**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. 7381 |  | Ильясов А.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

**Цель работы**

Исследование контейнеров стандартной библиотеки С++.

**Задание**

Необходимо реализовать контейнеры вектор и список. Поведение реализованных функций должно быть таким же, как у класса std::vector и std::list. Семантику реализованных функций нужно оставить без изменений.

**Ход работы**

Реализация класса vector:

Были реализованы различные конструкторы и деструктор класса аналогичные конструкторам vector из stl.

Далее были добавлены оператор присваивания, методы assign, resize, erase, insert и push\_bask.

В данной реализации класса vector отсутствует резервирование памяти под будущие элементы, и также отсутствует класс iterator.

Реализация класса list:

Были добавлены методы вставки и удаления из головы и хвоста списка, получения элемента из головы и хвоста, проверки размера списка.

Далее были добавлены конструкторы и деструктор аналогичные конструкторам std::list.

Также был реализован класс list\_iterator, с поведением аналогичным std::list.

В конце были добавлены методы вставки и удаления элементов с использованием итераторов списка.

Исходный код каждого написанного класса представлен в приложениях А и Б.

**Выводы**

В ходе написания лабораторной работы были изучены реализация контейнеров стандартной библиотеки С++ и написана собственная реализация упрощенных контейнеров.

**ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД 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):

m\_first(count ? new value\_type[count] : nullptr), m\_last(m\_first+count)

{}

template <typename InputIterator>

vector(InputIterator first, InputIterator last):

m\_first(new value\_type[last-first]), m\_last(m\_first+(last-first))

{

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

}

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

m\_first(new value\_type[init.size()]), m\_last(m\_first+init.size())

{

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

}

vector(const vector& other):

m\_first(other.size() ? new value\_type[other.size()] : nullptr), m\_last(m\_first+other.size())

{

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

}

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;

}

//assignment operators

vector& operator=(const vector& other)

{

delete [] m\_first;

m\_first = new value\_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)

{

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)

{

delete [] m\_first;

m\_first = new value\_type[last-first];

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

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

}

// resize methods

void resize(size\_t count)

{

if (size() == count)

return;

iterator tmp = new value\_type[count];

std::copy(m\_first, count > size() ? m\_last : m\_first+count, tmp);

delete [] m\_first;

m\_first = tmp;

m\_last = m\_first + count;

}

//erase methods

iterator erase(const\_iterator pos)

{

size\_t new\_size = size() - 1;

size\_t part\_before\_erase = pos - m\_first;

iterator tmp = new value\_type[new\_size];

std::copy(m\_first, m\_first+part\_before\_erase, tmp);

std::copy(m\_first+(part\_before\_erase+1), m\_last, tmp+part\_before\_erase);

delete [] m\_first;

m\_first = tmp;

m\_last = m\_first + new\_size;

return m\_first+part\_before\_erase;

}

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

{

size\_t erase\_part = last - first;

size\_t before\_erase = first - m\_first;

size\_t after\_erase = m\_last - last;

size\_t new\_size = size() - erase\_part;

iterator tmp = new value\_type[new\_size];

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

std::copy(m\_first+(before\_erase+erase\_part), m\_last, tmp+before\_erase);

delete [] m\_first;

m\_first = tmp;

m\_last = m\_first + new\_size;

return m\_first+before\_erase;

}

//insert methods

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

{

size\_t new\_size = size() + 1;

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

iterator tmp = new value\_type[new\_size];

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

tmp[insert\_pos] = value;

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

delete [] m\_first;

m\_first = tmp;

m\_last = m\_first + new\_size;

return m\_first+insert\_pos;

}

template <typename InputIterator>

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

{

size\_t insert\_size = last - first;

size\_t new\_size = size() + insert\_size;

iterator tmp = new value\_type[new\_size];

size\_t before\_insert = pos - m\_first;

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

std::copy(first, last, tmp+before\_insert);

std::copy(m\_first+before\_insert, m\_last, tmp+(before\_insert+insert\_size));

delete [] m\_first;

m\_first = tmp;

m\_last = m\_first + new\_size;

return m\_first+before\_insert;

}

//push\_back methods

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

{

resize(size()+1);

at(size()-1) = 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

**ПРИЛОЖЕНИЕ Б**

**ИСХОДНЫЙ КОД 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)

{

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 res\_it = \*this;

m\_node = m\_node->next;

return res\_it;

}

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), m\_size(0)

{

}

~list()

{

clear();

}

list(const list& other)

: m\_head(nullptr), m\_tail(nullptr), m\_size(0)

{

auto tmp = other.m\_head;

while (tmp != nullptr) {

push\_back(tmp->value);

tmp = tmp->next;

}

}

list(list&& other)

: m\_head(other.m\_head), m\_tail(other.m\_tail), m\_size(other.m\_size)

{

other.m\_head = nullptr;

other.m\_tail = nullptr;

}

list& operator= (const list& other)

{

clear();

auto tmp = other.m\_head;

while (tmp != nullptr) {

push\_back(tmp->value);

tmp = tmp->next;

}

return \*this;

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

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

{

if (empty()) {

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

m\_tail = m\_head;

} else {

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

m\_tail = m\_tail->next;

}

m\_size++;

}

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

{

if (empty()) {

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

m\_tail = m\_head;

} else {

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

m\_head = m\_head->prev;

}

m\_size++;

}

iterator insert(iterator pos, const Type& value)

{

if (pos == begin()) {

push\_front(value);

return begin();

}

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

push\_back(value);

return iterator(m\_tail);

}

else {

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 == begin()) {

pop\_front();

return iterator(m\_head);

}

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

return end();

}

else if (pos == iterator(m\_tail)) {

pop\_back();

return end();

}

else {

auto res\_it = pos.m\_node->next;

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

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

delete pos.m\_node;

return res\_it;

}

}

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 (empty()) return;

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

delete m\_head;

m\_tail = nullptr;

m\_head = nullptr;

} else {

m\_head = m\_head->next;

delete m\_head->prev;

m\_head->prev = nullptr;

}

m\_size--;

}

void pop\_back()

{

if (empty()) return;

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

delete m\_head;

m\_tail = nullptr;

m\_head = nullptr;

} else {

m\_tail = m\_tail->prev;

delete m\_tail->next;

m\_tail->next = nullptr;

}

m\_size--;

}

void clear()

{

if (empty()) return;

while (!empty())

pop\_back();

}

bool empty() const

{

return m\_head == nullptr;

}

size\_t size() const

{

return m\_size;

}

private:

//your private functions

size\_t m\_size;

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}// namespace stepik