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.

Студентка гр. 7304 Юруть Е.А.

Преподаватель Размочаева Н.В

Санкт-Петербург

2019

Цель работы.

Изучить реализация контейнеров list и vector в языке программирования C++. Протестировать полученные реализации на практике.

Задача.

Реализовать конструктор, деструктор , операторы присваивания, функцию assign, функцию resize, функцию earse, функцию insert и функцию push_back. Поведение реализованных функций должно быть таки же как и у std::vector. Реализовать список с функциями: вставка элемента в голову, вставка элемента в хвост, получение элемента из головы, получение элемента из хвоста, удаление из головы, из хвоста, очистка списка, проверка размера, деструктор, конструктор копирования, конструктор перемещения, оператор присваивания, insert, earse, а так же итераторы для списка: =, ==, !=, ++(постфиксный и префексный), *, ->. Поведение реализованных функций должно быть таким же, как у класса std::list.

Ход работы.

- List.

В ходе реализации list были созданы следующие функции:

✓ Функции вставки элемента в голову и в хвост.

Принимает на вход элемент и помещает его в вектор.

✓ Функции получение элемента из головы и из хвоста.

Возвращает элемент из головы или из хвоста.

✓ Функции удаления из головы, удаления из хвоста.

Совершает удаление элемента из начала или конца списка.

✓ Функции очистки списка, проверки размера.

✓ Деструктор, конструктор копирования, конструктор перемещения, оператор присваивания.

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

✓ Функции удаления элемента и вставка элемента в произвольное место.

Получает на вход элемент и помещает его в заданное место в массиве. Так же имеет возможно удалить элемент из заданного положения.

Поведение функций такое же, как у класса std::list.

- Vector.

В ходе реализации vector были созданы следующие функции:

✓ Конструкторы и деструктор для вектора.

Реализованные конструкторы включают в себя – конструктор копирования, присваивания и перемещения.

✓ Оператор присваивания и функция assign.

✓ Функции изменения размера и стирания элементов в массиве (resize, erase).

Resize – принимает на вход необходимый размер вектора, который будет присвоен текущему.

Erase – может принимать как одну переменную – индекс, начиная с которого произойдет очистка вектора, так из пару переменных – интервал в векторе, которой очистится.

Поведение функций такое же, как у класса std::vector.

Вывод:

Таким образом, в ходе лабораторной работы была подробно изучена реализация контейнеров list и vector. Поведение реализованных функций каждого из классов совпадает с реальным поведением функций из стандартной библиотеки C++. Полученные результаты были протестированы на практике.

Приложение 1. Исходный код.
Реализация 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
    if (count > 0)
        {
            m_first = new Type[count];
            m_last = m_first + count;
            for (size_t i = 0; i < count; i++)
                m_first[i] = 0;
        }
        else
        {
            m_first = nullptr;
            m_last = nullptr;
        }
    }

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

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

    vector(const vector& other)
        :  m_first((other.size()) ? new Type[other.size()] : nullptr)
        ,  m_last((other.size()) ? m_first + other.size() : nullptr)
    {
        // implement this
    try
    {
        std::copy(other.m_first, other.m_last, m_first);
```

```
    }
    catch(...)
    {
        delete[] m_first;
        throw;
    }
    }

    void swap(vector & other)
    {
    std::swap(this->m_first, other.m_first);
    std::swap(this->m_last, other.m_last);
    }

    vector(vector&& other)
        :  vector()
    {
        // implement this
        if (this != &other)
        swap(other);
    }

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

    //assignment operators
    vector& operator=(const vector& other)
    {
        // implement this
        if (this != &other)
        vector(other).swap(*this);
        return *this;
    }

    vector& operator=(vector&& other)
    {
        // implement this
        if (this != &other)
        swap(other);
        return *this;
    }

