**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

|  |  |  |
| --- | --- | --- |
| Студент гр. 7382 |  | Гаврилов А.В. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2019

**Цель работы.**

Изучить конейнеры std::list, std::vector, реализовать класс vector и класс list, поведение которых идентично поведению классов std::vector и std::list.

**Постановка задачи.**

1. Необходимо реализовать методы, операторы, конструкторы и деструктор для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector. ﻿Семантику реализованных функций нужно оставить без изменений.
2. Необходимо реализовать список. Поведение реализованных функций должно быть таким же, как у класса std::list. Семантику реализованных функций нужно оставить без изменений.

Требования к реализации: при выполнении этого задания вы можете определять любые вспомогательные функции. Не используйте функции из cstdlib (malloc, calloc, realloc и free).

**Ход работы.**

1. Реализация vector.

На первом этапе были реализованы конструкторы и деструктор для контейнера, на втором этапе операторы присваивания и метод assign, затем метод изменения размера resize и метод удаления элемента или элементов erase. На последнем этапе реализована вставка элементов insert и push\_back.

1. Реализация list.

На первом этапе реализовано:

1. вставка элементов в голову и в хвост(push\_back, push\_front),
2. получение элемента из головы и из хвоста(back, front),
3. удаление из головы, хвоста и очистка(pop\_front, pop\_back, clear)
4. проверка размера(size).

На втором:

1. деструктор
2. конструктор копирования,
3. конструктор перемещения,
4. оператор присваивания.﻿

На 3 этапе ﻿реализован итератор для списка. Реализованы операторы: =, ﻿==, !=, ++ (постфиксный и префиксный), \*, ->.

﻿На последнем шаге с ﻿использованием итераторов реализовано:

1. вставка 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).

**Выводы.**

В результате работы была изучена реализация контейнеров для списка и вектора, изучены методы для работы с данными контейнерами. Реализованы собственные контейнеры для двунаправленного списка и вектора.

**ПРИЛОЖЕНИЕ А**

**Исходный код**

vector.cpp:

#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)

{

my\_alloc(count);

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last)

{

my\_alloc(last-first);

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

}

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

{

my\_alloc(init.size());

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

}

vector(const vector& other)

{

my\_alloc(other.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;

}

void resize(size\_t count)

{

auto old\_first=m\_first;

auto old\_last=m\_last;

my\_alloc(count);

if(count>old\_last-old\_first){

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

}

else{

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

}

delete[] old\_first;

}

//insert methods

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

{

auto old\_first=m\_first;

auto old\_last=m\_last;

my\_alloc(m\_last-m\_first+1);

auto added=pos-old\_first;

std::copy(old\_first,old\_first+added,m\_first);

m\_first[added]=value;

std::copy(old\_first+added,old\_last,m\_first+added+1);

delete[] old\_first;

return m\_first+added;

}

template <typename InputIterator>

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

{

auto old\_first=m\_first;

auto old\_last=m\_last;

auto added\_begin=pos-old\_first;

if(first==last)

return m\_first+added\_begin;

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

my\_alloc(count);

std::copy(old\_first,old\_first+added\_begin,m\_first);

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

std::copy(old\_first+added\_begin,old\_last,m\_first+added\_begin+(last-first));

delete[] old\_first;

return m\_first+added\_begin;

}

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

{

auto count=m\_last-m\_first;

resize(count+1);

m\_first[count]=value;

}

vector& operator=(const vector& other)

{

delete[] m\_first;

my\_alloc(other.size());

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

}

vector& operator=(vector&& other)

{

delete[] m\_first;

m\_first=nullptr;

m\_last=nullptr;

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

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

}

iterator erase(const\_iterator pos)

{

auto old\_first=m\_first;

auto old\_last=m\_last;

my\_alloc(m\_last-m\_first-1);

auto deleted=pos-old\_first;

std::copy(old\_first,old\_first+deleted,m\_first);

std::copy(old\_first+deleted+1,old\_last,m\_first+deleted);

delete[] old\_first;

return m\_first+deleted;

