**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»**

|  |  |
| --- | --- |
| Студент гр. 7381 | Габов Е. С. |
| 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.

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 <class 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; |
|  | } |
|  |  |
|  | template <typename InputIterator> |
|  | vector(InputIterator first, InputIterator last){ |
|  | size\_t vectorSize; |
|  | vectorSize = last - first; |
|  | m\_first = new Type[vectorSize]; |
|  | m\_last = m\_first + vectorSize; |
|  | 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) noexcept{ |
|  | m\_first = other.m\_first; |
|  | m\_last = other.m\_last; |
|  | other.m\_first = nullptr; |
|  | other.m\_last = nullptr; |
|  | } |
|  |  |
|  | ~vector(){ |
|  | delete[] m\_first; |
|  | } |
|  |  |
|  | //assignment operators |
|  | vector& operator=(const vector& other){ |
|  | if(this != &other){ |
|  | delete[] m\_first; |
|  | size\_t vectorSize = other.m\_last - other.m\_first; |
|  | m\_first = new Type[vectorSize]; |
|  | m\_last = m\_first + vectorSize; |
|  | std::copy(other.m\_first, other.m\_last, m\_first); |
|  | return \*this; |
|  | } |
|  | } |
|  |  |
|  | vector& operator=(vector&& other) noexcept{ |
|  | if (this != &other) { |
|  | delete[] m\_first; |
|  | m\_first = other.m\_first; |
|  | m\_last = other.m\_last; |
|  | other.m\_first = nullptr; |
|  | other.m\_last = nullptr; |
|  | return \*this; |
|  | } |
|  | } |
|  |  |
|  | // assign method |
|  | template <typename InputIterator> |
|  | void assign(InputIterator first, InputIterator last){ |
|  | \*this = std::move(vector(first, last)); |
|  | } |
|  |  |
|  | void resize(size\_t count){ |
|  | iterator end = (count > size()) ? m\_last : m\_first + count; |
|  | vector tempVector(count); |
|  | std::move(m\_first, end, tempVector.m\_first); |
|  | std::swap(m\_first, tempVector.m\_first); |
|  | std::swap(m\_last, tempVector.m\_last); |
|  | } |
|  |  |
|  | //erase methods |
|  | iterator erase(const\_iterator pos){ |
|  | size\_t posNum = pos - m\_first; |
|  | std::rotate(const\_cast<iterator>(pos), const\_cast<iterator>(pos) + 1, m\_last); |
|  | resize(size() - 1); |
|  | return m\_first + posNum; // след после удаленного |
|  | } |
|  |  |
|  | iterator erase(const\_iterator first, const\_iterator last){ |
|  | iterator it = const\_cast<iterator>(first); |
|  | for(size\_t i = 0, count = last - first; i < count; ++i){ |
|  | it = erase(it); |
|  | } |
|  | return it; |
|  | } |
|  |  |
|  | iterator insert(const\_iterator pos, const Type& value){ |
|  | vector tempVector(size() + 1); |
|  | size\_t diff = pos - m\_first; |
|  | std::copy(m\_first, m\_first + diff, tempVector.m\_first); |
|  | \*(tempVector.begin() + diff) = value; |
|  | std::copy(m\_first + diff, m\_last, tempVector.begin() + diff + 1); |
|  | \*this = std::move(tempVector); |
|  | return m\_first + diff; |
|  | } |
|  |  |
|  | template <typename InputIterator> |
|  | iterator insert(const\_iterator pos, InputIterator first, InputIterator last){ |
|  | vector tempVector(size() + (last - first)); |
|  | size\_t diff = pos - m\_first; |
|  | std::copy(m\_first, m\_first + diff, tempVector.m\_first); |
|  | std::copy(first, last, tempVector.begin() + diff); |
|  | std::copy(m\_first + diff, m\_last, tempVector.begin() + diff + (last - first)); |
|  | \*this = std::move(tempVector); |
|  | return m\_first + diff; |
|  | } |
|  |  |
|  | void push\_back(const value\_type& value){ |
|  | insert(this->end(), value); |
|  | } |
|  |  |
|  | reference at(size\_t pos) |
|  | { |
|  | return checkIndexAndGet(pos); |
|  | } |
|  |  |
|  | const\_reference at(size\_t pos) const |
|  | { |
|  | return checkIndexAndGet(pos); |
|  | } |
|  |  |
|  | reference operator[](size\_t pos) |
|  | { |
|  | return m\_first[pos]; |
|  | } |
|  |  |
|  | const\_reference operator[](size\_t pos) const |
|  | { |
|  | return m\_first[pos]; |
|  | } |
|  |  |
|  | iterator begin() |
|  | { |
|  | return m\_first; |
|  | } |
|  |  |
|  | const\_iterator begin() const |
|  | { |
|  | return m\_first; |
|  | } |
|  |  |
|  | iterator end() |
|  | { |
|  | return m\_last; |
|  | } |
|  |  |
|  | const\_iterator end() const |
|  | { |
|  | return m\_last; |
|  | } |
|  |  |
|  | size\_t size() const |
|  | { |
|  | return m\_last - m\_first; |
|  | } |
|  |  |
|  | 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**

