**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.**

Студент гр.7303 Никитенко Д.А

Преподаватель Размочаева Н.В

Санкт-Петербург

2019 г.

**Цель работы**

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

**Задание**

Необходимо реализовать конструкторы и деструктор для контейнера вектор.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор.

Необходимо реализовать функции resize и erase для контейнера вектор.

Необходимо реализовать функции insert и push_back для контейнера вектор.

Необходимо реализовать список со следующими функциями: вставка элементов в голову и в хвост; получение элемента из головы и из хвоста; удаление из головы, хвоста и очистка; проверка размера.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции: деструктор; конструктор копирования; конструктор перемещения; оператор присваивания.

Необходимо реализовать операторы: =, ==, !=, ++ (постфиксный и префиксный), *, ->.

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).

**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.

**Вывод**

В ходе написания лабораторной работы были реализованы классы vector и list, аналогичные класса из стандартной библиотеки.

# ПРИЛОЖЕНИЕ А
# РЕАЛИЗАЦИЯ КЛАССА ВЕКТОР

```
#ifndef VECTOR_H
#define 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;
    typedef const value_type& const_reference;

    typedef std::ptrdiff_t difference_type;

    explicit vector(size_t count = 0)
    {
        m_first = new value_type[count];
        m_last = m_first + count;
    }
    template <typename InputIterator>
    vector(InputIterator first, InputIterator last)
    {
        int count = last - first;
        m_first = new value_type[count];
        m_last = m_first + count;
        std::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.m_first;
        m_last = other.m_last;
        other.m_first = other.m_last = nullptr;
    }

    void swap(vector &other)
    {
        std::swap(m_first, other.m_first);
        std::swap(m_last, other.m_last);
    }

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

```
//assignment operators
vector& operator=(const vector& other)
{
  if(m_first)
        delete [] m_first;
    if(other.size()){
        m_first = new Type[other.size()];
        m_last = m_first + other.size();
        std::copy(other.m_first,other.m_last,m_first);
    }
    else m_first = m_last = nullptr;
}

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

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    if(m_first)
        delete [] m_first;
    if(first!=last)
    {
        size_t s = last - first;
        m_first = new Type[s];
        m_last = m_first + s;
        std::copy(first, last, m_first);
    }
    else m_first = m_last = nullptr;
}

// resize methods
void resize(size_t count)
{
    if(count==0)
    {
        delete [] m_first;
        m_first = m_last = nullptr;
        return;
    }
    iterator new_first = new Type[count];
    std::copy(m_first,count>size()?m_last:m_first+count,new_first);
    delete [] m_first;
    m_first = new_first;
    m_last = m_first + count;
}

//erase methods
iterator erase(const_iterator pos)
{
    difference_type  d = pos - m_first;
```

```
        std::rotate(m_first + d, m_first + d + 1, m_last);
        resize(size()-1);
        return m_first+d;
    }

    iterator erase(const_iterator first, const_iterator last)
    {
        difference_type  d = first - m_first;
        difference_type  erase_d = last - first;
        std::rotate(m_first + d, m_first +d + erase_d, m_last);
        resize(m_last - m_first - erase_d);
        return m_first + d;
    }

    //insert methods
    iterator insert(const_iterator pos, const Type& value)
    {
        size_t len = size()+1;
        iterator new_m_first = new value_type[len];
        size_t first_part = pos-m_first;
        std::copy(m_first, m_first+first_part, new_m_first);
        new_m_first[first_part] = value;
        std::copy(m_first+first_part, m_last, new_m_first+first_part);
        delete [] m_first;
        m_first = new_m_first;
        m_last = m_first+len;
        return m_first+first_part;
    }

    template <typename InputIterator>
    iterator insert(const_iterator pos, InputIterator first, InputIterator
last)
    {
        size_t len = size()+(last-first);
        iterator new_m_first = new value_type[len];
        size_t first_part = pos-m_first;
        std::copy(m_first, m_first+first_part, new_m_first);
        std::copy(first, last, new_m_first+first_part);
        std::copy(m_first+first_part, m_last, new_m_first+first_part+(last-
first));
        delete[] m_first;
        m_first = new_m_first;
        m_last = m_first+len;
        return m_first+first_part;
    }

    //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];
  }

  //your private functions

private:
  iterator m_first;
  iterator m_last;
};
```

```
}// namespace stepik
```

#endif // VECTOR_H

# ПРИЛОЖЕНИЕ Б
# РЕАЛИЗАЦИЯ КЛАССА СПИСОК

```
#ifndef LIST_H
#define LIST_H
#include <assert.h>
#include <algorithm>
#include <stdexcept>
#include <cstddef>
#include <utility>
#include "sector.h"
#include "parall.h"
#include "ellipse.h"
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) {

        if (m_node != other.m_node) {
            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 ++ () {

        if (m_node)
            m_node = m_node->next;
        return *this;
    }

    list_iterator operator ++ (int) {

        list_iterator iterator = *this;
        ++(*this);
        return iterator;
    }

