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: Векторы и списки

Студент гр. 7304 _____ Пэтайчук Н.Г.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

Цель работы

Изучение стандартных контейнеров языка C++, а именно векторов и списков.

Постановка задачи

Необходимо реализовать шаблонные классы вектора и списка, при этом поведение данных классов должно быть идентично поведению стандартных контейнеров *std::vector* и *std::list* соответственно.

Описание решения

1. Первым делом был реализован класс вектора, который сдержит в себе в качестве полей два указателя на начало и конец вектора в памяти, что однозначно задаёт данный вектор. Спроектированный класс вектора содержит следующие методы:
   a) Конструктор от размера (элементы задаются собственными конструкторами по умолчанию);
   b) Конструктор от двух итераторов, указывающих на начало и конец области памяти, откуда будут скопированы элементы вектора;
   c) Конструктор от списка инициализации (так как список инициализации имеет методы получения итераторов на начало и конец, то вызывается конструктор от этих двух итераторов, описанный выше);
   d) Конструктор копирования (использует конструктор от двух итераторов);
   e) Конструктор перемещения;
   f) Деструктор;
   g) Метод изменения размера вектора;
   h) Методы вставки одного элемента или нескольких элементов, заданных при помощи двух итераторов на начало и конец той области памяти, где они находятся, на заданное итератором место. Также был реализован метод вставки элемента в конец вектора;
   i) Методы удаления одного элемента и интервала элементов;
   j) Перегруженный оператор [] и методы получения элемента по индексу;
   k) Методы получения итераторов на начало и конец;
   l) Метод получения размера вектора и метод, говорящий о том, пуст ли вектор;

   Дополнительно были реализованы приватные методы своппинга векторов и проверки индекса и интервала, на конец и начало которой указывают два итератора, на корректность;

2. Следующим шагом была реализация класса итератора для списка, который должен был содержать ряд перегруженных операторов, а именно операторов префиксного и постфиксного инкрементирования, операторов на равенство и неравенство, операторов * и ->. Также данный класс содержит в себе конструктор копирования, перегруженный оператор присваивания, конструктор по умолчанию и приватный конструктор от указателя на структуру узла списка, который будет использоваться классом списка, объявленном в данном классе как дружественный. Сама структура узла списка содержит в себе значение узла и два указателя на такие же структуры («следующий» и «предыдущий»);
3. Дальше был реализован класс списка, хранящий в себе указатели на первый и последний элементы списков. Спроектированный класс вектора содержит следующие методы:
   a) Конструктор по умолчанию;
   b) Конструктор копирования;
   c) Конструктор перемещения;
   d) Деструктор;
   e) Методы вставки и удаления элементов из начала и конца;
   f) Методы вставки элемента на заданную позицию и удаления элемента на заданной позиции;
   g) Метод очистки списка;
   h) Методы получения итераторов на начало и конец, а также методы получения значений элементов в начале и в конце;
   i) Метод получения размера вектора и метод, говорящий о том, пуст ли вектор;

   Дополнительно были реализованы приватные методы вставки первого элемента списка и удаления последнего элемента списка;
4. Реализация головной функции, демонстрирующей функционал спроектированных классов;

Результат работы программы

```
Sqaure and qube table:
1) 1 - 1 - 1
2) 2 - 4 - 8
3) 3 - 9 - 27
4) 4 - 16 - 64
5) 5 - 25 - 125
6) 6 - 36 - 216
Numeral list is now empty.
Linear and qube vectors are now empty.
1 2 3 4 5 6 1 4 9 16 25 36 1 8 27 64 125 216
```

Выводы

В ходе лабораторной работы были изучены стандартные контейнеры языка C++, а именно вектор и список, и были спроектированы собственные шаблонные классы, копирующие функционал данных контейнеров (конструкторы, деструкторы, методы удаления и вставки, методы получения итераторов и элементов головы и хвоста). Хоть работа с памятью в разработанных классах сильно упрощена, их поведение идентично поведению стандартных классов списка и вектора, что и было главной задачей при создании своих классов.

