**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №3**

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

Тема: «Контейнеры вектор и список»

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7381 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Кревчик А.Б. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Жангиров Т.Р. |

Санкт-Петербург

2019

**Цель работы.**

Изучить стандартные контейнерыvectorи list языка С++.

**Задание.**

Необходимо реализовать конструкторы, деструктор, оператор присваивания, функцииassign, 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 listбыли 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 ->.

Реализация класса представлена в приложении А.

**Вывод.**

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

**Приложение А. Файл vector.h.**

#include<assert.h>

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

#include<cstddef>// size\_t

#include<initializer\_list>

#include<stdexcept>

namespacestepik {

template <typename Type>

classvector {

public:

typedef Type\* iterator;

typedefconst Type\* const\_iterator;

typedef Type value\_type;

typedefvalue\_type& reference;

typedefconstvalue\_type&const\_reference;

typedefstd::ptrdiff\_tdifference\_type;

// Constructors and destructor

explicitvector(size\_t count = 0)

: m\_first(count ? newvalue\_type[count] : nullptr), m\_last(count ? m\_first + count : nullptr) {

}

template <typenameInputIterator>

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

for(inti = 0; i< last - first; i++)

m\_first[i] = (value\_type)first[i];

}

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

}

vector(const vector& other) : vector(other.m\_first, other.m\_last) {

}

vector(vector&& other) : vector() {

swap(\*this, other);

}

~vector() {

delete[]m\_first;

}

// Methods and operators

vector&operator=(const vector& other) {

if (this != &other) {

vector temp(other);

swap(\*this, temp);

}

return \*this;

}

vector&operator=(vector&& other) {

if (this != &other)

swap(\*this, other);

return \*this;

}

template <typenameInputIterator>

voidassign(InputIterator first, InputIterator last) {

delete[]m\_first;

m\_first = last - first ? newvalue\_type[last - first] : nullptr;

m\_last = last - first ? m\_first + (last - first) : nullptr;

for (inti = 0; i< last - first; i++)

m\_first[i] = (value\_type)first[i];

}

voidresize(size\_t count) {

if (m\_last - m\_first != count) {

vector temp(count);

std::copy(m\_first, m\_last - m\_first> count ? m\_first + count : m\_last, temp.m\_first);

swap(\*this, temp);

}

}

iterator erase(const\_iteratorpos) {

size\_t offset = pos - m\_first;

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

resize(size()-1);

returnm\_first + offset;

}

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

size\_t f = first - m\_first;

            size\_t l = last - m\_first;

            vector temp(\*this);

            std::rotate(temp.m\_first + f, temp.m\_first + l, temp.m\_last);

            temp.resize(temp.size() - l + f);

            \*this = std::move(temp);

            returnm\_first + f;

}

iterator insert(const\_iteratorpos, const Type& value) {

size\_t offset = pos - m\_first;

resize(size()+1);

\*(m\_last-1) = value;

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

returnm\_first + offset;

}

template <typenameInputIterator>

iterator insert(const\_iteratorpos, InputIterator first, InputIterator last) {

size\_t offset = pos - m\_first;

resize( size() + (last-first));

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

std::rotate(m\_first+offset, m\_last - (last-first) , m\_last);

returnm\_first + offset;

}

voidpush\_back(constvalue\_type& value) {

resize(size()+1);

\*(m\_last-1) = value;

}

reference at(size\_tpos) {

returncheckIndexAndGet(pos);

}

const\_referenceat(size\_tpos) const {

returncheckIndexAndGet(pos);

}

reference operator[](size\_tpos) {

returnm\_first[pos];

}

const\_referenceoperator[](size\_tpos) const {

returnm\_first[pos];

}

iterator begin() {

returnm\_first;

}

const\_iteratorbegin() const {

returnm\_first;

}

iterator end() {

returnm\_last;

}

const\_iteratorend() const {

returnm\_last;

}

size\_tsize() const {

returnm\_last - m\_first;

}

boolempty() const {

returnm\_first == m\_last;

}

private:

reference checkIndexAndGet(size\_tpos) const {

if (pos>= size())

throwstd::out\_of\_range("out of range");

returnm\_first[pos];

}

voidswap(vector& v1, vector& v2) {

            std::swap(v1.m\_first, v2.m\_first);

            std::swap(v1.m\_last, v2.m\_last);

