

std::string str2 = "zxc";

defaultList.pushBack(str1);

defaultList.pushBack(str2);

defaultList.popFront();

std::string actual = defaultList.toString();

std::string expected = str2 + ' ';

ASSERT\_EQ(actual, expected);

}

TEST(ASSERT\_EQUAL\_TESTS, clearTest)

{

List<std::string> defaultList;

std::string str1 = "qwe";

std::string str2 = "zxc";

defaultList.pushFront(str1);

defaultList.pushFront(str2);

defaultList.clear();

std::string actual = defaultList.toString();

int actualSize = 0;

std::string expected = "";

int expectedSize = 0;

ASSERT\_EQ(actual, expected);

ASSERT\_EQ(actualSize, expectedSize);

}

TEST(ASSERT\_EQUAL\_TESTS, NodeContructorTest)

{

std::string str1 = "qwe";

Node<std::string> defaultNode(str1);

std::string actual = defaultNode.data;

std::string expected = str1;

ASSERT\_EQ(actual, expected);

}

int main(int argc, char \*\*argv) {

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

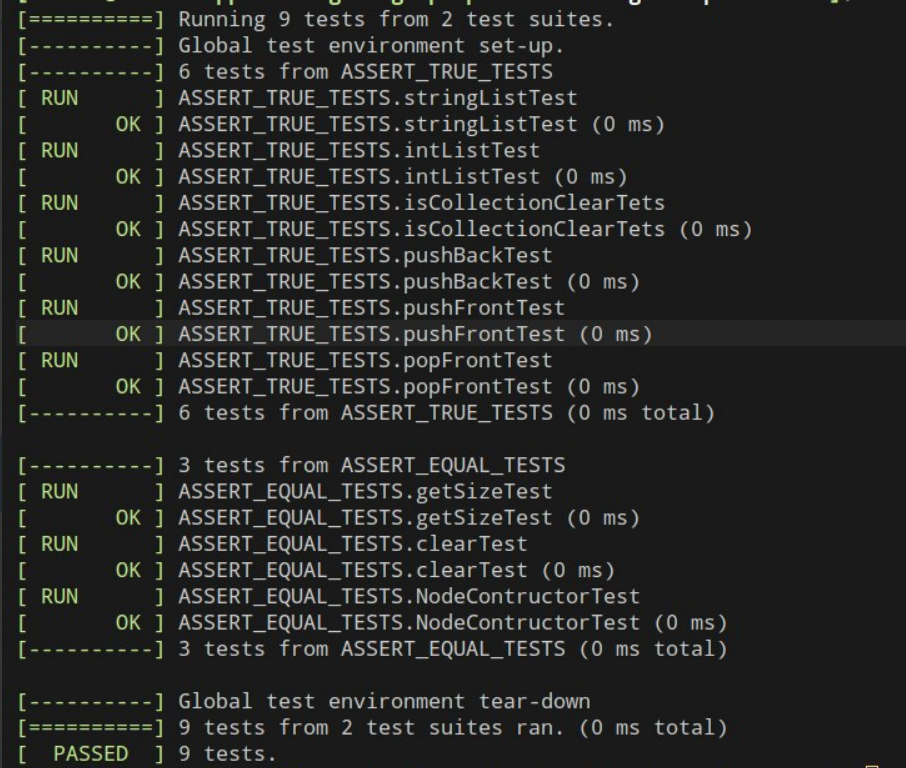


Рисунок 1 – прохождение тестов

**UML диаграмма**

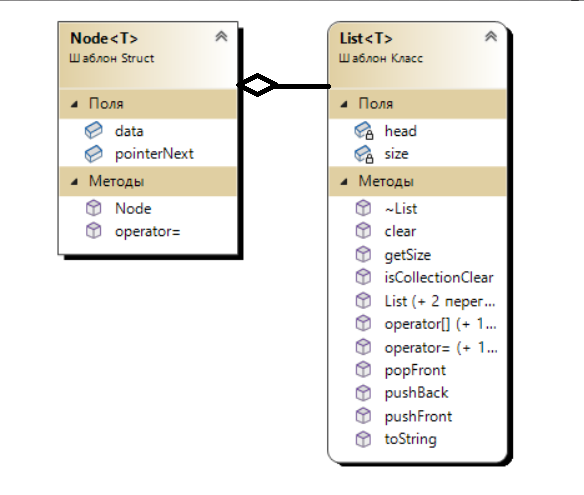
****

Рисунок 2 -Uml – диаграмма к классу SLL

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Рисунок 3 – approve задания.