Приложение А: Исходный код программы

- stepik_containers.h

```
#pragma once

#include <assert.h>
#include <algorithm>
#include <cstddef>
#include <initializer_list>
#include <stdexcept>

using std::rotate;
using std::copy;
using std::invalid_argument;
using std::length_error;
using std::out_of_range;

namespace stepik
{
    //Vector
    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)
        {
            if (count < 0)
                throw length_error("Incorrect vector length.");

            m_first = new Type[count];
            m_last = m_first + count;
        }

        template <typename InputIterator>
        vector(InputIterator first, InputIterator last)
        {
            if (last < first)
                throw invalid_argument("Incorrect interval for
creating vector.");
```

```
            iterator first_t = (iterator) first;
            iterator last_t = (iterator) last;
            m_first = new Type[last_t - first_t];
            m_last = m_first + (last_t - first_t);
            copy(first, last, m_first);
        }

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

        vector(const vector& other) :
          vector(other.m_first, other.m_last)
        {
        }

        vector(vector&& other) :
          m_first(other.begin()),
          m_last(other.end())
        {
            other.m_last = nullptr;
            other.m_first = nullptr;
        }

        ~vector()
        {
            delete [] m_first;
        }

        //resize method
        void resize(size_t count)
        {
            if (count == this->size())
                return;

            size_t copy_size = (count < this->size()) ? count :
this->size();
            vector resized_vector(count);
            copy(m_first, m_first + copy_size,
resized_vector.m_first);
            swap(*this, resized_vector);
        }

        //insert methods
        iterator insert(const_iterator pos, const Type& value)
        {
            if ((pos < m_first) || (pos > m_last))
                throw out_of_range("Iterator points on non-vector
element.");

            size_t pos_index = (iterator) pos - m_first;
```

```
            this->resize(this->size() + 1);
            copy(m_first + pos_index, m_last - 1, m_first +
pos_index + 1);
            *(m_first + pos_index) = value;
            return m_first + pos_index;
        }

        template <typename InputIterator>
        iterator insert(const_iterator pos, InputIterator first,
InputIterator last)
        {
            if ((last < first) || (pos < m_first) || (pos >
m_last))
                throw invalid_argument("Iterator points on non-
vector element or incorrect interval for inserting.");

            size_t offset = last - first;
            size_t pos_index = (iterator) pos - m_first;
            this->resize(this->size() + offset);
            copy(m_first + pos_index, m_last - offset, m_first +
pos_index + offset);
            copy(first, last, m_first + pos_index);
            return m_first + pos_index;
        }

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

        //erase methods
        iterator erase(const_iterator pos)
        {
            if ((pos < m_first) || (pos >= m_last))
                throw std::invalid_argument("Iterator points on
non-vector element.");

            size_t index = (iterator) pos - m_first;
            std::rotate(m_first + index,
                        m_first + index + 1, m_last);
            this->resize(this->size() - 1);
            return m_first + index;
        }

        iterator erase(const_iterator first, const_iterator last)
        {
            if (isInvalidInterval(first, last))
                throw std::invalid_argument("Incorrect interval
for erasing.");
);

            size_t first_index = (iterator) first - m_first;
```

```
            size_t last_index = (iterator) last - m_first;
            std::rotate(m_first + first_index,
                        m_first + last_index, m_last);
            this->resize(this->size() - (last_index -
first_index));
            return m_first + first_index;
        }

        //at methods
        reference at(size_t pos)
        {
            return checkIndexAndGet(pos);
        }

        const_reference at(size_t pos) const
        {
            return checkIndexAndGet(pos);
        }

        //operator[] methods
        reference operator[](size_t pos)
        {
            return m_first[pos];
        }

        const_reference operator[](size_t pos) const
        {
            return m_first[pos];
        }

        //begin iterator methods
        iterator begin()
        {
            return m_first;
        }

        const_iterator begin() const
        {
            return m_first;
        }

        //end iterator 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 out_of_range("out of range");

            return m_first[pos];
        }

        void swap(vector &first, vector &second)
        {
            std::swap(first.m_first, second.m_first);
            std::swap(first.m_last, second.m_last);
        }

        iterator m_first;
        iterator m_last;
    };

    //List
    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
    {
        return m_node == other.m_node;
    }

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

    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)
    {
```

```
            list_iterator prev_value(*this);
            ++(*this);
            return prev_value;
        }

    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()
        {
            clear();
        }

        list(const list& other) :
          m_head(nullptr),
          m_tail(nullptr)
        {
            for (auto iter = other.m_head; iter != nullptr; iter =
iter->next)
                push_back(iter->value);
        }

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

```
        list& operator=(const list& other)
        {
            if (this != &other)
            {
                clear();
                for (auto iter = other.m_head; iter != nullptr;
iter = iter->next)
                    push_back(iter->value);
            }
            return *this;
        }

        //iterators methods
        list::iterator begin()
        {
            return iterator(m_head);
        }

