**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе № 3**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: Вектор и список.**

Студент гр.7303 Батурин И.

|  |  |  |
| --- | --- | --- |
| Преподаватель |  | Размочаева Н.В |

Санкт-Петербург

2019 г.

**Цель работы**

Реализовать базовый функционал, семантически аналогичный функционалу из стандартной библиотеки шаблонов для классов вектор и список.

**Задание**

Необходимо реализовать конструкторы и деструктор для контейнера вектор.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор.

Необходимо реализовать функции resize и erase для контейнера вектор.

Необходимо реализовать функции insert и push\_back для контейнера вектор.

Необходимо реализовать список со следующими функциями: вставка элементов в голову и в хвост; получение элемента из головы и из хвоста; удаление из головы, хвоста и очистка; проверка размера.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции: деструктор; конструктор копирования; конструктор перемещения; оператор присваивания.

Необходимо реализовать операторы: =, ﻿==, !=, ++ (постфиксный и префиксный), \*, ->.

Необходимо реализовать: вставку элементов (Вставляет value перед элементом, на который указывает pos. Возвращает итератор, указывающий на вставленный value); удаление элементов (Удаляет элемент в позиции pos. Возвращает итератор, следующий за последним удаленным элементом).

**Требования к реализации**

При выполнении этого задания вы можете определять любые вспомогательные функции. Вводить или выводить что-либо **не нужно**. Реализовывать функцию main не нужно. Не используйте функции из cstdlib (malloc, calloc, realloc и free).

**Исходный код**

Код класса vector представлен в приложении А.

Код класса list представлен в приложении Б.

**Вывод**

В ходе написания лабораторной работы были реализованы классы vector и list, аналогичные класса из стандартной библиотеки.

**ПРИЛОЖЕНИЕ А**

**РЕАЛИЗАЦИЯ КЛАССА ВЕКТОР**

#include <assert.h>

#include <algorithm> // std::copy, std::rotate

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

namespace stepik

{

template <typename Type>

class vector

{

public:

typedef Type\* iterator;

typedef const Type\* const\_iterator;

typedef Type value\_type;

typedef value\_type& reference;

typedef const value\_type& const\_reference;

typedef std::ptrdiff\_t difference\_type;

explicit vector(size\_t count = 0)

{

m\_first = new Type[count];

m\_last = m\_first + count;

for (int i = 0; i < size(); i++)

m\_first[i] = 0;

// use previous step implementation

}

template <typename InputIterator>

/\*Создает контейнер с содержимым диапазона [first, last).\*/

vector(InputIterator first, InputIterator last) : vector(last-first)

{

std::copy(first, last, m\_first);

// use previous step implementation

}

vector(std::initializer\_list<Type> init) : vector(init.begin(), init.end())

{

// use previous step implementation

}

vector(const vector& other)

{

m\_first = new Type[other.size()];

m\_last = &(m\_first[other.size()]);

std::copy(other.m\_first, other.m\_last, m\_first);

// use previous step implementation

}

vector(vector&& other)

{

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

// use previous step implementation

}

~vector()

{

delete []m\_first;

// use previous step implementation

}

static void swap(vector& that, vector& other)

{

std::swap(that.m\_first, other.m\_first);

std::swap(that.m\_last, other.m\_last);

}

//assignment operators

vector& operator=(const vector& other)

{

if (this != &other) {

vector tmp(other);

swap(\*this, tmp);

}

return \*this;

// implement this

}

vector& operator=(vector&& other)

{

if (this != &other)

swap(\*this, other);

return \*this;

// implement this

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

vector tmp(first, last);

swap(\*this, tmp);

// implement this

}

void resize(size\_t count)

{

size\_t new\_size = count < (m\_last-m\_first) ? count : (m\_last-m\_first);

vector tmp(count);

std::copy(m\_first, m\_first + new\_size, tmp.m\_first);

swap(\*this, tmp);

// implement this

}

void push\_back(const Type& value)

{

resize(size() + 1);

\*(m\_last - 1) = value;

