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. 7382 _____ Филиппов И.С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Программирование собственных реализаций STL контейнеров vector и list.

Основные теоретические положения.

Необходимо реализовать конструкторы и деструктор 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 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

Необходимо реализовать список со следующими функциями: вставка элементов в голову и в хвост, получение элемента из головы и из хвоста, удаление из головы, хвоста и очистка проверка размера. Поведение реализованных функций должно быть таким же, как у класса std::list. Семантику реализованных функций нужно оставить без изменений.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции: деструктор конструктор копирования, конструктор перемещения, оператор присваивания.

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: 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).

Выводы.

В ходе выполнения лабораторной работы были реализованны собственные имплементации STL классов vector и list.

ПРИЛОЖЕНИЕ

Исходный код

vector.h

```
#include <assert.h>
#include <algorithm>
#include <cstddef>
#include <initializer_list>
#include <stdexcept>
#include <iostream>

#include <vector>
#include <bits/streambuf_iterator.h>

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)
      {
         allocate(count);
      }

      template <typename InputIterator>
      vector(InputIterator first, InputIterator last)
      {
         allocate(last - first);

         std::copy(first, last, m_first);
      }

      vector(std::initializer_list<Type> init)
      {
         allocate(init.size());

         std::copy(init.begin(), init.end(), m_first);
      }

      vector(const vector& other)
      {
```

```
    allocate(other.size());

    std::copy(other.begin(), other.end(), m_first);
}

vector(vector&& other) noexcept
{
    //allocate(other.size());

    m_first = m_last = nullptr;
    std::swap(m_first, other.m_first);
    std::swap(m_last, other.m_last);
}

~vector()
{
    delete_data();
}

/*vector& operator=(const vector& other)
{
    delete_data();

    allocate(other.size());

    std::copy(other.begin(), other.end(), m_first);
}

vector& operator=(vector&& other) noexcept
{
    delete_data();

    m_first = m_last = nullptr;
    std::swap(m_first, other.m_first);
    std::swap(m_last, other.m_last);
}*/

vector& operator=(vector other)
{
    delete_data();
    m_first = m_last = nullptr;

    std::swap(m_first, other.m_first);
    std::swap(m_last, other.m_last);
}

template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    delete_data();

    allocate(last - first);
```

```
            std::copy(first, last, m_first);
        }

        // resize methods
        void resize(size_t count)
        {
            Type* new_data = new Type[count]();

            if (count > m_last - m_first) // Увеличили
            {
                std::copy(m_first, m_last, new_data);
            }
            else // Уменьшили
            {
                std::copy(m_first, m_first + count, new_data);
            }

            delete_data();

            m_first = new_data;
            m_last = new_data + count;
        }

        //erase methods
        iterator erase(const_iterator pos)
        {
            auto new_vector_size = m_last - m_first - 1;
            Type* new_data = new Type[new_vector_size]();

            auto to_deleted_element = pos - m_first;
            std::copy(m_first, m_first + to_deleted_element, new_data);
            std::copy(m_first + to_deleted_element + 1, m_last, new_data +
to_deleted_element);

            delete_data();

            m_first = new_data;
            m_last = new_data + new_vector_size;

            return m_first + to_deleted_element;
        }

        iterator erase(const_iterator first, const_iterator last)
        {
            if (first == last)
                return m_first + (last - m_first);

            auto new_vector_size = m_last - m_first - (last - first);
            Type* new_data = new Type[new_vector_size]();

            auto num_before_deletion = first - m_first;
```

```
            std::copy(m_first, m_first + num_before_deletion, new_data);
            auto to_first_after_delete = last - m_first;
            std::copy(m_first + to_first_after_delete, m_last, new_data + num_before_deletion);

            delete_data();

            m_first = new_data;
            m_last = new_data + new_vector_size;

            return m_first + num_before_deletion;
        }

        //insert methods
        iterator insert(const_iterator pos, const Type& value)
        {
            auto new_vector_size = m_last - m_first + 1;
            Type* new_data = new Type[new_vector_size]();

            auto to_enserted_element = (pos - m_first);
            std::copy(m_first, m_first + to_enserted_element, new_data);
            new_data[to_enserted_element] = value;
            std::copy(m_first + to_enserted_element, m_last, new_data + to_enserted_element +
1);

            delete_data();

            m_first = new_data;
            m_last = new_data + new_vector_size;

            return m_first + to_enserted_element;
        }

        template <typename InputIterator>
        iterator insert(const_iterator pos, InputIterator first, InputIterator last)
        {
            if (first == last)
            {
                return m_first + (pos - m_first);
            }

            auto num_new_elements = last - first;

            auto new_vector_size = num_new_elements + m_last - m_first;
            Type* new_data = new Type[new_vector_size]();

            auto first_new_element = pos - m_first;
            std::copy(m_first, m_first + first_new_element, new_data);
            std::copy(first, last, new_data + first_new_element);
            std::copy(m_first + first_new_element, m_last, new_data + first_new_element +
num_new_elements);

            delete_data();
```

```
    m_first = new_data;
    m_last = new_data + new_vector_size;

    return m_first + first_new_element;
}

//push_back methods
void push_back(const value_type& value)
{
    auto new_vector_size = m_last - m_first + 1;
    Type* new_data = new Type[new_vector_size]();

    std::copy(m_first, m_last, new_data);

    new_data[new_vector_size - 1] = value;

    delete_data();

    m_first = new_data;
    m_last = new_data + new_vector_size;
}

//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];
    }

    void allocate(size_t memory_size)
    {
        Type* data = memory_size ? new Type[memory_size]() : nullptr;

        m_first = data;
        m_last = data + memory_size;
    }

    void delete_data()
    {
        delete[] m_first;
    }

private:
```

```
        iterator m_first;
        iterator m_last;
    };
}// namespace
```

