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: Контейнеры.

Студент гр.7304 _____ Сергеев И.Д.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

1. Постановка задачи

1.1. Цель работы

Исследование реализации контейнеров. Реализация контейнера vector и list на языке c++.

1.2. Формулировка задачи

Необходимо реализовать следующие конструкции для вектора: конструктор по умолчанию, копирования, перемещения, деструктор, оператор присваивания и перемещения, функцию assign, resize, insert, erase, push_back, size, empty. Поведение функций должно быть идентично поведению этих конструкций в классе std::vector.

Далее необходимо реализовать класс list и iterator со следующими конструкциями: конструктор по умолчанию, копирования, перемещения, деструктор, оператор присваивания, функции вставки элементов в конец, в начало и на позицию, функции удаления элементов из начала, конца и позиции, size, а также операторы для итератора списка ==, !=, ++(постфиксный и префиксный), *. ->. Поведение должно соответствовать классу std::list.

2. Ход работы

2.1. Был реализован класс вектор

2.1.1. Конструктор копирования, перемещения, деструктор.

2.1.2. Операторы присваивания и перемещения, функция assign.

2.1.3. Функции resize, size, insert, erase, push_back.

2.2. Был реализован класс список

2.2.1. Функции pop_back и pop_front, push_back и push_front, clear, size.

2.2.2. Конструктор копирования, перемещения, деструктор, оператор присваивания.

2.2.3. Операторы для итератора списка: ==, !=, ++, *, ->.

2.2.4. Функции удаления и вставки элемента на любую позицию.

3. Экспериментальные результаты

3.1. Вектор

```
PUSHING 1 AND 4
VECTOR SIZE 2
VECTOR ELEMENTS 1   4
```

3.2. Список

```
VECTOR SIZE AFTER ERASE 0
PUSHING 2 AND 4
INSERTING 5 at pos 0
FRONT AND BACK ELEMENTS 5   4
SIZE OF LIST 3
```

4. Вывод

В результате работы были изучены способы реализации контейнера vector и list, были реализованы функции для работы с этими контейнерами, такие как вставка и удаления элементов, получения размера, проверка на пустоту, конструкторы, операторы для итератора, деструкторы, функция очищения контейнера.

Приложение А:

