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 Батурин И.

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

```
#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 Type[count];
        m_last = m_first + count;
        for (int i = 0; i < size(); i++)
            m_first[i] = 0;
      // use previous step implementation
    }

    template <typename InputIterator>
/*Создает контейнер с содержимым диапазона [first, last).*/
    vector(InputIterator first, InputIterator last) : vector(last-
first)
    {
        std::copy(first, last, m_first);
      // use previous step implementation
    }

    vector(std::initializer_list<Type> init) : vector(init.begin(),
init.end())
    {
      // use previous step implementation
```

```
}

vector(const vector& other)
{
    m_first = new Type[other.size()];
    m_last = &(m_first[other.size()]);
    std::copy(other.m_first, other.m_last, m_first);
  // use previous step implementation
}

vector(vector&& other)
{
    m_first = other.m_first;
    m_last = other.m_last;
    other.m_first = nullptr;
    other.m_last = nullptr;
  // use previous step implementation
}

~vector()
{
    delete []m_first;
  // use previous step implementation
}


static void swap(vector& that, vector& other)
{
    std::swap(that.m_first, other.m_first);
    std::swap(that.m_last, other.m_last);
}
//assignment operators
vector& operator=(const vector& other)
{
    if (this != &other) {
        vector tmp(other);
        swap(*this, tmp);
    }
    return *this;
  // implement this
}

vector& operator=(vector&& other)
{
    if (this != &other)
        swap(*this, other);
    return *this;
```

```
      // implement this
    }

    // assign method
    template <typename InputIterator>
    void assign(InputIterator first, InputIterator last)
    {
        vector tmp(first, last);
        swap(*this, tmp);
      // implement this
    }

    void resize(size_t count)
    {
        size_t new_size = count < (m_last-m_first) ? count : (m_last-
m_first);
        vector tmp(count);
        std::copy(m_first, m_first + new_size, tmp.m_first);
        swap(*this, tmp);
      // implement this
    }

    void push_back(const Type& value)
    {
        resize(size() + 1);
        *(m_last - 1) = value;
      // implement this
    }

    iterator insert(const_iterator pos, const Type& value)
    {
        vector temp_vector(pos - m_first);
        difference_type tmp = pos - m_first;
        std::copy(m_first, tmp + m_first, temp_vector.m_first);
        temp_vector.push_back(value);
        temp_vector.resize(size()+1);
        std::copy(m_first + tmp, m_last, temp_vector.m_first + 1);
        *this = std::move(temp_vector);
        return tmp + m_first;
      // implement this
    }

    template <typename InputIterator>
    iterator insert(const_iterator pos, InputIterator first,
InputIterator last)
    {
        difference_type tmp_pos = pos - m_first;
```

```
        difference_type tmp_range = last - first;
        vector temp_vector(size() + tmp_range);
        std::copy(m_first, tmp_pos + m_first, temp_vector.m_first);
        std::copy(first, last, temp_vector.m_first + tmp_pos);
        std::copy(m_first + tmp_pos, m_last, temp_vector.m_first +
tmp_pos + tmp_range);
        *this = std::move(temp_vector);
        return tmp_pos + m_first;
        // implement this
    }

    //push_back methods

    //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
```

ПРИЛОЖЕНИЕ Б
РЕАЛИЗАЦИЯ КЛАССА СПИСОК

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

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;
  // implement this
}

bool operator == (const list_iterator& other) const
{
    return m_node == other.m_node;
  // implement this
}

bool operator != (const list_iterator& other) const
{
    return m_node != other.m_node;
  // implement this
}

reference operator * ()
{
    return m_node->value;
  // implement this
}

pointer operator -> ()
{
    return &(m_node->value);
  // implement this
}

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

list_iterator operator ++ (int)
{
    list_iterator temp(*this);
    ++(*this);
    return *this;
  // implement this
}
```

```
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();
    // use previous step implementation
  }

   list(const list& other) : m_head(nullptr), m_tail(nullptr)
  {
      auto elem = other.m_head;
              while (elem) {
                    push_back(elem->value);
                    elem = elem->next;
              }
    // implement this
  }

  list(list&& other) : m_head(nullptr), m_tail(nullptr)
  {
      swap(other);
    // implement this
  }
```

```
void swap(list& other) {
    std::swap(m_head, other.m_head);
    std::swap(m_tail, other.m_tail);
}

list& operator= (const list& other)
{
    if(this != &other){
        clear();
        auto cur = other.m_head;
        while(cur){
            push_back(cur->value);
            cur = cur->next;
        }
    }
    return *this;
  // implement this
}

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

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

void pop_front()
{
    if (!empty()) {
        if (size() == 1) {
            node<Type >* elem = m_head;
            m_head = NULL;
            m_tail = NULL;
        }
        else {
            node<Type >* elem = m_head;
            elem = m_head->next;
            elem->prev = nullptr;
            delete m_head;
            m_head = elem;
        }
    }
    // implement this
```

```
}

void pop_back()
{
    if (!empty()) {
        if (size() == 1) {
            node<Type >* elem = m_head;
            m_head = NULL;
            m_tail = NULL;
        }
        else {
            node<Type >* elem = m_tail;
            elem = m_tail->prev;
            elem->next = nullptr;
            delete m_tail;
            m_tail = elem;
        }
    }
  // implement this
}

void push_back(const value_type& value)
{
    if (empty()) {
        node<Type>* elem = new node<Type>(value, nullptr, nullptr);
        m_head = elem;
        m_tail = elem;
    }
    else {
        node<Type>* elem = new node<Type>(value, nullptr, m_tail);
        m_tail->next = elem;
        m_tail = elem;
    }
  // use previous step implementation
}

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

```
        }
      // implement this
    }

    iterator insert(iterator pos, const Type& value)
    {
        if (pos.m_node == NULL)
        {
            push_back(value);
            return iterator(m_tail);
        }// implement this
        else if (pos.m_node->prev == NULL) {
            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);
        }
      // implement this
    }

    iterator erase(iterator pos)
    {
        if (pos.m_node == NULL)
           {
                return NULL;
           }
           else if (pos.m_node->prev == NULL)
           {
                pop_front();
                return iterator(m_head);
           }
           else if (pos.m_node->next == NULL)
           {
                pop_back();
                return iterator(m_tail);
           }
           else
           {
                pos.m_node->next->prev = pos.m_node->prev;
                pos.m_node->prev->next = pos.m_node->next;
                node<Type>* temp = pos.m_node;
                iterator new_pos(pos.m_node->next);
```

```
            delete temp;
            return new_pos;
        }
    }

    reference front()
    {
        return m_head->value;
      // use previous step implementation
    }

    reference back()
    {
        return m_tail->value;
      // use previous step implementation
    }

      void clear()
    {
        while(m_head) {
            m_tail = m_head->next;
            delete m_head;
            m_head = m_tail;
        }
      // implement this
    }

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

      size_t size() const
    {
        node<Type>* elem = m_head;
        size_t i = 0;
        while (elem != NULL) {
            i++;
            elem = elem->next;
        }
        return i;
      // implement this
    }

private:
  //your private functions
```

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

}// namespace stepik
```