**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**ОТЧЕТ**

**по лабораторной работе №3**

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

**Тема: «Вектор и список»**

|  |  |  |  |
| --- | --- | --- | --- |
| Студент гр. 7381 | |  | Вологдин М.Д. |
| Преподаватель |  |  | Жангиров Т. Р. |

Санкт-Петербург

2019

**Цель работы:**

Изучить реализацию контейнеров vector и list в языке С++ и реализовать семантически аналогичный функционал.

**Задание.**

1. Вектор
   1. Необходимо реализовать конструкторы и деструктор для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector
   2. Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.
   3. Необходимо реализовать функции resize и erase для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector
   4. Необходимо реализовать функции insert и push\_back для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector
2. Список

Необходимо реализовать список со следующими функциями:

* 1. Вставка элементов в голову и в хвост
  2. Получение элемента из головы и из хвоста
  3. Удаление из головы, хвоста и очистка
  4. Проверка размера
  5. Деструктор
  6. Конструктор копирования
  7. Конструктор перемещения
  8. Оператор присваивания

Реализовать итератор для списка, а именно:

1. Операторы: =, ==, !=, ++(постфиксный и префиксный), \*, ->.
2. Вставка элементов (Вставляет value перед элементом, на который указывает pos. Возвращает итератор, указывающий на вставленный value)
3. Удаление элементов (Удаляет элемент в позиции pos. Возвращает итератор, следующий за последним удаленным элементом).

**Ход работы.**

Реализация классов осуществлена упрощением кода из библиотек vector и list.

1. Вектор

Операция вставки реализуется путём сдвига всего содержимого вектора, начиная с позиции вставки, вправо на единицу/на длину вставляемого интервала значений. Удаление элемента в заданной позиции или интервала реализуется сдвигом хранимых значений в векторе влево на один элемент/длину интервала. Остальные операции тривиальны.

1. Список

В списке дополнительно хранится количество элементов для более быстрого определения размера списка, а также для более простой реализации других методов. Вставка элемента осуществляется связыванием двух элементов слева и справа с вставляемым. Удаление элемента из списка осуществляется связыванием двух элементов справа и слева от удаляемого, после чего освобождается память от удаляемого элемента. Остальные операции тривиальны.

Код представлен в приложениях А и Б.

**Выводы:**

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

**Приложение А**

**Исходный код. Вектор.**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

#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 Type[count] : nullptr), m\_last(m\_first + count)

{}

template <typename InputIterator>

vector(InputIterator first, InputIterator last)

{

int count = std::distance(first, last);

m\_first = new value\_type[count];

m\_last = m\_first + count;

std::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.m\_first;

m\_last = other.m\_last;

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

}

~vector()

{

delete [] m\_first;

}

//assignment operators

vector& operator=(const vector& other)

{

if(this != &other)

{

delete[] m\_first;

m\_first = new value\_type[other.size()];

m\_last = m\_first + other.size();

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

}

return \*this;

}

vector& operator=(vector&& other)

{

if(this != &other)

{

delete[] m\_first;

m\_first=m\_last=nullptr;

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!=nullptr)

delete [] m\_first;

m\_first = new value\_type[last-first];

m\_last = m\_first + std::distance(first,last);

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

}

// resize methods

void resize(size\_t count)

{

if(count == size())

return;

Type\* temp = new Type[count];

if(count > size())

std::copy(m\_first, m\_last, temp);

else

std::copy(m\_first, m\_first+count, temp);

delete[] m\_first;

m\_first = temp;

m\_last = m\_first+count;

}

//erase methods

iterator erase(const\_iterator pos)

{

iterator \_Pos = iterator(pos);

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

return (\_Pos);

}

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

{

iterator \_First = iterator(first);

iterator \_Last = iterator(last);

std::rotate(\_First, \_Last, m\_last);

m\_last -= std::distance(first,last);

return (\_First);

}

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

{

size\_t len = pos - m\_first;

resize(size() + 1);

iterator newPos = m\_first + len;

std::rotate(newPos, m\_last - 1, m\_last);

m\_first[newPos - m\_first] = value;

return newPos;

}

template <typename InputIterator>

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

{

size\_t len = std::distance(first,last);

iterator newPos = const\_cast<iterator>(pos);

while (len != 0)