Исходный код

Файл lr3_1.cpp

```
#include <assert.h>
#include <algorithm> // std::copy, std::rotate
#include <cstddef> // size_t
#include <initializer_list>
#include <stdexcept>
#include <iostream>
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) : n(count), arr(n ? new value_type[n]() : nullptr)
  {
     m_first = arr;
     m_last = arr + count;
  }

  template <typename InputIterator>
  vector(InputIterator first, InputIterator last) : n(last-first), arr(n ? new value_type[n]() : nullptr)
  {
```

```
 if (arr != nullptr){
    for (InputIterator it = first;it!=last;++it){
       arr[it-first] = *it;
    }
 }
 m_first = arr;
 m_last = n + arr;
}

vector(std::initializer_list<Type> init) : n(init.size()), arr(n ? new value_type[n]() : nullptr)
{
 if (arr != nullptr){
    int ind = 0;
    for (auto &elem : init){
       arr[ind] = elem;
       ind++;
    }
 }
 m_first = arr;
 m_last = n + arr;
}

vector(const vector& other)
{
 vector tmp(other.n);
 for (size_t i = 0; i < tmp.n;i++){
    tmp.arr[i] = other.arr[i];
 }
 n = tmp.n;
 arr = n ? new value_type[n]() : nullptr;
 for (size_t i = 0;i < n;i++){
    arr[i] = tmp.arr[i];
 }
 m_first = arr;
 m_last = n + arr;
}

vector(vector&& other)
{
 arr = other.arr;
 n = other.n;
 other.arr = nullptr;
 other.m_first = nullptr;
 other.m_last = nullptr;
 m_first = arr;
 m_last = n + arr;
}

~vector()
{
   if (arr)
      delete[] arr;
}

vector& operator=(const vector& other)
{
 if (this != &other)
    vector(other).swap(*this);
 m_first = arr;
 m_last = n + arr;
 return *this;
}
```

```
vector& operator=(vector&& other)
{
  // Проверка на самоприсваивание
if (&other == this)
  return *this;
  // Удаляем всё, что может хранить указатель до этого момента
delete[] arr;
  m_first = nullptr;
  m_last = nullptr;
  n = other.n;
arr = other.arr;
  m_first = arr;
  m_last = n + arr;
  other.m_first = nullptr;
  other.m_last = nullptr;
other.arr = nullptr;
  return *this;
}

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
  delete[] arr;
  m_first = nullptr;
  m_last = nullptr;
  n = last-first;
  arr = n ? new value_type[n]() : nullptr;
  if (arr != nullptr){
     for (InputIterator it = first;it!=last;++it){
           arr[it-first] = *it;
     }
  }
  m_first = arr;
  m_last = arr + n;
}

 void resize(size_t count)
{
  vector tmp(n);
  for (size_t i = 0; i < tmp.n;i++){
     tmp.arr[i] = arr[i];
  }
  delete [] arr;
  arr = count ? new value_type[count]() : nullptr;
  if (count <= n){
     for (size_t i=0;i<count;i++)
        arr[i] = tmp.arr[i];
  }
  else if (count > n){
     for (size_t i=0;i<n;i++)
        arr[i] = tmp.arr[i];
  }
  n = count;
  m_first = arr;
  m_last = arr + n;
}

//erase methods
iterator erase(const_iterator pos)
{
  int dif = pos - m_first;
  n--;
```

```
  vector tmp(n);
  int i=0;
  for (const_iterator it=begin();it!=end();++it){
     if (it == pos)
        continue;
     tmp.arr[i] = *it;
     i++;
  }
  delete [] arr;
  arr = n ? new value_type[n]() : nullptr;
  for (size_t i = 0; i < n;i++){
     arr[i] = tmp.arr[i];
  }
  m_first = arr;
  m_last = arr + n;
  return arr + dif;
}

iterator erase(const_iterator first, const_iterator last)
{
  int dif = first-m_first;
  vector tmp(n);
  size_t j = 0;
  for (const_iterator it = begin();it!=end();++it){
     if (it >= first && it < last){
        continue;
     }
     tmp.arr[j] = *it;
     j++;
  }
  delete [] arr;
  n -= (last-first);
  arr = n ? new value_type[n]() : nullptr;
  for (size_t i = 0; i < n;i++){
     arr[i] = tmp.arr[i];
  }
  m_first = arr;
  m_last = arr + n;
  return arr + dif;
}

iterator insert(const_iterator pos, const Type& value)
{
  int dif = pos - m_first,i = 0;
  n++;
  vector tmp(n);
  for (const_iterator it=begin();it!=end();++it){
     if (it == pos){
        tmp.arr[i] = value;
        i++;
     }
     tmp.arr[i] = *it;
     i++;
  }
  if (pos == end()){
     tmp.arr[i] = value;
  }
  delete [] arr;
  arr = n ? new value_type[n]() : nullptr;
  for (int j=0;j<n;j++){
     arr[j] = tmp.arr[j];
  }
  m_first = arr;
```

```
  m_last = arr + n;
  return arr + dif;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
  int dif = pos - m_first,count = 0,i = 0;
  count = last-first;
  vector tmp(n+count);
  for (const_iterator it=begin();it!=end();++it){
     if (it == pos){
        for (InputIterator temp=first;temp!=last;++temp){
           tmp.arr[i] = *temp;
           i++;
        }
     }
     tmp.arr[i] = *it;
     i++;
  }
  if (pos == end()){
     for (InputIterator temp2=first;temp2!=last;++temp2){
           tmp.arr[i] = *temp2;
           i++;
     }
  }
  n += count;
  delete [] arr;
  arr = n ? new value_type[n]() : nullptr;
  for (int j=0;j<n;j++){
     arr[j] = tmp.arr[j];
  }
  m_first = arr;
  m_last = arr + n;
  return arr + dif;

}

//push_back methods
void push_back(const value_type& value)
{
  n++;
  vector tmp(n);
  for (int i=0;i<n-1;i++){
     tmp.arr[i] = arr[i];
  }
  tmp.arr[n-1] = value;
  delete [] arr;
  arr = n ? new value_type[n]() : nullptr;
  for (int j=0;j<n;j++){
     arr[j] = tmp.arr[j];
  }
  m_first = arr;
  m_last = arr + n;
}

//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
  void swap(vector& v){
    size_t const t1 = n;
```

```
    n = v.n;
    v.n = t1;
    Type * const t2 = arr;
    arr = v.arr;
    v.arr = t2;
  }
  private:
   size_t n;
   value_type* arr;
   iterator m_first;
   iterator m_last;
  };
}// namespace stepik

int main(){
  stepik::vector<int> vec(2);
  vec.push_back(1);
  vec.push_back(4);
  cout << vec.at(0) << "   " << vec.at(1) << endl;
  vec.erase(vec.begin(),vec.end());
  cout << vec.size() << endl;
  return 0;
}
```

