**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. 6304 |  | Ковынев М.В. |
| Преподаватель |  | Терентьев А.О. |

Санкт-Петербург

2018

**Цель работы.**

Изучить контейнеры C++ на примере *vector* и *list*

**Основные теоретические положения.**

Класс *vector* — это класс шаблона контейнеров последовательностей для хранения элементов заданного типа в линейном порядке и быстрого произвольного доступа к любому элементу. Он является наиболее подходящим типом контейнера для последовательности, когда на первом месте стоит производительность произвольного доступа. Для векторов время выполнения вставок и удалений элементов в конце последовательности является постоянной величиной. Время вставки и удаления элементов в середине вектора меняется линейно. Расширение вектора происходит, когда функции-члену требуется увеличить последовательность в объекте вектора сверх его текущей емкости. Другие операции вставки и стирания могут изменять различные адреса хранения внутри последовательности. Во всех таких случаях итераторы или ссылки, указывающие на изменившиеся части последовательности, становятся недействительными. Если расширения не происходит, действительными остаются только итераторы и ссылки перед точкой вставки или удаления.

Класс *list* — это класс шаблона контейнеров последовательностей, содержащих элементы в линейном порядке и позволяющие вставлять и удалять элементы в любом месте в пределах этих последовательностей. Последовательность сохраняется в виде двунаправленного связанного списка элементов, каждый из которых содержит член какого-либо типа *типа*.

Выбор типа контейнера должен в общем случае производиться на основе типа поиска и вставки, который требуется приложению. Векторы должны быть предпочитаемыми контейнерами для управления последовательностями, когда важен произвольный доступ к любому элементу, а вставка и удаление элементов требуется лишь в конце последовательности. Производительность контейнера класса двусторонней очереди выше, когда требуется произвольный доступ, а также вставка и удаление элементов и в начале последовательности, и в ее конце.

**Постановка задачи.**

Необходимо реализовать вектор со следующими функциями

* конструкторы
* деструктор
* операторы присваивания
* функция *assign*
* функция *resize*
* функция *erase*
* функция *insert*
* функция *push\_back*

Поведение реализованных функций должно быть таким же, как у класса [std::vector](http://ru.cppreference.com/w/cpp/container/vector). Семантику реализованных функций нужно оставить без изменений.

Необходимо реализовать список со следующими функциями:

* вставка элементов в голову и в хвост,
* получение элемента из головы и из хвоста,
* удаление из головы, хвоста и очистка
* проверка размера.
* деструктор
* конструктор копирования,
* конструктор перемещения,
* оператор присваивания.
* операторы: =, ==, !=, ++ (постфиксный и префиксный), \*, ->
* 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](http://ru.cppreference.com/w/cpp/container/list). Семантику реализованных функций нужно оставить без изменений.

**Требования к реализации:** при выполнении этого задания вы можете 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.**

1. Написан класс *Vector*. Класс хранит следующие переменные:

* Iterator m\_first — указатель на начало вектора
* Iterator m\_last— указатель на конец вектора

В классе реализованы следующие методы:

* Стандартный конструктор класса
* Конструктор с параметрами диапазона [first, last) *vector(InputIterator first, InputIterator last) :*
* Конструктор, создающий контейнер с содержимым списка инициализации *init*
* Конструктор копирования
* Конструктор перемещения
* Деструктор
* Оператор присваивания
* Оператор перемещения
* Функция *assign(InputIterator first, InputIterator last),* выполняющая замену содержимого в векторе
* Функция *resize(size\_t count),* изменяющая размер
* Функция *erase(const\_iterator pos),* которая удаляет заданный элемент
* Функция *insert(const\_iterator pos, const Type& value),* вставляющая значение value перед элементом, на который указывает pos
* Функция *insert(const\_iterator pos, InputIterator first, InputIterator last)* для диапазона [first, last)
* Функция *push\_back(const value\_type& value),* которая добавляет элемент в конце вектора
* Функция *at(size\_t pos)*
* Оператор индексирования
* Функции begin(), end(), size(), empty()

1. Код представлен в приложении А.
2. Реализована структура node, хранящая значение, указатель на следующий и предыдущий элементы.
3. Написан класс *list\_iterator.* Класс хранит следующие переменные:
   * Node<Type>\* m\_node
   * friend class list<Type>;

В классе реализованы следующие методы:

* Стандартный конструктор класса
* Оператор присваивания
* Операторы сравнения !=, ==
* Оператор\*, возвращающий ссылку на хранимое значение
* Оператор->, возвращающий указатель на хранимое значение
* Постфиксный и префиксный инкремент

1. Написан класс *list.* Класс хранит следующие переменные:
   * Node<Type>\* m\_head — указатель на начальный элемент списка
   * Node<Type>\* m\_tail — указатель на конечный элемент списка

В классе реализованы следующие методы:

* Стандартный конструктор класса
* Деструктор
* Конструктор копирования
* Конструктор перемещения
* Оператор присваивания
* Функция *push\_back(const value\_type& value),* которая добавляет элемент в конце списка
* Функция *push\_front(const value\_type& value),* которая добавляет элемент в к списка
* Функция front(), возвращающая значение первого элемента
* Функция back(), возвращающая значение последнего элемента
* Функция pop\_front(), удаляющая первый элемент
* Функция pop\_back(), удаляющая последний элемент
* Функция очистки списка clear()
* Функции size(), empty(), begin(), end()
* Функция *insert(iterator pos, const Type& value),* вставляющая значение value перед элементом, на который указывает pos
* Функция *erase(\_iterator pos),* которая удаляет заданный элемент

1. Код представлен в приложении Б.

