

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

Каляева А.В

Преподаватель

Размочаева Н.В

г. Санкт-Петербург

2019 г.

Цель работы:

Изучить реализацию контейнеров `vector` и `list` в языке программирования C++.

Задание:

Реализовать конструкторы, деструктор, операторы присваивания, функцию `assign`, функции `resize` и `erase`, функции `insert` и `push_back` для контейнера `vector`. Поведение реализованных функций должно быть таким же, как у класса `std::vector`.

Реализовать список со следующими функциями: вставка элементов в голову и в хвост, получение элемента из головы и из хвоста, удаление из головы, хвоста, проверка размера. Так же необходимо реализовать деструктор, конструктор копирования, оператор присваивания, оператор перемещения, операторы `=`, `==`, `!=`, `++`, `*`, `->`. Реализовать вставку элементов и удаление элементов. Поведение реализованных функций должно быть таким же, как у класса `std::list`.

Ход работы:

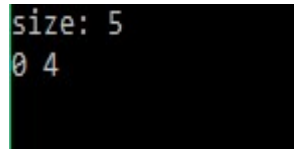
- 1) Реализация контейнера `vector`. Все функции данного контейнера были реализованы в соответствии с поведением класса `std::vector`.
 - a. На первом шаге были реализованы конструкторы и деструктор для контейнера `vector`.
 - b. На втором шаге были реализованы операторы присваивания и функция `assign`.
 - c. На третьем шаге были реализованы функции `resize` и `erase`.
 - d. На четвертом шаге были реализованы функции `insert` и `push_back`.
- 2) Реализация контейнера `list`. Все функции данного контейнера были реализованы в соответствии с поведением класса `std::list`.
 - a. На первом шаге были реализованы функции вставки элементов в голову и хвост, получения элемента из головы и из хвоста, удаления из головы и хвоста, проверки размера.
 - b. На втором шаге были реализованы деструктор, конструктор копирования, оператор присваивания, оператор перемещения.
 - c. На третьем шаге были реализованы операторы `=`, `==`, `!=`, `++`, `*`, `->`.
 - d. На четвертом шаге были реализованы функции для вставки элементов и удаления элементов.

Примеры работы программы:

1) Был создан вектор `vec`, в который с помощью функции `push_back` были добавлены элементы от 0 до 6, после с помощью функции `size()` был выведен размер полученного вектора, а затем и сам вектор был выведен на экран.

```
size: 7
0 1 2 3 4 5 6
```

2) Был создан список `lst`, в который помощью функции `push_back` были добавлены элементы от 0 до 4, после с помощью функции `size()` был выведен на экран размер списка, а затем с помощью функций `front()` и `back()` были выведены на экран первый и последний элемент списка.



```
size: 5
0 4
```

Заключение:

В ходе выполнения лабораторной работы были изучены и реализованы такие контейнеры, как вектор и список. Для заданных контейнеров были реализованы основные функции для работы с ними. Поведение реализованных функций соответствует классам `std::vector` и `std::list`.

Приложение А

Исходный код файла my_vector.hpp

```
#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){
                m_first=new Type[count];
                m_last=m_first+count;
            }
            else{
                m_first=nullptr;
                m_last=nullptr;
            }
        }

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

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

        vector(const vector& other): vector(other.size())
        {
            std::copy(other.m_first, other.m_last, m_first);
        }
    };
}
```

```

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

~vector()
{
    delete [] m_first;
    m_first=nullptr;
    m_last=nullptr;
}
//assignment operators
vector& operator=(const vector& other)
{
    if(this!=&other){
        vector tmp(other);
        tmp.swap(*this);
    }
    return *this;
}

vector& operator=(vector&& other)
{
    if(this!=&other){
        delete [] m_first;
        m_first=other.m_first;
        m_last=other.m_last;
        other.m_first=nullptr;
        other.m_last=nullptr;
    }
    return *this;
}

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    vector tmp (first, last);
    tmp.swap(*this);
}
// resize methods
void resize(size_t count)
{
    vector tmp(count);
    if(count>=size()){
        std::copy(m_first, m_last, tmp.m_first);
    }
    else{
        std::copy(m_first, m_first+count, tmp.m_first);
    }
    swap(tmp);
}

```

```

    }

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

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

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

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

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

private:
    iterator m_first;
    iterator m_last;
};
} // namespace stepik

```


Приложение В

Исходный код файла my_list.hpp

```
#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)
        {
            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 tmp(*this);
    ++(*this);
    return tmp;
}

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)
{
    copy(const_cast<list&>(other));
}

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

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

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

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

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

```

```

    }
}

void push_front(const value_type& value)
{
    if(m_head){
        node<value_type> *tmp=new node<value_type>(value, m_head, nullptr);
        m_head->prev=tmp;
        m_head=tmp;
    }
    else{
        m_head=new node <value_type>(value, nullptr, nullptr);
        m_tail=m_head;
    }
}

void pop_front()
{
    node<value_type> *tmp=m_head;
    m_head=m_head->next;
    if(m_head){
        m_head->prev=nullptr;
    }
    else{
        m_head=nullptr;
        m_tail=nullptr;
    }
    delete tmp;
}

void pop_back()
{
    node<value_type> *tmp=m_tail;
    m_tail=m_tail->prev;
    if(m_tail){
        m_tail->next=nullptr;
    }
    else{
        m_head=nullptr;
        m_tail=nullptr;
    }
    delete tmp;
}

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

```

```

        return iterator(m_head);
    }
    else{
        node<Type>* tmp = new node<Type>(value, pos.m_node, pos.m_node->prev);
        pos.m_node->prev = pos.m_node->prev->next= tmp;
        return iterator(tmp);
    }
}

iterator erase(iterator pos)
{
    if (!pos.m_node){
        return nullptr;
    }
    else if (!pos.m_node->prev){
        pop_front();
        return iterator(m_head);
    }
    else if (!pos.m_node->next){
        pop_back();
        return iterator(m_tail);
    }
    else{
        node<Type>* tmp = pos.m_node;
        pos.m_node->next->prev = pos.m_node->prev;
        pos.m_node->prev->next = pos.m_node->next;
        iterator n(pos.m_node->next);
        delete tmp;
        return n;
    }
}

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

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

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

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

void clear()

```

```

{
    while(m_head){
        pop_back();
    }
    m_head=nullptr;
    m_tail=nullptr;
}

bool empty() const
{
    if(m_head){
        return false;
    }
    else{
        return true;
    }
}

size_t size() const
{
    size_t count=0;
    node<value_type> *tmp=m_head;
    while(tmp!=nullptr){
        count++;
        tmp=tmp->next;
    }
    return count;
}

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

void copy(list &other){
    node<value_type> *tmp=other.m_head;
    while(tmp){
        push_back(tmp->value);
        tmp=tmp->next;
    }
}

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

} // namespace stepik

```