**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**по практической работе №3**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема**: **Контейнеры. Вектор. Список**

|  |  |  |
| --- | --- | --- |
| Студент гр. 7382 |  | Дрозд А.С. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2019

**Цель работы.**

Изучить стандартные контейнеры vector и list языка С++.

**Постановка задачи.**

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 В.

**Выводы.**

В ходе выполнения лабораторной работы была изучена реализация контейнеров vector и list.

**ПРИЛОЖЕНИЕ А**

Исходный код list.cpp

#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\_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;

}

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 retval(m\_node);

m\_node=m\_node->next;

return retval;

}

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;

list(): m\_head(nullptr), m\_tail(nullptr)

{

copies=new int;

\*copies=1;

}

~list()

{

\*copies--;

if(\*copies==0)

clear();

}

void copy\_list\_surface(const list &other)

:m\_head(other.m\_head), m\_tail(other.m\_tail), copies(other.copies)

{

m\_head=other.m\_head;

m\_tail=other.m\_tail;

copies=other.copies;

\*copies++;

/\* node<Type>\* element = other.m\_head;

while(element!= NULL)

{

push\_back(element->value);

element = element->next;

}

\*/ }

list(const list &other)

:m\_head(nullptr), m\_tail(nullptr)

{

node<Type>\* element = other.m\_head;

while(element!= NULL)

{

push\_back(element->value);

element = element->next;

}

}

list(list &&other)

:m\_head(other.m\_head), m\_tail(other.m\_tail)

{

other.m\_head=NULL;

other.m\_tail=NULL;

}

void push\_back(const value\_type &value)

{

if(empty())

{

m\_tail=new node<Type>(value,NULL,NULL);

m\_head=m\_tail;

}

else

{

m\_tail->next=new node<Type>(value,NULL,m\_tail);

m\_tail=m\_tail->next;

}

}

void push\_front(const value\_type &value)

{

if(empty())

{

m\_tail=new node<Type>(value,NULL,NULL);

m\_head=m\_tail;

}

else

{

m\_head->prev=new node<Type>(value,m\_head,NULL);

m\_head=m\_head->prev;

}

}

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(m\_head==m\_tail)

{

delete(m\_head);

m\_head=m\_tail=NULL;

}

else

{

m\_head=m\_head->next;

delete(m\_head->prev);

m\_head->prev=NULL;

}

}

}

void pop\_back()

{

if(!empty())

{

if(m\_head == m\_tail)

{

delete(m\_head);

m\_head=m\_tail=NULL;

}

else

{

m\_tail=m\_tail->prev;

delete(m\_tail->next);

m\_tail->next=NULL;

}

}

}

void clear()

{

node<Type>\* elem;

node<Type>\* element=m\_head;

while(element!=NULL)

{

elem=element;

element=element->next;

delete elem;

}

m\_head=NULL;

m\_tail=NULL;

}

bool empty() const

{

return m\_head==NULL;

}

list& operator=(const list &other)

{

if (this!=&other)

{

clear();

node<Type>\* element=other.m\_head;

while(element!=NULL)

{

push\_back(element->value);

element=element->next;

}

}

return \*this;

}

size\_t size() const

{

size\_t size\_ = 0;

node<Type> \*element = m\_head;

while(element!=NULL)

{

element=element->next;

size\_++;

}

return size\_;

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

iterator insert(iterator pos, const Type& value)

{

if(pos.m\_node==NULL || pos.m\_node==m\_head)

push\_front(value);

else

{

node<Type>\* insertElem=new node<Type>(value,pos.m\_node,pos.m\_node->prev);

pos.m\_node->prev->next=insertElem;

pos.m\_node->prev=insertElem;

}

return iterator(pos.m\_node->prev);

}

iterator erase(iterator pos)

{

iterator ret = pos.m\_node->next;

if(pos.m\_node == m\_head)

pop\_front();

else

{

if (pos.m\_node == m\_tail)

{

pop\_back();

ret = m\_tail;

}

else

{

pos.m\_node->prev->next = pos.m\_node->next;

pos.m\_node->next->prev = pos.m\_node->prev;

}

}

return ret;

}

private:

//your private functions

node<Type>\* m\_head;

node<Type>\* m\_tail;

int \* copies;

};

}// namespace stepik

**ПРИЛОЖЕНИЕ Б**

Исходный код программы vector.cpp

#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(count ? new value\_type[count] : nullptr),

m\_last(m\_first ? m\_first+count : nullptr)

{

copies=new int;

\*copies=1;

}

template <typename InputIterator>

vector(InputIterator first,InputIterator last)

: vector(last-first)

{

if (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=other.m\_first),(m\_last=other.m\_last)

{

\*copies++;

}

vector(vector &&other)

{

m\_first=other.begin();

m\_last=other.end();

other.m\_first=nullptr;

other.m\_last=nullptr;

}

~vector()

{

\*copies--;

if(\*copies==0)

{

delete[] m\_first;

m\_last=nullptr;

m\_first=nullptr;

}

}

vector& operator=(const vector& other)

{

assign(other.m\_first, other.m\_last);

}

vector& operator=(vector&& other)

{

if(m\_first)

delete[] m\_first;

m\_first=other.m\_first;

m\_last=other.m\_last;

other.m\_first=nullptr;

other.m\_last=nullptr;

return \*this;

}

void assign(InputIterator first, InputIterator last)

{

vector a(first, last);

(\*this)=std::move(a);

}

void resize(size\_t count)

{

int size\_vec=size();

if(count!=size\_vec)

{

if(count<size\_vec)

m\_last=m\_first+count;

else

{

vector a(count);

std::copy(m\_first,m\_last,a.m\_first);

(\*this)=std::move(a);

}

}

}

iterator erase(const\_iterator pos)

{

iterator element=m\_first+(pos-m\_first);

std::rotate(element,element+1,m\_last);

m\_last--;

return element;

}

iterator erase(const\_iterator first, const\_iterator last)

{

iterator begin=m\_first+(first-m\_first);

iterator end=m\_first+(last-m\_first);

std::rotate(begin,end,m\_last);

m\_last=m\_last-(last-first);

return begin;

}

//insert methods

iterator insert(const\_iterator pos, const Type& value)

{

size\_t index=pos-m\_first;

resize(size()+1);

m\_first[size()-1]=value;

std::rotate((iterator)(m\_first + index),m\_last,m\_last);

return m\_first+index;

}

template <typename InputIterator>

iterator insert(const\_iterator pos, InputIterator first, InputIterator last)

{

size\_t index=pos-m\_first;

size\_t old\_size=size();

resize(size()+last-first);

std::copy(first,last,m\_first+old\_size);

std::rotate((iterator)(m\_first+index),(iterator)(m\_first+old\_size),m\_last);

return m\_first+index;

}

//push\_back methods

void push\_back(const value\_type& value)

{

insert(m\_last,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

private:

iterator m\_first;

iterator m\_last;

int\* copies;

};

}// namespace stepik