**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №3**

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

Тема: «Контейнеры. Вектор. Список»

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Давкаева В.С. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Жангиров Т.М. |

Санкт-Петербург

2019

**Цель работы.**

Изучить стандартные контейнеры vector и list языка С++.

**Задание.**

Необходимо реализовать конструкторы, деструктор, оператор присваивания, функции assign, resize, erase, insert и push\_back для контейнера вектор (в данном уроке предполагается реализация упрощенной версии, без резервирования памяти под будущие элементы).

Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать список со следующими функциями: вставка элементов в голову и в хвост, получение элемента из головы и из хвоста, удаление из головы и из хвоста, очистка, проверка размера, деструктор, конструктор копирования, конструктор перемещения, оператор присваивания.

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

С использованием итераторов 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).

Поведение реализованных функций должно быть таким же, как у класса std::list. Семантику реализованных функций нужно оставить без изменений.

При выполнении этого задания можно 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.**

Был реализован класс vector; поведение реализованных функций аналогично поведению функций класса std::vector.

Класс vector содержит два поля: указатели на начало и конец массива данных в памяти. Были реализованы деструктор и следующие конструкторы: конструктор от размера массива, от двух итераторов, от списка инициализации, копирования и перемещения. Также были реализованы методы изменения размера, удаления одного элемента или интервала элементов, вставки одного элемента или нескольких элементов, заданных при помощи двух итераторов, на заданное итератором место и вставки одного элемента в конец вектора.

Реализация класса представлена в приложении А.

Класс list имеет аналогичные поля, как и у класса vector, но данные содержатся не в массиве, а в двусвязном списке. Для класса list были реализованы деструктор и следующие конструкторы: стандартный, копирования и перемещения. Также был реализованы оператор присваивания и методы для вставки, получения и удаления элементов из головы и из хвоста, очистки списка и проверки размера. Поведение реализованных функций аналогично поведению функций класса std::list.

Итератор для списка содержит одно поле – указатель на элемент контейнера list. Для итератора был перегружен ряд операторов: =, ==, !=, ++ (постфиксный и префиксный), \* и ->.

Реализация класса представлена в приложении Б.

**Вывод.**

В ходе выполнения лабораторной работы была изучена реализация контейнеров vector и list.

**Приложение А. Файл vector.h.**

#include "pch.h"

#include <iostream>

#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(int count = 0)

{

m\_first = new Type[count];

m\_last = m\_first + count;

std::cout << "конструктор от размера массва\n";

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last)

{

Type \*first\_t = (Type\*)first;

Type \*last\_t = (Type\*)last;

m\_first = new Type[last\_t - first\_t];

m\_last = m\_first + (last\_t - first\_t);

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

std::cout << "конструктор от 2х итераторов \n";

}

vector(std::initializer\_list<Type> init) :vector(init.begin(), init.end()) { std::cout << "конструктор списка инициализации\n"; }

vector(const vector& other) :vector(other.m\_first, other.m\_last) { std::cout << "конструктор копирования \n"; }

vector(vector&& other)

{

swap(\*this, other);

std::cout << "перемещение\n";

}

~vector()

{

delete[] m\_first;

m\_first = nullptr;

m\_last = nullptr;

std::cout << "деструктор\n";

}

vector& operator=(const vector& other)

{

if (((void\*)this) == ((void\*)(&other)))return \*this;

vector v(other);

swap(\*this, v);

std::cout << "присваивание копированием\n";

return \*this;

}

vector& operator=(vector&& other)

{

if (((void\*)this) == ((void\*)(&other)))return \*this;

swap(\*this, other);

std::cout << "присваивание перемещением\n";

return \*this;

}

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

vector v(first, last);

swap(\*this, v);

std::cout << "assign\n";

}

void resize(size\_t count)

{

size\_t c = m\_last - m\_first;

if (count == c)return;

vector v(count);

size\_t ms = count < c ? count : c;

std::copy(m\_first, m\_first + ms, v.m\_first);

swap(\*this, v);

}

iterator erase(const\_iterator pos)

{

size\_t p = (Type\*)pos - m\_first;

vector v = \*this;

std::rotate(v.m\_first + p, v.m\_first + p + 1, v.m\_last);

v.resize(v.size() - 1);

\*this = std::move(v);

std::cout << "удаление 1го элемента\n";

return m\_first + p;

}

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

{

size\_t f = (Type\*)first - m\_first;

size\_t l = (Type\*)last - m\_first;

vector v = \*this;

std::rotate(v.m\_first + f, v.m\_first + l, v.m\_last);

v.resize(v.size() - l + f);

\*this = std::move(v);

std::cout << "удаление диапозона элементов\n";

return m\_first + f;

}

//insert methods

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

{

size\_t f = (Type\*)pos - m\_first;

vector v(size() + 1);

std::copy(m\_first, m\_first + f, v.m\_first);

\*(v.m\_first + f) = value;

std::copy(m\_first + f, m\_last, v.m\_first + f + 1);

\*this = std::move(v);

std::cout << "вставка одного элемента\n";

return m\_first + f;

}

template <typename InputIterator>

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

{

size\_t f = (Type\*)pos - m\_first;

size\_t l = f + (last - first);

vector v(size() + last - first);

std::copy(m\_first, m\_first + f, v.m\_first);

std::copy(first, last, v.m\_first + f);

std::copy(m\_first + f, m\_last, v.m\_first + l);

\*this = std::move(v);

std::cout << "вставка диапозона элементов\n";

return m\_first + f;

