VOID 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: «Вектор и список»

Студент гр. 7381 Минуллин М. А.
Преподаватель Жангиров Т. Р.

Санкт-Петербург
2019

Цель работы.

Реализовать базовый функционал, семантически аналогичный функционалу из стандартной библиотеки шаблонов для классов вектор и линейный список.

Задание.

Необходимо реализовать конструкторы и деструктор для контейнера вектор. Предполагается реализация упрощенной версии вектора, без резервирования памяти под будущие элементы.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции resize и erase для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции insert и push_back для контейнера вектор.

Поведение реализованных функций должно быть таким же, как у класса std::vector (http://ru.cppreference.com/w/cpp/container/vector). Семантику реализованных функций нужно оставить без изменений.

Необходимо реализовать список со следующими функциями:

1. Вставка элементов в голову и в хвост;
2. Получение элемента из головы и из хвоста;
3. Удаление из головы, хвоста и очистка;
4. Проверка размера.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции:

1. Деструктор;
2. Конструктор копирования;
3. Конструктор перемещения;

4. Оператор присваивания.

На данном шаге необходимо реализовать итератор для списка. Для краткости реализации можно ограничиться однонаправленным изменяемым (неконстантным) итератором. Необходимо реализовать операторы: =, ==, !=, ++ (постфиксный и префиксный), *, ->.

На данном шаге с использованием итераторов необходимо реализовать:

1. Вставку элементов (Вставляет value перед элементом, на который указывает pos. Возвращает итератор, указывающий на вставленный value),

2. 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 (http://ru.cppreference.com/w/cpp/container/list). Семантику реализованных функций нужно оставить без изменений.

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.

Реализация класса vector была получена путём разбора и упрощения кода из библиотеки stl. Удаление элемента в заданной позиции или интервала реализуется арифметическим сдвигом хранимых значений в векторе влево на один элемент или длину интервала соответственно. В аналогичных условиях операция вставки реализуется путём арифметического сдвига всего содержимого вектора начиная с позиции вставки вправо на единицу или на длину вставляемого интервала значений соответственно. Это происходит в случае, если зарезервированной памяти для вставки достаточно, в противном случае выделяется новый участок памяти, туда копируется старое содержимое

вектора до позиции вставки, затем копируется вставляемое значение или интервал значений, а затем оставшаяся часть старого вектора.

Обычно, резервируется в полтора раза больше памяти, чем необходимо в данный момент. Если это невозможно, то резервируется максимально возможный объём памяти.

Изменения размера происходит путём изменения значения размера массива (не путать с объёмом выделенной памяти).

Реализация остальных функций достаточно тривиальна. Таким образом, для работы вектора необходимо хранить информацию о размере используемой памяти, о размере выделенной памяти и указатель на участок памяти, в котором память, собственно, выделена.

Для реализации класса список хранятся указатели на голову списка, на хвост списка и количество элементов в списке. Последняя информация позволяет за $O(1)$ определить размер списка (против $O(n)$ в случае линейного подсчёта элементов списка).

Удаление элемента из списка осуществляется связыванием двух элементов справа и слева от удаляемого переброской указателей. После чего освобождается память для удаляемого элемента и возвращается указатель на следовавший за ним элемент.

Все остальные методы реализуются тривиальным образом.

Исходный код.

Код класса vector представлен в приложении А.

Код класса list представлен в приложении Б.

Выводы.

В ходе написания лабораторной работы были реализованы классы вектор и список, аналогичные классам из стандартной библиотеки. Полученные знания из предыдущих лабораторных работ были применены в ходе работы над этой работой.

