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: «Вектор и список»

Студент гр. 7381 _____ Тарасенко Е.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

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

Постановка задачи.

Необходимо реализовать конструкторы и деструктор для контейнера вектор. Предполагается реализация упрощенной версии вектора, без резервирования памяти под будущие элементы.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции resize и erase для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции insert и push_back для контейнера вектор.

Поведение реализованных функций должно быть таким же, как у класса std::vector (http://ru.cppreference.com/w/cpp/container/vector). Семантику реализованных функций нужно оставить без изменений.

Необходимо реализовать список со следующими функциями:

1. Вставка элементов в голову и в хвост;
2. Получение элемента из головы и из хвоста;
3. Удаление из головы, хвоста и очистка;
4. Проверка размера.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции:

1. Деструктор;
2. Конструктор копирования;

3. Конструктор перемещения;

4. Оператор присваивания.

На данном шаге необходимо реализовать итератор для списка. Для краткости реализации можно ограничиться однонаправленным изменяемым (неконстантным) итератором. Необходимо реализовать операторы: =, ==, !=, ++ (постфиксный и префиксный), *, ->.

На данном шаге с использованием итераторов необходимо реализовать:

1. Вставку элементов (Вставляет value перед элементом, на который указывает pos. Возвращает итератор, указывающий на вставленный value),

2. Удаление элементов (Удаляет элемент в позиции pos. Возвращает итератор, следующий за последним удаленным элементом).

Поведение реализованных функций должно быть таким же, как у класса std::list (http://ru.cppreference.com/w/cpp/container/list). Семантику реализованных функций нужно оставить без изменений.

Решение должно содержать:

- условие задания;
- UML диаграмму разработанных классов;
- текстовое обоснование проектных решений;
- реализацию классов на языке C++.

Вариант 17: квадрат, эллипс, ромб.

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).

Ход работы.

В ходе данной лабораторной работы были реализованы классы для работы с векторами и списками.

Для класса “vector” было создано 5 различных конструкторов и деструктор; 2 оператора присваивания (для присваивания константного значения и неконстантного (другого вектора)); методы assign (заполнение указанной области вектора новыми значениями), resize (изменение размера вектора), erase (удаление указанного элемента или диапазона элементов из вектора), insert (вставка заданного элемента в указанную область вектора) и push_back (для вставки элемента в конец вектора).

Для класса “list” были реализованы методы вставки в голову списка и в его конец (push_front и push_back соответственно), получение значений головного и хвостового элементов списка (front и back), удаление элементов из головы и хвоста (pop_front и pop_back) и получения размера списка (size); конструкторы копирования и перемещения, деструктор и оператор присваивания. Также присутствуют операторы сравнений на равенство и на неравенство, увеличения на единицу (префиксальный и постфиксальный) и пр. Реализованы методы для вставки и удаления элементов.

Вывод.

В ходе написания данной лабораторной работы были реализованы классы вектор и список, аналогичные классам из стандартной библиотеки. Были получены необходимые знания об устройстве (реализации) и функционировании стандартных векторов и списков в языке C++.

ПРИЛОЖЕНИЕ А.

Исходный код для класса “vector”

```
#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)
    {
      // implement this
        m_first = new value_type[count]();
        m_last = m_first + count;
    }

    template <typename InputIterator>
    vector(InputIterator first, InputIterator last)
    {
      // implement this
        size_t count = std::distance(first, last);
        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())
    {
      // implement this
    }

    vector(const vector& other) : vector(other.begin(), other.end())
```

```
{
  // implement this
}

vector(vector&& other)
{
  // implement this
    m_first = other.m_first;
    m_last = other.m_last;
    other.m_first = nullptr;
    other.m_last = nullptr;
}

~vector()
{
  // implement this
    delete[] m_first;
}

  //assignment operators
vector& operator=(const vector& other)
{
  // implement this
    if(other.size() > 0){
        delete[] m_first;
        m_first = new value_type[other.size()];
        m_last = m_first + other.size();
        std::copy(other.begin(), other.end(), m_first);
    }
    return *this;
}

vector& operator=(vector&& other)
{
  // implement this
    if(this != &other) {
        delete[] m_first;
        m_first = other.m_first;
        m_last = other.m_last;
        other.m_first = nullptr;
        other.m_last = nullptr;
```

```
    }
}

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

  // resize methods
void resize(size_t count)
{
  // implement this
    if (count > 0) {
        if (count == size()) return;
        iterator tmp = new value_type[count]();
        if (count > size()) std::copy(m_first, m_last, tmp);
        if (count < size()) std::copy(m_first, m_first + count, tmp);
        delete[] m_first;
            m_first = tmp;
            m_last = m_first + count;
    }
    else {
        delete[] m_first;
        m_first = nullptr;
        m_last = nullptr;
    }
}

//erase methods
```

```
iterator erase(const_iterator pos)
{
  // implement this
    difference_type difference = pos - m_first;
    std::rotate(m_first + difference, m_first + difference + 1, m_last);
    resize(m_last - m_first - 1);
    return (m_first + difference);
}

iterator erase(const_iterator first, const_iterator last)
{
  // implement this
    difference_type difference = first - m_first;
    difference_type erase_zone = last - first;
    std::rotate(m_first + difference, m_first + difference + erase_zone, m_last);
    resize(m_last - m_first - erase_zone);
    return (m_first + difference);
}

   //insert methods
iterator insert(const_iterator pos, const Type& value)
{
  // implement this
    difference_type difference = pos - m_first;
    resize(size() + 1);
    m_first[size() - 1] = value;
    std::rotate(m_first + difference, m_last - 1, m_last);
    iterator new_el = m_first + difference;
    return new_el;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
  // implement this
    size_t difference = pos - m_first;
    resize(size() + (last - first));
    std::copy(first, last, m_last - (last - first));
    std::rotate(m_first + difference, m_last - (last - first), m_last);
    iterator new_el = m_first + difference;
    return new_el;
```

```
}

//push_back methods
void push_back(const value_type& value)
{
  // implement this
    resize(size() + 1);
    m_first[size() - 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
```

