**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. 7304 _____ Есиков О.И.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2019

**Цель работы.**

Изучить реализацию контейнеров в языке программирования c++.

**Задача.**

Реализовать конструкторы, деструктор, операторы присваивания, функцию assign, функцию resize, функцию erase, функцию insert и функцию push_back. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Реализовать список со следующими функциями: вставка элемента в голову, вставка элемента в хвост, получение элемента из головы, получение элемента из хвоста, удаление из головы, из хвоста, очистка списка, проверка размера, деструктор, конструктор копирования, конструктор перемещения, оператор присваивания, insert, erase, а также итераторы для списка: =, ==, !=, ++ (постфиксный и префиксный), *, ->. Поведение реализованных функций должно быть таким же, как у класса std::list.

**Ход работы.**

Vector:

-создаётся вектор «a» из пяти элементов;

-он инициализируется по порядку числами от 1 до 5;

-удаляется последний элемент;

-в конец добавляется 10;

-с помощью конструктора копирования создаётся вектор «b» из вектора «a»;

-создаётся вектор «c» из восьми элементов;

-с помощью оператора присваивания вектор «c» становится таким же, как и «a»;

-создаётся вектор «d», состоящий из элементов 7, 8, 9;

-в вектор «c», начиная с третьего элемента вставляются элементы вектора «d»;

-элементы вектора «c» заменяются элементами вектора «a».

List:

-создаётся список «a»;

-последовательно добавляются в конец списка 10 и 5;

-в начало списка добавляется 1;

-удаляется элемент из головы списка;

-создаётся список «b» из списка «a» с помощью конструктора копирования;

-удаляются все элементы из списка «a»;

-создаётся список «c» из списка «b» с помощью конструктора копирования;

-с помощью оператора присваивания список «c» становится таким же, как и «a»;

-происходит сравнение итераторов, указывающих на начало списков «c» и «a»;

-в начало списка «c» вставляется 8;

-снова происходит сравнение итераторов, указывающих на начало списков «c» и «a»;

-в конец списка «c» добавляется 20;

-в начало списка «c» добавляется 3;

-в список «c», с помощью префиксного итератора ++ добавляется вторым элементом 1.

**Результат работы.**

Vector:

vector a: 1 2 3 4 5

a.erase(a.end()): 1 2 3 4

a.push_back(10): 1 2 3 4 10

a.insert(a.begin(), 5): 5 1 2 3 4 10

vector b(a): 5 1 2 3 4 10

vector c(8); c = a: 5 1 2 3 4 10

vecotr d: 9 8 7

c.insert(c.begin() + 2, d.begin(), d.end()): 5 1 9 8 7 2 3 4 10

c.assign(a.begin(), a.end()): 5 1 2 3 4 10

List:

a:

a.push_back(10): 10

a.push_back(5): 10 5

a.push_front(1): 1 10 5

a.pop_front(): 10 5

b(a): 10 5

a.clear():

c(b): 10 5

c=a:

a.begin() == c.begin()

c.insert(c.begin(), 8): 8

a:

a.begin() != c.begin()

c.push_back(20): 8 20

c.push_front(3): 3 8 20

c.insert(++c.begin(), 1): 3 1 8 20

**Выводы.**

В ходе выполнения данной лабораторной работы была изучена реализация таких контейнеров, как вектор и список, были реализованы основные функции для работы с этими контейнерами, как вставка в произвольное место, удаление произвольного элемента, изменение размера, необходимые конструкторы и итераторы для работы с этими контейнерами.

## Приложение А.

## Исходный код.

### Файл myvector.h.

```
#ifndef MYVECTOR_H
#define MYVECTOR_H

#include <assert.h>
#include <algorithm>        // std::copy, std::rotate
#include <cstddef>          // size_t
#include <initializer_list>
#include <stdexcept>
#include <iostream>

using std::copy;
using std::swap;
using std::rotate;

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;

    //constructors
    explicit vector(size_t count = 0) : m_first(new Type[count]),
m_last(&(m_first[count])) {}

    template <typename InputIterator>
    vector(InputIterator first, InputIterator last) :  m_first(new Type[last-
first]), m_last(m_first + (last-first))
    {
            copy(first, last, m_first);
    }

    vector(std::initializer_list<Type> init) : vector(init.begin(), init.end())
{}

    vector(const vector& other) : m_first(new Type[other.size()]),
m_last(&(m_first[other.size()]))
    {
            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)
{
    if(this != &other)
    {
        vector temp(other);
        swap(m_first, temp.m_first);
        swap(m_last, temp.m_last);
    }
    return *this;
}

vector& operator = (vector&& other)
{
    if(this != &other)
    {
        swap(m_first, other.m_first);
        swap(m_last, other.m_last);
    }
    return *this;
}

//assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    vector temp(first, last);
    swap(m_first, temp.m_first);
    swap(m_last, temp.m_last);
}

//resize methods
void resize(size_t count)
{
    vector temp(count);
    size_t delta = count < this->size() ? count : this->size();
    copy(m_first, m_first + delta, temp.m_first);
    swap(m_first, temp.m_first);
    swap(m_last, temp.m_last);
}

//erase methods
iterator erase(const_iterator pos)
{
    difference_type result = pos - m_first;
    rotate(m_first + result, m_first + result + 1, m_last);
    resize(size() - 1);
    return m_first + result;
}

iterator erase(const_iterator first, const_iterator last)
{
    difference_type result = first - m_first;
    difference_type delta = last - first;
    rotate(m_first + result, m_first + result + delta, m_last);
    resize(size() - delta);
    return m_first + result;
}

//insert methods
iterator insert(const_iterator pos, const Type& value)
{
```

```
    difference_type result = pos - m_first;
    resize(size() + 1);
    rotate(m_first + result, m_last - 1, m_last);
    m_first[result] = value;
    return m_first + result;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
    difference_type i = last - first;
    iterator temp = const_cast<iterator>(pos);
    while(i)
    {
        i--;
        temp = insert(temp, *(first + i));
    }
    return temp;
}

//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 void operator << (std::ostream& os, vector& element)
    {
        for(size_t i = 0; i < element.size(); i++)
            os << element.m_first[i] << " ";
        os << std::endl;
    }

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

#endif // MYVECTOR_H
```

