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 Бондарчук Н.Р.

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>
#include <cstddef>
#include <initializer_list>
#include <stdexcept>
using namespace std;
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;
   // implement this
  }

  template <typename InputIterator>
  vector(InputIterator first, InputIterator last): vector(last-first)
  {
   copy(first, last, m_first);
  }

  vector(std::initializer_list<Type> init) : vector(init.begin(), init.end())
  {
   // implement this
  }

  vector(const vector& other)
  {
     m_first = new Type[other.size()];
     m_last = &(m_first[other.size()]);
     copy(other.m_first, other.m_last, m_first);
   // implement this
```

```
}

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

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

//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;
}
//assignment operators
    vector& operator=(const vector& other)
    {  if(this != &other)
    {
       vector a(other);
       swap(this->m_first, a.m_first);
       swap(this->m_last, a.m_last);
    }
        return *this;
      // implement this
    }

    vector& operator=(vector&& other)
    {    if(this != &other)
    {
        swap(this->m_first, other.m_first);
       swap(this->m_last, other.m_last);
    }
        return *this;

      // implement this
    }

    // assign method
    template <typename InputIterator>
    void assign(InputIterator first, InputIterator last)
    {
       vector a(first, last);
       swap(this->m_first, a.m_first);
       swap(this->m_last, a.m_last);
```

```
  // implement this
}
void resize(size_t count)
    {
        vector a(count);
        if(count < (m_last-m_first))
        {
            copy(m_first, m_first + count, a.m_first);
            swap(this->m_first, a.m_first);
            swap(this->m_last, a.m_last);
        }
        else
        {
            copy(m_first, m_first + (m_last-m_first), a.m_first);
            swap(this->m_first, a.m_first);
            swap(this->m_last, a.m_last);
        }
      // implement this
    }

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

            while (a != pos) {
                a++;
                i++;
            }
            rotate(a, a + 1, m_last);

            resize(size - 1);
            return  m_first + i;
      // implement this
    }

    iterator erase(const_iterator first, const_iterator last)
    {
            difference_type  size = last - first;
            iterator a = m_first;

            if (size == 0)
                return a;

           while(a!= first)
              a++;
            for (size_t i = 0; i < size; i++) {
                a = erase(a);
            }
```

```
            return a;
          // implement this
        }
        iterator insert(const_iterator pos, const Type& value)
          {
            difference_type size = pos - m_first;
            resize(this->size()+1);
            *(m_last-1) = value;
            rotate(m_first+size, m_last-1, m_last);
            return m_first + size;

           // implement this
          }


          template <typename InputIterator>
          iterator insert(const_iterator pos, InputIterator first, InputIterator last)
          {
             difference_type size = pos - m_first;
             resize(this->size() + (last-first));
             copy(first, last, m_last - (last-first));
             rotate(m_first+size, m_last - (last-first) , m_last);
                return m_first + size;
            // implement this
          }

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


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;
```

```
  };

}

#endif // VECTOR_H
```

ПРИЛОЖЕНИЕ Б
РЕАЛИЗАЦИЯ КЛАССА СПИСОК

```
#ifndef LIST_H
#define LIST_H
#include <assert.h>
#include <algorithm>
#include <stdexcept>
#include <cstddef>

using namespace std;
{
  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 m_node;
  // 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* next_node = new list_iterator(*this);
    m_node = m_node->next;
    return *next_node;
    // 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;

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

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

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

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

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

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

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

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

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

```
  // implement this
}

void clear()
{
    while(!empty())
        pop_front();
  // implement this
}

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

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

   list(const list& other): list()
   {
      node<Type>* new_node = other.m_head;
      while(new_node != NULL){
         push_back(new_node->value);
         new_node = new_node->next;
      }
     // implement this
   }

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

   list& operator= (const list& other)
```

```
{
    if (this != &other) {
        clear();
       node<Type>* new_node = other.m_head;
       while(new_node != NULL){
       push_back(new_node->value);
       new_node = new_node->next;
       }
 }
   return *this;
 // implement this
}

iterator insert(iterator pos, const Type& value)
  {
     if (pos.m_node == NULL)
     {
        push_back(value);
        return iterator(m_tail);
     }
     else if (pos.m_node->prev == NULL) {
        push_front(value);
        return iterator(m_head);
     }
     else {
        node<Type>* a = new node<Type>(value, pos.m_node, pos.m_node->prev);
        pos.m_node->prev->next = a;
        pos.m_node->prev = a;
        return iterator(a);
     }
   // 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>* a = pos.m_node;
            iterator pos_next(pos.m_node->next);
            delete a;
            return pos_next;
          }
        // implement this
      }


  private:
    //your private functions

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

}
#endif // LIST_H
```