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. Список»

Студентка гр. 7382 _____ Лящевская А.П.

Преподаватель _____ Жангиров Т.М.

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 А.

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 vector.h.

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

    template <typename InputIterator>
    vector(InputIterator first, InputIterator last)
    {
        size_t dis = std::distance(first, last);
```

```
    if(all_mem(dis))
    {
        m_last += dis;
        std::copy(first, last, m_first);
    }
}

vector(std::initializer_list<Type> init)
{
    if(all_mem(init.size()))
    {
        m_last += init.size();
        std::copy(init.begin(), init.end(), m_first);
    }
}

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

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

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

```
//assignment operators
vector& operator=(const vector& other)
{
    if(m_first == other.m_first && size() == other.size())
        return *this;
    if(m_first != m_last)
    {
        delete [] m_first;
        m_last = m_first;
    }
    if(all_mem(other.size()))
    {
        m_last += other.size();
        std::copy(other.begin(), other.end(), m_first);
    }
    return *this;
}

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

// assign method
template <typename InputIterator>
void assign(InputIterator first, InputIterator last)
{
    if(m_first != m_last)
    {
        delete [] m_first;
```

```
        m_last = m_first;
    }
    size_t size = std::distance(first, last);
    if(all_mem(size))
    {
        m_last += size;
        std::copy(first, last, m_first);
    }
}

// resize methods
void resize(size_t count)
{
    size_t old_size = size();
    if(count > old_size)
    {
        iterator new_vec = new Type[count];
        std::copy(m_first, m_last, new_vec);
        if(m_first != m_last)
        {
            delete [] m_first;
        }
        m_first = new_vec;
        m_last = new_vec + count;
    }
    else
    {
        m_last -= (old_size - count);
    }
}


//erase methods
iterator erase(const_iterator pos)
{
    iterator p = iterator(pos);
    std::rotate(p, p + 1, m_last);
```

```
    resize(size() - 1);
    return p;
}

iterator erase(const_iterator first, const_iterator last)
{
    difference_type dist = std::distance(first, last);
    std::rotate(iterator(first), iterator(last), m_last);
    resize(size() - dist);
    return iterator(first);

}

//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:
   bool all_mem(size_t N)
  {
      m_first = m_last = iterator();
      if(!N) return (false);
```

```
            m_first = new Type[N];
            m_last = m_first;
            return (true);
        }

        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;
      };
}// namespace stepik
```

Приложение Б. Файл list.h.

```
#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* old = new list_iterator(*this);
      m_node = m_node->next;
      return *old;
  }

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)
    : m_head(), m_tail()
{
    for(node<value_type>* tmp = other.m_head; tmp; tmp = tmp->next)
        push_back(tmp->value);
}

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

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

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

list::iterator end()
```

```
{
  return iterator();
}

void push_back(const value_type& value)
{
    node<value_type>* new_node = new node<value_type>(value, nullptr, m_tail);
    if(!m_head) m_head = new_node;
    else m_tail->next = new_node;
    m_tail = new_node;
}

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

void pop_front()
{
    if(!m_head) return;
    node<value_type>* new_head = m_head->next;
    delete m_head;
    if(!new_head)
        m_tail = new_head;
    else
        new_head->prev = nullptr;
    m_head = new_head;
}
void pop_back()
{
    if(!m_tail) return;
    node<value_type>* new_tail = m_tail->prev;
    delete m_tail;
    if(!new_tail)
```

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

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

```
    }
    pos_node->prev->next = pos_node->next;
    pos_node->next->prev = pos_node->prev;
    iterator ret = iterator(pos_node->next);
    delete pos_node;
    return ret;
}

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_head;
}

size_t size() const
{
    size_t size = 0;
    node<value_type>* tmp = m_head;
```

```
        while(tmp)
        {
            tmp = tmp->next;
            size++;

        }
        return size;
    }
  private:
    void clear()
    {
        while (m_head)
            pop_back();
    }

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

}// namespace stepik
```