# 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 |  | Мининг М.С. |
| Преподаватель |  | Размочаева Н.В. |

Санкт-Петербург 2019

**Цель работы.**

Реализовать базовый функционал, семантически аналогичный функционалу из стандартной библиотеки шаблонов для классов вектор и линейный 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).](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).](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.**

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

**Исходный код.**

Код класса vector представлен в приложении А. Код класса list представлен в приложении Б.

**Выводы.**

В ходе написания лабораторной работы были реализованы классы вектор и список, аналогичные классам из стандартной библиотеки. 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 <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)

{

if (count > 0){

m\_first = new Type[count];

m\_last = m\_first + count;

}

else {

m\_first = nullptr;

m\_last = nullptr;

}

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last)

{

size\_t size = last - first;

if (size > 0){

m\_first = new Type[size];

m\_last = m\_first + size;

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

}

else{

m\_first = nullptr;

m\_last = nullptr;

}

}

vector(std::initializer\_list<Type> init)

{

if (init.size()){

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

m\_last = m\_first + init.size();

std::copy(init.begin(), init.end(), this->m\_first);

}

else{

m\_first = nullptr;

m\_last = nullptr;

}

}

vector(const vector& other)

{

if (other.size()){

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

this->m\_last = this->m\_first + other.size();

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

}

else{

m\_first = nullptr;

m\_last = nullptr;

}

}

vector(vector&& other): m\_last(nullptr), m\_first(nullptr)

{

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

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

}

~vector()

{

if (m\_first != nullptr){

delete [] m\_first;

m\_first = nullptr;

m\_last = nullptr;

}

}

//assignment operators

vector& operator=(const vector& other)

{

if (m\_first != nullptr) delete [] m\_first;

if (other.size()) {

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

this->m\_last = this->m\_first + other.size();

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

}

else {

m\_first = nullptr;

m\_last = nullptr;

}

return \*this;

}

vector& operator=(vector&& other)

{

if (m\_first != nullptr) delete [] m\_first;

m\_first = nullptr;

m\_last = nullptr;

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

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

return \*this;

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

if (m\_first != nullptr) delete [] m\_first;

size\_t size = last- first;

this->m\_first = new Type[ size ];

this->m\_last = m\_first + size;

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

}

void resize(size\_t count)

{

if (count == 0) {

delete [] m\_first;

m\_first = nullptr;

m\_last = nullptr;

return;

}

if (count == size()) return;

Type\* buffer = new Type[count];

if (count > size()) {

std::copy(m\_first, m\_last, buffer);

} else {

std::copy(m\_first, m\_first + count, buffer);

}

delete[] m\_first;

m\_first = buffer;

m\_last = m\_first + count;

}

//push\_back methods

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

{

resize(size() + 1);

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

}

//insert methods

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

{

size\_t index = 0;

iterator current = m\_first;

while (current != pos) {

current++;

index++;

}

push\_back(value);

std::rotate(m\_first + index, m\_last - 1, m\_last);

return m\_first + index;

}

template <typename InputIterator>

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

{

size\_t index = 0;

iterator current = m\_first;

while (current != pos) {

current++;

index++;

}

difference\_type size = last - first;

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

push\_back(\*first);

first++;

}

std::rotate(m\_first + index, m\_last - size, m\_last);

return m\_first + index;

}

//erase methods

iterator erase(const\_iterator pos)

{

difference\_type delta = pos - m\_first;

std::rotate(m\_first + delta, m\_first + delta + 1, m\_last);

resize(size() - 1);

return (m\_first + delta);

}

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

{

difference\_type delta = first - m\_first;

difference\_type erase\_size = last - first;

std::rotate(m\_first + delta, m\_first + delta + erase\_size, m\_last);

resize(size() - erase\_size);

return m\_first + delta;

}

//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)

{

if (this != &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;

}

pointer operator -> ()

{

return &(m\_node->value);

}

list\_iterator& operator ++ ()

{

m\_node = m\_node->next;

return \*this;

}

list\_iterator operator ++ (int)

{

auto temp = \*this;

m\_node = m\_node->next;

return temp;

}

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()

{

if (!empty()) clear();

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

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

{

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

if (!empty()) {

temp->prev= m\_tail;

m\_tail->next = temp;

m\_tail = temp;

} else {

m\_head = temp;

m\_tail = temp;

}

}

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

{

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

if(!empty()) {

temp->next = m\_head;

m\_head->prev = temp;

m\_head = temp;

} else {

m\_head = temp;

m\_tail = temp;

}

}

void pop\_front()

{

if(!empty()) {

if (m\_head == m\_tail){

delete m\_head;

m\_head = nullptr;

m\_tail = nullptr;

} else {

node<Type>\* temp = m\_head->next;

delete m\_head;

m\_head = temp;

m\_head->prev = nullptr;

}

}

}

void pop\_back()

{

if(!empty()) {

if (m\_head == m\_tail){

delete m\_head;

m\_head = nullptr;

m\_tail = nullptr;

} else {

node<Type>\* temp = m\_tail->prev;

delete m\_tail;

m\_tail = temp;

m\_tail->next = nullptr;

}

}

}

iterator insert(iterator pos, const Type& value)

{

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

push\_back(value);

return iterator(m\_tail);

}

if (pos == begin()){

push\_front(value);

return begin();

}

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 pos.m\_node->prev;

}

iterator erase(iterator pos)

{

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

pop\_front();

return iterator(m\_head);

}

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

pop\_back();

return iterator(m\_tail);

}

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

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

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

delete pos.m\_node;

return iterator(next);

}

void clear()

{

node<Type>\* temp;

while(m\_head != nullptr){

temp = m\_head->next;

delete m\_head;

m\_head = temp;

}

m\_head = nullptr;

m\_tail = nullptr;

}

bool empty() const

{

if (size() == 0) return true;

else return false;

}

size\_t size() const

{

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

size\_t size = 0;

while(temp != nullptr) {

size++;

temp = temp->next;

}

return size;

}

reference front()

{

return m\_head->value;

}

reference back()

{

return m\_tail->value;

}

private:

//your private functions

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}// namespace stepik