// implement this

}

iterator insert(const\_iterator pos, const Type& value)

{

vector temp\_vector(pos - m\_first);

difference\_type tmp = pos - m\_first;

std::copy(m\_first, tmp + m\_first, temp\_vector.m\_first);

temp\_vector.push\_back(value);

temp\_vector.resize(size()+1);

std::copy(m\_first + tmp, m\_last, temp\_vector.m\_first + 1);

\*this = std::move(temp\_vector);

return tmp + m\_first;

// implement this

}

template <typename InputIterator>

iterator insert(const\_iterator pos, InputIterator first, InputIterator last)

{

difference\_type tmp\_pos = pos - m\_first;

difference\_type tmp\_range = last - first;

vector temp\_vector(size() + tmp\_range);

std::copy(m\_first, tmp\_pos + m\_first, temp\_vector.m\_first);

std::copy(first, last, temp\_vector.m\_first + tmp\_pos);

std::copy(m\_first + tmp\_pos, m\_last, temp\_vector.m\_first + tmp\_pos + tmp\_range);

\*this = std::move(temp\_vector);

return tmp\_pos + m\_first;

// implement this

}

//push\_back methods

//at methods

reference at(size\_t pos)

{

return checkIndexAndGet(pos);

}

const\_reference at(size\_t pos) const

{

return checkIndexAndGet(pos);

}

//[] operators

reference operator[](size\_t pos)

{

return m\_first[pos];

}

const\_reference operator[](size\_t pos) const

{

return m\_first[pos];

}

//\*begin methods

iterator begin()

{

return m\_first;

}

const\_iterator begin() const

{

return m\_first;

}

//\*end methods

iterator end()

{

return m\_last;

}

const\_iterator end() const

{

return m\_last;

}

//size method

size\_t size() const

{

return m\_last - m\_first;

}

//empty method

bool empty() const

{

return m\_first == m\_last;

}

private:

reference checkIndexAndGet(size\_t pos) const

{

if (pos >= size())

{

throw std::out\_of\_range("out of range");

}

return m\_first[pos];

}

//your private functions

private:

iterator m\_first;

iterator m\_last;

};

}// namespace stepik

**ПРИЛОЖЕНИЕ Б**

**РЕАЛИЗАЦИЯ КЛАССА СПИСОК**

#include <assert.h>

#include <algorithm>

#include <stdexcept>

#include <cstddef>

#include <utility>

namespace stepik

{

template <class Type>

struct node

{

Type value;

node\* next;

node\* prev;

node(const Type& value, node<Type>\* next, node<Type>\* prev)

: value(value), next(next), prev(prev)

{

}

};

template <class Type>

class list; //forward declaration

template <class Type>

class list\_iterator

{

public:

typedef ptrdiff\_t difference\_type;

typedef Type value\_type;

typedef Type\* pointer;

typedef Type& reference;

typedef size\_t size\_type;

typedef std::forward\_iterator\_tag iterator\_category;

list\_iterator()

: m\_node(NULL)

{

}

list\_iterator(const list\_iterator& other)

: m\_node(other.m\_node)

{

}

list\_iterator& operator = (const list\_iterator& other)

{

m\_node = other.m\_node;

return \*this;

// implement this

}

bool operator == (const list\_iterator& other) const

{

return m\_node == other.m\_node;

// implement this

}

bool operator != (const list\_iterator& other) const

{

return m\_node != other.m\_node;

// implement this

}

reference operator \* ()

{

return m\_node->value;

// implement this

}

pointer operator -> ()

{

return &(m\_node->value);

// implement this

}

list\_iterator& operator ++ ()

{

m\_node = m\_node->next;

return \*this;

// implement this

}

list\_iterator operator ++ (int)

{

list\_iterator temp(\*this);

++(\*this);

return \*this;

// implement this

}

private:

friend class list<Type>;

list\_iterator(node<Type>\* p)

: m\_node(p)

{

}

node<Type>\* m\_node;

