**void 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: «Вектор и список»**

|  |  |
| --- | --- |
| Студентка гр. 7381 | Машина Ю. Д. |
| Преподаватель | Жангиров Т. Р. |

Санкт-Петербург

2019

**Цель работы.**

Реализовать базовый функционал, семантически аналогичный функционалу из стандартной библиотеки шаблонов для классов вектор и линейный список.

**Задание.**

Необходимо реализовать конструкторы и деструктор для контейнера вектор. Предполагается реализация упрощенной версии вектора, без резервирования памяти под будущие элементы.

Необходимо реализовать операторы присваивания и функцию assign для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции resize и erase для контейнера вектор. Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать функции insert и push\_back для контейнера вектор.

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:

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.

Необходимо добавить к сделанной на прошлом шаге реализации списка следующие функции:

1. Деструктор;
2. Конструктор копирования;
3. Конструктор перемещения;
4. Оператор присваивания.

На данном шаге необходимо реализовать итератор для списка. Для краткости реализации можно ограничиться однонаправленным изменяемым (неконстантным) итератором. Необходимо реализовать операторы: =, ==, !=, ++ (постфиксный и префиксный), \*, ->.

На данном шаге с использованием итераторов необходимо реализовать:

1. Вставку элементов (Вставляет value перед элементом, на который указывает pos. Возвращает итератор, указывающий на вставленный value),
2. 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.**

Реализация класса vector была получена путём разбора и упрощения кода из библиотеки stl. Удаление элемента в заданной позиции или интервала реализуется арифметическим сдвигом хранимых значений в векторе влево на один элемент или длину интервала соответственно. В аналогичных условиях операция вставки реализуется путём арифметического сдвига всего содержимого вектора начиная с позиции вставки вправо на единицу или на длину вставляемого интервала значений соответственно. Это происходит в случае, если зарезервированной памяти для вставки достаточно, в противном случае выделяется новый участок памяти, туда копируется старое содержимое вектора до позиции вставки, затем копируется вставляемое значение или интервал значений, а затем оставшаяся часть старого вектора.

Обычно, резервируется в полтора раза больше памяти, чем необходимо в данный момент. Если это невозможно, то резервируется максимально возможный объём памяти.

Изменения размера происходит путём изменения значения размера массива (не путать с объёмом выделенной памяти).

Реализация остальных функций достаточно тривиальна. Таким образом, для работы вектора необходимо хранить информацию о размере используемой памяти, о размере выделенной памяти и указатель на участок памяти, в котором память, собственно, выделена.

Для реализации класса список хранятся указатели на голову списка, на хвост списка и количество элементов в списке. Последняя информация позволяет за определить размер списка (против в случае линейного подсчёта элементов списка).

Удаление элемента из списка осуществляется связыванием двух элементов справа и слева от удаляемого переброской указателей. После чего освобождается память для удаляемого элемента и возвращается указатель на следовавший за ним элемент.

Все остальные методы реализуются тривиальным образом.

**Исходный код.**

Код класса vector представлен в приложении А.

Код класса list представлен в приложении Б.

**Выводы.**

В ходе написания лабораторной работы были реализованы классы вектор и список, аналогичные классам из стандартной библиотеки. Полученные знания из предыдущих лабораторных работ были применены в ходе работы над этой работой.

**ПРИЛОЖЕНИЕ A**

**РЕАЛИЗАЦИЯ КЛАССА VECTOR**

#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)

{

// use previous step implementation

if (\_Buy(count))

{

m\_last += count;

}

}

template <typename InputIterator>

vector(InputIterator first, InputIterator last)

{

// use previous step implementation

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

if (\_Buy(\_Newsize))

{

m\_last += \_Newsize;

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

}

}

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

{

// use previous step implementation

if (\_Buy(init.size()))

{

m\_last += init.size();

std::copy(init.begin(), init.end(), m\_first);

}

}

vector(const vector& other)

{

// use previous step implementation

if (\_Buy(other.capacity()))

{

m\_last += other.size();

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

}

}

vector(vector&& other): m\_first(), m\_last(), m\_end()

{

// use previous step implementation

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

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

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

}

~vector()

{

// use previous step implementation

\_Tidy();

}

//insert methods

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

{

// implement this

iterator \_Pos = iterator(pos);

size\_t \_Offset = std::distance(m\_first, \_Pos);

if (!\_Has\_unused\_capacity())

{

\_Resize(size() + 1);

\_Pos = m\_first + \_Offset;

}

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

\*\_Pos = value;

return (\_Pos);

}

template <typename InputIterator>

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

{

// implement this

iterator \_Pos = iterator(pos);

iterator \_First = iterator(first);

iterator \_Last = iterator(last);

size\_t \_Offset = std::distance(\_First, \_Last);

size\_t \_Tillpos = std::distance(m\_first, \_Pos);

if (\_Unused\_capacity() < \_Offset)

{

\_Resize(size() + \_Offset);

\_Pos = m\_first + \_Tillpos;

}

std::rotate(\_Pos, \_Pos + size() - \_Offset - \_Tillpos, m\_last);

std::copy(first, last, \_Pos);

return (\_Pos);