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

    // resize methods
    void resize(size_t count)
    {
        // implement this
        vector buf(count);
        std::move(m first, (count > size()) ? m last : m first + count,
buf.m_first);
        swap(buf);
    }
```

```
    //erase methods
    iterator erase(const_iterator pos)
    {
        // implement this
        size_t new_pos = pos - m_first;
         std::rotate(const_cast<iterator>(pos), const_cast<iterator>(pos) + 1,
m_last);
        resize(size()-1);
        return m_first + new_pos;
    }

    iterator erase(const_iterator first, const_iterator last)
    {
        // implement this
        size_t new_pos = last - first;
        iterator _first = const_cast<iterator>(first);
        while(new_pos--)
            _first = erase(_first);
        return _first;
    }

    //insert methods
    iterator insert(const_iterator pos, const Type& value)
    {
        // implement this
        size_t buf = pos - m_first;
        resize(size() + 1);
        iterator new_pos = const_cast<iterator>(m_first + buf);
        std::rotate(new_pos, m_last - 1, m_last);
        *new_pos = value;
        return new_pos;
    }

    template <typename InputIterator>
    iterator insert(const_iterator pos, InputIterator first, InputIterator
last)
    {
        // implement this
        size_t distance = last - first;
        iterator new_pos = const_cast<iterator>(pos);
        while(distance)
            new_pos = insert(new_pos, *(first+(--distance)));
        return new_pos;
    }

    //push_back methods
    void push_back(const value_type& value)
    {
        // implement this
        insert(end(), 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
```

```
#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)
    {
        // implement this
        m_node = other.m_node;
        return *this;
    }

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

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

```
  }

  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 buf(*this);
      ++(*this);
      return buf;
  }

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

  list(const list& other)
      :  m_head(nullptr)
      ,  m_tail(nullptr)
```

```
{
    // implement this
    try
    {
        copy(const_cast<node<Type>*>(other.m_head));
    }
    catch(...)
    {
        delete[] m_head;
        throw;
    }
}

list(list&& other)
    :  list()
{
    // implement this
    if (this != &other)
        swap(other);
}

list& operator= (const list& other)
{
    // implement this
    if(this != &other)
        list(other).swap(*this);
    return *this;
}

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

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

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

void push_front(const value_type& value)
{
    // implement this
    if (empty())
      {
        node<Type> *buf = new node<Type>(value, nullptr, nullptr);
          m_head = buf;
```

```
            m_tail = buf;
        }
    else
    {
        node<Type> *buf = new node<Type>(value, m_head, nullptr);
        m_head->prev = buf;
        m_head = buf;
    }
}

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 (!empty())
    {
        if (size() == 1)
        {
            node<Type>* buf = m_head;
            m_head = NULL;
            m_tail = NULL;
        }
        else
        {
            node<Type>* buf = m_head->next;
            buf->prev = nullptr;
            delete m_head;
            m_head = buf;
        }
    }
}

void pop_back()
{
    // implement this
    if (!empty())
    {
        if (size() == 1)
        {
            node<Type>* buf = m_head;
```

```
                m_head = NULL;
                m_tail = NULL;
            }
            else
            {
                node<Type>* buf = m_tail->prev;
                buf->next = nullptr;
                delete m_tail;
                m_tail = buf;
            }
        }
    }

    iterator insert(iterator pos, const Type& value)
    {
        // implement this
        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>* buf = new node<Type>(value, pos.m_node, pos.m_node-
>prev);
            pos.m_node->prev->next = buf;
            pos.m_node->prev = buf;
            return iterator(buf);
        }
    }

    iterator erase(iterator pos)
    {
        // implement this
        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>* buf = pos.m_node;
            iterator new pos(pos.m node->next);
            delete buf;
            return new_pos;
        }
    }
```

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

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

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

    void copy(node<Type> * other)
    {
    while(other)
    {
        push_back(other->value);
        other = other->next;
    }
    }

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

  private:
    //your private functions

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

}// namespace stepik
```