Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«**Пермский национальный исследовательский политехнический университет»**

Кафедра «Информационные технологии и автоматизированные системы»

**ОТЧЕТ**

Дисциплина: «Основы алгоритмизации и программирования»

Тема: АТД. Контейнеры

Семестр 2

Выполнил работу

Студент группы РИС-22-1Б

Токарев Павел Аркадьевич

Проверил

Доцент кафедры ИТАС

Полякова Ольга Андреевна

Г. Пермь-2023

**Постановка задачи**

1.Определить класс-контейнер.

2.Реализовать конструкторы, деструкторы, операции ввода-вывода, операцию присваивания.

3.Перегрузить операции: доступа по индексу, определения размера вектора, сложения элементов векторов

4.Реализовать класс-итератор. Реализовать с помощью его операции последовательного доступа.

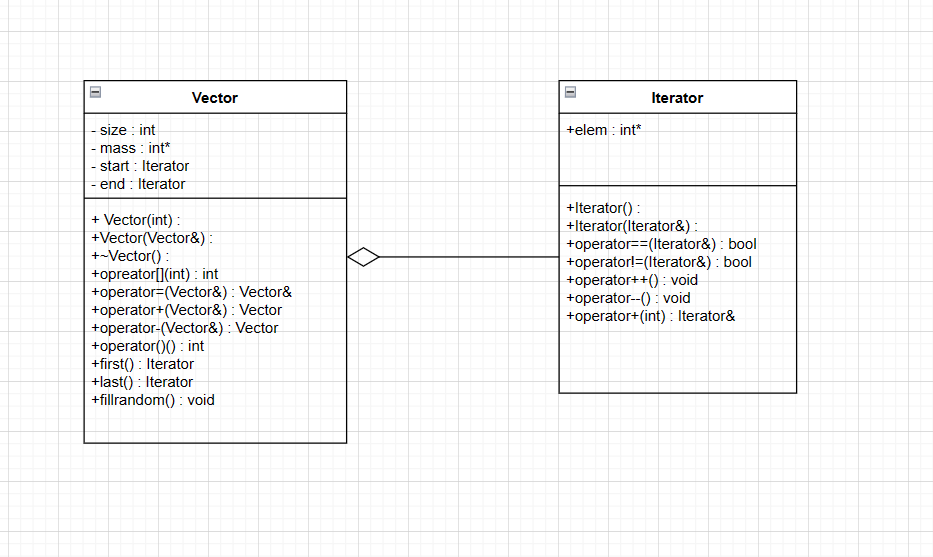
**Анализ**

1.Определим класс Vector, полями которого будут размер size, динамический массив mass и объекты класса Iterator start и end.

2.Класс Iterator содержит указатель на элемент массива.

3.В классе Vector перегрузим операции из задания.

**Диаграмма UML**

****

**Код программы**

main.cpp

#include<iostream>

#include"iterator.h"

#include"vector.h"

using namespace std;

int main() {

setlocale(LC\_ALL, "rus");

vector a(10);

a.fillrandom(a());

cout << "Заполним вектор случайными числами" <<endl;

cout << a<<endl;

cout <<"Первый элемент: " << \*(a.first())<<endl;

cout << "Последний элемент: " << \*(a.last()) << endl;

cout << "Выедите вектор b из 10 элементов" << endl;

vector b(10);

cin >> b;

cout << "6 элемент: " <<b[5] << endl;

cout << "Выведем вектор с помощью итератора" << endl;

Iterator iter = b.first();

for (int i = 0;i < 10;i++) {

cout << \*(iter)<<" ";

iter = iter + 1;

}

cout <<endl<< "Сложим векторы a и b" << endl;

vector c = a + b;

cout << c;

}

Vector.h

#pragma once

#include<iostream>

#include "iterator.h"

using namespace std;

class vector

{

public:

vector(int t);

vector(const vector& t);

~vector();

int operator[](int index);

vector& operator=(const vector& t);

vector operator+(const vector& t);

vector operator-(const vector& t);

int operator() ();

friend ostream& operator<<(ostream& out, const vector& t);

friend istream& operator>>(istream& in, vector& t);

Iterator first();

Iterator last();

void fillrandom(int sz);

private:

int size;

int\* mass;

Iterator start;

Iterator end;

};

Vector.cpp

#include "vector.h"

#include"iterator.h"

#include<iostream>

using namespace std;

vector::vector(int sz) {

size = sz;

mass = new int[sz];

for (int i = 0;i < sz;i++) {

mass[i] = 0;

}

start.elem = &mass[0];

end.elem = &mass[sz - 1];

}

vector::vector(const vector& t) {

size = t.size;

mass = new int[size];

for (int i = 0;i < size;i++) {

mass[i] = t.mass[i];

}

start = t.start;

end = t.end;

}

vector::~vector() {

delete[] mass;

size = 0;

}

vector& vector::operator=(const vector& t) {

size = t.size;

for (int i = 0;i < size;i++) {

mass[i] = t.mass[i];

}

start = t.start;

end = t.end;

return \*this;

}

int vector::operator[](int index) {

if (index < size) {

return mass[index];

