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. Список»

Студентка гр. 7382 _____ Дерябина П.С,

Преподаватель _____ Жангиров Т.М.

Санкт-Петербург

2019

Цель работы

Изучить работу стандартных контейнеров vector и list языка C++.

Задание

1. Необходимо реализовать класс для контейнера вектор, составляющие класса: конструкторы, деструктор, операторы присваивания, функцию assign, функции resize и erase, функции insert и push_back. Поведение реализованных функций должно быть таким же, как у класса std::vector.

2. Необходимо реализовать класс для контейнера список, составляющие класса: деструктор, конструктор копирования, конструктор перемещения,о ператор присваивания; функции вставки элементов в голову и в хвост, получения элемента из головы и из хвоста, удаления из головы и хвоста, очистки, проверки размера; итератор.

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

1. Класс vector кроме требуемых функций содержит 2 приватных поля: указатель на первый и последний элемент данных, а также 1 приватную функцию проверки на выход за рамки вектора. Реализация класса представлена в приложении А.

2. Класс list кроме требуемых функций содержит 2 приватных поля: указатель на первый и последний узел списка. В классе list_iterator содержится одно поле — указатель на узел списка. Реализация класса представлена в приложении Б.

Вывод

Были изучены и реализованы собственные контейнеры вектора и списка.

ПРИЛОЖЕНИЕ А

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

    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())
{
    int count = 0;
    for (auto& element : init) {
        m_first[count] = element;
        ++count;
    }
}

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

vector(vector&& other)
{
    std::swap(m_first, other.m_first);
    std::swap(m_last, other.m_last);
}
~vector()
{
    delete[] m_first;
    m_first = m_last = nullptr;
```

```
}

vector& operator=(const vector& other)
{
   delete[] m_first;
   m_first = new Type[other.size()];
   m_last = m_first + other.size();
   std::copy(other.begin(), other.end(), m_first);
   return *this;
}

vector& operator=(vector&& other)
{
   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)
{
   delete[] m_first;
   int size = last - first;
   m_first = new Type[size];
   m_last = m_first + size;
   std::copy(first, last, m_first);
}
```

```
//insert methods
iterator insert(const_iterator pos, const Type& value)
{
    size_t ind = pos - m_first;
    size_t old_size = m_last - m_first;
    size_t new_size = old_size + 1;

    resize(new_size);
    std::copy(m_first + ind, m_last, m_first + ind + 1);
    m_first[ind] = value;

    return m_first + ind;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
    size_t ind = pos - m_first;
    size_t add = last - first;
    size_t old_size = m_last - m_first;

    resize(old_size + add);

    for (int i = old_size + add - 1; i != ind; i--) {
        m_first[i] = m_first[i - add];
    }
    std::copy(first, last, m_first + ind);

    return m_first + ind;
```

```
}

//push_back methods
void push_back(const value_type& value)
{
   insert(m_last, 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

```
#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(m_node);
        tmp = tmp->next;
        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;

    list()
        : m_head(nullptr)
        , m_tail(nullptr)
    {
    }

    ~list()
    {
        clear();
    }

    list(const list& other)
        : m_head(nullptr)
        , m_tail(nullptr)
    {
```

```
    node<Type>* tmp = other.m_head;
    while (tmp) {
        push_back(tmp->value);
        tmp = tmp->next;
    }
}

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

list& operator=(const list& other)
{
    clear();
    node<Type>* tmp = other.m_head;
    while (tmp) {
        push_back(tmp->value);
        tmp = tmp->next;
    }

    return *this;
}

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>* tmp = pos.m_node->prev;
    pos.m_node->prev = new node<Type>(value, pos.m_node, tmp);
    tmp->next = pos.m_node->prev;

    return iterator(pos.m_node->prev);
}

iterator erase(iterator pos)
{
    if (pos.m_node->next == nullptr) {
        pop_back();
        return iterator();
    }

    if (pos.m_node->prev == nullptr) {
        pop_front();
        return iterator(m_head);
    }
```

```
    node<Type>* tmp = pos.m_node;
    delete pos.m_node;
    tmp->prev->next = tmp->next;
    tmp->next->prev = tmp->prev;

    return tmp->next;
}

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

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

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 pop_front()
{
    if (empty())
        return;

    if (m_head == m_tail) {
        delete m_head;
        m_head = m_tail = nullptr;
        return;
    }
```

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

void pop_back()
{
    if (empty())
        return;

    if (m_head == m_tail) {
        delete m_head;
        m_head = m_tail = nullptr;
        return;
    }

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

void clear()
{
    node<Type>* tmp = m_head;
    node<Type>* next = nullptr;
    while (tmp) {
        next = tmp->next;
        delete tmp;
```

```
        tmp = next;
    }
    m_head = m_tail = nullptr;
}

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

size_t size() const
{
    size_t size = 0;
    node<Type>* el = m_head;
    while (el) {
        size++;
        el = el->next;
    }

    return size;
}

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

list::iterator end()
{
```

```
        return iterator();
    }

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

} // namespace stepik
```