}

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

{

if(first==last)

return m\_first;

auto old\_first=m\_first;

auto old\_last=m\_last;

my\_alloc(m\_last-m\_first-(last-first));

auto deleted\_begin=first-old\_first;

auto deleted\_end=last-old\_first;

std::copy(old\_first,old\_first+deleted\_begin,m\_first);

std::copy(old\_first+deleted\_end,old\_last,m\_first+deleted\_begin);

delete[] old\_first;

return m\_first+deleted\_begin;

}

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

delete[] m\_first;

my\_alloc(last-first);

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) const

{

if (pos >= size())

{

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

}

return m\_first[pos];

}

void my\_alloc(size\_t size){

m\_first=size ? new Type[size]() : nullptr;

m\_last=m\_first+size;

}

private:

iterator m\_first;

iterator m\_last;

};

}

**list.cpp:**

#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 this->m\_node==other.m\_node;

}

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

{

return this->m\_node!=other.m\_node;

}

reference operator \* ()

{

return m\_node->value;

}

pointer operator -> ()

{

return &m\_node->value;

}

list\_iterator& operator ++ ()

{

if(m\_node!=nullptr)

m\_node=m\_node->next;

return \*this;

}

list\_iterator operator ++ (int)

{

if(m\_node!=nullptr)

return list\_iterator(m\_node->next);

return \*this;

}

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()

{

auto pos=m\_head;

while(pos!=nullptr){

delete pos;

pos=pos->next;

}

}

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

{

auto pos=other.m\_head;

while(pos!=nullptr){

push\_back(pos->value);

pos=pos->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)

{

clear();

auto pos=other.m\_head;

while(pos!=nullptr){

push\_back(pos->value);

pos=pos->next;

}

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

void clear()

{

auto pos=m\_head;

while(pos!=nullptr){

delete pos;

pos=pos->next;

}

m\_tail=nullptr;

m\_head=nullptr;

}

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

{

auto el=new node<value\_type>(value,nullptr,nullptr);

if(empty()){

m\_head=el;

m\_tail=el;

}

else{

m\_tail->next=el;

el->prev=m\_tail;

m\_tail=el;

}

}

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

{

auto el=new node<value\_type>(value,nullptr,nullptr);

if(empty()){

m\_head=el;

m\_tail=el;

}

else{

m\_head->prev=el;

el->next=m\_head;

m\_head=el;

}

}

bool empty() const

{

if(m\_head==nullptr)

return true;

return false;

}

iterator insert(iterator pos, const Type& value)

{

if(size()<2 || pos.m\_node==m\_head){

push\_front(value);

return iterator(m\_head);

}

auto cur=m\_head;

while(cur->next!=pos.m\_node){

cur=cur->next;

}

auto el=new node<value\_type>(value, pos.m\_node,cur);

cur->next=el;

pos.m\_node->prev=el;

return iterator(el);

}

size\_t size() const

{

auto pos=m\_head;

size\_t count=0;

while(pos!=nullptr){

count++;

pos=pos->next;

}

return count;

}

iterator erase(iterator pos)

{

auto& deleted = pos.m\_node;

if (deleted == nullptr)

return pos;

if (deleted->next == nullptr)

{

pop\_back();

return nullptr;

}

if (deleted->prev == nullptr)

{

pop\_front();

return iterator(m\_head);

}

deleted->prev->next = deleted->next;

deleted->next->prev = deleted->prev;

auto next = deleted->next;

delete deleted;

return iterator(next);

}

void pop\_front()

{

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

delete m\_head;

m\_head=nullptr;

m\_tail=nullptr;

}

else{

auto el=m\_head;

m\_head=m\_head->next;

m\_head->prev=nullptr;

delete el;

}

}

void pop\_back()

{

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

delete m\_head;

m\_head=nullptr;

m\_tail=nullptr;

}

else{

auto el=m\_tail;

m\_tail=m\_tail->prev;

m\_tail->next=nullptr;

delete el;

}

}

reference front()

{

return m\_head->value;

}

reference back()

{

return m\_tail->value;

}

const\_reference front() const

{

return m\_head->value;

}

const\_reference back() const

{

return m\_tail->value;

}

private:

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}