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: «Вектор и список»

Студентка гр. 7381 Кушкоева А.О.
Преподаватель Жангиров Т. Р.

Санкт-Петербург
2019

Цель работы.

Реализовать базовый функционал, семантически аналогичный функционалу из стандартной библиотеки шаблонов для классов вектор и линейный список.

Задание.

Необходимо реализовать конструкторы и деструктор для контейнера вектор. Предполагается реализация упрощенной версии вектора, без резервирования памяти под будущие элементы.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции resize и erase для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции insert и push_back для контейнера вектор.

Поведение реализованных функций должно быть таким же, как у класса std::vector (http://ru.cppreference.com/w/cpp/container/vector). Семантику реализованных функций нужно оставить без изменений.

Необходимо реализовать список со следующими функциями:

1. Вставка элементов в голову и в хвост;
2. Получение элемента из головы и из хвоста;
3. Удаление из головы, хвоста и очистка;
4. Проверка размера.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции:

1. Деструктор;
2. Конструктор копирования;
3. Конструктор перемещения;

4. Оператор присваивания.

На данном шаге необходимо реализовать итератор для списка. Для краткости реализации можно ограничиться однонаправленным изменяемым (неконстантным) итератором. Необходимо реализовать операторы: =, ==, !=, ++ (постфиксный и префиксный), *, ->.

На данном шаге с использованием итераторов необходимо реализовать:

1. Вставку 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).

Поведение реализованных функций должно быть таким же, как у класса std::list (http://ru.cppreference.com/w/cpp/container/list). Семантику реализованных функций нужно оставить без изменений.

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).

Исходный код.

Код класса vector представлен в приложении А.

Код класса list представлен в приложении Б.

Выводы.

В ходе написания лабораторной работы были реализованы классы вектор и список, аналогичные классам из стандартной библиотеки. Полученные знания из предыдущих лабораторных работ были применены в ходе работы над этой работой.