ПРИЛОЖЕНИЕ Б.

Исходный код для класса “list”

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

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
{
  // implement this
    if(m_node == other.m_node) return true;
    else return false;
}

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

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

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

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

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

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;

  list()
    : m_head(nullptr), m_tail(nullptr)
  {
  }

    ~list()
```

```
{
  // implement this
    clear();
}

list(const list& other) : m_head(nullptr), m_tail(nullptr)
{
  // implement this
    node<Type>* tmp = other.m_head;
    while(tmp){
        push_back(tmp->value);
        tmp = tmp->next;
    }
}

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

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

void push_back(const value_type& value)
{
  // implement this
    node<Type>* new_element = new node<Type>(value, nullptr, m_tail);
    if (!m_head) m_head = new_element;
```

```
    else m_tail->next = new_element;
    m_tail = new_element;
}

void push_front(const value_type& value)
{
  // implement this
    node<Type>* new_element = new node<Type>(value, m_head, nullptr);
    if (!m_tail) m_tail = new_element;
    else m_head->prev = new_element;
    m_head = new_element;
}

   iterator insert(iterator pos, const Type& value)
{
  // implement this
    node<Type>* pos_node = pos.m_node;
    if (!pos_node) {
        push_back(value);
        return iterator(m_tail);
    }
    if (pos_node == m_head) {
        push_front(value);
        return begin();
    }
    node<Type>* new_node = new node<Type>(value, pos_node, pos_node->prev);
    pos_node->prev->next = new_node;
    pos_node->prev = new_node;
    return iterator(new_node);
}

iterator erase(iterator pos)
{
  // implement this
    node<Type>* pos_node = pos.m_node;
    if (!pos_node)
        return pos;
    if (!(pos_node->prev)) {
        pop_front();
        return begin();
    }
```

```
    if (!(pos_node->next)) {
        pop_back();
        return end();
    }
    pos_node->prev->next = pos_node->next;
    pos_node->next->prev = pos_node->prev;
    iterator next_pos = iterator(pos_node->next);
    delete pos_node;
    return next_pos;
}


reference front()
{
  // implement this
    return m_head->value;
}

const_reference front() const
{
  // implement this
    return m_head->value;
}

reference back()
{
  // implement this
    return m_tail->value;
}

const_reference back() const
{
  // implement this
    return m_tail->value;
}

void pop_front()
{
  // implement this
    if (m_head){
        node<Type>* new_head = m_head->next;
```

```
        if (m_head->next) new_head->prev = nullptr;
        else m_tail = new_head;
        delete m_head;
        m_head = new_head;
    }
    else return;
}

void pop_back()
{
  // implement this
    if(m_tail){
        node<Type>* new_tail = m_tail->prev;
        if(m_tail->prev) m_tail->prev->next = nullptr;
        else m_head = new_tail;
        delete m_tail;
        m_tail = new_tail;
    }
    else return;
}

void clear()
{
  // implement this
    m_tail = nullptr;
    delete[] m_head;
}

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

size_t size() const
{
  // implement this
    int s = 0;
    node<Type>* tmp = m_head;
    while(tmp){
```

```
            s++;
            tmp = tmp->next;
        }
        return s;
    }

  private:
    //your private functions

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

}// namespace stepik
```