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»

Студент гр. 7304 _____ Комаров А.О.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы:

Изучить реализацию контейнеров 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:

```
VECTOR
Object id: 0
(x, y): 5, 3
Цвет фигуры: 32 32 43
Координаты фигуры:
(12.3623;-11.8286)
(-6.58935;-10.3863)
(-4.42983;-24.2188)
Object id: 1
(x, y): 5, 3
Цвет фигуры: 43 54 12
Координаты фигуры:
(-76.302;31.7174)
(-24.5497;-0.860114)
(-29.9266;9.86785)
```

List:

```
LIST

Object id: 3
(x, y): 40, 40
Цвет фигуры: 32 32 43
Координаты фигуры:
(82.0704;-44.735)
(-26.2249;-36.4934)
(-13.8847;-115.536)
 Object id: 4
(x, y): 1720, 160
Цвет фигуры: 43 54 12
Координаты фигуры:
(-1532.08;1308.7)
```

Вывод.

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

```
#include "Vector.h"
using namespace std;
template <typename Type>
Vector<Type>::Vector(size_t count)
{
    m_first = new Type[count];
    m_last = &(m_first[count]);
}
template <typename Type>
template <typename InputIterator>
Vector<Type>::Vector(InputIterator first, InputIterator last)
{
    m_first = new value_type[last-first];
    m_last = m_first + (last-first);
    std::copy(first, last, m_first);
}
template <typename Type>
Vector<Type>::Vector(std::initializer_list<Type> init) :
Vector<Type>::Vector(init.begin(), init.end())
{}
template <typename Type>
Vector<Type>::Vector(const Vector& other) : Vector(other.begin(),
other.end())
{}
template <typename Type>
Vector<Type>::Vector(Vector&& other) : m_first(other.begin()),
m_last(other.end())
{
    other.m_first = other.m_last = NULL;
}
template <typename Type>
Vector<Type>::~Vector()
{
    delete [] m_first;
    m_first = m_last = NULL;
}
//assignment operators
template <typename Type>
Vector<Type>& Vector<Type>::operator=(const Vector<Type> &other)
{
    if ( ((void *)this) == ((void *)(&other)) )
        return *this;
    Vector a(other);
    swap(*this, a);
    return *this;
}
// Оператор перемещения
template <typename Type>
Vector<Type>& Vector<Type>::operator=(Vector<Type>&& other)
{
    if ( ((void *)this) == ((void *)(&other)) )
        return *this;
    swap( *this, other);
    return *this;
}
```

```
// assign method
template <typename Type>
template <typename InputIterator>
void Vector<Type>::assign(InputIterator first, InputIterator last)
{
    Vector a(first, last);
    swap(*this, a);
}
// resize methods
template <typename Type>
void Vector<Type>::resize(size_t count)
{
    if ( count == (m_last-m_first) ){
        return;
    }
    if ( count < (size()) )
    {
        Vector<Type> a(m_first, m_first+count);
        swap(*this, a);
    }
    else
    {
        Vector<Type> a(count);
        std::copy(m_first, m_last, a.m_first);
        swap(*this, a);
    }
}
// Удаление элемента
template <typename Type>
typename Vector<Type>::iterator
Vector<Type>::erase(Vector<Type>::const_iterator pos)
{
    size_t offset = pos-m_first;
    std::rotate( m_first+offset, m_first+offset+1, m_last);
    resize(size()-1);
    return m_first + offset;
}
// Удаление элементов
template <typename Type>
typename Vector<Type>::iterator
Vector<Type>::erase(Vector<Type>::const_iterator first,
Vector<Type>::const_iterator last)
{
    size_t offset = first-m_first;
    std::rotate( m_first + offset, m_first + (last-m_first), m_last);
    resize(size() - (last-first));
    return m_first + offset;
}
// Вставляет value перед элементом, на который указывает pos.
template <typename Type>
typename Vector<Type>::iterator
Vector<Type>::insert(Vector<Type>::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;
}
// Вставляет элементы из диапазона [first, last) перед элементом, на который
указывает pos
```

```
template <typename Type>
template <typename InputIterator>
typename Vector<Type>::iterator
Vector<Type>::insert(Vector<Type>::const_iterator pos, InputIterator first,
InputIterator last)
{
    size_t offset = pos - m_first;
    resize( size() + (last-first));
    std::copy(first, last, m_last - (last-first));
    std::rotate(m_first+offset, m_last - (last-first) , m_last);