Файл lr3_2.cpp

```
#include <assert.h>
#include <algorithm>
#include <stdexcept>
#include <cstddef>
#include <utility>
#include <iostream>
using namespace std;
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 (this != &other){
       m_node = other.m_node;
    }
    return *this;
 }

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

 bool operator != (const list_iterator& other) const
 {
  if (!(*this == other))
     return true;
  else return false;
 }

 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)
 {
  m_node = m_node->next;
  return *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), m_size(0)
  {
  }

  ~list()
  {
    clear();
  }

  list(const list& other)
  {
    if (other.m_head == nullptr){
        m_head = nullptr;
        m_tail = nullptr;
        m_size = 0;
    }
    else{
        m_size++;
        m_head = new node<Type>(other.m_head->value,nullptr,nullptr);
        m_tail = m_head;
        if (other.m_head->next == nullptr){
            return;
        }
        else{
            node<Type>* tmp = other.m_head->next;
            while (tmp != nullptr){
                m_size++;
                m_tail->next = new node<Type>(tmp->value,nullptr,m_tail);
                m_tail = m_tail->next;
                tmp = tmp->next;
            }
        }
    }
  }

  list(list&& other)
  {
    m_head = other.m_head;
    m_tail = other.m_tail;
    m_size = other.m_size;
    other.m_head = nullptr;
    other.m_tail = nullptr;
    other.m_size = 0;
  }

  list& operator= (const list& other)
  {
    clear();
    if (other.m_head == nullptr){
        m_head = nullptr;
        m_tail = nullptr;
```

```
    m_size = 0;
  }
  else{
    m_size++;
    m_head = new node<Type>(other.m_head->value,nullptr,nullptr);
    m_tail = m_head;
    node<Type>* tmp = other.m_head->next;
    if (other.m_head->next == nullptr){
      return *this;
    }
    else{
      node<Type>* tmp = other.m_head->next;
      while (tmp != nullptr){
        m_size++;
        m_tail->next = new node<Type>(tmp->value,nullptr,m_tail);
        m_tail = m_tail->next;
        tmp = tmp->next;
      }
    }
  }
  return *this;
}

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

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

void push_back(const value_type& value)
{
  m_size++;
  node<Type>* ptr = new node<Type>(value,nullptr,m_tail);
  if (m_head == nullptr){
    m_head = ptr;
  }
  m_tail = ptr;
  if (m_tail->prev != nullptr){
    m_tail->prev->next = ptr;
  }
}

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

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

void clear()
{
  node<Type>* ptr;
  while (m_head != nullptr){
    ptr = m_head;
    m_head = m_head->next;
    if (m_head == nullptr){
```

```
            m_tail = nullptr;
        }
        else m_head->prev = nullptr;
        delete ptr;
    }
    m_size = 0;
}

bool empty() const
{
    return m_head == nullptr;
}

size_t size() const
{
    return m_size;
}

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

iterator erase(iterator pos)
{
    if(pos.m_node == m_head)
    {
        pop_front();
        pos.m_node = m_head;
        return pos;
    }
    else if(pos.m_node == nullptr)
        {
            return pos;
        }
        else if(pos.m_node->next == nullptr)
            {
            pop_back();
            pos.m_node = m_tail;
            return pos;
            }
            else
            {
                pos.m_node->prev->next = pos.m_node->next;
```

```
                pos.m_node->next->prev = pos.m_node->prev;
                delete pos.m_node;
                pos.m_node = pos.m_node->next;
                return pos;
            }
    }

    void pop_front()
    {
      if (m_size == 0)
          return;
      m_size--;
      node<Type>* ptr = m_head;
      m_head = m_head->next;
      if (m_head == nullptr)
          m_tail = nullptr;
      else m_head->prev = nullptr;
      delete ptr;
    }

    void pop_back()
    {
      if (m_size == 0)
          return;
      m_size--;
      node<Type>* ptr = m_tail;
      m_tail = m_tail->prev;
      if (m_tail == nullptr)
          m_head = nullptr;
      else m_tail->next = nullptr;
      delete ptr;
    }

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

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

}// namespace stepik

int main(){
    stepik::list<int> my = stepik::list<int>();
    my.push_back(2);
    my.push_back(4);
    cout << "PUSHING 2 AND 4\n";
    cout << "INSERTING 5 at pos 0\n";
    my.insert(my.begin(),5);
    cout << "FRONT AND BACK ELEMENTS " << my.front() <<  "   " << my.back() << endl;
```

```
    cout << "SIZE OF LIST " << my.size() << endl;
}
```