}

//push\_back methods

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

{

// implement this

if (!\_Has\_unused\_capacity())

{

\_Resize(size() + 1);

--m\_last;

}

++m\_last;

\*(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;

}

// capacity method

size\_t capacity() const

{

return m\_end - m\_first;

}

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

bool \_Buy(size\_t \_Newcapacity)

{ // allocate array with \_Newcapacity elements

m\_first = m\_last = m\_end = iterator();

if (!\_Newcapacity)

return (false);

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

m\_last = m\_first;

m\_end = m\_first + \_Newcapacity;

return (true);

}

void \_Tidy()

{ // free all storage

if (m\_first != iterator())

{ // deallocate old array

delete[] m\_first;

m\_first = m\_last = m\_end = iterator();

}

}

void \_Change\_array(const iterator \_Newvec, const size\_t \_Newsize, const size\_t \_Newcapacity)

{ // discard old array, acquire new array

\_Tidy();

m\_first = \_Newvec;

m\_last = \_Newvec + \_Newsize;

m\_end = \_Newvec + \_Newcapacity;

}

bool \_Has\_unused\_capacity() const noexcept

{ // micro-optimization for capacity() != size()

return m\_end != m\_last;

}

size\_t \_Unused\_capacity() const noexcept

{ // micro-optimization for capacity() - size()

return m\_end - m\_last;

}

size\_t \_Calculate\_growth(const size\_t \_Newsize) const

{ // given \_Oldcapacity and \_Newsize, calculate growth

const size\_t \_Oldcapacity = capacity();

const size\_t \_Growth = \_Oldcapacity + \_Oldcapacity / 2;

if (\_Growth < \_Newsize)

{

return (\_Newsize); // \_Growth is not sufficient

}

return (\_Growth); // \_Growth is sufficient

}

void \_Resize(const size\_t \_Newsize)

{ // trim or append elements, provide strong guarantee

const size\_t \_Oldsize = size();

const size\_t \_Oldcapacity = capacity();

if (\_Newsize > \_Oldcapacity)

{ // reallocate

const size\_t \_Newcapacity = \_Calculate\_growth(\_Newsize);

iterator \_Newvec = new value\_type[\_Newcapacity];

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

\_Change\_array(\_Newvec, \_Newsize, \_Newcapacity);

}

else if (\_Newsize > \_Oldsize)

{ // append

m\_last += (\_Newsize - \_Oldsize);

}

else if (\_Newsize == \_Oldsize)

{ // nothing to do, avoid invalidating iterators

}

else

{ // trim

m\_last -= (\_Oldsize - \_Newsize);

}

}

private:

iterator m\_first;

iterator m\_last;

iterator m\_end;

};

}// namespace stepik**ПРИЛОЖЕНИЕ Б**

**РЕАЛИЗАЦИЯ КЛАССА LIST**

#include <iostream>

#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;

template <class Type>

class list\_iterator

{

public:

friend class list<Type>;

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(nullptr)

{

}

list\_iterator(node<Type>\* p)

: m\_node(p)

{

}

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);

}

list\_iterator& operator++()

{

m\_node = m\_node->next;

return (\*this);

}

list\_iterator operator++(int)

{

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

m\_node = m\_node->next;

return (\*old\_iterator);

}

private:

node<Type>\* m\_node;

};

template <class Type>

class list

{

public:

typedef list\_iterator<Type> iterator;

typedef Type value\_type;

typedef value\_type& reference;

typedef const value\_type& const\_reference;

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

{

}

~list()

{

clear();

}

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);

}

iterator insert(iterator pos, const Type& value)

{

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

if (Pnode == nullptr)

{

push\_back(value);

return iterator(m\_tail);

}

else if (Pnode->prev == nullptr)

{

push\_front(value);

return iterator(m\_head);

}

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);

}

iterator erase(iterator pos)

{

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

if (Pnode == nullptr)

return pos;

if (Pnode->prev == nullptr)

{

pop\_front();

return iterator(m\_head);

}

if (Pnode->next == nullptr)

{

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);

}

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

{

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

if (m\_size == 0)

{

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 (m\_size == 0)

{

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

}

else

{

m\_head->prev = new\_node;

m\_head = new\_node;

}

++m\_size;

}

void pop\_front()

{

--m\_size;

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

delete m\_head;

m\_head = new\_head;

if (m\_size == 0)

{

m\_tail = nullptr;

}

else

{

m\_head->prev = nullptr;

}

}

void pop\_back()

{

--m\_size;

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

delete m\_tail;

m\_tail = new\_tail;

if (m\_size == 0)

{

m\_head = nullptr;

}

else

{

m\_tail->next = nullptr;

}

}

void clear()

{

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

node<Type>\* Pnext;

while (Pnode)

{

Pnext = Pnode->next;

delete Pnode;

Pnode = Pnext;

}

m\_head = m\_tail = nullptr;

m\_size = 0;

}

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\_size);

}

size\_t size() const

{

return (m\_size);

}

iterator begin() const

{

return iterator(m\_head);

}

iterator end() const

{

return iterator();

}

private:

node<Type>\* m\_head;

node<Type>\* m\_tail;

size\_t m\_size;

};

}// namespace stepik