**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**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Еременко А.А. |
| 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 vector.h

#include <assert.h>

#include <algorithm> // std::copy, std::rotate

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

using namespace std;

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] : nullptrptr),

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

{

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last) : vector(last - first)

{

if (last - first)

copy(first, last, m\_first);

}

vector(std::initializer\_list<Type> init) : vector(init.begin(), init.end())

{

}

vector(const vector &other) : vector(other.begin(), other.end())

{

}

vector(vector &&other)

{

m\_first = other.begin();

m\_last = other.end();

other.m\_first = other.m\_last = nullptr;

}

~vector()

{

delete[] m\_first;

m\_last = m\_first = nullptr;

}

//assignment operators

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 = nullptrptr;

other.m\_last = nullptrptr;

return \*this;

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

vector a(first, last);

(\*this) = move(a);

}

// resize methods

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);

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

(\*this) = move(a);

}

}

}

//erase methods

iterator erase(const\_iterator pos)

{

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

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);

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;

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;

};

} // namespace stepik

**ПРИЛОЖЕНИЕ Б**

Исходный код программы list.h

#include <assert.h>

#include <algorithm>

#include <stdexcept>

#include <cstddef>

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)

{

if (this != &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->prev;

return \*this;

}

list\_iterator operator--(int)

{

m\_node = m\_node->prev;

return m\_node->next;

}

list\_iterator &operator++()

{

m\_node = m\_node->next;

return \*this;

}

list\_iterator operator++(int)

{

m\_node = m\_node->next;

return m\_node->prev;

}

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)

{

}

~list()

{

clear();

}

list(const list &other) : list()

{

auto cur = other.m\_head;

while (cur != nullptr)

{

push\_back(cur->value);

cur = cur->next;

}

}

list(list &&other)

{

m\_head = other.m\_head;

m\_tail = other.m\_tail;

other.m\_head = nullptr;

other.m\_tail = nullptr;

}

list &operator=(const list &other)

{

if (this != &other)

{

clear();

for (node<Type> \*element = other.m\_head; element != NULL; push\_back(element->value), element = element->next)

;

}

return \*this;

}

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

{

if (empty())

{

m\_head = new node<value\_type>(value, nullptr, nullptr);

m\_tail = m\_head;

return;

}

m\_tail->next = new node<value\_type>(value, nullptr, m\_tail);

m\_tail = m\_tail->next;

}

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

{

if (empty())

{

m\_head = new node<value\_type>(value, nullptr, nullptr);

m\_tail = m\_head;

return;

}

m\_head->prev = new node<value\_type>(value, m\_head, nullptr);

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 = nullptr;

}

else

{

m\_head = m\_head->next;

delete (m\_head->prev);

m\_head->prev = nullptr;

}

}

}

void pop\_back()

{

if (!empty())

{

if (m\_head == m\_tail)

{

delete (m\_head);

m\_head = m\_tail = nullptr;

}

else

{

m\_tail = m\_tail->prev;

delete (m\_tail->next);

m\_tail->next = nullptr;

}

}

}

void clear()

{

node<Type> \*past;

for (node<Type> \*element = m\_head; element != nullptr; past = element, element = element->next, delete past)

;

m\_head = m\_tail = nullptr;

}

bool empty() const

{

return m\_head == nullptr;

}

size\_t size() const

{

size\_t size = 0;

for (node<Type> \*element = m\_head; element != nullptr; element = element->next, size++)

;

return size;

}

private:

//your private functions

node<Type> \*m\_head;

node<Type> \*m\_tail;

};

} // namespace stepik