VOID 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 Мищенко М. А.
Преподаватель Размочаева Н.В.

Санкт-Петербург
2019

Цель работы.

Реализовать базовый функционал, семантически аналогичный функционалу из стандартной библиотеки шаблонов для классов вектор и линейный список.

Задание.

Необходимо реализовать конструкторы и деструктор для контейнера вектор. Предполагается реализация упрощенной версии вектора, без резервирования памяти под будущие элементы.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции resize и erase для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции insert и push_back для контейнера вектор.

Поведение реализованных функций должно быть таким же, как у класса std::vector (http://ru.cppreference.com/w/cpp/container/vector). Семантику реализованных функций нужно оставить без изменений.

Необходимо реализовать список со следующими функциями:

1. Вставка элементов в голову и в хвост;
2. Получение элемента из головы и из хвоста;
3. Удаление из головы, хвоста и очистка;
4. Проверка размера.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции:

1. Деструктор;
2. Конструктор копирования;
3. Конструктор перемещения;
4. Оператор присваивания.

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

На данном шаге с использованием итераторов необходимо реализовать:

1. Вставку элементов (Вставляет value перед элементом, на который указывает pos. Возвращает итератор, указывающий на вставленный value),

2. 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 (http://ru.cppreference.com/w/cpp/container/list). Семантику реализованных функций нужно оставить без изменений.

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).

Ход работы.

Реализован базовый функционал, семантически аналогичный функционалу из стандартной библиотеки шаблонов для классов вектор и линейный список.

Исходный код.

Код класса vector представлен в приложении А.

Код класса list представлен в приложении Б.

Выводы.

В ходе написания лабораторной работы были реализованы классы вектор и список, аналогичные классам из стандартной библиотеки. над этой работой.

ПРИЛОЖЕНИЕ А

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)
        {  if (count>0)
        { m_first=new Type[count];
            m_last=m_first+count;}
        else  m_first=m_last=nullptr;
        }

        template <typename InputIterator>
        vector(InputIterator first, InputIterator last)
        {
            if((last-first)>0)
            { m_first=new Type[last-first];
                m_last=m_first+(last-first);
                std::copy(first,last,this->m_first);}
            else m_first= m_last=nullptr;
        }

        vector(std::initializer_list<Type> init)
        {
            if((init.size())>0)
            { m_first=new Type[init.size()];
                m_last=m_first+init.size();
```

```
                std::copy(init.begin(),init.end(),this-
>m_first);}
            else m_first=m_last=nullptr;
        }

        vector(const vector& other)
        {
            if (other.size()>0){
                this->m_first=new Type[other.size()];
                this->m_last=this->m_first+other.size();
                std::copy(other.m_first,other.m_last,this-
>m_first);}
            else m_first=m_last=nullptr;
        }

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

        ~vector()
        {
            if( m_first!=nullptr)
            { delete [] m_first;
                m_first=m_last=nullptr;}
        }


        //insert methods
        iterator insert(const_iterator pos, const Type& value)
        {iterator Pos=iterator(pos);
            size_t sz=size();
            size_t count = std::distance(m_first,Pos);
            Pos=m_first+count;
            iterator new_arr=new Type[size()+1];
            iterator Pos1=new_arr;
            Pos1=Pos1+count;
            if (Pos!=m_first)
                std::copy(m_first,Pos,new_arr);
            *Pos1=value;
            std::copy(Pos,m_last,Pos1+1);
            delete [] this->m_first;
            this->m_first=nullptr;
            std::swap(m_first,new_arr);
            m_last=m_first+sz+1;
            return m_first+count;
        }
```

```
template <typename InputIterator>
        iterator insert(const_iterator pos, InputIterator first,
InputIterator last)
        {  size_t sz=size();
            iterator Pos = iterator(pos);
            iterator First = iterator(first);
            iterator Last = iterator(last);
            size_t count1 = std::distance(First,Last);
            size_t count2 = std::distance(m_first,Pos);

            Pos=m_first+count2;
            iterator new_arr=new Type[size()+count1];
            iterator Pos1=new_arr;
            Pos1=Pos1+count2;
            if (Pos!=m_first)
                std::copy(m_first,Pos,new_arr);
            std::copy(First,Last,Pos1);
            std::copy(Pos,m_last,Pos1+count1);
            delete [] this->m_first;
            this->m_first=nullptr;
            std::swap(m_first,new_arr);
            m_last=m_first+sz+count1;
            return m_first+count2;
        }