        }

private:

iterator m\_first;

iterator m\_last;

};

}

**Приложение Б. Файл list.h.**

#include<assert.h>

#include<algorithm>

#include<stdexcept>

#include<cstddef>

namespacestepik {

template <classType>

structnode {

Type value;

node\* next;

node\* prev;

node(const Type& value, node<Type>\* next, node<Type>\* prev)

: value(value), next(next), prev(prev) {

}

};

template <classType>

classlist;

template <classType>

classlist\_iterator {

public:

typedefptrdiff\_tdifference\_type;

typedef Type value\_type;

typedef Type\* pointer;

typedef Type& reference;

typedefsize\_tsize\_type;

typedefstd::forward\_iterator\_tagiterator\_category;

list\_iterator() : m\_node(nullptr) {

}

list\_iterator(constlist\_iterator& other) : m\_node(other.m\_node) {

}

list\_iterator&operator = (constlist\_iterator& other) {

m\_node = other.m\_node;

return \*this;

}

booloperator == (constlist\_iterator& other) const {

returnm\_node == other.m\_node;

}

booloperator != (constlist\_iterator& other) const {

returnm\_node != other.m\_node;

}

reference operator \* () {

returnm\_node->value;

}

pointer operator -> () {

return&(m\_node->value);

}

list\_iterator&operator ++ () {

m\_node = m\_node->next;

return \*this;

}

list\_iteratoroperator ++ (int) {

list\_iteratortemp(\*this);

++(\*this);

return temp;

}

private:

friendclasslist<Type>;

list\_iterator(node<Type>\* p) : m\_node(p) {

}

node<Type>\* m\_node;

};

template <classType>

classlist {

public:

typedef Type value\_type;

typedefvalue\_type& reference;

typedefconstvalue\_type&const\_reference;

typedeflist\_iterator<Type> iterator;

// Constructors and destructor

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

}

list(const list& other) : list() {

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

push\_back(ptr->value);

}

list(list&& other) : list() {

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

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

}

~list() {

clear();

}

// Methods and operators

list&operator=(const list& other) {

if(this != &other){

clear();

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

push\_back(ptr->value);

}

return \*this;

}

list::iterator begin() {

returniterator(m\_head);

}

list::iterator end() {

returniterator();

}

voidpush\_back(constvalue\_type& value) {

if(m\_head == nullptr) {

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

m\_tail = m\_head;

} else {

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

m\_tail = m\_tail->next;

}

}

voidpush\_front(constvalue\_type& value) {

if(m\_head == nullptr) {

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

m\_tail = m\_head;

} else {

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

m\_head = m\_head->prev;

}

}

iterator insert(iterator pos, const Type& value) {

if (pos.m\_node == nullptr) {

push\_back(value);

returniterator(m\_tail);

}

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

push\_front(value);

returniterator(m\_head);

}

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

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

pos.m\_node->prev = temp;

returniterator(temp);

}

iterator erase(iterator pos) {

if (pos.m\_node == nullptr)

returnpos;

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

pop\_back();

returnnullptr;

}

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

pop\_front();

returniterator(m\_head);

}

node<Type>\* temp = pos.m\_node->next;

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

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

deletepos.m\_node;

return temp;

}

reference front() {

returnm\_head->value;

}

const\_referencefront() const {

returnm\_head->value;

}

reference back() {

returnm\_tail->value;

}

const\_referenceback() const {

returnm\_tail->value;

}

voidpop\_front() {

if(!empty()){

if (m\_head == m\_tail){

deletem\_head;

m\_head = nullptr;

m\_tail = nullptr;

} else {

m\_head = m\_head->next;

deletem\_head->prev;

m\_head->prev = nullptr;

}

}

}

voidpop\_back() {

if(!empty()){

if (m\_head == m\_tail){

deletem\_head;

m\_head = nullptr;

m\_tail = nullptr;

} else {

m\_tail = m\_tail->prev;

deletem\_tail->next;

m\_tail->next = nullptr;

}

}

}

voidclear() {

while (!empty())

pop\_front();

}

boolempty() const {

returnm\_tail == nullptr;

}

size\_tsize() const {

size\_t result = 0;

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

while( ptr != nullptr) {

result++;

ptr = ptr->next;

}

return result;

}

private:

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}