list.h

```
#include <iostream>
#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 Value>
    class list_iterator
    {
    public:
        friend class list<Value>;

        typedef ptrdiff_t difference_type;
        typedef Value value_type;
        typedef Value* pointer;
        typedef Value& 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(node<Value>* p) : m_node(p)
        {}
```

```
    list_iterator& operator = (const list_iterator& other)
    {
        if (*this != 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 !(*this == other);
    }

    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 = m_node;
        m_node = m_node->next;

        return tmp;
    }

private:
    node<Value>* 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)
{}

~list()
{
   clear();
}

list(const list& other) : m_head(nullptr), m_tail(nullptr)
{
   if (!other.empty())
   {
      for (auto it = other.m_head; it != other.m_tail; it = it->next)
      {
         push_back(it->value);
      }
      push_back(other.m_tail->value); // For tail
   }
}

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

iterator insert(iterator pos, const Type& value)
{
   auto& pos_node = pos.m_node;

   if (pos_node == nullptr)
   {
      push_back(value);

      return iterator(m_tail);
   }

   if (pos_node->prev == nullptr)
   {
      push_front(value);
      return iterator(m_head);
   }

   auto new_node = new node<Type>(value, pos_node, pos_node->prev);

   pos_node->prev->next = new_node;
   pos_node->prev = new_node;

   return iterator(new_node);
```

```
}

iterator erase(iterator pos)
{
   auto& pos_node = pos.m_node;

   if (pos_node == nullptr)
      return pos;

   if (pos_node->next == nullptr)
   {
      pop_back();

      return nullptr;
   }

   if (pos_node->prev == nullptr)
   {
      pop_front();

      return iterator(m_head);
   }

   auto ret = pos_node->next;
   pos_node->prev->next = pos_node->next;
   pos_node->next->prev = pos_node->prev;

   delete pos_node;

   return ret;
}

void push_back(const value_type& value)
{
   if (!empty())
   {
      auto old_tail = m_tail;
      m_tail = new node<value_type>(value, nullptr, nullptr);

      m_tail->prev = old_tail;
      old_tail->next = m_tail;
   }
   else
   {
      create_first_node(value);
   }
}

void push_front(const value_type& value)
{
   if (!empty())
   {
```

```
        m_head->prev = new node<value_type>(value, m_head, nullptr);

        m_head = m_head->prev;
    }
    else
    {
        create_first_node(value);
    }
}

void pop_front()
{
    if (only_one_node())
    {
        delete_last_node();
    }
    else
    {
        auto head_to_del = m_head;
        m_head = m_head->next;
        m_head->prev = nullptr;

        delete head_to_del;
    }
}

void pop_back()
{
    if (only_one_node())
    {
        delete_last_node();
    }
    else
    {
        auto tail_to_del = m_tail;
        m_tail = m_tail->prev;
        m_tail->next = nullptr;

        delete tail_to_del;
    }
}

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

    m_head = m_tail = nullptr;
}
```

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

    size_t size() const
    {
        if (empty())
            return 0;

        size_t size = 0;
        for (auto it = m_head; it != m_tail; it = it->next)
        {
            size++;
        }
        size++; // For tail

        return size;
    }

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

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

private:
    node<Type>* m_head;
    node<Type>* m_tail;
private:
    void create_first_node(const value_type& value)
    {
        m_head = new node<value_type>(value, nullptr, nullptr);
        m_tail = m_head;
    }

    void delete_last_node()
    {
        delete m_head;
        m_head = m_tail = nullptr;
    }

    bool only_one_node()
    {
        return m_head == m_tail && m_head != nullptr;
    }
};
```

```
}// namespace stepik
```