**МИНОБРНАУКИ РОССИИ**
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**
**Кафедра математического обеспечения и применения ЭВМ**

# ОТЧЕТ

**по практической работе №3**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема**: **Контейнеры. Вектор. Список**

Студент гр. 7382 _____ Токарев А.П.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

**Цель работы.**

Изучить стандартные контейнеры vector и list языка C++.

**Постановка задачи.**

Необходимо реализовать конструкторы, деструктор, оператор присваивания, функции assign, resize, erase, insert и push_back для контейнера вектор (в данном уроке предполагается реализация упрощенной версии, без резервирования памяти под будущие элементы).

Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать список со следующими функциями: вставка элементов в голову и в хвост, получение элемента из головы и из хвоста, удаление из головы и из хвоста, очистка, проверка размера, деструктор, конструктор копирования, конструктор перемещения, оператор присваивания.

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

С использованием итераторов 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).

Поведение реализованных функций должно быть таким же, как у класса std::list. Семантику реализованных функций нужно оставить без изменений.

При выполнении этого задания можно 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.**

Был реализован класс vector; поведение реализованных функций аналогично поведению функций класса std::vector.

Класс vector содержит два поля: указатели на начало и конец массива данных в памяти. Были реализованы деструктор и следующие конструкторы: конструктор от размера массива, от двух итераторов, от списка инициализации, копирования и перемещения. Также были реализованы методы изменения размера, удаления одного элемента или интервала элементов, вставки одного элемента или нескольких элементов, заданных при помощи двух итераторов, на заданное итератором место и вставки одного элемента в конец вектора.

Реализация класса представлена в приложении А.

Класс list имеет аналогичные поля, как и у класса vector, но данные содержатся не в массиве, а в двусвязном списке. Для класса list были реализованы деструктор и следующие конструкторы: стандартный, копирования и перемещения. Также был реализованы оператор присваивания и методы для вставки, получения и удаления элементов из головы и из хвоста, очистки списка и проверки размера. Поведение реализованных функций аналогично поведению функций класса std::list.

Итератор для списка содержит одно поле – указатель на элемент контейнера list. Для итератора был перегружен ряд операторов: =, ==, !=, ++ (постфиксный и префиксный), * и ->. Класс list объявлен в данном классе, как дружественный, так как используется в функциях для вставки и удаления элементов из списка.

Реализация класса представлена в приложении В.

**Выводы.**

В ходе выполнения лабораторной работы была изучена реализация контейнеров vector и list.

## ПРИЛОЖЕНИЕ А

Исходный код list.h

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

namespace stepik
{
    template <class Type, template <class Type1>>

    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;

    template <class Value>
    class list_iterator
    {
    public:
        friend class list<Value>;

        typedef ptrdiff_t difference_type;
        typedef Value value_type;
        typedef Value* pointer;
        typedef Value& 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(node<Value>* p) : m_node(p)
        {}

        list_iterator& operator = (const list_iterator& other)
        {
            if (*this != other)
                m_node = other.m_node;

            return *this;
        }
```

```
    bool operator ==(const list_iterator& other) const
    {
        return m_node == other.m_node;
    }

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

    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)
    {
        auto cur = m_node;
        m_node = m_node->next;

        return cur;
    }

private:
    node<Value>* m_node;
};

template <class Type>
class list
{
public:
    typedef list_iterator<Type> iterator;
    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)
    {
        if (!other.empty())
        {
```

```
                for (auto item = other.m_head; item != other.m_tail; item =
item->next)
                {
                    push_back(item->value);
                }
                push_back(other.m_tail->value);
            }
        }

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

        iterator insert(iterator pos, const Type& value)
        {
            auto& pos_node = pos.m_node;
            if (pos_node == nullptr)
            {
                push_back(value);
                return iterator(m_tail);
            }

            if (pos_node->prev == nullptr)
            {
                push_front(value);
                return iterator(m_head);
            }
            auto 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)
        {
            auto& pos_node = pos.m_node;

            if (pos_node == nullptr)
                return pos;

            if (pos_node->next == nullptr)
            {
                pop_back();
                return nullptr;
            }

            if (pos_node->prev == nullptr)
            {
                pop_front();
                return iterator(m_head);
            }

            auto res = pos_node->next;
            pos_node->prev->next = pos_node->next;
            pos_node->next->prev = pos_node->prev;
            delete pos_node;
            return res;
        }
```

```
void push_back(const value_type& value)
{
    if (!empty())
    {
        auto prev_tail = m_tail;
        m_tail = new node<value_type>(value, nullptr, nullptr);

        m_tail->prev = prev_tail;
        prev_tail->next = m_tail;
        m_tail->next = nullptr;
    }
    else
    {
        init_first(value);
    }
}