}

else {

return -1;

}

}

vector vector::operator-(const vector& t) {

vector tmp(t.size);

for (int i = 0;i < t.size;i++) {

tmp.mass[i] = mass[i] - t.mass[i];

}

return tmp;

}

vector vector::operator+(const vector& t) {

vector tmp(t.size);

for (int i = 0;i < t.size;i++) {

tmp.mass[i] = mass[i] + t.mass[i];

}

return tmp;

}

int vector::operator()() {

return size;

}

ostream& operator<<(ostream& out, const vector& t) {

for (int i = 0;i < t.size;i++) {

out << t.mass[i] << " ";

}

return out;

}

istream& operator>>(istream& in,vector& t) {

for (int i = 0;i < t.size;i++) {

cout << (i+1)<<":";

in >> t.mass[i];

}

return in;

}

Iterator vector::first() {

return start;

}

Iterator vector::last() {

return end;

}

void vector::fillrandom(int sz) {

for (int i = 0;i < sz;i++) {

mass[i] = rand() % 100;

}

}

Iterator.h

#pragma once

class Iterator {

friend class vector;

public:

Iterator();

Iterator(const Iterator& t);

bool operator==(const Iterator& t);

bool operator!=(const Iterator& t);

void operator++();

void operator--();

int operator\*()const;

Iterator& operator+(const int t);

private:

int\* elem;

};

Iterator.cpp

#include "iterator.h"

#include"vector.h"

Iterator::Iterator() {

elem = nullptr;

}

Iterator::Iterator(const Iterator& t) {

elem = t.elem;

}

bool Iterator::operator==(const Iterator& t) {

return(elem = t.elem);

}

bool Iterator::operator!=(const Iterator& t) {

return(elem != t.elem);

}

void Iterator::operator++() {

elem++;

}

void Iterator::operator--() {

elem--;

}

int Iterator::operator\*()const{

return \*elem;

}

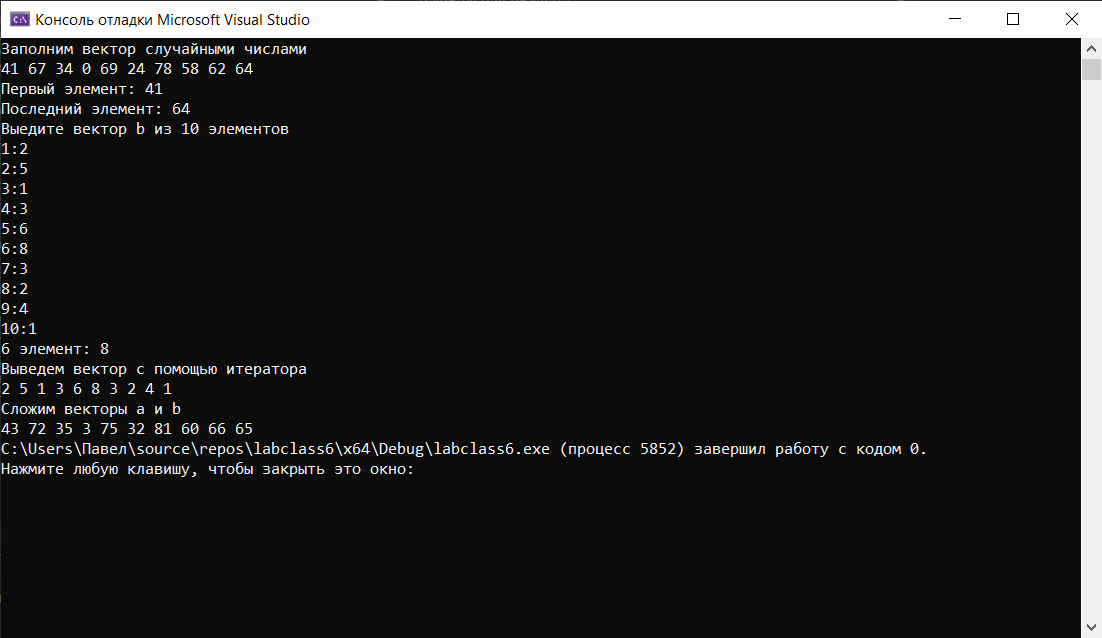
Iterator& Iterator::operator+(const int t) {

elem = elem + t;

return \*this;

}

**Результат работы программы**

****

**Анализ результатов**

Удалось успешно реализовать класс-контейнер и класс-итератор. Удаётся без проблем осуществлять доступ к элементу вектор через перегруженные операторы.

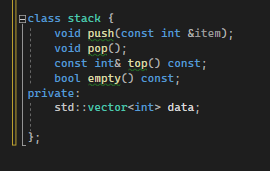
**Контрольные вопросы**

1.Абстрактный тип данных (АТД) - это тип данных, определенный не через его реализацию, а через набор операций, которые могут выполняться над этим типом данных. Операции определяются интерфейсом АТД и скрывают детали его реализации.

