**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 С++.

**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