**void 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. 7381 | Кортев Ю. В. |
| Преподаватель | Жангиров Т. Р. |

Санкт-Петербург

2019

**Цель работы.**

Реализовать базовый функционал для классов вектор и список.

**Задание.**

Необходимо реализовать конструкторы и деструктор для контейнера вектор. Предполагается реализация упрощенной версии вектора, без резервирования памяти под будущие элементы.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции resize и erase для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

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

Поведение реализованных функций должно быть таким же, как у класса std::vector (<http://ru.cppreference.com/w/cpp/container/vector>). Семантику реализованных функций нужно оставить без изменений.

Необходимо реализовать список со следующими функциями:

1. Вставка элементов в голову и в хвост;
2. Получение элемента из головы и из хвоста;
3. Удаление из головы, хвоста и очистка;
4. Проверка размера.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции:

1. Деструктор;
2. Конструктор копирования;
3. Конструктор перемещения;
4. Оператор присваивания.

На данном шаге необходимо реализовать итератор для списка. Для краткости реализации можно ограничиться однонаправленным изменяемым (неконстантным) итератором. Необходимо реализовать операторы: =, ==, !=, ++ (постфиксный и префиксный), \*, ->.

На данном шаге с использованием итераторов необходимо реализовать:

1. Вставку элементов (Вставляет value перед элементом, на который указывает pos. Возвращает итератор, указывающий на вставленный value),
2. Удаление элементов (Удаляет элемент в позиции pos. Возвращает итератор, следующий за последним удаленным элементом).

Поведение реализованных функций должно быть таким же, как у класса std::list (<http://ru.cppreference.com/w/cpp/container/list>). Семантику реализованных функций нужно оставить без изменений.

**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).

**Ход работы.**

Реализация класса vector:

Были реализованы различные конструкторы и деструктор класса аналогичные конструкторам vector из stl.

Добавлены оператор присваивания, методы assign, resize, erase, insert и push\_bask.

Реализация класса list:

Были добавлены методы вставки и удаления из головы и хвоста списка, получения элемента из головы и хвоста, проверки размера списка.

Добавлены конструкторы и деструктор аналогичные std::list.

Также был реализован класс list\_iterator, аналогичный std::list

Добавлены методы вставки и удаления элементов с использованием итераторов списка.

Исходный код каждого написанного класса представлен в приложениях А и Б.

**Выводы.**

В ходе написания лабораторной работы были реализованы классы вектор и список, аналогичные классам из стандартной библиотеки. Полученные знания из предыдущих лабораторных работ были применены в ходе работы над этой работой.

**ПРИЛОЖЕНИЕ A**

**РЕАЛИЗАЦИЯ КЛАССА VECTOR**

#include <assert.h>

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

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

namespace stepik

{

template <class 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;

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last) {

size\_t count;

count = last - first;

m\_first = new Type[count];

m\_last = m\_first + count;

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

}

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

vector(const vector& other) : vector(other.begin(), other.end()) {}

vector(vector&& other) {

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

}

//assignment operators

vector& operator=(const vector& other) {

if (this != &other) {

delete[] m\_first;

size\_t count = other.m\_last - other.m\_first;

m\_first = new Type[count];

m\_last = m\_first + count;

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

return \*this;

}

}

vector& operator=(vector&& other) {

if (this != &other) {

delete[] m\_first;

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

return \*this;

}

}

~vector() {

delete[] m\_first;

}

//insert methods

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

{

size\_t num = pos - m\_first;

vector temp(\*this);

delete[]m\_first;

m\_first = new value\_type[temp.size() + 1];

m\_last = m\_first + temp.size() + 1;

std::copy(temp.begin(), temp.begin() + num, m\_first);

\*(m\_first + num) = value;

std::copy(temp.begin() + num, temp.end(), m\_first + num + 1);

return m\_first + num;

}

template <typename InputIterator>

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

{

size\_t num = pos - m\_first;

size\_t count = last - first;

vector temp(\*this);

delete[]m\_first;

m\_first = new value\_type[temp.size() + last - first];

m\_last = m\_first + temp.size() + (last - first);

std::copy(temp.begin(), temp.begin() + num, m\_first);

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

std::copy(temp.begin() + num, temp.end(), m\_first + num + count);

return num + m\_first;

}

#include <assert.h>

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

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

namespace stepik

{

template <class 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;

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last) {

size\_t count;

count = last - first;

m\_first = new Type[count];

m\_last = m\_first + count;