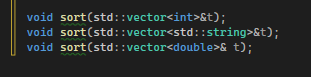
Примеры абстрактных типов данных:

* Очередь - это АТД, который поддерживает операции добавления и удаления элементов в двух концах. Примеры реализации: очередь на основе массива, очередь на основе связного списка.
* Список - это АТД, который представляет собой набор элементов, упорядоченных в определенной последовательности. Он поддерживает операции добавления, удаления и поиска элементов. Примеры реализации:

2.Пример абстракции через параметризацию:



3.Функция "сортировка" может быть специфицирована для различных типов данных и алгоритмов сортировки.



4.Контейнер - это объект, который содержит и управляет другими объектами (элементами), обеспечивая доступ к ним и их хранение.

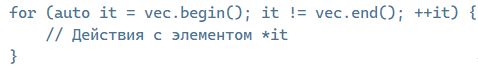
* std::vector - динамический массив, который может изменять свой размер во время выполнения программы.
* std::list - двусвязный список, который позволяет быстро добавлять и удалять элементы в любом месте списка.
* std::map - ассоциативный массив, который хранит пары ключ-значение и обеспечивает быстрый доступ к значению по ключу.
* std::set - упорядоченное множество уникальных элементов, которые могут быть быстро добавлены или удалены из множества.
* std::queue - очередь, которая работает по принципу "первым пришел - первым обслужен".
* std::stack - стек, который работает по принципу "последним пришел - первым обслужен".

5.В контейнерах выделяют три группы операций:

* Операции добавления и удаления элементов - включают в себя методы pushback, pushfront, insert, erase и другие.
* Операции доступа к элементам - включают в себя методы operator, at, front, back и другие.
* Операции изменения размера контейнера - включают в себя методы resize, reserve и другие.

6.Существуют следующие виды доступа к элементам контейнера:

* Доступ по индексу - осуществляется с помощью оператора или метода at. Например, для вектора vec доступ к элементу с индексом i можно получить следующим образом: veci или vec.at(i).
* Доступ к первому и последнему элементам - осуществляется с помощью методов front и back соответственно. Например, для списка lst доступ к первому и последнему элементу можно получить следующим образом: lst.front() и lst.back().
* Итераторный доступ - осуществляется с помощью итераторов, которые позволяют перебирать элементы контейнера по порядку. Например, для вектора vec итераторный доступ можно реализовать следующим образом:



7.Итератор - это объект, который используется для перебора элементов контейнера. Он представляет собой указатель на текущий элемент и может быть инкрементирован или декрементирован, чтобы переместиться к следующему или предыдущему элементу. Итераторы могут быть константными или неконстантными, в зависимости от того, можно ли изменять элементы контейнера через итератор или нет. Итераторы позволяют работать с контейнером без необходимости знать его внутреннюю структуру и реализацию.

8. Итератор может быть реализован как отдельный класс или структура, который содержит указатель на текущий элемент контейнера и методы для перемещения по контейнеру. Методы могут включать инкрементирование, декрементирование, получение значения текущего элемента, проверку на равенство и неравенство итераторов и т.д. Также может быть реализовано несколько различных типов итераторов для разных целей, таких как итераторы только для чтения, итераторы для изменения элементов и т.д.

9.Существует несколько способов объединения контейнеров, в зависимости от требуемой функциональности и типа контейнеров:

* Использование алгоритмов стандартной библиотеки. Стандартная библиотека C++ предоставляет множество алгоритмов для работы с контейнерами, включая алгоритмы для объединения контейнеров. Например, функция std::merge() объединяет два отсортированных контейнера в один отсортированный.
* Использование специальных контейнеров. Некоторые контейнеры, такие как std::set и std::map, предоставляют методы для объединения с другими контейнерами. Например, метод std::set::insert() может быть использован для объединения двух множеств.
* Ручное объединение. Если необходимо объединить контейнеры с определенной логикой, можно написать свою функцию или метод для объединения. Например, для объединения двух списков можно создать новый список и добавлять в него элементы из обоих списков поочередно.
* Использование шаблонов. Если требуется объединить контейнеры разных типов, можно использовать шаблоны. Например, можно написать шаблонную функцию, которая принимает два контейнера и возвращает новый контейнер, содержащий элементы обоих контейнеров.

10.Контейнер, состоящий из элементов "ключ-значение", предоставляет доступ к элементам по ключу. Например, в контейнере std::map можно получить значение элемента по его ключу с помощью метода operator. Также доступ к элементам можно получить с помощью итераторов, которые указывают на пары "ключ-значение".

11.Такой контейнер называется стек (stack).

12.Контейнером является int mas[100]; - это массив, который может содержать до 100 элементов типа int.

13.Объект int mas; не является контейнером, так как это просто переменная типа int, которая может хранить только одно значение.

14.Доступ к элементам контейнера осуществляется по индексу, используя оператор или метод at(). Например, для доступа к элементу с индексом i в контейнере container необходимо использовать выражение containeri или container.at(i).

15.Доступ к элементам контейнера, реализованного как линейный список, осуществляется по индексу через итераторы. Для доступа к элементу с индексом i необходимо пройти по списку от начала до i-го элемента, используя итераторы. Таким образом, сложность доступа к элементу в линейном списке равна O(n), где n - количество элементов в списке.