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

        //insert methods
        void push_front(const value_type &value)
        {
            if (empty())
            {
                node<Type> *first_elem = new node<Type>(value,
nullptr, nullptr);
                m_head = first_elem;
                m_tail = first_elem;
                return;
            }

            m_head->prev = new node<Type>(value, m_head, nullptr);
            m_head = m_head->prev;
        }

        void push_back(const value_type& value)
        {
            if (empty())
            {
                create_first_elem(value);
                return;
            }

            m_tail->next = new node<Type>(value, nullptr, m_tail);
            m_tail = m_tail->next;
        }

        iterator insert(iterator pos, const Type& value)
        {
```

```
            if (pos == m_head)
            {
                push_front(value);
                return iterator(m_head);
            }
            else if (pos == nullptr)
            {
                push_back(value);
                return iterator(m_tail);
            }

            for (iterator iter(m_head); iter != nullptr; ++iter)
                if (iter == pos)
                {
                    auto new_elem = new node<Type>(value,
pos.m_node, pos.m_node->prev);
                    new_elem->next->prev = new_elem;
                    new_elem->prev->next = new_elem;
                    return iterator(new_elem);
                }
            throw out_of_range("Iterator points on non-list
element.");
            return iterator();
        }

        //erase methods
        void pop_front()
        {
            if (empty())
                return;
            else if (m_head == m_tail)
            {
                delete_last_elem();
                return;
            }

            m_head = m_head->next;
            delete m_head->prev;
            m_head->prev = nullptr;
        }

        void pop_back()
        {
            if (empty())
                return;
            else if (m_head == m_tail)
            {
                delete_last_elem();
                return;
            }

            m_tail = m_tail->prev;
            delete m_tail->next;
```

```
            m_tail->next = nullptr;
        }

        iterator erase(iterator pos)
        {
            if (pos == m_head)
            {
                pop_front();
                return iterator(m_head);
            }
            else if (pos == m_tail)
            {
                pop_back();
                return iterator(m_tail);
            }

            for (iterator iter(m_head); iter != nullptr; ++iter)
                if (iter == pos)
                {
                    auto old_elem = pos.m_node;
                    old_elem->next->prev = old_elem->prev;
                    old_elem->prev->next = old_elem->next;
                    iterator next_elem(old_elem);
                    delete old_elem;
                    return next_elem;
                }
            throw out_of_range("Iterator points on non-list
element.");
            return iterator();
        }

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

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

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

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

```
        const_reference back() const
        {
            return m_tail->value;
        }

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

        //size method
        size_t size() const
        {
            size_t size = 0;
            node<Type> *iter = m_head;
            while (iter != nullptr)
            {
                 size++;
                 iter = iter->next;
            }
            return size;
        }

    private:
        void delete_last_elem()
        {
            delete m_head;
            m_head = nullptr;
            m_tail = nullptr;
        }

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

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

- main.cpp

```
#include <iostream>
#include "stepik_containers.h"

using namespace stepik;
```

```
using std::cout;
using std::endl;

int main()
{
    list<int> numeral_list;
    vector<int> linear_vector = {1, 2, 3, 4, 5};
    vector<int> square_vector(linear_vector);

    for (int i = 1; i < 6; i++)
    {
        numeral_list.push_back(i + 1);
        square_vector[i - 1] *= square_vector[i - 1];
    }

    vector<int> qube_vector(square_vector.begin(),
square_vector.end());
    for (int i = 0; i < 5; i++)
        qube_vector[i] *= linear_vector[i];

    linear_vector.push_back(6);
    square_vector.push_back(36);
    qube_vector.push_back(216);
    numeral_list.push_front(1);

    cout << "Square and qube table:" << endl;
    for (int i = 0; i < 6; i++)
    {
        cout << i + 1 << ") " << linear_vector[i] << " - "
             << square_vector[i] << " - " << qube_vector[i] <<
endl;
        if (i % 2 == 0)
            numeral_list.pop_front();
        else
            numeral_list.pop_back();
    }

    if (numeral_list.empty())
        cout << "Numeral list is now empty." << endl;

    vector<int> all_element_vector = square_vector;
    all_element_vector.insert(all_element_vector.begin(),
linear_vector.begin(), linear_vector.end());
    linear_vector.erase(linear_vector.begin(),
linear_vector.end());
    for (int i = 0; i < 6; i++)
    {
        all_element_vector.insert(all_element_vector.end(),
qube_vector[0]);
        qube_vector.erase(qube_vector.begin());
    }

    if (linear_vector.empty() && qube_vector.empty())
```

```
        cout << "Linear and qube vectors are now empty." << endl;
    for (unsigned int i = 0; i < all_element_vector.size(); i++)
        cout << all_element_vector[i] << " ";
    cout << endl;

    return 0;
}
```