};

template <class Type>

class list

{

public:

typedef Type value\_type;

typedef value\_type& reference;

typedef const value\_type& const\_reference;

typedef list\_iterator<Type> iterator;

list()

: m\_head(nullptr), m\_tail(nullptr)

{

}

~list()

{

clear();

// use previous step implementation

}

list(const list& other) : m\_head(nullptr), m\_tail(nullptr)

{

auto elem = other.m\_head;

while (elem) {

push\_back(elem->value);

elem = elem->next;

}

// implement this

}

list(list&& other) : m\_head(nullptr), m\_tail(nullptr)

{

swap(other);

// implement this

}

void swap(list& other) {

std::swap(m\_head, other.m\_head);

std::swap(m\_tail, other.m\_tail);

}

list& operator= (const list& other)

{

if(this != &other){

clear();

auto cur = other.m\_head;

while(cur){

push\_back(cur->value);

cur = cur->next;

}

}

return \*this;

// implement this

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

void pop\_front()

{

if (!empty()) {

if (size() == 1) {

node<Type >\* elem = m\_head;

m\_head = NULL;

m\_tail = NULL;

}

else {

node<Type >\* elem = m\_head;

elem = m\_head->next;

elem->prev = nullptr;

delete m\_head;

m\_head = elem;

}

}

// implement this

}

void pop\_back()

{

if (!empty()) {

if (size() == 1) {

node<Type >\* elem = m\_head;

m\_head = NULL;

m\_tail = NULL;

}

else {

node<Type >\* elem = m\_tail;

elem = m\_tail->prev;

elem->next = nullptr;

delete m\_tail;

m\_tail = elem;

}

}

// implement this

}

void push\_back(const value\_type& value)

{

if (empty()) {

node<Type>\* elem = new node<Type>(value, nullptr, nullptr);

m\_head = elem;

m\_tail = elem;

}

else {

node<Type>\* elem = new node<Type>(value, nullptr, m\_tail);

m\_tail->next = elem;

m\_tail = elem;

}

// use previous step implementation

}

void push\_front(const value\_type& value)

{

if (empty()) {

node<Type>\* elem = new node<Type>(value, nullptr, nullptr);

m\_head = elem;

m\_tail = elem;

}

else {

node<Type>\* elem = new node<Type>(value, m\_head, nullptr);

m\_head->prev = elem;

m\_head = elem;

}

// implement this

}

iterator insert(iterator pos, const Type& value)

{

if (pos.m\_node == NULL)

{

push\_back(value);

return iterator(m\_tail);

}// implement this

else if (pos.m\_node->prev == NULL) {

push\_front(value);

return iterator(m\_head);

}

else {

node<Type>\* temp = new node<Type>(value, pos.m\_node, pos.m\_node->prev);

pos.m\_node->prev->next = temp;

pos.m\_node->prev = temp;

return iterator(temp);

}

// implement this

}

iterator erase(iterator pos)

{

if (pos.m\_node == NULL)

{

return NULL;

}

else if (pos.m\_node->prev == NULL)

{

pop\_front();

return iterator(m\_head);

}

else if (pos.m\_node->next == NULL)

{

pop\_back();

return iterator(m\_tail);

}

else

{

pos.m\_node->next->prev = pos.m\_node->prev;

pos.m\_node->prev->next = pos.m\_node->next;

node<Type>\* temp = pos.m\_node;

iterator new\_pos(pos.m\_node->next);

delete temp;

return new\_pos;

}

}

reference front()

{

return m\_head->value;

// use previous step implementation

}

reference back()

{

return m\_tail->value;

// use previous step implementation

}

void clear()

{

while(m\_head) {

m\_tail = m\_head->next;

delete m\_head;

m\_head = m\_tail;

}

// implement this

}

bool empty() const

{

return m\_head == NULL;

// implement this

}

size\_t size() const

{

node<Type>\* elem = m\_head;

size\_t i = 0;

while (elem != NULL) {

i++;

elem = elem->next;

}

return i;

// implement this

}

private:

//your private functions

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}// namespace stepik