    return m_first + offset;
}
// Добавляет данный элемент value до конца контейнера.
template <typename Type>
void Vector<Type>::push_back(const value_type& value)
{
    resize(size()+1);
    *(m_last-1) = value;
}
//at methods
template <typename Type>
typename Vector<Type>::reference Vector<Type>::at(size_t pos)
{
    return checkIndexAndGet(pos);
}
template <typename Type>
typename Vector<Type>::const_reference Vector<Type>::at(size_t pos) const
{
    return checkIndexAndGet(pos);
}
template <typename Type>
//[] operators
typename Vector<Type>::reference Vector<Type>::operator[](size_t pos)
{
    return
            m_first[pos];
}
template <typename Type>
typename Vector<Type>::const_reference Vector<Type>::operator[](size_t pos)
const
{
    return m_first[pos];
}
template <typename Type>
//*begin methods
typename Vector<Type>::iterator Vector<Type>::begin()
{
    return m_first;
}
template <typename Type>
typename Vector<Type>::const_iterator Vector<Type>::begin() const
{
    return m_first;
}
template <typename Type>
//*end methods
typename Vector<Type>::iterator Vector<Type>::end()
{
    return m_last;
}
template <typename Type>
```

```
typename Vector<Type>::const_iterator Vector<Type>::end() const
{
    return m_last;
}
template <typename Type>
//size method
size_t Vector<Type>::size() const
{
    return m_last - m_first;
}
template <typename Type>
//empty method
bool Vector<Type>::empty() const
{
    return m_first == m_last;
}
template<typename Type>
typename Vector<Type>::reference Vector<Type>::checkIndexAndGet(size_t pos)
const
{
    if (pos >= size())
    {
        throw std::out_of_range("out of range");
    }
    return m_first[pos];
}
template <typename Type>
void Vector<Type>::swap(Vector &v1,Vector &v2)
{
    std::swap(v1.m_first, v2.m_first);
    std::swap(v1.m_last, v2.m_last);
}

#include "list_basic.h"

  template <class Type>
    node<Type>::node(const Type& value, node<Type>* next, node<Type>* prev)
      : value(value), next(next), prev(prev)
    {
    }
    template <class Type>
      list_iterator<Type>::list_iterator()
        : m_node(NULL)
      {
      }
      template <class Type>
      list_iterator<Type>::list_iterator(const list_iterator& other)
        : m_node(other.m_node)
      {
      }
        template <class Type>
       list_iterator<Type>& list_iterator<Type>::operator=(const
list_iterator<Type>& other)
      {
        m_node = other.m_node;
        return *this;
      }
     template <class Type>
      bool list_iterator<Type>::operator == (const list_iterator<Type>&
other) const
      {
```

```
        return m_node == other.m_node;
      }
    template <class Type>
      bool list_iterator<Type>::operator != (const list_iterator<Type>&
other) const
      {
        return m_node != other.m_node;
      }
      template <class Type>
      typename list_iterator<Type>::reference list_iterator<Type>::operator *
()
      {
        return m_node->value;
      }
        template <class Type>
      typename list_iterator<Type>::pointer list_iterator<Type>::operator ->
()
          {
            return &(m_node->value);
          }
        //prefix
      template <class Type>
      list_iterator<Type>& list_iterator<Type>::operator ++ ()
      {
          m_node=m_node->next;
          return *this;
      }
    //postfix
      template <class Type>
      list_iterator<Type> list_iterator<Type>::operator ++ (int)
      {
          list_iterator tmp(*this);
          ++(*this);
          return tmp;
      }
      template <class Type>
        list_iterator<Type>::list_iterator(node<Type>* p)
        : m_node(p)
      {
      }
  template <class Type>
     List<Type>::List()
      : m_head(nullptr), m_tail(nullptr)
    {
    }
     template <class Type>
        List<Type>::~List(){
            clear();
        }
        template <class Type>
           List<Type>::List(const List<Type>& other) : m_head(nullptr),
m_tail(nullptr){
            copy(const_cast<List&>(other));
        }
           template <class Type>
              List<Type>::List(List&& other) : List(){
