

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

Студент гр. 7304

Шарапенков И.И.

Преподаватель

Размочаева Н.В.

Санкт-Петербург

2019

Цель работы.

Изучить и реализовать контейнеры вектор и список.

Задача.

Вектор

Необходимо реализовать конструкторы и деструктор для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса `std::vector`. Семантику реализованных функций нужно оставить без изменений. В данном уроке предполагается реализация упрощенной версии вектора, без резервирования памяти под будущие элементы.

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`).

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

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`).

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

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`).

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

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`).

Список

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.

Поведение реализованных функций должно быть таким же, как у класса `std::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`).

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

1. деструктор
2. конструктор копирования,
3. конструктор перемещения,
4. оператор присваивания.

Поведение реализованных функций должно быть таким же, как у класса `std::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`).

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

Поведение реализованных функций должно быть таким же, как у класса `std::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`). 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`. 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`).

Описание результатов.

Вектор

1. Были реализованы конструкторы и деструктор для контейнера вектор.
2. Были реализованы операторы присваивания и функция assign для контейнера вектор.
3. Были реализованы функции resize и erase для контейнера вектор.
4. Были реализованы функции insert и push_back для контейнера вектор.

Список

1. Были реализованы следующий функционал: вставка элементов в голову и хвост, получение элемента из головы и хвоста, удаление из головы, хвоста и очистка, проверка размера.
2. Были реализованы деструктор, конструктор копирования, конструктор перемещения и оператор присваивания.
3. Был реализован итератор для списка, а также операторы =, ==, !=, ++, *, -> для класса итератора.
4. Были реализованы функции вставки и удаления элементов с использованием итераторов.

Вывод.

В ходе выполнения данной лабораторной работы были изучены такие структуры данных как вектор и список. Данные структуры были реализованы средствами C++ с использованием шаблонов классов.

Исходный код

Вектор

```
#include <assert.h>
#include <algorithm> // std::copy, std::rotate
#include <cstddef> // size_t
#include <initializer_list>
#include <stdexcept>
#include <iostream>
#include <vector>

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 <= 0) {
                m_size = 0;
                pointer = nullptr;
            } else {
                m_size = count;
                pointer = new Type[m_size];
            }
            m_first = pointer;
            m_last = pointer + m_size;
        }

        template<typename InputIterator>
        vector(InputIterator first, InputIterator last) {
            size_t size = last - first;
            if (size <= 0) {
                m_size = 0;
                pointer = nullptr;
            } else {
                m_size = size;
                pointer = new Type[m_size];
            }
            for (auto it = first; it != last; it++)
                pointer[it - first] = *it;
            m_first = pointer;
            m_last = pointer + m_size;
        }

        vector(std::initializer_list<Type> init) {
            m_size = init.size();
            pointer = new Type[m_size];
            int i = 0;
            for (auto const &it : init)
                pointer[i++] = it;
            m_first = pointer;
            m_last = pointer + m_size;
        }
    };
}
```

```

    }

    vector(const vector &other) {
        m_size = other.m_size;
        if (m_size == 0)
            pointer = nullptr;
        else
            pointer = new Type[m_size];
        for (auto i = 0; i < m_size; i++)
            pointer[i] = other.pointer[i];
        m_first = pointer;
        m_last = pointer + m_size;
    }

    vector(vector &&other) noexcept {
        m_size = other.m_size;
        pointer = other.pointer;
        m_first = other.m_first;
        m_last = other.m_last;

        other.pointer = nullptr;
        other.m_first = nullptr;
        other.m_last = nullptr;
    }

    ~vector() {
        delete[] pointer;
    }

    //assignment operators
    vector& operator=(const vector& other)
    {
        delete[] pointer;
        m_size = other.m_size;
        if (m_size == 0)
            pointer = nullptr;
        else
            pointer = new Type[m_size];
        for (auto i = 0; i < m_size; i++)
            pointer[i] = other.pointer[i];
        m_first = pointer;
        m_last = pointer + m_size;

        return *this;
    }

    vector& operator=(vector&& other) noexcept
    {
        delete[] pointer;
        m_size = other.m_size;
        pointer = other.pointer;
        m_first = other.m_first;
        m_last = other.m_last;

        other.pointer = nullptr;
        other.m_first = nullptr;
        other.m_last = nullptr;

        return *this;
    }

    // assign method
    template <typename InputIterator>