{

len--;

newPos = insert(newPos, \*(first + len));

}

return newPos;

}

//push\_back methods

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

{

resize(size()+1);

\*(m\_last-1) = 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. Лист.**

#include <assert.h>

#include <algorithm>

#include <stdexcept>

#include <cstddef>

#include <iostream>

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 ++ ()

{

if (m\_node != nullptr)

m\_node = m\_node->next;

return \*this;

}

list\_iterator operator ++ (int)

{

list\_iterator\* temp = new list\_iterator(\*this);

++(\*this);

return (\*temp);

}

//////////////////////

list\_iterator& operator -- ()

{

if (m\_node != nullptr)

m\_node = m\_node->prev;

return \*this;

}

list\_iterator operator -- (int)

{

list\_iterator\* temp = new list\_iterator(\*this);

--(\*this);

return (\*temp);

}

//////////////////////

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), m\_size(0)

{

}

~list()

{

clear();

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

list(const list& other) : m\_head(nullptr), m\_tail(nullptr), m\_size(0)

{

for (node<Type>\* Pnode = other.m\_head; Pnode; Pnode = Pnode->next)

push\_back(Pnode->value);

}

list(list&& other): m\_head(nullptr), m\_tail(nullptr), m\_size(0)

{

std::swap(m\_head, other.m\_head);

std::swap(m\_tail, other.m\_tail);

std::swap(m\_size, other.m\_size);

}

list& operator= (const list& other)

{

clear();

for (node<Type>\* Pnode = other.m\_head; Pnode; Pnode = Pnode->next)

push\_back(Pnode->value);

return (\*this);

}

list& operator=(list&& other)

{

std::swap(m\_head, other.m\_head);

std::swap(m\_tail, other.m\_tail);

std::swap(m\_size, other.m\_size);

return (\*this);

}

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

{

node<Type>\* new\_node = new node<Type>(value, nullptr, m\_tail);

if (empty())

{

m\_head = m\_tail = new\_node;

}

else

{

m\_tail->next = new\_node;

m\_tail = new\_node;

}

++m\_size;

}

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

{

node<Type>\* new\_node = new node<Type>(value, m\_head, nullptr);

if (empty())

{

m\_head = m\_tail = new\_node;

}

else

{

m\_head->prev = new\_node;

m\_head = new\_node;

}

++m\_size;

}

iterator insert(iterator pos, const Type& value)

{

node<Type>\* Pnode = pos.m\_node;

if (Pnode)

{

if (Pnode == m\_head)

{

push\_front(value);

return iterator(m\_head);

}

else

{

node<Type>\* new\_node = new node<Type>(value, Pnode, Pnode->prev);

Pnode->prev->next = new\_node;

Pnode->prev = new\_node;

++m\_size;

return iterator(new\_node);

}

}

else

{

push\_back(value);

return iterator(m\_tail);

}

}

iterator erase(iterator pos)

{

node<Type>\* Pnode = pos.m\_node;

if (Pnode == nullptr)

return pos;

if (Pnode == m\_head)

{

pop\_front();

return iterator(m\_head);

}

if (Pnode == m\_tail)

{

pop\_back();

return iterator();

}

node<Type>\* Pnext = Pnode->next;

Pnode->next->prev = Pnode->prev;

Pnode->prev->next = Pnode->next;

delete Pnode;

--m\_size;

return iterator(Pnext);

}

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())

{

--m\_size;

if (m\_head==m\_tail)

{

delete m\_head;

m\_head=m\_tail=nullptr;

}

else

{

node<Type>\* temp=m\_head->next;

delete m\_head;

m\_head=temp;

m\_head->prev = nullptr;

}

}

}

void pop\_back()

{

if(!empty())

{

--m\_size;

if (m\_head==m\_tail)

{

delete m\_head;

m\_head=m\_tail=nullptr;

}

else

{

node<Type>\* temp=m\_tail->prev;

delete m\_tail;

m\_tail=temp;

m\_tail->next = nullptr;

}

}

}

void clear()

{

while(!empty())

pop\_back();

}

bool empty() const

{

return !m\_size;

}

size\_t size() const

{

return m\_size;

}

private:

//your private functions

size\_t m\_size;

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}// namespace stepik