            if(this!= &other){
                swap(other);
            }
        }
        template <class Type>
```

```
List<Type>& List<Type>::operator= (const List<Type>& other){
    if(this!= &other){
        List tmp(other);
        tmp.swap(*this);
    }
    return *this;
}
template <class Type>
typename List<Type>::iterator List<Type>::begin()
    {
      return iterator(m_head);
    }
    template <class Type>
    typename List<Type>::iterator List<Type>::end()
    {
      return iterator();
    }
template <class Type>
void List<Type>::push_back(const value_type& value){
        node<Type>* tmp = new node<Type>(value, nullptr, nullptr);
        if(empty()){
            m_head=m_tail=tmp;
        }
        else{
            m_tail->next=tmp;
            tmp->prev = m_tail;
            m_tail = tmp;
        }
    }
template <class Type>
    void  List<Type>::push_front(const value_type& value){
         node<Type>* tmp =new node<Type>(value, nullptr, nullptr);
         if(empty()){
             m_head=m_tail=tmp;
         }
         else{
             m_head->prev=tmp;
             tmp->next = m_head;
             m_head = tmp;
         }
    }
    template <class Type>
    void List<Type>::pop_front(){
        if(!empty()){
            if(size()!=1){
                node<Type>* tmp = m_head->next;
                tmp->prev = nullptr;
                delete m_head;
                m_head = tmp;
            }
            else{
                delete m_head;
                m_head = m_tail = nullptr;
            }
        }
    }
    template <class Type>
    void List<Type>::pop_back(){
        if(!empty()){
            if(size()!=1){
                node<Type>* tmp = m_tail->prev;
```

```
                    tmp->next = nullptr;
                    delete m_tail;
                    m_tail = tmp;
                }
                else{
                    delete m_head;
                    m_head = m_tail = nullptr;
                }
            }
        }
        template <class Type>
        typename List<Type>::iterator
List<Type>::insert(List<Type>::iterator pos, const Type& value)
            {
                if(!pos.m_node){
                    push_back(value);
                    return iterator(m_tail);
                }
                else if(!pos.m_node->prev){
                    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 = pos.m_node->prev->next = tmp;
                    return iterator(tmp);
                }
            }
            template <class Type>
            typename List<Type>::iterator List<Type>::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 newest(pos.m_node->next);
                    delete tmp;
                    return newest;
                }
            }
    template <class Type>
    typename List<Type>::reference List<Type>::front(){
            return m_head->value;
        }
        template <class Type>
       typename List<Type>::const_reference List<Type>::front() const{
            return m_head->value;
        }
            template <class Type>
```

```
typename List<Type>:: reference List<Type>::back(){
     return m_tail->value;
 }
     template <class Type>
typename List<Type>:: const_reference List<Type>::back() const{
     return m_tail->value;
 }
     template <class Type>
 bool List<Type>::empty() const{
   return (m_head == nullptr);
 }
     template <class Type>
 size_t List<Type>::size() const{
   node<Type>* tmp = m_head;
   size_t i = 0;
   while(tmp){
       i++;
       tmp = tmp->next;
   }
   return i;
 }
 template <class Type>
void List<Type>::copy(List & lst){
       node<Type>* tmp = lst.m_head;
       while(tmp){
           push_back(tmp->value);
           tmp=tmp->next;
       }
 }
 template <class Type>
void List<Type>::swap(List & lst)
 {
     std::swap(m_head, lst.m_head);
     std::swap(m_tail, lst.m_tail);
 }
 template <class Type>
void List<Type>::clear(){
      while(m_head){
          m_tail = m_head->next;
          delete m_head;
          m_head = m_tail;
      }
 }
```