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. Список»

Студентка гр. 7381 _____ Алясова А.Н.

Преподаватель _____ Жангиров Т.М.

Санкт-Петербург

2019

Цель работы.

Изучить стандартные контейнеры vector и list языка C++.

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 ->.

Реализация класса представлена в приложении Б.

Вывод.

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

Приложение А.

Файл vector.h.

```
#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; //ссылка
		typedef const value_type& const_reference;//константная ссылка

		typedef std::ptrdiff_t difference_type; //указатель на разные типы

		explicit vector(size_t count = 0){ //явный
конструктор(неконвертирующийся конструктор)
			m_first = new Type[count];
			m_last = m_first + count;// use previous step implementation
		}

		template <typename InputIterator>
		vector(InputIterator first, InputIterator last){
			Type *first_t = (Type*)first;
			Type *last_t = (Type*)last;

			m_first = new Type[last_t - first_t];
			m_last = m_first + (last_t - first_t);
			std::copy(first_t, last_t, m_first);// use previous step
implementation
		}

		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);// use previous step implementation
}

~vector(){
     delete[] m_first;
     m_first = nullptr;
     m_last = nullptr;// use previous step implementation
}

vector& operator=(const vector& other){
     if (((void*)this) == ((void*)(&other)))return *this;
     vector v(other);
     swap(*this, v);
     //std::cout « "присваивание копированием\n";
     return *this;
}

vector& operator=(vector&& other){
     if (((void*)this) == ((void*)(&other)))return *this;
     swap(*this, other);
     //std::cout « "присваивание перемещением\n";
     return *this;
}

template <typename InputIterator>
void assign(InputIterator first, InputIterator last){
     vector v(first, last);
     swap(*this, v);
     //std::cout « "assign\n";
}

void resize(size_t count){
     size_t c = m_last - m_first;
     if (count == c)return;
```

```
            vector v(count);
            size_t ms = count < c ? count : c;
            std::copy(m_first, m_first + ms, v.m_first);
            swap(*this, v);
        }

        //insert methods
        iterator insert(const_iterator pos, const Type& value){
            size_t f = (Type*)pos - m_first;
            vector v(size() + 1);
            std::copy(m_first, m_first + f, v.m_first);
            *(v.m_first + f) = value;
            std::copy(m_first + f, m_last, v.m_first + f + 1);
            *this = std::move(v);
            //std::cout « "вставка одного элемента\n";
            return m_first + f;// implement this
        }

        template <typename InputIterator>
        iterator insert(const_iterator pos, InputIterator first,
InputIterator last){
            size_t f = (Type*)pos - m_first;
            size_t l = f + (last - first);
            vector v(size() + last - first);
            std::copy(m_first, m_first + f, v.m_first);
            std::copy(first, last, v.m_first + f);
            std::copy(m_first + f, m_last, v.m_first + l);
            *this = std::move(v);
            //std::cout « "вставка диапозона элементов\n";
            return m_first + f;// implement this
        }

        //push_back methods
        void push_back(const value_type& value){
            resize(size() + 1);
            *(m_last - 1) = value;// implement this
        }

        //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];
        }

        static void swap(vector& a, vector& b) {
            std::swap(a.m_first, b.m_first);
            std::swap(a.m_last, b.m_last);
        }
        //your private functions

    private:
        iterator m_first = nullptr;
        iterator m_last = nullptr;
    };
}// namespace stepik
```

Приложение Б.

Файл list.h.

```
#include "pch.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();
        }

        iterator begin()
        {
            return iterator(m_head);
        }

        iterator end()
        {
            return iterator();
        }
```

```
	list(const list& other)
	{
		auto c = other.m_head;
		while (c != nullptr) {
			push_back(c->value);
			c = c->next;
		}
	}

	list(list&& other)
	{
		m_head = other.m_head;
		m_tail = other.m_tail;
		other.m_tail = nullptr;
		other.m_head = nullptr;
	}

	list& operator= (const list& other)
	{
		if (this == &other)return *this;
		clear();
		for (auto r = other.m_head; r != nullptr; r = r-
>next)push_back(r->value);
		return *this;
	}

	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;
	}

	iterator insert(iterator pos, const Type& value)
	{
		auto ref = pos.m_node;
		if (ref == nullptr) {
```

```
                push_back(value);
                return iterator(m_tail);
            }
            if (ref == m_head) {
                push_front(value);
                return iterator(m_head);
            }
            auto n = new node<Type>(value, ref, ref->prev);
            n->next->prev = n;
            n->prev->next = n;
            return iterator(n);
        }

        iterator erase(iterator pos)
        {
            auto ref = pos.m_node;
            if (ref == m_head) {
                pop_front();
                return iterator(m_head);
            }
            if (ref == m_tail) {
                pop_back();
                return iterator(m_tail);
            }
            ref->prev->next = ref->next;
            ref->next->prev = ref->prev;
            iterator i(ref->next);
            delete ref;
            return i;
        }


        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;
        }
```

```
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 = 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;
        }

        void clear()
        {
            while (!empty())pop_back();
        }

        bool empty() const
        {
            return m_head == nullptr;
        }

        size_t size() const
        {
            auto c = m_head;
            size_t s = 0;
            while (c != nullptr) {
                s++;
                c = c->next;
            }
            return s;
        }

    private:
        //your private functions

        node<Type>* m_head = nullptr;
        node<Type>* m_tail = nullptr;
    };

}// namespace stepik
```