**Выводы.**

В ходе выполнения данной лабораторной работы были изучены контейнеры C++ на примере *vector* и *list*

Приложение А

**КОД КОНТЕЙНЕРА VECTOR**

#include <assert.h>

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

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

#include <iostream>

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 = new Type[count];

m\_last = &(m\_first[count]);

for (int i = 0; i < count; i++)

{

m\_first[i] = 0;

}

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last) :

vector(last - first)

{

copy(first, last, m\_first);

}

vector(std::initializer\_list<Type> init) :

vector(init.begin(), init.end()) {}

vector(const vector& other)

{

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

m\_last = &(m\_first[other.size()]);

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

}

vector(vector&& other)

{

m\_first = other.m\_first;

m\_last = other.m\_last;

other.m\_first = nullptr;

other.m\_last = nullptr;

}

~vector()

{

delete[] m\_first;

}

vector& operator=(const vector& other)

{

if (this != &other)

vector(other).swap(\*this);

return \*this;

}

vector& operator=(vector&& other)

{

if (this != &other)

swap(other);

return \*this;

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

vector<Type>(first, last).swap(\*this);

}

void resize(size\_t count)

{

vector bufferVector(count);

if (count > size())

{

std::move(m\_first, m\_last, bufferVector.m\_first);

}

else

{

std::move(m\_first, m\_first + count, bufferVector.m\_first);

}

swap(bufferVector);

}

iterator erase(const\_iterator pos)

{

size\_t newPos = pos - m\_first;

std::rotate(const\_cast<iterator>(pos), const\_cast<iterator>(pos) + 1, m\_last);

resize(size() - 1);

return m\_first + newPos;

}

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

{

size\_t newPos = last - first;

iterator newFirst = const\_cast<iterator>(first);

while (newPos != 0)

{

newFirst = erase(newFirst);

newPos--;

}

return newFirst;

}

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

{

size\_t insertSize = pos - m\_first;

resize(size() + 1);

iterator newPos = m\_first + insertSize;

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)

{

int insertSize = last - first;

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

while (insertSize != 0)

{

insertSize--;

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

}

return newPos;

}

//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];

}

void swap(vector & other)

{

std::swap(this->m\_first, other.m\_first);

std::swap(this->m\_last, other.m\_last);

}

private:

iterator m\_first;

iterator m\_last;

};

}// namespace stepik

Приложение А

**КОД КОНТЕЙНЕРА LIST**

#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);

}

//Prefix

list\_iterator& operator ++ ()

{

m\_node = m\_node->next;

return \*this;

}

//Postfix

list\_iterator operator ++ (int)

{

list\_iterator buf(\*this);

++(\*this);

return buf;

}

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(nullptr), m\_tail(nullptr)

{

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

while (current)

{

push\_back(current->value);

current = current->next;

}

}

list(list&& other) : list()

{

if (this != &other)

swap(other);

}

list& operator= (const list& other)

{

if (this != &other)

list(other).swap(\*this);

return \*this;

}

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

{

if (!empty())

{

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

m\_tail->next = buf;

m\_tail = buf;

}

else

{

node<Type> \*buf = new node<Type>(value, nullptr, nullptr);

m\_head = buf;

m\_tail = buf;

}

}

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

{

if (!empty())

{

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

m\_head->prev = buf;

m\_head = buf;

}

else

{

node<Type> \*buf = new node<Type>(value, nullptr, nullptr);

m\_head = buf;

m\_tail = buf;

}

}

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 (size() == 1)

{

m\_head = NULL;

m\_tail = NULL;

}

else

{

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

buf->prev = nullptr;

delete m\_head;

m\_head = buf;

}

}

}

void pop\_back()

{

if (!empty())

{

if (size() == 1)

{

m\_head = NULL;

m\_tail = NULL;

}

else

{

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

buf->next = nullptr;

delete m\_tail;

m\_tail = buf;

}

}

}

void clear()

{

while (m\_head != NULL)

{

m\_tail = m\_head->next;

delete m\_head;

m\_head = m\_tail;

}

}

bool empty() const

{

return m\_head == NULL;

}

size\_t size() const

{

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

size\_t countSize = 0;

do

{

countSize++;

buf = buf->next;

} while (buf != NULL);

return countSize;

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

iterator insert(iterator pos, const Type& value)

{

if (pos.m\_node == NULL)

{

push\_back(value);

return iterator(m\_tail);

}

else

if (pos.m\_node->prev == NULL)

{

push\_front(value);

return iterator(m\_head);

}

else

{

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

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

pos.m\_node->prev = buf;

return iterator(buf);

}

}

iterator erase(iterator pos)

{

if (pos.m\_node == NULL)

{

return NULL;

}

else

if (pos.m\_node->prev == NULL)

{

pop\_front();

return iterator(m\_head);

}

else

if (pos.m\_node->next == NULL)

{

pop\_back();

return iterator(m\_tail);

}

else

{

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

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

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

iterator new\_pos(pos.m\_node->next);

delete buf;

return new\_pos;

}

}

private:

//your private functions

node<Type>\* m\_head;

node<Type>\* m\_tail;

void swap(list& lst)

{

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

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

}

};

}// namespace stepik