ПРИЛОЖЕНИЕ А

РЕАЛИЗАЦИЯ КЛАССА VECTOR

```
#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 (_Buy(count))
        {
            m_last += count;
        }
    }

    template <typename InputIterator>
    vector(InputIterator first, InputIterator last)
    {
        size_t _Newsize = std::distance(first, last);
        if (_Buy(_Newsize))
        {
            m_last += _Newsize;
            std::copy(first, last, m_first);
        }
    }

    vector(std::initializer_list<Type> init)
    {
        if (_Buy(init.size()))
        {
```

```
        m_last += init.size();
        std::copy(init.begin(), init.end(), m_first);
    }
}

vector(const vector& other)
{
    if (_Buy(other.capacity()))
    {
        m_last += other.size();
        std::copy(other.begin(), other.end(), m_first);
    }
}

vector(vector&& other) : m_first(), m_last(), m_end()
{
    std::swap(m_first, other.m_first);
    std::swap(m_last, other.m_last);
    std::swap(m_end, other.m_end);
}

~vector()
{
    _Tidy();
}

//assignment operators
vector& operator=(const vector& other)
{
    _Tidy();
    if (_Buy(other.capacity()))
    {
        m_last += other.size();
        std::copy(other.begin(), other.end(), m_first);
    }
    return *this;
}

vector& operator=(vector&& other)
{
    _Tidy();
    std::swap(m_first, other.m_first);
    std::swap(m_last, other.m_last);
    std::swap(m_end, other.m_end);
    return *this;
}

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
```

```
    _Tidy();
    size_t _Newsize = std::distance(first, last);
    size_t _Newcapacity = _Calculate_growth(_Newsize);
    if (_Buy(_Newcapacity))
    {
        m_last += _Newsize;
        std::copy(first, last, m_first);
    }
}

// resize methods
void resize(size_t count)
{
    _Resize(count);
}

//erase methods
iterator erase(const_iterator pos)
{
    iterator _Pos = iterator(pos);
    std::rotate(_Pos, _Pos + 1, m_last);
    --m_last;
    return (_Pos);
}

iterator erase(const_iterator first, const_iterator last)
{
    size_t _Offset = std::distance(first, last);
    iterator _First = iterator(first);
    iterator _Last = iterator(last);
    std::rotate(_First, _Last, m_last);
    m_last -= _Offset;
    return (_First);
}

//insert methods
iterator insert(const_iterator pos, const Type& value)
{
    iterator _Pos = iterator(pos);
    size_t _Offset = std::distance(m_first, _Pos);
    if (!_Has_unused_capacity())
    {
        _Resize(size() + 1);
        _Pos = m_first + _Offset;
    }
    std::rotate(_Pos + 1, _Pos + size() - 1, m_last);
    *_Pos = value;
    return (_Pos);
}

template <typename InputIterator>
```

```
    iterator insert(const_iterator pos, InputIterator first,
InputIterator last)
    {
        iterator _Pos = iterator(pos);
        iterator _First = iterator(first);
        iterator _Last = iterator(last);

        size_t _Offset = std::distance(_First, _Last);
        size_t _Tillpos = std::distance(m_first, _Pos);
        if (_Unused_capacity() < _Offset)
        {
            _Resize(size() + _Offset);
            _Pos = m_first + _Tillpos;
        }

        std::rotate(_Pos, _Pos + size() - _Offset - _Tillpos, m_last);
        std::copy(first, last, _Pos);
        return (_Pos);
    }

    //push_back methods
    void push_back(const value_type& value)
    {
        if (!_Has_unused_capacity())
        {
            _Resize(size() + 1);
            --m_last;
        }
        ++m_last;
        *(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;
    }

    // capacity method
    size_t capacity() const
    {
        return m_end - 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];
    }

    bool _Buy(size_t _Newcapacity)
    {   // allocate array with _Newcapacity elements
        m_first = m_last = m_end = iterator();
        if (!_Newcapacity)
            return (false);
        m_first = new value_type[_Newcapacity];
        m_last = m_first;
        m_end = m_first + _Newcapacity;
        return (true);
    }

    void _Tidy()
    {   // free all storage
        if (m_first != iterator())
        {   // deallocate old array
            delete[] m_first;
            m_first = m_last = m_end = iterator();
        }
    }

    void _Change_array(const iterator _Newvec, const size_t _Newsize,
const size_t _Newcapacity)
    {   // discard old array, acquire new array
        _Tidy();
        m_first = _Newvec;
        m_last = _Newvec + _Newsize;
        m_end = _Newvec + _Newcapacity;
    }

    size_t _Calculate_growth(const size_t _Newsize) const
    {   // given _Oldcapacity and _Newsize, calculate geometric growth
        const size_t _Oldcapacity = capacity();

        const size_t _Geometric = _Oldcapacity + _Oldcapacity / 2;

        if (_Geometric < _Newsize)
        {
            return (_Newsize);  // geometric growth would be
insufficient
        }

        return (_Geometric);    // geometric growth is sufficient
    }

    size_t _Unused_capacity() const noexcept
    {   // micro-optimization for capacity() - size()
        return m_end - m_last;
    }
```

```
    bool _Has_unused_capacity() const noexcept
    {   // micro-optimization for capacity() != size()
        return m_end != m_last;
    }

    void _Resize(const size_t _Newsize)
    {   // trim or append elements, provide strong guarantee
        const size_t _Oldsize = size();
        const size_t _Oldcapacity = capacity();

        if (_Newsize > _Oldcapacity)
        {   // reallocate
            const size_t _Newcapacity = _Calculate_growth(_Newsize);
            iterator _Newvec = new value_type[_Newcapacity];
            std::copy(m_first, m_last, _Newvec);
            _Change_array(_Newvec, _Newsize, _Newcapacity);
        }
        else if (_Newsize > _Oldsize)
        {   // append
            m_last += (_Newsize - _Oldsize);
        }
        else if (_Newsize == _Oldsize)
        {   // nothing to do, avoid invalidating iterators
        }
        else
        {   // trim
            m_last -= (_Oldsize - _Newsize);
        }
    }
private:
    iterator m_first;
    iterator m_last;
    iterator m_end;
};
}// namespace stepik
```

