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. 7303 _____ Ермолаев Д.В.

Преподаватель _____ Размочаева Н.В.

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 Б.

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 list.cpp.

```
#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::iterator begin() {
            return iterator(m_head);
        }

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

        ~list() {
            clear();
        }

        list(const list& other)
            : m_head(nullptr), m_tail(nullptr) {
            auto tmp = other.m_head;

            while (tmp != nullptr) {
                push_back(tmp->value);
                tmp = tmp->next;
            }
```

```
}

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

list& operator= (const list& other) {
    clear();
    auto tmp = other.m_head;

    while (tmp != nullptr) {
        push_back(tmp->value);
        tmp = tmp->next;
    }

    return *this;
}

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

    m_tail->next = new node<Type>(value, nullptr, m_tail);
    m_tail=m_tail->next;
}

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

    m_head = new node<Type>(value, m_head, nullptr);
    m_head->next->prev = m_head;
}

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

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

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

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

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

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

        size_t size() const {
            size_t size = 0;

            for(node<Type>* currentNode = m_head; currentNode; currentNode =
currentNode->next, ++size);

            return size;
        }

        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;
            }

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

}// namespace stepik
```

Приложение Б.

Файл vector.cpp.

```
#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) {
                size_t vectorSize;
                vectorSize = last - first;
                m_first = new Type[vectorSize];
                m_last = m_first + vectorSize;
                std::copy(first, last, m_first);
            }

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

            vector(const vector& other) : vector(other.begin(), other.end()) {}

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

            ~vector() {
                delete[] m_first;
            }

            //assignment operators
            vector& operator=(const vector& other) {
                if(this != &other) {
                    delete[] m_first;
                    size_t vectorSize = other.m_last - other.m_first;
```

```
            m_first = new Type[vectorSize];
            m_last = m_first + vectorSize;
            std::copy(other.m_first, other.m_last, m_first);
            return *this;
        }
    }

    vector& operator=(vector&& other) noexcept {
        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) {
        *this = std::move(vector(first, last));
    }

    void resize(size_t count) {
        iterator end = (count > size()) ? m_last : m_first + count;
        vector tempVector(count);
        std::move(m_first, end, tempVector.m_first);
        std::swap(m_first, tempVector.m_first);
        std::swap(m_last, tempVector.m_last);
    }

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

    iterator erase(const_iterator first, const_iterator last) {
        iterator it = const_cast<iterator>(first);

        for(size_t i = 0, count = last - first; i < count; ++i) {
            it = erase(it);
        }

        return it;
    }

    iterator insert(const_iterator pos, const Type& value) {
        vector tempVector(size() + 1);
        size_t diff = pos - m_first;
        std::copy(m_first, m_first + diff, tempVector.m_first);
        *(tempVector.begin() + diff) = value;
        std::copy(m_first + diff, m_last, tempVector.begin() + diff + 1);
        *this = std::move(tempVector);
```

```
            return m_first + diff;
        }

        template <typename InputIterator>
        iterator insert(const_iterator pos, InputIterator first, InputIterator last) {
            vector tempVector(size() + (last - first));
            size_t diff = pos - m_first;
            std::copy(m_first, m_first + diff, tempVector.m_first);
            std::copy(first, last, tempVector.begin() + diff);
            std::copy(m_first + diff, m_last, tempVector.begin() + diff + (last -
first));
            *this = std::move(tempVector);
            return m_first + diff;
        }

        void push_back(const value_type& value) {
            insert(this->end(), 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("Index out of range");
                }

                return m_first[pos];
            }

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