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 Субботин А.С.

Преподаватель Размочаева Н.В.

Санкт-Петербург
2019

Цель работы

Изучить реализацию контейнеров в языке программирования C++

Формулировка задачи

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

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 списка, проверка размера, деструктор, конструкторы копирования и перемещения, оператор присваивания, insert, erase, а также итераторы для списка: =, ==, !=, ++, *, ->. Поведение реализованных функций должно быть таким же, как у класса std::list.

Ход работы

- Написаны методы классов, соответствующие заданным требованиям.

– Для вектора:

1) Создан вектор из пяти элементов, он проинициализирован по порядку числами от 0 до 4.
2) Удалён последний элемент.
3) В конец добавлен элемент со значением 585.
4) С помощью конструктора копирования создаётся второй вектор, аналогичный первому.
5) Создаётся третий вектор из 1 элемента. С помощью оператора присваивания он становится подобным первому вектору.
6) Второй вектор вставляется в вектор 1, начиная со второго элемента.

Результаты работы программы:

step 1: 0 1 2 3 4

step 2: 0 1 2 3

step 3: 0 1 2 3 585

step 4: 0 1 2 3 585

step 5: 0 1 2 3 585

step 6: 0 0 1 2 3 585 1 2 3 585

– Для списка:

1) Создан список.
2) В конец подряд добавлены числа 3, 6, 9. В начало – 2 и 1.
3) С помощью конструктора копирования создаётся второй список, из конца списка удаляется элемент.

4) Удаляются все элементы первого списка. Вставка в хвост элемента 0.

5) Сравнение итераторов, указывающих на начала первого и второго списков.

6) В первый список происходит вставка элемента 7 с помощью префиксного итератора ++.

Результаты работы программы:

```
step 1:
step 2: 1 2 3 6 9
step 3: 1 2 3 6
step 4: 0
step 5: a.begin()  != c.begin()
step 6: 0 7
```

Выводы

В ходе выполнения данной лабораторной работы была изучена реализация умного таикх контейнеров, как вектор и список, были реализованы основные методы для работы с этими контейнерами.

Приложение А. Код программы (вектор)

```
#include <iostream>
#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) : m_first(new Type[last -
first]), m_last(m_first + (last - first))
    {
      std::copy(first, last, m_first);
    }

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

    vector(const vector& other) : m_first(new Type[other.size()]),
m_last(&(m_first[other.size()]))
    {
      std::copy(other.m_first, other.m_last, m_first);
    }

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

    ~vector()
    {
      delete[] m_first;
      m_first = nullptr;
      m_last = nullptr;
    }

    vector& operator=(const vector& other)
    {
      if(this != &other){
        vector temp(other);
        std::swap(m_first, temp.m_first);
        std::swap(m_last, temp.m_last);
      }
      return *this;
    }
```

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

template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
  vector temp(first, last);
  std::swap(m_first, temp.m_first);
  std::swap(m_last, temp.m_last);
}

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

iterator erase(const_iterator pos)
{
  difference_type index = pos - m_first;
  std::rotate(m_first + index, m_first + index + 1, m_last);
  resize(size() - 1);
  return m_first + index;
}

iterator erase(const_iterator first, const_iterator last)
{
  difference_type index = first - m_first;
  difference_type delta = last - first;
  std::rotate(m_first + index, m_first + index + delta, m_last);
  resize(size() - delta);
  return m_first + index;
}

iterator insert(const_iterator pos, const Type& value)
{
  difference_type index = pos - m_first;
  resize(size() + 1);
  std::rotate(m_first + index, m_last - 1, m_last);
  m_first[index] = value;
  return m_first + index;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
  difference_type index = pos - m_first;
  resize(size() + (last - first));
  std::copy(first, last, m_last - (last - first));
  std::rotate(m_first + index, m_last - (last - first) , m_last);
  return m_first + index;
}
```