ПРИЛОЖЕНИЕ Б

РЕАЛИЗАЦИЯ КЛАССА LIST

```
#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:
        friend class list<Type>;

        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(node<Type>* p)
        : m_node(p)
        {
        }
```

```
    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);
    }

    list_iterator& operator++()
    {
        m_node = m_node->next;
        return (*this);
    }

    list_iterator operator++(int)
    {
        list_iterator* old_iterator = new list_iterator(*this);
        m_node = m_node->next;
        return (*old_iterator);
    }

private:
    node<Type>* m_node;
};

template <class Type>
class list
```

```
    {
    public:
        typedef list_iterator<Type> iterator;
        typedef Type value_type;
        typedef value_type& reference;
        typedef const value_type& const_reference;

        list() : m_head(nullptr), m_tail(nullptr), m_size(0)
        {
        }

        ~list()
        {
            clear();
        }

        list(const list& other) : m_head(nullptr), m_tail(nullptr),
m_size(0)
        {
            for (node<Type>* Pnode = other.m_head; Pnode; Pnode =
Pnode->next)
                push_back(Pnode->value);
        }

        list(list&& other) : m_head(nullptr), m_tail(nullptr),
m_size(0)
        {
            std::swap(m_head, other.m_head);
            std::swap(m_tail, other.m_tail);
            std::swap(m_size, other.m_size);
        }

        list& operator=(const list& other)
        {
            clear();
            for (node<Type>* Pnode = other.m_head; Pnode; Pnode =
Pnode->next)
                push_back(Pnode->value);
            return (*this);
        }

        list& operator=(list&& other)
        {
            std::swap(m_head, other.m_head);
            std::swap(m_tail, other.m_tail);
            std::swap(m_size, other.m_size);
            return (*this);
        }

        iterator insert(iterator pos, const Type& value)
        {
```

```
            node<Type>* Pnode = pos.m_node;
            if (Pnode == nullptr)
            {
                push_back(value);
                return iterator(m_tail);
            }
            else if (Pnode->prev == nullptr)
            {
                push_front(value);
                return iterator(m_head);
            }
            node<Type>* new_node = new node<Type>(value, Pnode, Pnode-
>prev);
            Pnode->prev->next = new_node;
            Pnode->prev = new_node;
            ++m_size;
            return iterator(new_node);
        }

        iterator erase(iterator pos)
        {
            node<Type>* Pnode = pos.m_node;

            if (Pnode == nullptr)
                return pos;

            if (Pnode->prev == nullptr)
            {
                pop_front();
                return iterator(m_head);
            }

            if (Pnode->next == nullptr)
            {
                pop_back();
                return iterator();
            }

            node<Type>* Pnext = Pnode->next;
            Pnode->next->prev = Pnode->prev;
            Pnode->prev->next = Pnode->next;
            delete Pnode;
            --m_size;
            return iterator(Pnext);
        }

        void push_back(const value_type& value)
        {
            node<Type>* new_node = new node<Type>(value, nullptr,
m_tail);
            if (m_size == 0)
```

```
            {
                m_head = m_tail = new_node;
            }
            else
            {
                m_tail->next = new_node;
                m_tail = new_node;
            }
            ++m_size;
        }

        void push_front(const value_type& value)
        {
            node<Type>* new_node = new node<Type>(value, m_head,
nullptr);
            if (m_size == 0)
            {
                m_head = m_tail = new_node;
            }
            else
            {
                m_head->prev = new_node;
                m_head = new_node;
            }
            ++m_size;
        }

        void pop_front()
        {
            --m_size;
            node<Type>* new_head = m_head->next;
            delete m_head;
            m_head = new_head;
            if (m_size == 0)
            {
                m_tail = nullptr;
            }
            else
            {
                m_head->prev = nullptr;
            }
        }

        void pop_back()
        {
            --m_size;
            node<Type>* new_tail = m_tail->prev;
            delete m_tail;
            m_tail = new_tail;
            if (m_size == 0)
            {
```

```
            m_head = nullptr;
        }
        else
        {
            m_tail->next = nullptr;
        }
    }

    void clear()
    {
        node<Type>* Pnode = m_head;
        node<Type>* Pnext;
        while (Pnode)
        {
            Pnext = Pnode->next;
            delete Pnode;
            Pnode = Pnext;
        }
        m_head = m_tail = nullptr;
        m_size = 0;
    }

    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);
    }

    bool empty() const
    {
        return (!m_size);
    }

    size_t size() const
    {
        return (m_size);
    }
```

```
        iterator begin() const
        {
            return iterator(m_head);
        }

        iterator end() const
        {
            return iterator();
        }

    private:
        node<Type>* m_head;
        node<Type>* m_tail;
        size_t m_size;
    };
}// namespace stepik
```