**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: Контейнеры вектор и лист

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Головина Е.С. |
| Преподаватель |  | Жангиров Т. Р. |

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 ->. Класс list объявлен в данном классе, как дружественный, так как используется в функциях для вставки и удаления элементов из списка.

Реализация класса представлена в приложении Б.

**Выводы.**

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

**ПРИЛОЖЕНИЕ А**

**КОНТЕЙНЕР VECTOR**

#include <cstddef>

#include <stdexcept>

#include <initializer\_list>

#include <utility>

#include <algorithm>

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 value\_type[count]();

m\_last=m\_first+count;

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last){

difference\_type size=last-first;

m\_first = new value\_type[size]();

m\_last = m\_first+size;

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

}

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

size\_t size = init.size();

m\_first = new value\_type[size]();

m\_last = m\_first+size;

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

}

vector(const vector& other){

size\_t size = other.size();

m\_first= new value\_type[size]();

m\_last = m\_first+size;

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

}

vector( vector&& other){

m\_first=nullptr;

m\_last=nullptr;

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

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

}

~vector(){

delete [] m\_first;

}

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

difference\_type offset = pos-m\_first;

size\_t size = this->size()+1;

if (0<= offset && offset <= this->size()){

iterator tmp = m\_first;

m\_first = new value\_type[size]();

std::copy(tmp,tmp+offset,m\_first);

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

std::copy(tmp+offset,m\_last,m\_first+offset+1);

m\_last=m\_first+size;

delete [] tmp;

}

return m\_first+offset;

}

template <typename InputIterator>

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

difference\_type offset = pos-m\_first;

difference\_type quantity = last-first;

size\_t size = this->size()+quantity;

if (0 <= offset && offset <= this->size()){

iterator tmp = m\_first;

m\_first = new value\_type[size]();

std::copy(tmp,tmp+offset,m\_first);

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

std::copy(tmp+offset,m\_last,m\_first+offset+quantity);

m\_last=m\_first+size;

delete [] tmp;

}

return m\_first+offset;

}

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

iterator tmp = m\_first;

size\_t size = this->size()+1;

m\_first = new value\_type[size]();

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

m\_last=m\_first+size;

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

delete [] tmp;

}

void resize(size\_t count){

if (count != size()){

size\_t copy\_size= size() < count ? size(): count;

iterator tmp\_first=m\_first;

m\_first = count ? new value\_type[count](): nullptr;

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

m\_last = m\_first+count;

delete [] tmp\_first;

}

}

iterator erase(const\_iterator pos){

difference\_type offset= pos - m\_first;

iterator tmp = m\_first+offset;

std::cout<<"offset = "<<offset<<'\n';

if (0 <= offset && offset < size()){

std::copy(tmp+1,m\_last,m\_first+offset);

m\_last-=1;

}

return m\_first+offset;

}

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

difference\_type offset = first-m\_first, end = m\_last-last;

if (offset>=0 && end>=0){

iterator tmp=m\_last-end;

std::copy(tmp,m\_last,m\_first+offset);

m\_last-=(last-first);

}

return m\_first+offset;

}

vector& operator=(const vector& other){

if (m\_first != other.m\_first){

iterator tmp\_first = m\_first;

try{

m\_first= other.size() ? new value\_type[other.size()]() : nullptr;

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

}

catch(std::exception &e){

if (m\_first != tmp\_first){

delete [] m\_first;

m\_first=tmp\_first;

}

throw;

}

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

delete [] tmp\_first;

}

return \*this;

}

vector& operator=(vector&& other){

iterator tmp\_first = m\_first;

try{

m\_first=nullptr;

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

}

catch(std::exception &e){

if (m\_first != tmp\_first){

m\_first=tmp\_first;

}

throw;

}

m\_last=nullptr;

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

delete [] tmp\_first;

return \*this;

}

template <typename InputIterator>

void assign(InputIterator first, InputIterator last){

difference\_type size = last - first;

delete [] m\_first;

m\_first = new value\_type[size]();

m\_last = m\_first + size;

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

}

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){

if ( pos>= size() ){

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

}

return m\_first[pos];