}

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

{

resize(size() + 1);

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

std::cout << "вставка элемента в конец\n";

}

reference at(size\_t pos)

{

return checkIndexAndGet(pos);

}

const\_reference at(size\_t pos) const

{

return checkIndexAndGet(pos);

}

reference operator[](size\_t pos)

{

return m\_first[pos];

}

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

{

return m\_first[pos];

}

iterator begin()

{

return m\_first;

}

const\_iterator begin() const

{

return m\_first;

}

iterator end()

{

return m\_last;

}

const\_iterator end() const

{

return m\_last;

}

size\_t size() const

{

return m\_last - m\_first;

}

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];

}

static void swap(vector& a, vector& b) {

std::swap(a.m\_first, b.m\_first);

std::swap(a.m\_last, b.m\_last);

}

private:

iterator m\_first = nullptr;

iterator m\_last = nullptr;

};

}

int main()

{

setlocale(LC\_ALL, "Russian");

stepik::vector<int> words1{ 1, 2, 3, 4, 5 };

//for (int i = 0; i < words1.size(); i++)

//std::cout << words1[i] << std::endl;

stepik::vector<int> words2(words1.begin(), words1.end());

//for (int i = 0; i < words2.size(); i++)

//std::cout << words2[i] << std::endl;

stepik::vector<int> words3(5);

//for (int i = 0; i < words3.size(); i++)

//std::cout << words3[i] << std::endl;

stepik::vector<int> words4(words1);

//for (int i = 0; i < words4.size(); i++)

//std::cout << words4[i] << std::endl;

stepik::vector<int> v(std::move(words1));

//for (int i = 0; i < v.size(); i++)

//std::cout << v[i] << std::endl;

words1 = words3;

//for (int i = 0; i < words1.size(); i++)

//std::cout << words1[i] << std::endl;

words1 = std::move(words2);

//for (int i = 0; i < words1.size(); i++)

//std::cout << words1[i] << std::endl;

int myints[] = { 1776,7,4 ,78,5,134};

words1.assign(myints, myints + 5);

//for (int i = 0; i < words1.size(); i++)

//std::cout << words1[i] << std::endl;

words1.erase(words1.begin()+1);

//for (int i = 0; i < words1.size(); i++)

//std::cout << words1[i] << std::endl;

words1.erase(words1.begin() + 1, words1.begin() + 3);

//for (int i = 0; i < words1.size(); i++)

//std::cout << words1[i] << std::endl;

words1.push\_back(23);

words1.insert(words1.begin()+1, 345);

words1.insert(words1.begin(), words3.begin(), words3.begin() + 2);

for (int i = 0; i < words1.size(); i++)

std::cout << words1[i] << std::endl;

}

**Приложение Б. Файл list.h.**

#include "pch.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;

}

pointer operator -> ()

{

return &(m\_node->value);

}

list\_iterator& operator ++ ()

{

m\_node = m\_node->next;

return \*this;

}

list\_iterator operator ++ (int)

{

list\_iterator tmp(\*this);

++(\*this);

return tmp;

}

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();

}

iterator begin()

{

return iterator(m\_head);

}

iterator end()

{

return iterator();

}

list(const list& other)

{

auto c = other.m\_head;

while (c != nullptr) {

push\_back(c->value);

c = c->next;

}

}

list(list&& other)

{

m\_head = other.m\_head;

m\_tail = other.m\_tail;

other.m\_tail = nullptr;

other.m\_head = nullptr;

}

list& operator= (const list& other)

{

if (this == &other)return \*this;

clear();

for (auto r = other.m\_head; r != nullptr; r = r->next)push\_back(r->value);

return \*this;

}

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

{

if (empty()) {

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

return;

}

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

m\_tail = m\_tail->next;

}

iterator insert(iterator pos, const Type& value)

{

auto ref = pos.m\_node;

if (ref == nullptr) {

push\_back(value);

return iterator(m\_tail);

}

if (ref == m\_head) {

push\_front(value);

return iterator(m\_head);

}

auto n = new node<Type>(value, ref, ref->prev);

n->next->prev = n;

n->prev->next = n;

return iterator(n);

}

iterator erase(iterator pos)

{

auto ref = pos.m\_node;

if (ref == m\_head) {

pop\_front();

return iterator(m\_head);

}

if (ref == m\_tail) {

pop\_back();

return iterator(m\_tail);

}

ref->prev->next = ref->next;

ref->next->prev = ref->prev;

iterator i(ref->next);

delete ref;

return i;

}

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

{

if (empty()) {

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

return;

}

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

m\_head->next->prev = m\_head;

}

reference front()

{

return m\_head->value;

}

const\_reference front() const

{

return m\_head->value;

}

reference back()

{

return m\_tail->value;

}

const\_reference back() const

{

return m\_tail->value;

}

void pop\_front()

{

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

delete m\_head;

m\_head = m\_tail = nullptr;

return;

}

m\_head = m\_head->next;

delete m\_head->prev;

m\_head->prev = nullptr;

}

void pop\_back()

{

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

delete m\_head;

m\_head = m\_tail = nullptr;

return;

}

m\_tail = m\_tail->prev;

delete m\_tail->next;

m\_tail->next = nullptr;

}

void clear()

{

while (!empty())pop\_back();

}

bool empty() const

{

return m\_head == nullptr;

}

size\_t size() const

{

auto c = m\_head;

size\_t s = 0;

while (c != nullptr) {

s++;

c = c->next;

}

return s;

}

private:

//your private functions

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

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

};

}// namespace stepik