ПРИЛОЖЕНИЕ А

РЕАЛИЗАЦИЯ КЛАССА VECTOR

```
#include "pch.h"
#include <iostream>
#include <assert.h>
#include <algorithm> // std::copy, std::rotate
#include <cstddef> // size_t
#include <initializer_list>
#include <stdexcept>
using namespace std;


//функция компаратор для int
int comp(const void* x1, const void* x2) {
	return *(const int*)x1 - *(const int*)x2;
}

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 = nullptr;
				m_last = nullptr;
			}
			else {
				m_first = new Type[count];
				m_last = m_first + count;
			}
		}

		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.size())
		{
			size_t i = 0;
			for (auto& element : init) {
```

```
            m_first[i] = element;
            i++;
      }
}

vector(const vector& other)
      : vector(other.begin(), other.end())
{

}

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)
{
      if (&other != this) {
            size_t size = other.size();
            delete[] m_first;
            m_first = new Type[size];
            std::copy(other.begin(), other.end(), m_first);
            m_last = m_first + size;
      }
      return *this;
}

vector& operator=(vector&& other)
{
      if (&other != this) {
            delete[] m_first;
            m_first = other.begin();
            m_last = other.end();
            other.m_first = nullptr;
            other.m_last = nullptr;
      }
      return *this;
}

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
      size_t size = last - first;
      delete[] m_first;
      m_first = new Type[size];
      std::copy(first, last, m_first);
      m_last = m_first + size;
```

```
        }

        // resize methods
        void resize(size_t count)
        {
              if (count == m_last - m_first)
                    return;
              size_t size = this->size();
              Type* data = new Type[size];
              std::copy(m_first, m_last, data);
              delete[] m_first;
              m_first = new Type[count];
              m_last = m_first + count;
              if (count > size) {
                    std::copy(data, data + size, m_first);
              }
              else {
                    std::copy(data, data + count, m_first);
              }// implement this

              delete[] data;
        }


        iterator erase(const_iterator pos)
        {
              size_t pos_index = pos - m_first;
              for (size_t i = pos_index; i < size() - 1; i++) {
                    m_first[i] = m_first[i + 1];
              }
              m_last--;
              resize(size());
              return m_first + pos_index;
        }

        iterator erase(const_iterator first, const_iterator last)
        {
              iterator tmp = const_cast<iterator>(first);
              for (size_t i = 0, between = last - first; i <
between; ++i) {
                    tmp = erase(tmp);
              }
              return tmp;
        }

        //insert methods
        iterator insert(const_iterator pos, const Type& value)
        {
              size_t size = m_last - m_first;
              size_t ptr = pos - m_first;
              Type* data = new Type[size];
              std::copy(m_first, m_last, data);
              delete[] m_first;
              m_first = new Type[size + 1];
              m_last = m_first + size + 1;
              std::copy(data, data + ptr, m_first);
              *(m_first + ptr) = value;
              std::copy(data + ptr, data + size, m_first + ptr + 1);
```

```
            return m_first + ptr;// implement this
        }

        template <typename InputIterator>
        iterator insert(const_iterator pos, InputIterator first,
InputIterator last)
        {
            size_t size = m_last - m_first;
            size_t dif = last - first;
            size_t ptr = pos - m_first;
            Type* data = new Type[size];
            std::copy(m_first, m_last, data);
            delete[] m_first;
            m_first = new Type[size + dif];
            m_last = m_first + size + dif;
            std::copy(data, data + ptr, m_first);
            std::copy(first, last, m_first + ptr);
            std::copy(data + ptr, data + size, m_first + ptr +
dif);
            return m_first + ptr;
        }

        //push_back methods
        void push_back(const value_type& value)
        {
            insert(m_last, 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;
        }

        friend vector operator + (vector& a, vector& b) {
            vector <Type> c(a.size() + b.size());
            std::copy(a.m_first, a.m_last, c.m_first);
            std::copy(b.m_first, b.m_last, c.m_first + a.size());
            return c;
        }


    private:
        reference checkIndexAndGet(size_t pos) const
        {
            if (pos >= size())
            {
                throw std::out_of_range("out of range");
            }

            return m_first[pos];
        }

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

ПРИЛОЖЕНИЕ Б

РЕАЛИЗАЦИЯ КЛАССА LIST

```
#include <assert.h>
#include <algorithm>
#include <stdexcept>
#include <cstddef>

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(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;

        list()
                : m_head(nullptr), m_tail(nullptr)
        {
        }

        ~list()
        {
                clear();
        }
```

```
            list(const list& other)
                  : m_head(nullptr), m_tail(nullptr)
            {
                  node<Type>* ptr = other.m_head;
                  while (ptr) {
                        push_back(ptr->value);
                        ptr = ptr->next;
                  }
            }

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

            list& operator= (const list& other)
            {
                  clear();
                  node<Type>* tmp = other.m_head;
                  while (tmp) {
                        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_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;
            }

            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;
        }

        iterator insert(iterator pos, const Type& value)
        {
            if (pos.m_node == nullptr) {
                push_back(value);
                return iterator(m_tail);
            }

            if (pos.m_node->prev == nullptr) {
                push_front(value);
                return iterator(m_head);
            }

            node<Type>* tmp = pos.m_node->prev;
            pos.m_node->prev = new node<Type>(value, pos.m_node,
tmp);
            tmp->next = pos.m_node->prev;

            return iterator(pos.m_node->prev);
        }

        iterator erase(iterator pos)
        {
            if (pos.m_node->next == nullptr) {
                pop_back();
                return iterator();
            }

            if (pos.m_node->prev == nullptr) {
                pop_front();
                return iterator(m_head);
            }

            node<Type>* tmp = pos.m_node;
            delete pos.m_node;
            tmp->prev->next = tmp->next;
            tmp->next->prev = tmp->prev;

            return tmp->next;
        }

        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
    {
        size_t count = 0;
        node<Type>* tmp = m_head;
        while (tmp != nullptr) {
            count++;
            tmp = tmp->next;
        }

        return count;
    }

private:
    //your private functions

    node<Type>* m_head;
    node<Type>* m_tail;
};
```

```
}// namespace stepik
```