**РЕАЛИЗАЦИЯ КЛАССА 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; |
|  | } |
|  |  |
|  | // Указатель на хранимое |
|  | pointer operator-> (){ |
|  | return &(m\_node->value); |
|  | } |
|  |  |
|  | // ++object |
|  | list\_iterator& operator ++(){ |
|  | m\_node = m\_node->next; |
|  | return \*this; |
|  | } |
|  |  |
|  | // object++ |
|  | list\_iterator operator ++ (int){ |
|  | auto ret = \*this; |
|  | m\_node = m\_node->next; |
|  | return ret; |
|  | } |
|  |  |
|  | 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::iterator begin() |
|  | { |
|  | return iterator(m\_head); |
|  | } |
|  |  |
|  | list::iterator end() |
|  | { |
|  | return iterator(); |
|  | } |
|  |  |
|  | void push\_back(const value\_type& value){ |
|  | if(empty()){ |
|  | m\_tail = m\_head = new node<Type>(value, nullptr, nullptr); |
|  | return; |
|  | } |
|  | m\_tail->next = new node<Type>(value, nullptr,m\_tail); |
|  | m\_tail=m\_tail->next; |
|  | } |
|  |  |
|  | void pop\_front(){ |
|  | if(m\_head == m\_tail){ |
|  | delete m\_head; |
|  | m\_head = m\_tail = nullptr; |
|  | return; |
|  | } |
|  | m\_head = m\_head->next; |
|  | delete m\_head->prev; |
|  | m\_head->prev= nullptr; |
|  | } |
|  |  |
|  | void pop\_back(){ |
|  | if(m\_head == m\_tail){ |
|  | delete m\_head; |
|  | m\_head = m\_tail = nullptr; |
|  | return; |
|  | } |
|  | m\_tail = m\_tail->prev; |
|  | delete m\_tail->next; |
|  | m\_tail->next = nullptr; |
|  | } |
|  |  |
|  | reference front(){ |
|  | return m\_head->value; |
|  | } |
|  |  |
|  | reference back(){ |
|  | return m\_tail->value; |
|  | } |
|  |  |
|  |  |
|  | void push\_front(const value\_type& value){ |
|  | if(empty()){ |
|  | m\_tail = m\_head = new node<Type>(value, nullptr, nullptr); |
|  | return; |
|  | } |
|  | m\_head = new node<Type>(value, m\_head, nullptr); |
|  | m\_head->next->prev = m\_head; |
|  | } |
|  |  |
|  | iterator insert(iterator pos, const Type& value){ |
|  | if (pos.m\_node == NULL){ |
|  | push\_back( value); |
|  | return iterator(m\_tail); |
|  | } |
|  | if (pos.m\_node == m\_head){ |
|  | push\_front(value); |
|  | return iterator(m\_head); |
|  | } |
|  | pos.m\_node->prev->next = new node<Type>( value, pos.m\_node, pos.m\_node->prev); |
|  | pos.m\_node->prev = pos.m\_node->prev->next; |
|  | return iterator(pos.m\_node->prev); |
|  | } |
|  |  |
|  | iterator erase(iterator pos){ |
|  | if(pos.m\_node == m\_head){ |
|  | pop\_front(); |
|  | return iterator(m\_head); |
|  | } |
|  | if(pos.m\_node == m\_tail){ |
|  | pop\_back(); |
|  | return iterator(m\_tail); |
|  | } |
|  | node<Type> \*temp = pos.m\_node->next; |
|  | temp->prev = pos.m\_node->prev; |
|  | temp->prev->next = temp; |
|  | delete pos.m\_node; |
|  | return iterator(temp); |
|  | } |
|  |  |
|  | private: |
|  | //your private functions |
|  | void clear(){ |
|  | while(!empty()) |
|  | pop\_back(); |
|  | } |
|  | bool empty() const{ |
|  | return m\_head == nullptr; |
|  | } |
|  | node<Type>\* m\_head; |
|  | node<Type>\* m\_tail; |
|  | }; |
|  |  |
|  | }// namespace stepik |