**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. Векторы. Список**

|  |  |  |
| --- | --- | --- |
| Студент гр. 7382 |  | Находько А.Ю. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

**Цель работы.**

Ознакомится с работой стандартных контейнеров vector и list языка C++, написать собственную реализацию данных контейнеров.

**Задание.**

**Задание 1.**

Необходимо реализовать класс для контейнера vector, в качестве составляющих класса должны присутствовать: конструкторы, деструкторы, операторы присваивания, функция assign, функция insert, функция push\_back. Поведение реализованных функций должно быть таким же как у класса std::vector.

**Задание 2.**

Необходимо реализовать класс для контейнера list, в качестве составляющих класса должны присутствовать: конструкторы, деструкторы, операторы присваивания, функции вставки элементов в голову и в хвост, функции получения элемента из головы и хвоста, функции удаления элементов из головы и хвоста, очистки, проверки размера, итератор. Поведение реализованных функций должно быть таким же как у класса std::list.

**Требования к реализации**:

При выполнении этого задания вы можете 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.**

**Задание 1.**

Класс vector содержит 2 приватных поля: указатель на первый и последний элемент. Также помимо требуемых в задании функций была реализована приватная функция проверки выхода вектора за рамки. Реализация требуемого класса vector содержится в приложении А.

**Задание 2.**

Класс list содержит 2 приватных поля: указатель на первый и последний узел списка. Также приватное поле содержит класс list\_iterator, это поле является указателем на узел списка. Реализация требуемого класса list содержится в приложении Б.

**Выводы.**

В результате выполнения лабораторной работы ознакомился с работой стандартных контейнеров vector и list языка C++, написал собственную реализацию данных контейнеров.

Приложение А

Код программы

**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)

{

if(count != 0)

{

m\_first = new Type[count];

m\_last = m\_first + count;

}

else

{

m\_first = nullptr;

m\_last = nullptr;

}

}

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.m\_first, other.m\_last) {}

vector(vector&& other)

{

swap(\*this, other);

other.m\_first = nullptr;

other.m\_last = nullptr;

}

~vector()

{

delete [] m\_first;

m\_first = nullptr;

m\_last = nullptr;

}

//insert methods

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

{

size\_t offset = pos - m\_first;

resize(size() + 1);

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

std::rotate(m\_first + offset, m\_last - 1, m\_last);

return m\_first + offset;

}

template <typename InputIterator>

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

{

size\_t offset = pos - m\_first;

size\_t insert\_size = last - first;

resize(size() + insert\_size);

std::copy(first, last, m\_last - insert\_size);

std::rotate(m\_first + offset, m\_last - insert\_size, m\_last);

return m\_first + offset;

}

//push\_back methods

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

{

resize(size() + 1);

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

}

//assignment operators

vector& operator=(const vector& other)

{

if(this != &other)

{

vector v1(other);

swap(\*this, v1);

}

return \*this;

}

vector& operator=(vector&& other)

{

if(this != &other)

{

swap(\*this, other);

}

return \*this;

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

vector v1(first, last);

swap(\*this, v1);

}

// resize methods

void resize(size\_t count)

{

vector v1(count);

if(count >= size())

{

std::copy(m\_first, m\_last, v1.m\_first);

}

else

{

std::copy(m\_first, m\_first + count, v1.m\_first);

}

swap(\*this,v1);

}

//erase methods

iterator erase(const\_iterator pos)

{

size\_t elem = pos - m\_first;

std::rotate(m\_first + elem, m\_first + elem + 1, m\_last);

resize(size()-1);

return (m\_first + elem);

}

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

{

size\_t start = first - m\_first;

size\_t erase\_size = last - first;

std::rotate(m\_first + start, m\_first + start + erase\_size, m\_last);

resize(m\_last - m\_first - erase\_size);

return m\_first + start;

}

void swap(vector& a, vector& b)

{

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

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

}

//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

Код программы

#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)

{

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();

}

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

{

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

while (el)

{

push\_back(el->value);

el = el->next;

}

}

list(list&& other) : list()

{

if (this != &other)

swap(other);

}

list& operator= (const list& other)

{

if (this != &other)

list(other).swap(\*this);

return \*this;

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

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

{

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

if(empty())

{

m\_head = tmp;

m\_tail = tmp;

}

else

{

m\_tail->next = tmp;

tmp->prev = m\_tail;

m\_tail = tmp;

}

}

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

{

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

if(empty())

{

m\_head = tmp;

m\_tail = tmp;

}

else

{

m\_head->prev = tmp;

tmp->next = m\_head;

m\_head = tmp;

}

}

iterator insert(iterator pos, const Type& value)

{

if(pos.m\_node == nullptr)

{

push\_back(value);

return iterator(m\_tail);

}

else

{

if(pos.m\_node->prev == nullptr)

{

push\_front(value);

return iterator(m\_head);

}

else

{

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

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

pos.m\_node->prev = tmp;

return iterator(tmp);

}

}

}

iterator erase(iterator pos)

{

if(!pos.m\_node)

{

return nullptr;

}

else if(!pos.m\_node->prev)

{

pop\_front();

return iterator(m\_head);

}

else if(!pos.m\_node->next)

{

pop\_back();

return iterator(m\_tail);

}

else

{

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

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

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

iterator new\_l(pos.m\_node->next);

delete tmp;

return new\_l;

}

}

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 = nullptr;

m\_tail = nullptr;

}

else

{

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 = nullptr;

m\_tail = nullptr;

}

else

{

m\_tail = m\_tail->prev;

delete m\_tail->next;

m\_tail->next = nullptr;

}

}

void clear()

{

while(m\_head)

{

pop\_back();

}

m\_head = nullptr;

m\_tail = nullptr;

}

bool empty() const

{

return m\_head == nullptr;

}

size\_t size() const

{

node<Type> \*list\_el = m\_head;

size\_t i = 0;

while(list\_el)

{

i++;

list\_el = list\_el->next;

}

return i;

}

private:

//your private functions

void swap(list& other)

{

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

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

}

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}// namespace stepik