```



```

void assign(InputIterator first, InputIterator last)
{
    size_t size = last - first;
    delete[] pointer;
    if (size <= 0) {
        m_size = 0;
        pointer = nullptr;
    } else {
        m_size = size;
        pointer = new Type[m_size];
    }
    for (auto it = first; it != last; it++)
        pointer[it - first] = *it;
    m_first = pointer;
    m_last = pointer + m_size;
}

// resize methods
void resize(size_t count)
{
    if(count <= m_size) {
        m_size = count;
        if(count != m_size) delete[] (pointer + m_size);
        m_first = pointer;
        m_last = pointer + m_size;
    } else {
        size_t old_size = m_size;
        m_size = count;
        Type* new_pointer;
        if(m_size <= 0)
            new_pointer = nullptr;
        else
            new_pointer = new Type[m_size];
        for(auto i = 0; i < m_size; i++) {
            if(i >= old_size) new_pointer[i] = 0;
            else new_pointer[i] = pointer[i];
        }
        delete[] pointer;
        pointer = new_pointer;
        m_first = pointer;
        m_last = pointer + m_size;
    }
}

//erase methods
iterator erase(const_iterator pos)
{
    auto position = pos - m_first;
    for(auto i = position; i < m_size - 1; i++)
        pointer[i] = pointer[i+1];
    resize(m_size - 1);
    return m_first + position;
}

iterator erase(const_iterator first, const_iterator last)
{
    auto position = first - m_first;
    for(auto i = position; i < m_size; i++)
        pointer[i] = pointer[i + last - m_first - position];
    resize(m_size - (last - first));
    return m_first + position;
}

```

```

//insert methods
iterator insert(const_iterator pos, const Type& value)
{
    size_t position = pos - m_first;
    resize(m_size + 1);
    for(auto i = m_size - 1; i > position; i--)
        pointer[i] = pointer[i - 1];
    pointer[position] = value;
    return pointer + position;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator
last)
{
    size_t position = pos - m_first;
    resize(m_size + (last - first));
    for(auto i = m_size - (last - first); i + 1 > position; i--)
        pointer[i + last - first - 1] = pointer[i - 1];
    for(auto i = position; i < position + (last - first); i++)
        pointer[i] = *(first + i - position);
    return pointer + position;
}

//push_back methods
void push_back(const value_type& value)
{
    resize(m_size + 1);
    pointer[m_size - 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

private:
    iterator m_first;
    iterator m_last;
    Type *pointer;
    size_t m_size;
};
} // namespace stepik

```

Список

```

#include <assert.h>
#include <algorithm>
#include <stdexcept>
#include <cstdint>
#include <iostream>
#include <list>

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)
    {
        auto tmp = *this;

        m_node = m_node->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;
    typedef list_iterator<Type> iterator;

    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)
    {
        auto current = other.m_head;

        while(current) {
            push_back(current->value);
            current = current->next;
        }
    }

    list(list&& other) noexcept
        : m_head(nullptr), m_tail(nullptr), m_size(0)
    {
        m_size = other.m_size;
        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 current = other.m_head;

        while(current) {
            push_back(current->value);
            current = current->next;
        }

        return *this;
    }
};

```

```

}

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

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

void push_back(const value_type& value)
{
    auto new_node = new node<Type>(value, nullptr, nullptr);

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

    m_size++;
}

void push_front(const value_type& value)
{
    auto new_node = new node<Type>(value, nullptr, nullptr);

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

    m_size++;
}

iterator insert(iterator pos, const Type& value)
{
    auto new_node = new node<Type>(value, nullptr, nullptr);
    auto node = pos.m_node;

    if(node) {
        if(node->prev) {
            node->prev->next = new_node;
            new_node->prev = node->prev;
            new_node->next = node;
            node->prev = new_node;
        } else {
            new_node->prev = node->prev;
            new_node->next = node;
            node->prev = new_node;
            m_head = new_node;
        }
    } else {
        if(empty()) {

```

```

        m_head = new_node;
        m_tail = new_node;
    } else {
        new_node->prev = m_tail;
        m_tail->next = new_node;
        m_tail = new_node;
    }
}

m_size++;
return iterator(new_node);
}

iterator erase(iterator pos)
{
    auto node = pos.m_node;

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

    m_size--;
    auto next = node->next;
    return iterator(next);
}

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()
{
    node<Type> *tmp = m_head;

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

```

```

    }

    void pop_back()
    {
        node<Type> *tmp = m_tail;

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

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

    bool empty() const
    {
        return !m_head;
    }

    size_t size() const
    {
        return m_size;
    }

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

} // namespace stepik

```