}

iterator m\_first;

iterator m\_last;

};

**ПРИЛОЖЕНИЕ Б**

**КОНТЕЙНЕР LIST**

#include <assert.h>

#include <algorithm>

#include <stdexcept>

#include <cstddef>

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;

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(nullptr)

{

}

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<Type> tmp(m\_node);

m\_node = m\_node->next;

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()

{

erase\_nodes(m\_head);

}

list::iterator begin(){

return iterator(m\_head);

}

list::iterator end(){

return iterator();

}

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

{

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

while (tmp != nullptr){

push\_back(tmp->value);

tmp=tmp->next;

}

}

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

{

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

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

}

list& operator= (const list& other)

{

if (m\_head != other.m\_head){

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

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

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

try{

m\_head = m\_tail = nullptr;

while (tmp != nullptr){

push\_back(tmp->value);

tmp=tmp->next;

}

}

catch(std::exception &e){

if (saved\_head != m\_head){

erase\_nodes(m\_head);

m\_head = saved\_head;

}

if (saved\_tail != m\_tail) m\_tail = saved\_tail;

throw;

}

erase\_nodes(saved\_head);

}

return \*this;

}

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

{

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

if (m\_tail != nullptr) m\_tail->next = tmp;

m\_tail = tmp;

if (m\_head == nullptr) m\_head = tmp;

}

iterator insert(iterator pos, const Type& value)

{

node<Type> \* tmp = new node<Type>(value,pos.m\_node,nullptr);

if (begin() == pos){

if (m\_head != nullptr){

m\_head->prev = tmp;

}

else m\_tail = tmp;

m\_head = tmp;

}

else if (end() == pos){

m\_tail->next = tmp;

tmp->prev = m\_tail;

tmp = m\_tail;

}

else{

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

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

pos.m\_node->prev = tmp;

}

return iterator(tmp);

}

iterator erase(iterator pos)

{

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

node<Type> \* next = nullptr;

if (tmp != nullptr) next = tmp->next;

if (m\_head != nullptr){

if (begin() == pos){

m\_head = m\_head->next;

if (m\_head != nullptr){

m\_head->prev = nullptr;

}

else m\_tail = m\_head;

}

else{

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

if (pos.m\_node->next != nullptr){

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

}

else m\_tail = pos.m\_node->prev;

}

delete tmp;

}

return iterator(next);

}

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

{

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

if (m\_head != nullptr) m\_head->prev = tmp;

m\_head = tmp;

if (m\_tail == nullptr) m\_tail = tmp;

}

reference front()

{

if (m\_head != nullptr) return m\_head->value;

}

const\_reference front() const

{

if (m\_head != nullptr) return m\_head->value;

}

reference back()

{

if (m\_tail != nullptr) return m\_tail->value;

}

const\_reference back() const

{

if (m\_tail != nullptr) return m\_tail->value;

}

void pop\_front()

{

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

if (tmp != nullptr){

m\_head = m\_head->next;

delete tmp;

if (m\_head != nullptr) m\_head->prev = nullptr;

else m\_tail = nullptr;

}

}

void pop\_back()

{

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

if (tmp != nullptr){

m\_tail=m\_tail->prev;

delete tmp;

if (m\_tail != nullptr) m\_tail->next = nullptr;

else m\_head = nullptr;

}

}

void clear()

{

if (!empty()){

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

while (tmp->next != nullptr){

tmp = tmp->next;

delete tmp->prev;

}

delete tmp;

}

m\_head = nullptr;

m\_tail = m\_head;

}

bool empty() const

{

if (m\_head == nullptr) return true;

else return false;

}

size\_t size() const

{

if (!empty()){

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

size\_t counter=0;

while (tmp != nullptr){

counter++;

tmp = tmp->next;

}

return counter;

}

else return 0;

}

private:

void erase\_nodes(node<Type> \* start){

if (start != nullptr){

node<Type> \* tmp = start;

while (tmp->next != nullptr){

tmp = tmp->next;

delete tmp->prev;

}

delete tmp;

}

}

//your private functions

node<Type>\* m\_head;

node<Type>\* m\_tail;

};