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

        void resize(size_t count)
        {
            if( count>size()){
                iterator new_arr=new Type[count];
                std::copy(this->m_first,this->m_last,new_arr);
                delete [] this->m_first;
                this->m_first=nullptr;
                std::swap(m_first,new_arr);
                m_last=m_first+count;}

            if( count<size()){
                iterator new_arr=new Type[count];
                m_last=m_first+count;
                std::copy(this->m_first,this->m_last,new_arr);
                delete [] this->m_first;
                this->m_first=nullptr;
                std::swap(m_first,new_arr);
                m_last=m_first+count;}
        }
        iterator erase(const_iterator pos)
        {  iterator Pos = iterator(pos);
```

```
            std::rotate(Pos, Pos + 1, m_last);
            --m_last;
            return Pos;
        }

        iterator erase(const_iterator first, const_iterator
last)
        {
            iterator Pos1 = iterator(first);
            iterator Pos2 = iterator(last);
            std::rotate(Pos1, Pos2, m_last);
            m_last =m_last-(last-first);
            return (Pos1);
        }

        vector& operator=(const vector& other)
        {  if (m_first!=nullptr)
            delete [] m_first;

            if (other.size()>0){
                this->m_first=new Type[other.size()];
                this->m_last=this->m_first+other.size();
                std::copy(other.m_first,other.m_last,this-
>m_first);}
            else m_first=m_last=nullptr;

            return *this;
        }

        vector& operator=(vector&& other)
        {   if (m_first!=nullptr)
            delete [] m_first;

            m_first=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)
        {
            if (m_first!=nullptr)
                delete [] m_first;

            this->  m_first=new Type[last-first];
            this->   m_last=m_first+(last-first);
            std::copy(first,last,this->m_first);
        }
```

```
//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;
        }
        iterator test;
        iterator itr;
    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
```

ПРИЛОЖЕНИЕ Б

РЕАЛИЗАЦИЯ КЛАССА LIST

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

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)
        {
            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 temp(*this);
        ++(*this);
        return temp;
    }
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::iterator begin()
        {
            return iterator(m_head);
        }

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



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

        iterator erase(iterator pos)
        {
            if(pos.m_node ==nullptr)
                return nullptr;

            if(pos.m_node == m_head)
            {pop_front();
                return iterator(m_head);}
```

```
            if(pos.m_node == m_tail)
            {pop_back();
                return iterator(m_tail);}

            else
            {pos.m_node->prev->next = pos.m_node->next;
                pos.m_node->next->prev = pos.m_node->prev;
                node<Type>* temp = pos.m_node;
                iterator itr(pos.m_node->next);
                delete temp;
                return itr;}
        }

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

        }
        void push_front(const value_type& value)
        {
            node<Type> *temp = new node<Type>(value, nullptr,
nullptr);
            if(!empty())
            {temp->next = m_head;
                m_head->prev = temp;
                m_head = temp;}
            else
            {m_head = temp;
                m_tail = temp;}
        }
        reference front()
        {
            return m_head->value;
        }

        reference back()
        {
            return m_tail->value;
        }
        void clear()
        {
            node<Type>* temp;
```

```
    while(m_head != nullptr)
    {temp = m_head->next;
        delete m_head;
        m_head=temp;
    }
    m_head = m_tail = nullptr;
}
void pop_front()
{
    if(!empty()){
        if (m_head==m_tail){
            delete m_head;
            m_head=m_tail=nullptr;}
        else {
            node<Type>* temp=m_head->next;
            delete m_head;
            m_head=temp;
            m_head->prev = nullptr;}
    }

}

void pop_back()
{
    if(!empty()){
        if (m_head==m_tail){
            delete m_head;
            m_head=nullptr;m_tail=nullptr;}
        else {
            node<Type>* temp=m_tail->prev;
            delete m_tail;
            m_tail=temp;
            m_tail->next = nullptr;}
    }

}
bool empty() const
{
    if (m_head==nullptr && m_tail==nullptr)
        return true;
    else return  false;
}
size_t size() const
{
    node<Type>* temp = m_head;
    size_t size = 0;
    while(temp !=nullptr)
    {
        size++;
        temp = temp->next;
    }
    return size;
```

```
        }



    private:
        //your private functions

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

}// namespace stepik
```