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 _____ Абдульманов Э.М

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Изучить реализацию таких контейнеров как vector и list 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: Все функции были реализованы в соответствие с поведением класса std::vector

a. Были реализованы конструкторы копирования и перемещения.
b. Были реализованы операторы присвоения и функция assign.
c. Были реализованы следующие функции: resize,erase,push_back,insert.

List: Все функции были реализованы в соответствие с поведением класса std::list

a. Были реализованы функции: вставка элемента в голову, вставлка элемент в хвост, получение элемента из головы, получение элемента из хвоста, удаление из головы, удаление из хвоста, очистка списка, проверка размера.

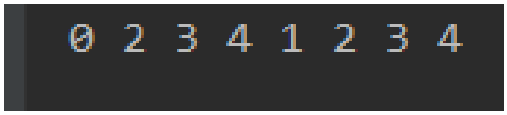
b. Были реализованы: деструктор, конструктор копирования, конструктор перемещения, оператор присвоения.
c. Были реализованы опреторы для итератора списка: =, ==, !=, ++ (постфиксный и префиксный), *, ->.
d. Были реализованы функции удаления элемента и вставка элемента в произвольное место.

Результат работы.

Vector:

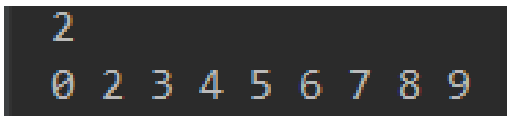
1. Был создан vector1(5), инициализирован, далее был создан vector2(vector1).

2. Был создан vector1(5), инициализирован, далее была вызвана функция vector1.insert(vector1.begin()+1,vector1.begin()+2,vector1.begin()+5);

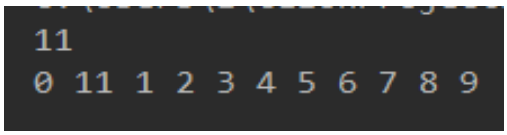


List:

1. Был удален элемент 1.



2. Был вставлен элемент 11 перед элементом 1.



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 vector.h.

```
#ifndef VECTOR_VECTOR_H
#define VECTOR_VECTOR_H
#include <assert.h>
#include <algorithm> // std::copy, std::rotate
#include <cstddef> // size_t
#include <initializer_list>
#include <stdexcept>

using namespace std;
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)
            memoryReserve(count);
        else
            m_first=m_last=NULL;
    }

    template <typename InputIterator>
    vector(InputIterator first, InputIterator last):vector(last-first)
    {
        if (last - first)
            copy(first,last,m_first);
    }

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

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

    vector(vector&& other)
    {
        m_first=other.begin();
        m_last=other.end();
        other.m_first=other.m_last=NULL;
    }

    ~vector()
    {
        delete [] m_first;
        m_last=m_first= NULL;
    }
```

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

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

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    vector a(first,last);
    (*this)=move(a);
}

 void resize(size_t count)
{
    int size_vec=size();
    if(count!=size_vec){
        if(count<size_vec)
            m_last=m_first+count;
        else{
            vector a(count);
            copy(m_first,m_last,a.m_first);
            (*this)=move(a);
        }
    }
}


iterator erase(const_iterator pos)
{
    iterator element=m_first+(pos-m_first);
    rotate(element,element+1,m_last);
    m_last--;
    return element;
}

iterator erase(const_iterator first, const_iterator last)
{
    iterator begin=m_first+(first-m_first);
    iterator end=m_first+(last-m_first);
    rotate(begin,end,m_last);
    m_last=m_last-(last-first);
    return begin;
}

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

```
    size_t offset=(pos-m_first);
    vector a(m_first,m_first+offset);
    a.push_back(value);
    for(iterator it=m_first+(pos-m_first);it!=m_last;it++)
        a.push_back(*it);
    (*this)=move(a);
    return m_first+offset;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
    size_t offset=(pos-m_first);
    vector a(m_first,m_first+offset);
    for(InputIterator it=first;it!=last;it++)
        a.push_back(*it);
    for(iterator it=m_first+offset;it!=m_last;it++)
        a.push_back(*it);
    (*this)=move(a);
    return m_first+offset;
}

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

    void memoryReserve(size_t count){
        m_first = new value_type[count];
        m_last = m_first + count;
    }
    //your private functions

private:
    iterator past;
    iterator m_first;
    iterator m_last;
};
#endif //VECTOR_VECTOR_H
```

Файл list.h.

```
#ifndef VECTOR_LIST_H
#define VECTOR_LIST_H
#include <assert.h>
#include <algorithm>
#include <stdexcept>
#include <cstddef>
#include <iostream>

using  namespace std;
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;

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)
    {
        if(this!=&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->prev;
        return *this;
    }

    list_iterator operator -- (int)
    {
        m_node=m_node->prev;
        return m_node->next;
    }

    list_iterator& operator ++ ()
    {
        m_node=m_node->next;
        return *this;
    }

    list_iterator operator ++ (int)
    {
        m_node=m_node->next;
        return m_node->prev;
    }

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)
    {
        for(node<Type>* element=other.m_head;element!=NULL;push_back(element-
>value),element=element->next);
    }
```

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

    list& operator= (const list& other)
    {
        if(this!=&other){
            clear();
            for(node<Type>* element=other.m_head;element!=NULL;push_back(element-
>value),element=element->next);
        }
        return *this;
    }

    void push_back(const value_type& value)
    {
        if(empty())
            m_head=m_tail=new node<Type>(value,NULL,NULL);
        else{
            m_tail->next=new node<Type>(value,NULL,m_tail);
            m_tail=m_tail->next;
        }
    }

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

    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(m_head==m_tail){
                delete(m_head);
```

```
                m_head=m_tail=NULL;
            }
            else{
                m_head=m_head->next;
                delete(m_head->prev);
                m_head->prev=NULL;
            }
        }
    }

    void pop_back()
    {
        if(!empty()){
            if(m_head==m_tail){
                delete(m_head);
                m_head=m_tail=NULL;
            }
            else{
                m_tail=m_tail->prev;
                delete(m_tail->next);
                m_tail->next=NULL;
            }
        }
    }

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

    }

    iterator erase(iterator pos)
    {
        iterator ret=pos.m_node->next;
        if(pos.m_node==m_head)
            pop_front();
        else if(pos.m_node==m_tail) {
            pop_back();
            ret=m_tail;
        }
        else {
            pos.m_node->prev->next = pos.m_node->next;
            pos.m_node->next->prev = pos.m_node->prev;
        }
        return ret;
    }

    void clear()
    {
        node<Type>* past;
        for(node<Type>* element=m_head;element!=NULL;past=element,element=element-
>next,delete past);
        m_head=m_tail=NULL;
```

```
    }

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

    size_t size() const
    {
        size_t size=0;
        for(node<Type>* element=m_head;element!=NULL;element=element->next,size++);
        return size;
    }

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

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

private:
    //your private functions
    node<Type>* m_head;
    node<Type>* m_tail;
};
#endif //VECTOR_LIST_H
```