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 _____ Адамов Я.В.

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 реализованы 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 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;


        // Constructors and destructor

        explicit vector(size_t count = 0)
        : m_first(count ? new value_type[count] : nullptr), m_last(count ? m_first +
count : nullptr) {
        }

        template <typename InputIterator>
        vector(InputIterator first, InputIterator last) : vector(last - first) {
            for(int i = 0; i < last - first; i++)
                m_first[i] = (value_type)first[i];
        }

        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) : vector() {
            swap(*this, other);
        }

        ~vector() {
            delete[] m_first;
        }
```

```
        // Methods and operators

        vector& operator=(const vector& other) {
            if (this != &other) {
                vector temp(other);
                swap(*this, temp);
            }
            return *this;
        }

        vector& operator=(vector&& other) {
            if (this != &other)
                swap(*this, other);
            return *this;
        }

        template <typename InputIterator>
        void assign(InputIterator first, InputIterator last) {
            delete[] m_first;
            m_first = last - first ? new value_type[last - first] : nullptr;
            m_last = last - first ? m_first + (last - first) : nullptr;
            for (int i = 0; i < last - first; i++)
                m_first[i] = (value_type)first[i];
        }

        void resize(size_t count) {
            if (m_last - m_first != count) {
                vector temp(count);
                std::copy(m_first, m_last - m_first > count ? m_first + count : m_last,
temp.m_first);
                swap(*this, temp);
            }
        }

        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 f = first - m_first;
            size_t l = last - m_first;
            vector temp(*this);
```

```
    std::rotate(temp.m_first + f, temp.m_first + l, temp.m_last);
    temp.resize(temp.size() - l + f);
    *this = std::move(temp);
    return m_first + f;
}

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;
}

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

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

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

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

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

iterator begin() {
    return m_first;
}
```

```
        const_iterator begin() const {
            return m_first;
        }

        iterator end() {
            return m_last;
        }

        const_iterator end() const {
            return m_last;
        }

        size_t size() const {
            return m_last - m_first;
        }

        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];
        }

        void swap(vector& v1, vector& v2) {
            std::swap(v1.m_first, v2.m_first);
            std::swap(v1.m_last, v2.m_last);
        }


    private:
        iterator m_first;
        iterator m_last;
    };
}
```

Приложение Б. Файл list.h.

```
#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;



    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(nullptr) {
        }

        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;
```

```
// Constructors and destructor

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

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() {
    clear();
}


// Methods and operators

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;
}

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

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

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

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;
}
```

```
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()){
        if (m_head == m_tail){
            delete m_head;
            m_head = nullptr;
            m_tail = nullptr;
        } else {
            m_head = m_head->next;
            delete m_head->prev;
            m_head->prev = nullptr;
        }
    }
}

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

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

bool empty() const {
```

```
            return m_tail == nullptr;
        }

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

    private:

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

}
```