

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ
по лабораторной работе №3
по дисциплине «Объектно-ориентированное программирование»
Тема: «Контейнеры вектор и список»

Студент гр. 7381

Павлов А.П.

Преподаватель

Жангиров Т.М.

Санкт-Петербург

2019

Цель работы.

Изучить стандартные контейнеры `vector` и `list` языка C++.

Задание.

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 реализован 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 А.

Вывод.

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

Приложение А. Файл lab_vector.h.

```
#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)
        : m_first(count ? new Type[count] : nullptr), m_last(m_first + count)
        {}

        template <typename InputIterator>
        vector(InputIterator first, InputIterator last)
        {
            int count = last - first;
            if (count > 0){
                m_first = new Type[count];
                m_last = m_first + count;
            }
        }
    };
}
```

```

        std::copy(first, last, m_first);
    }
    else{
        m_first = nullptr;
        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(), m_first);
    }
    else{
        m_first = nullptr;
        m_last = nullptr;
    }
}

vector(const vector& other)
: vector(other.begin(), other.end())
{
    // use previous step implementation
}

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

~vector()
{
    if (m_first){

```

```

        delete [] m_first;
        m_first = m_last = nullptr;
    }
}

//assignment operators
vector& operator=(const vector& other)
{
    if(this != &other)
    {
        delete[] m_first;
        m_first = new value_type[other.size()];
        m_last = m_first + other.size();
        std::copy(other.m_first, other.m_last, m_first);
    }
    return *this;
}

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

    m_first = nullptr;
    m_last = nullptr;
    std::swap(m_first, other.m_first);
    std::swap(m_last, other.m_last);
    return *this;
}

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    if (m_first!=nullptr)
        delete [] m_first;
    m_first = new value_type[last-first];

```

```

    m_last = m_first + (last-first);
    std::copy(first, last, m_first);
}

// resize methods
void resize(size_t count)
{
    if(count == size())
        return;
    iterator temp = new Type[count];
    if(count > size())
        std::copy(m_first, m_last, temp);
    else
        std::copy(m_first, m_first+count, temp);
    delete[] m_first;
    m_first = temp;
    m_last = m_first+count;
}

//erase methods
iterator erase(const_iterator pos)
{
    size_t offset = pos - m_first;
    std::rotate( m_first+offset, m_first+offset+1, m_last);
    resize(size()-1);
    return m_first + offset;
}

iterator erase(const_iterator first, const_iterator last)
{
    size_t new_first = first - m_first;
    size_t new_last = last - m_first;
    std::rotate(m_first + new_first, m_first + new_last, m_last);
    resize(size() - new_last + new_first);
    return m_first + new_first;
}

```



```

//insert methods
iterator insert(const_iterator pos, const Type& value)
{
    size_t offset = pos - m_first;
    resize(size()+1);
    *(m_last-1) = value;
    std::rotate(m_first+offset, m_last-1, m_last);
    return m_first + offset;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
    size_t offset = pos - m_first;
    resize( size() + (last-first));
    std::copy(first, last, m_last - (last-first));
    std::rotate(m_first+offset, m_last - (last-first) , m_last);
    return m_first + offset;
}

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

```

Приложение Б. Файл lab_list.h.

```
#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)
{
    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 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();
    }

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

    list(const list& other):list()
    {
        for(node<Type> *ptr = other.m_head; ptr != nullptr; ptr = ptr->next)
            push_back(ptr->value);
    }

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

    list& operator= (const list& other)
    {

```

```

    if(this != &other){
        clear();
        for (node<Type> *ptr = other.m_head; ptr != nullptr; ptr = ptr->next)
            push_back(ptr->value);
    }
    return *this;
}

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 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;
    }
}

bool empty() const
{
    if (m_head == nullptr && m_tail == nullptr)
        return true;
    else
        return false;
}

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;
        }
    }
}

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

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

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;
    }
}

iterator insert(iterator pos, const Type& value)
{
    if (pos.m_node == nullptr){
        push_back(value);
        return iterator(m_tail);
    }
    if (pos.m_node->prev == nullptr){
        push_front(value);
    }
}

```



```

        return iterator(m_head);
    }
    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 pos;
    if (pos.m_node->next == nullptr){
        pop_back();
        return nullptr;
    }
    if (pos.m_node->prev == nullptr){
        pop_front();
        return iterator(m_head);
    }
    node<Type>* temp = pos.m_node->next;
    pos.m_node->prev->next = temp;
    temp->prev = pos.m_node->prev;
    delete pos.m_node;
    return temp;
}

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

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

private:
    //your private functions

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

} // namespace stepik

```