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 _____ Овчинников Н.В.

Преподаватель _____ Размочаева Н.В

Санкт-Петербург

2019

Цель работы

Изучить реализацию контейнеров vector и list на языке программирования C++.

Задание

Реализовать конструкторы, деструктор, оператор присваивания, функцию assign, функции resize и erase, функции insert и push_back для контейнера vector. Поведение реализованных функций должно быть таким же, как у класса std::vector.

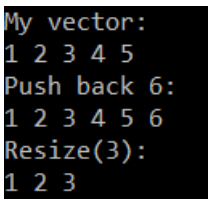
Реализовать список со следующими функциями: вставка элементов в голову и в хвост, получение элемента из головы и из хвоста, удаление из головы, хвоста, проверка размера. Так же необходимо реализовать деструктор, конструктор копирования, оператор присваивания, оператор перемещения, операторы =, ==, !=, ++, *, ->. Реализовать вставку элементов и удаление элементов. Поведение реализованных функций должно быть таким же, как у класса std::list.

Ход работы

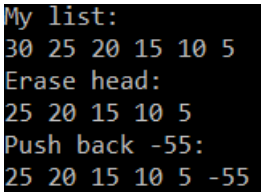
1. Реализация контейнера vector. Все функции данного контейнера были реализованы в соответствии с поведением класса std::vector.
    a. На первом шаге были реализованы конструкторы и деструктор для контейнера vector.
    b. На втором шаге были реализованы операторы присваивания и функция assign.
    c. На третьем шаге были реализованы функции resize и erase.
    d. На четвертом шаге были реализованы функции insert и push_back.
2. Реализация контейнера list. Все функции данного контейнера были реализованы в соответствии с поведением класса std::list.
    a. На первом шаге были реализованы функции вставки элементов в голову и хвост, получение элемента из головы и из хвоста, удаления из головы и хвоста, проверки размера.
    b. На втором шаге были реализованы деструктор, конструктор копирования, оператор присваивания, оператор перемещения.
    c. На третьем шаге были реализованы операторы =, ==, !=, ++, *, ->.
    d. На четвертом шаге были реализованы функции вставки и удаление элемента.

Пример работы программы

1. Был создан vector с пятью последовательными элементами типа int. Вызван метод push_back для 6 и метод изменения размера resize для трёх.



2. Был создан лист с шестью элементами (30, 25, 20, 15, 10, 5), удалена голова списка методом pop_front и добавлено число -55 в конец списка методом push_back.



Вывод

В ходе выполнения лабораторной работы были изучены и реализованы такие контейнеры, как вектор и список. Для заданных контейнеров были реализованы основные функции для работы с ними. Поведение реализованных функций соответствует классам std::vector и std::list.

Приложение №1. Реализация классов

1. Класс vector.

```
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)
  {
      count ? m_first = new Type[count] : m_first = nullptr;
      count ? m_last = m_first + count : m_last = nullptr;
  }

  template <typename InputIterator>
  vector(InputIterator first, InputIterator last) : vector(last-first)
  {
      std::copy(first, last, m_first);
  }

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

  vector(const vector& other) : vector(other.size())
  {
      std::copy(other.begin(), other.end(), m_first);
  }

  vector(vector&& other) : vector(other.size())
  {
      std::copy(other.begin(), other.end(), m_first);
      other.clear();
  }

  ~vector()
  {
      this->clear();
  }

  // resize methods
  void resize(size_t count)
  {
      if(count == this->size())
          return;
      if(count == 0)
      {
          this->clear();
          return;
      }
      if(count > this->size())
      {
          iterator tmp = new Type[count];
          for(size_t i=0; i<size(); i++)
          {
              tmp[i] = *(m_first+i);
```

```
        }
        clear();
        m_first = tmp;
        m_last = tmp + count;
    }
    else
    {
        iterator tmp = new Type[count];
        for(size_t i = 0; i < count; i++)
        {
             tmp[i] = m_first[i];
        }
        clear();
        m_first = tmp;
        m_last = m_first + count;
    }
}

//insert methods
iterator insert(const_iterator pos, const Type& value)
{
    size_t i = 0;
    while(m_first + i != pos) i++;
    resize(size() + 1);
    *(m_last - 1) = value;
    std::rotate(m_first+ i, m_last-1, m_last);
    return m_first + i;
}

template <typename InputIterator>
iterator insert(const_iterator pos, InputIterator first, InputIterator last)
{
    size_t i = 0, digit = last - first;
    while(m_first + i != pos) i++;
    resize(size() + digit);
    for(size_t tmp = 0; tmp < digit; tmp++)
        *(m_last - digit + tmp) = *(first + tmp);
    for(size_t tmp = 0; tmp < digit; tmp++)
        std::rotate(m_first + i, m_last -1, m_last);
    return m_first + i;
}

//push_back methods
void push_back(const value_type& value)
{
    insert(m_last, (Type)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
  void clear()
  {
       if(m_first)
       {
            delete [] m_first;
            m_first = nullptr;
            m_last = nullptr;
        }
   }

private:
  iterator m_first;
  iterator m_last;
};
```

2. Класс list.

```
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)
  {
      this->m_node = other.m_node;
  }

  bool operator == (const list_iterator& other) const
  {
      return this->m_node == other.m_node;
  }

  bool operator != (const list_iterator& other) const
  {
      return this->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)
  {
      node<Type> *tmp = m_node;
      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)
  {
  }

  ~list()
  {
      clear();
  }

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

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

  reference front()
  {
      if(m_head != nullptr)
          return this->m_head->value;
  }

  const_reference front() const
  {
      if(m_head != nullptr)
          return this->m_head->value;
  }
```

```
reference back()
{
    if(m_tail != nullptr)
        return this->m_tail->value;
}

const_reference back() const
{
    if(m_tail != nullptr)
        return this->m_tail->value;
}

void pop_front()
{
    if(m_tail)
    {
        if(m_head == m_tail)
        {
            delete m_head;
            m_head = m_tail = nullptr;
            return;
        }
        m_head = m_head->next;
        delete m_head->prev;
        m_head->prev->prev = m_head->prev->next = nullptr;
        m_head->prev = nullptr;
    }
}

void pop_back()
{
    if(m_tail)
    {
        if(m_head == m_tail)
        {
            delete m_head;
            m_head = m_tail = nullptr;
            return;
        }
        m_tail = m_tail->prev;
        delete m_tail->next;
        m_tail->next->next = m_tail->next->prev = nullptr;
        m_tail->next = nullptr;
    }
}

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

bool empty() const
{
    if(m_head == nullptr)
        return true;
    return false;
}

size_t size() const
{
    size_t i = 0;
    node<Type> *tmp = m_head;
    while(tmp != nullptr)
    {
        i++;
        tmp = tmp->next;
    }
```

```
      return i;
  }

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

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

  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);
      }

      node<Type> *tmp = pos.m_node;
      node<Type> *ins = new node<Type>(value, tmp, tmp->prev);
      tmp->prev->next = ins;
      tmp->prev = ins;
      return iterator(ins);
  }

  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();
      }

      node<Type> *tmp = pos.m_node;
      iterator it = iterator(pos.m_node->next);
      tmp->prev->next = tmp->next;
      tmp->next->prev = tmp->prev;
      delete tmp;
      tmp = nullptr;
      return it;
  }

private:
  //your private functions

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