## Файл mylist.h.

```
#ifndef MYLIST_H
#define MYLIST_H

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

using std::swap;

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 temp(*this);
```

```
        ++(*this);
        return temp;
    }

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) : list()
    {
        node<Type>* temp = other.m_head;
        while(temp)
        {
            push_back(temp->value);
            temp = temp->next;
        }
    }

    list(list&& other) : list()
    {
        if(this != &other)
        {
            std::swap(m_head, other.m_head);
            std::swap(m_tail, other.m_tail);
        }
    }

    list& operator = (const list& other)
    {
        if(this != &other)
        {
            list temp(other);
            std::swap(m_head, temp.m_head);
            std::swap(m_tail, temp.m_tail);
        }
        return *this;
    }

    void push_back(const value_type& value)
    {
        node<Type> *temp = new node<Type>(value, nullptr, nullptr);
        if(!empty())
        {
```

```
            temp->prev = m_tail;
            m_tail->next = temp;
            m_tail = temp;
        }
        else
        {
            m_head = temp;
            m_tail = temp;
        }
    }

    void push_front(const value_type& value)
    {
        node<Type> *temp = new node<Type>(value, nullptr, nullptr);
        if(!empty())
        {
            temp->next = m_head;
            m_head->prev = temp;
            m_head = temp;
        }
        else
        {
            m_head = temp;
            m_tail = temp;
        }
    }

    iterator insert(iterator pos, const Type& value)
    {
        if(pos.m_node == nullptr)
        {
            push_back(value);
            return iterator(m_tail);
        }
        if(pos.m_node == m_head)
        {
            push_front(value);
            return iterator(m_head);
        }
        else
        {
            node<Type>* temp = new node<Type>(value, pos.m_node, pos.m_node-
>prev);
            pos.m_node->prev->next = temp;
            pos.m_node->prev = temp;
            return iterator(temp);
        }
    }

    iterator erase(iterator pos)
    {
        if(pos.m_node == NULL)
        {
            return NULL;
        }
        if(pos.m_node == m_head)
        {
            pop_front();
            return iterator(m_head);
        }
        if(pos.m_node == m_tail)
        {
            pop_back();
```

```
        return iterator(m_tail);
    }
    else
    {
        pos.m_node->prev->next = pos.m_node->next;
        pos.m_node->next->prev = pos.m_node->prev;
        node<Type>* temp = pos.m_node;
        iterator result(pos.m_node->next);
        delete temp;
        return result;
    }
}

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())
    {
        if(size() == 1)
        {
            m_head = NULL;
            m_tail = NULL;
        }
        else
        {
            node<Type>* temp = m_head->next;
            temp->prev = nullptr;
            delete m_head;
            m_head = temp;
        }
    }
}

void pop_back()
{
    if(!empty())
    {
        if(size() == 1)
        {
            m_head = NULL;
            m_tail = NULL;
        }
        else
        {
```

```
                node<Type>* temp = m_tail->prev;
                temp->next = nullptr;
                delete m_tail;
                m_tail = temp;
            }
        }
    }

    void clear()
    {
        while(m_head != NULL)
        {
            m_tail = m_head->next;
            delete m_head;
            m_head = m_tail;
        }
    }

    bool empty() const
    {
        if(m_head)
            return false;
        else
            return true;
    }

    size_t size() const
    {
        node<Type>* temp = m_head;
        size_t count = 0;
        while(temp != NULL)
        {
            count++;
            temp = temp->next;
        }
        return count;
    }

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

    list::iterator end()
    {
        return iterator();
    }

    friend void operator << (std::ostream& os, list& element)
    {
            node<Type>* temp = element.m_head;
            while(temp)
            {
                os << temp->value << " ";
                temp = temp->next;
            }
            os << std::endl;
    }

private:
    //your private functions

    node<Type>* m_head;
```

```
    node<Type>* m_tail;
    };
}// namespace stepik

#endif // MYLIST_H
```