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

}

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

vector(const vector& other) : vector(other.begin(), other.end()) {}

vector(vector&& other) {

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

}

//assignment operators

vector& operator=(const vector& other) {

if (this != &other) {

delete[] m\_first;

size\_t count = other.m\_last - other.m\_first;

m\_first = new Type[count];

m\_last = m\_first + count;

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

return \*this;

}

}

vector& operator=(vector&& other) {

if (this != &other) {

delete[] m\_first;

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

return \*this;

}

}

~vector() {

delete[] m\_first;

}

// resize methods

void resize(size\_t count)

{

vector temp(\*this);

delete[]m\_first;

m\_first = new value\_type[count];

m\_last = m\_first + count;

std::copy(temp.begin(), count < temp.end() - temp.begin() ? temp.begin() + count : temp.end(), m\_first);

}

//erase methods

iterator erase(const\_iterator pos)

{

size\_t num = pos - m\_first;

std::rotate(const\_cast<iterator>(pos), const\_cast<iterator>(pos) + 1, m\_last);

resize(size() - 1);

return m\_first + num;

}

iterator erase(const\_iterator first, const\_iterator last)

{

iterator del = const\_cast<iterator>(first);

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

del = erase(del);

return del;

}

//at methods

reference at(size\_t pos)

{

return checkIndexAndGet(pos);

}

const\_reference at(size\_t pos) const

{

return checkIndexAndGet(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

//push\_back methods

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

{

insert(this->end(), value);

}

//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 LIST**

#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;

}

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

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

}

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

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

}

reference operator\* () {

return m\_node->value;

}

// Указатель на хранимое

pointer operator-> () {

return &(m\_node->value);

}

// ++object

list\_iterator& operator ++() {

m\_node = m\_node->next;

return \*this;

}

// object++

list\_iterator operator ++ (int) {

auto ret = \*this;

m\_node = m\_node->next;

return ret;

}

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();

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

iterator insert(iterator pos, const Type& value)

{

if (pos.m\_node == nullptr) {

push\_back(value);

return iterator(m\_tail);

}

if (pos.m\_node == m\_head) {

push\_front(value);

return iterator(m\_head);

}

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

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

return iterator(pos.m\_node->prev);

}

iterator erase(iterator pos)

{

if (pos.m\_node == m\_head) {

pop\_front();

return iterator(m\_head);

}

if (pos.m\_node == m\_tail) {

pop\_back();

return iterator(m\_tail);

}

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

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

temp->prev->next = temp;

delete pos.m\_node;

return iterator(temp);

}

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

{

node<Type>\* cur = other.m\_head;

while (cur) {

push\_back(cur->value);

cur = cur->next;

}

}

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

{

m\_head = other.m\_head;

m\_tail = other.m\_tail;

other.m\_head = nullptr;

other.m\_tail = nullptr;

}

list& operator= (const list& other)

{

clear();

node<Type>\* cur = other.m\_head;

while (cur) {

push\_back(cur->value);

cur = cur->next;

}

}

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

{

if (empty()) {

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

m\_tail = m\_head;

return;

}

m\_tail->next = new node<Type>(value, nullptr, m\_tail);

m\_tail = m\_tail->next;

}

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

{

if (empty()) {

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

m\_tail = m\_head;

return;

}

m\_head->prev = new node<Type>(value, m\_head, nullptr);

m\_head = m\_head->prev;

}

reference front()

{

return m\_head->value;

}

reference back()

{

return m\_tail->value;

}

void pop\_front()

{

if (size() == 1) {

m\_tail = nullptr;

delete m\_head;

m\_head = nullptr;

}

else {

m\_head = m\_head->next;

delete m\_head->prev;

m\_head->prev = nullptr;

}

}

void pop\_back()

{

if (size() == 1) {

m\_tail = nullptr;

delete m\_head;

m\_head = nullptr;

}

else {

m\_tail = m\_tail->prev;

delete m\_tail->next;

m\_tail->next = nullptr;

}

}

void clear()

{

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

node<Type>\* next;

while (cur) {

next = cur->next;

delete cur;

cur = next;

}

m\_head = m\_tail = nullptr;

}

bool empty() const

{

if (m\_head == nullptr)

return true;

else return false;

}

size\_t size() const

{

if (empty())

return 0;

size\_t i = 1;

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

while (tpr != m\_tail) {

tpr = tpr->next;

i++;

}

return i;

}

private:

//your private functions

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}// namespace stepik