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

void pop_front()
{
    if (is_single())
    {
        delete_last();
    }
    else
    {
        auto del_head = m_head;
        m_head = m_head->next;
        m_head->prev = nullptr;

        delete del_head;
    }
}

void pop_back()
{
    if (is_single())
    {
        delete_last();
    }
    else
    {
        auto del_tail = m_tail;
        m_tail = m_tail->prev;
        m_tail->next = nullptr;

        delete del_tail;
    }
}
```

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

        m_head = m_tail = nullptr;
    }

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

    size_t size() const
    {
        if (empty())
            return 0;

        size_t size = 0;
        for (auto it = m_head; it != m_tail; it = it->next)
        {
            size++;
        }
        size++;

        return size;
    }

    iterator begin() const
    {
        return iterator(m_head);
    }

    iterator end() const
    {
        return iterator();
    }
    void print()
     {
         std::cout << "List: ";
         for(auto i = m_head; i->next != nullptr; i = i->next)
             std::cout << i->value << ", ";
         std::cout << std::endl;
         std::cout << "size of list: " << size() << std::endl;
     }

private:
    node<Type>* m_head;
    node<Type>* m_tail;
    void init_first(const value_type& value)
    {
        m_head = new node<value_type>(value, nullptr, nullptr);
        m_tail = m_head;
    }

    void delete_last()
    {
        delete m_head;
        m_head = m_tail = nullptr;
```

```
        }

        bool is_single()
        {
            return m_head == m_tail && m_head != nullptr;
        }
    };

}// namespace stepik
```

## ПРИЛОЖЕНИЕ Б

Исходный код программы vector.h

```
#include <assert.h>
#include <iostream>
#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)
    {
      new_memory(count);
    }

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

    vector(std::initializer_list<Type> init)
    {
        new_memory(init.size());
        std::copy(init.begin(), init.end(), m_first);
    }

    vector(const vector& other)
    {
        new_memory(other.size());
        std::copy(other.begin(), other.end(), m_first);
    }

    vector(vector&& other)
    {
        m_first = nullptr;
        m_last = nullptr;
        std::swap(m_first, other.m_first);
```

```
    std::swap(m_last, other.m_last);
}

~vector()
{
    delete_memory();
}
  vector& operator=(vector other)
  {
      delete_memory();
      m_first = nullptr;
      m_last = nullptr;
      std::swap(m_first, other.m_first);
      std::swap(m_last, other.m_last);
      return *this;
  }

template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    delete_memory();
    new_memory(last - first);
    std::copy(first, last, m_first);
}
// resize methods
void resize(size_t count)
{
    auto* buf = new Type[count]();
    if (count > m_last - m_first)
            std::copy(m_first, m_last, buf);
    else
            std::copy(m_first, m_first + count, buf);
    delete_memory();
    m_first = buf;
    m_last = m_first + count;
}

//erase methods
iterator erase(const_iterator pos)
{
    auto mod_size = m_last - m_first - 1;
    auto mod_vector = new Type[mod_size]();
    auto erase_el = pos - m_first;
    auto del_element = pos - m_first;
    std::copy(m_first, m_first + erase_el, mod_vector);
    std::copy(m_first + erase_el + 1, m_last, mod_vector + erase_el);

    delete_memory();

    m_first = mod_vector;
    m_last = mod_vector + mod_size;

   return m_first + erase_el;
}

iterator erase(const_iterator first, const_iterator last)
{
    if (first == last)
            return m_first + (last - m_first);
    auto mod_size = m_last - m_first - (last - first);
    auto mod_vector = new Type[mod_size]();
```

```
        auto p1 = first - m_first, p2 = last - m_first;
        std::copy(m_first, m_first + p1, mod_vector);
        std::copy(m_first + p2, m_last, mod_vector + p1);

        delete_memory();
        m_first = mod_vector;
        m_last = mod_vector + mod_size;
       return m_first + p1;

    }

    //insert methods
    iterator insert(const_iterator pos, const Type& value)
    {
      auto mod_size = m_last - m_first + 1;
            Type* mod_data = new Type[mod_size]();

            auto insert_el = (pos - m_first);
            std::copy(m_first, m_first + insert_el, mod_data);
            mod_data[insert_el] = value;
            std::copy(m_first + insert_el, m_last, mod_data + insert_el + 1);

            delete_memory();

            m_first = mod_data;
            m_last = mod_data + mod_size;

            return m_first + insert_el;
    }

    template <typename InputIterator>
    iterator insert(const_iterator pos, InputIterator first, InputIterator
last)
    {
      if (first == last)
            {
                return m_first + (pos - m_first);
            }

            auto num_els = last - first;

            auto mod_size = num_els + m_last - m_first;
            Type* mod_data = new Type[mod_size]();

            auto first_in = pos - m_first;
            std::copy(m_first, m_first + first_in, mod_data);
            std::copy(first, last, mod_data + first_in);
            std::copy(m_first + first_in, m_last, mod_data + first_in +
num_els);

            delete_memory();

            m_first = mod_data;
            m_last = mod_data + mod_size;

            return m_first + first_in;
    }

    //push_back methods
```

```
void push_back(const value_type& value)
{
  auto mod_size = m_last - m_first + 1;
  auto mod_vector = new Type[mod_size]();
  std::copy(m_first, m_last, mod_vector);
  mod_vector[mod_size - 1] = value;
  delete_memory();
  m_first = mod_vector;
  m_last = mod_vector + mod_size;
}

//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;
    }
    void print()
    {
        for(int i=0; i<size(); i++)
            std::cout << m_first[i] << ", ";
        std::cout << std::endl;
    }

  private:
    reference checkIndexAndGet(size_t pos) const
    {
      if (pos >= size())
      {
        throw std::out_of_range("out of range");
      }

      return m_first[pos];
    }

  private:
        void new_memory(size_t memory)
        {
            auto* mem = memory ? new Type[memory]() : nullptr;

            m_first = mem;
            m_last = mem + memory;
        }
      void delete_memory()
      {
          delete[] m_first;
      }
    iterator m_first;
    iterator m_last;
  };
}// namespace stepik
```