```
  void push_back(const value_type& value)
  {
    insert(m_last, 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];
    }

    friend void operator << (std::ostream& os, vector& v)
    {
        for(size_t i(0); i < v.size(); i++)
            os << v.m_first[i] << " ";
        os << std::endl;
    }

    iterator m_first;
    iterator m_last;
  };
}

using namespace stepik;
using std::cout;

int main(){
    vector<int> vec1(5);
        vec1[0] = 0;
        vec1[1] = 1;
        vec1[2] = 2;
        vec1[3] = 3;
        vec1[4] = 4;
        cout << "step 1: " << vec1;
        vec1.erase(vec1.end());
        cout << "step 2: " << vec1;
        vec1.push_back(585);
        cout << "step 3: " << vec1;

        vector<int> vec2(vec1);
        cout << "step 4: " << vec2;
        vector<int> vec3(1);
        vec3 = vec1;
        cout << "step 5: " << vec3;

        vec1.insert(vec1.begin() + 1, vec2.begin(), vec2.end());
        cout << "step 6: " << vec1;

    return 0;
}
```

Приложение В. Код программы (список)

```
#include <iostream>
#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;

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()
  {
    clear();
  }

  list(const list& other) : list()
  {
    node<Type>* temp = other.m_head;
    while(temp){
      push_back(temp->value);
      temp = temp->next;
    }
  }

  list(list&& other) : list()
  {
    if(this != &other){
      std::swap(m_head, other.m_head);
      std::swap(m_tail, other.m_tail);
    }
  }

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

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

```
    }

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

    iterator insert(iterator pos, const Type& value)
    {
        if(pos.m_node == nullptr){
            push_back(value);
            return iterator(m_tail);
        }
        if(pos.m_node == m_head){
            push_front(value);
            return iterator(m_head);
        }
        else{
            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 nullptr;
        }
        if(pos.m_node == m_head)
        {
            pop_front();
            return iterator(m_head);
        }
        if(pos.m_node == m_tail)
        {
            pop_back();
            return iterator(m_tail);
        }
        else
        {
            pos.m_node->prev->next = pos.m_node->next;
            pos.m_node->next->prev = pos.m_node->prev;
            node<Type>* temp = pos.m_node;
            iterator result(pos.m_node->next);
            delete temp;
            return result;
        }
    }
```

```
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(size() == 1)
    {
      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(size() == 1)
    {
      m_head = nullptr;
      m_tail = nullptr;
    }
    else
    {
      m_tail = m_tail->prev;
      delete m_tail->next;
      m_tail->next = nullptr;
    }
  }
}

void clear()
{
  if(!m_head)
    return;
  while(m_head != m_tail){
    m_head = m_head->next;
```

```
        delete m_head->prev;
        m_head->prev = nullptr;
      }
      delete m_head;
      m_head = nullptr;
      m_tail = nullptr;
    }

    size_t size() const
    {
      node<Type>* temp = m_head;
      size_t count = 0;
      while(temp){
        count++;
        temp = temp->next;
      }
      return count;
    }

    bool empty() const
    {
      return m_head ? false:true;
    }

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

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

  private:
    friend void operator << (std::ostream& os, list& element)
    {
            node<Type>* temp = element.m_head;
            while(temp)
            {
                os << temp->value << " ";
                temp = temp->next;
            }
            os << std::endl;
    }

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

}




using namespace stepik;
using namespace std;

int main()
{
```

```
    list<int> a;
        cout << "step 1: " << a;
        a.push_back(3);
        a.push_back(6);
        a.push_back(9);
        a.push_front(2);
        a.push_front(1);
        cout << "step 2: " << a;



        list<int> b(a);
        b.pop_back();
        cout << "step 3: " << b;
        a.clear();
        a.push_back(0);
        cout << "step 4: " << a;

        if(a.begin() == b.begin())
            cout << "step 5: a.begin() == c.begin()" << endl;
        else
            cout << "step 5: a.begin() != c.begin()" << endl;
        a.insert(++a.begin(), 7);
        cout << "step 6: " << a;
    return 0;
}
```