**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 и список»

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7381 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Трушников А.П. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Жангиров Т.Р. |

Санкт-Петербург

2019

**Цель работы.**

Изучить стандартные контейнерыvectorи list языка С++.

**Задание.**

Необходимо реализовать конструкторы, деструктор, оператор присваивания, функцииassign, 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 listбыли 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 А.

**Вывод.**

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

**Приложение А.**

**Файл vector.h.**

#ifndef VECTOR\_H

#define VECTOR\_H

#include <assert.h>

#include <iostream>

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

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

#include <map>

#include <set>

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;

}

friend vector operator^ (const vector <Type> &v1, const vector <Type> &v2){

vector <Type> tmp1(0), tmp2(0);

tmp1 = v1&v2;

tmp2.insert(tmp2.begin(),v1.begin(),v1.end());

tmp2.insert(tmp2.end(),v2.begin(),v2.end());

for (size\_t i = 0; i < tmp2.size();i++)

for (size\_t j = 0; j < tmp1.size(); j++)

if(tmp2[i] == tmp1[j]){

tmp2.erase(tmp2.begin() + i);

break;

}

return tmp2;

}

vector operator^= (vector <Type> &v){

\*this = (\*this)^(v);

return \*this;

}

friend vector operator| (const vector <Type> &v1, const vector <Type> &v2){

std::set<Type> m;

m.insert(v1.begin(),v1.end());

m.insert(v2.begin(),v2.end());

vector <Type> tmp(0);

for (auto it: m) {

tmp.push\_back(it);

}

return tmp;

}

vector operator|= (vector <Type> &v){

\*this = (\*this)|(v);

return \*this;

}

friend vector operator& (const vector <Type> &v1, const vector <Type> &v2){

std::map<Type,Type> m;

vector <Type> tmp(0);

for (int i=0; i<v1.size(); i++)

for (int j=0; j<v2.size(); j++){

if (v1.m\_first[i]==v2.m\_first[j])

m[v1.m\_first[i]]=0;

}

for (typename std::map<Type,Type>::iterator it=m.begin(); it!=m.end(); it++){

Type num = it->first;

tmp.push\_back(num);

}

/\*

for (auto it:m){

Type num = it.first;

tmp.push\_back(num);

}

\*/

return tmp;

}

vector operator&= (vector <Type> &v){

\*this = (\*this)&(v);

return \*this;

}

template <typename InputIterator>

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

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;

}

~**vector**(){

delete []m\_first;

}

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){

delete []m\_first;

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

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

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;

}

// assign method

template <typename InputIterator>

void **assign**(InputIterator first, InputIterator last){

vector tmp(first,last);

swap(*\**this,*tmp*);

}

// resize methods

void **resize**(size\_t count){

iterator end = (count > size()) ? m\_last : m\_first + count;

vector temp(count);

std::move(m\_first, end, temp.m\_first);

swap(*\**this,*temp*);

}

//erase methods

iterator **erase**(const\_iterator pos){

size\_t tmp = pos - m\_first;

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

resize(size() - 1);

return m\_first + tmp;

}

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

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

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

it = erase(it);

}

return it;

}

//insert methods

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

size\_t s\_pos = pos - m\_first;

vector tmp(size()+1);

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

\*(tmp.m\_first + s\_pos) = value;

std::copy(m\_first+s\_pos,m\_last,tmp.m\_first + s\_pos+1);

swap(*\**this,*tmp*);

return m\_first + s\_pos;

}

template <typename InputIterator>

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

size\_t s\_pos = pos - m\_first;

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

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

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

std::copy(m\_first + s\_pos, m\_last, tmp.begin() + s\_pos + (last - first));

swap(*\**this,*tmp*);

return m\_first + s\_pos;

}

//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

#endif // VECTOR\_H

**Приложение Б.**

**Файл list.h.**

#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();

}

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;

}

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;

}

reference **front**(){

return m\_head->value;

}

reference **back**(){

return m\_tail->value;

}

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;

}

iterator **insert**(iterator pos, const Type& value){

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

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);

}

private:

//your private functions

void **clear**(){

while(!empty())

pop\_back();

}

bool **empty**() const{

return m\_head == nullptr;

}

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}// namespace stepik