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 Шаталов Э.В.

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)
    {
      // implement this
        m_first = new Type[count];
        m_last = m_first + count;
    }

    template <typename InputIterator>
    vector(InputIterator first, InputIterator last)
    {
      // implement this
        size_t dis = std::distance(first,last);
        m_first = new Type[dis];
        std::copy(first, last, m_first);
        m_last = m_first + dis;

    }

    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;
    }

    ~vector()
    {
       delete[] m_first;
    }

    //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];
    }

 vector& operator=(const vector& other)
    {
      // implement this
         if (this != &other){
            if (other.size()) {
                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){
        std::swap(m_first,other.m_first);
        std::swap(m_last,other.m_last);}
        return *this;
    }

    // assign method
    template <typename InputIterator>
    void assign(InputIterator first, InputIterator last)
    {
      // implement this
       *this = vector(first,last);
    }
   void resize(size_t count)
    {
        // implement this
        vector new_vector(count);
           if (count > size())
                std::move(m_first, m_last, new_vector.m_first);
           else
                std::move(m_first, m_first + count,
new_vector.m_first);
           std::swap(m_first,new_vector.m_first);
        std::swap(m_last,new_vector.m_last);
    }

    //erase methods
  iterator erase(const_iterator pos)
          {
                iterator iter = m_first;
                difference_type size = m_last - m_first;
                size_t i = 0;

                while (iter != pos) {
                     iter++;
                     i++;
                }
                std::rotate(iter, iter + 1, m_last);

                resize(size - 1);
                return  m_first + i;
          }
```

```
        iterator erase(const_iterator first, const_iterator last)
        {
              difference_type  n = last - first;
              iterator iter = m_first;

              if (n == 0)
                    return iter;

              for (; iter != first; iter++);
              for (size_t i = 0; i < n; i++) {
                    iter = erase(iter);
              }
              return iter;
        }   iterator insert(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;
      }

      template <typename InputIterator>
      iterator insert(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;
       }


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

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.other;
      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);
  }

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

  list_iterator& operator ++ ()
  {
    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()
  {
     clear();
  }

  list(const list& other):m_head(nullptr), m_tail(nullptr)
  {
      node<Type>* new_node = other.m_head;
      for (;new_node;new_node = new_node->next)
              push_back(new_node->value);

  }

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

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

```
            return *this;
}
void push_back(const value_type& value)
{
    node<Type>* new_node = new node<Type>(value, nullptr, m_tail);
       if (!m_head) m_head = new_node;
       else m_tail->next = new_node;
       m_tail = new_node;
}

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

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) return;
      node<Type>* new_head = m_head->next;
         delete m_head;
       if (new_head)
           new_head->prev = nullptr;
         else m_tail = new_head;
         m_head = new_head;
```

```
    }

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

    void clear()
    {
      while(!empty())
          pop_back();
    }

    bool empty() const
    {
      return !m_head;
    }

 iterator insert(iterator pos, const Type& value)
    {
        node<Type>* pos_node = pos.m_node;
           if (pos_node == nullptr) {
                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)
    {
          node<Type>* pos_node = pos.m_node;
           if (pos_node == nullptr)
                     return pos;
```

```
            if (pos_node->prev == nullptr) {
                  pop_front();
                  return begin();
               }
               if (pos_node->next == nullptr) {
                  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;
    }


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

  private:
    //your private functions

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

}// namespace stepik
```