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

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

list(const list& other)
{
    m_head = new node <value_type>(other.m_head->value, nullptr, nullptr);
    node <value_type>* cur_prev = m_head;
    node <value_type>* cur = m_head->next;
    node <value_type>* other_cur = other.m_head->next;

    while (other_cur)
    {
        cur = new node <value_type>(other_cur->value, nullptr, cur_prev);
        cur_prev->next = cur;
        other_cur = other_cur->next;
        cur_prev = cur;
    }

    m_tail = cur;
}

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

list& operator= (const list& other)
{
    if (m_head == other.m_head)
    {

        return *this;
    }

    if (!other.m_head) {

        m_head = nullptr;
        m_tail = nullptr;

        return *this;
    }

    this->clear();

    m_head = new node <value_type>(other.m_head->value, nullptr, nullptr);
    node <value_type>* cur_prev = m_head;
    node <value_type>* cur = m_head->next;
    node <value_type>* other_cur = other.m_head;

    while (other_cur) {

        cur = new node <value_type>(other_cur->value, nullptr, cur_prev);
        cur_prev->next = cur;
        other_cur = other_cur->next;
```

```
            cur_prev = cur;
        }

        m_tail = cur;
        return *this;
    }

    void push_back(const value_type& value)
    {
        node <value_type>* elem = new node <value_type>(value, nullptr,
m_tail);
        if (empty())
        {
            m_head = m_tail = elem;
            return;
        }
        m_tail->next = elem;
        m_tail = elem;
    }

    void push_front(const value_type& value)
    {
         node <value_type>* elem = new node <value_type>(value, m_head,
nullptr);
         if (empty())
         {
            m_head = m_tail = elem;
            return;
        }
        m_head->prev = elem;
        m_head = elem;
    }


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

    iterator erase(iterator pos)
    {
        if (pos.m_node == m_head) {

            pop_front();
            return iterator(m_head);
        }
        else if (pos.m_node == m_tail) {

            pop_back();
```

```
        return iterator(m_tail);
    }

    pos.m_node->prev->next = pos.m_node->next;
    pos.m_node->next->prev = pos.m_node->prev;

    iterator next = pos.m_node->next;
    delete pos.m_node;
    return iterator(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;
    }
    else{
        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;
    }
    else{
        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;
    }

     friend const list<Type> operator+(const list<Type>& left, const
list<Type>& right)
     {
         list<Type> res;
         Sector* sector = new Sector();
         Parall* parall = new Parall();
         Ellipse* ellipse = new Ellipse();
         list<Shape*>::iterator cur = left.begin();
         for (size_t i = 0; i < left.size();++i)
         {
             if ((*cur)->shape() == "Sector")
             {
                 for (size_t j = 0;j < (*cur)->getCoords().size();++j)
                 {
                    sector->setCoord(j,sector->getCoords()[j] + (*cur)-
>getCoords()[j]);

                 }
             }
             if ((*cur)->shape() == "Parall")
             {
                 for (size_t j = 0;j < (*cur)->getCoords().size();++j)
                 {
                     parall->setCoord(j,parall->getCoords()[j]+(*cur)-
>getCoords()[j]);
                 }
             }
             if ((*cur)->shape() == "Ellipse")
             {
                 for (size_t j = 0;j < (*cur)->getCoords().size();++j)
                 {
                     ellipse->setCoord(j,ellipse->getCoords()[j]+(*cur)-
>getCoords()[j]);
                 }
             }
             cur++;
         }
         cur = right.begin();
         for (size_t i = 0; i < right.size();++i)
         {
             if ((*cur)->shape() == "Sector")
             {
                 for (size_t j = 0;j < (*cur)->getCoords().size();++j)
                 {
                    sector->setCoord(j,sector->getCoords()[j] + (*cur)-
>getCoords()[j]);
                 }
             }
             if ((*cur)->shape() == "Parall")
             {
                 for (size_t j = 0;j < (*cur)->getCoords().size();++j)
                 {
                     parall->setCoord(j,parall->getCoords()[j]+(*cur)-
>getCoords()[j]);
                 }
             }
             if ((*cur)->shape() == "Ellipse")
             {
```

```
                for (size_t j = 0;j < (*cur)->getCoords().size();++j)
                {
                    ellipse->setCoord(j,ellipse->getCoords()[j]+(*cur)-
>getCoords()[j]);
                }
             }
             cur++;
         }
         res.push_back(sector);
         res.push_back(parall);
         res.push_back(ellipse);
         return res;

     }


    size_t size() const
    {
        size_t i = 0;
        node <value_type> * el = m_head;
            while (el)
            {
                el = el->next;
                i++;
            }
        return i;
    }

  private:
    //your private functions
    node<Type>* m_head;
    node<Type>* m_tail;
  };


}// namespace stepik
#endif // LIST_H
```