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 ПО ЛАБОРАТОРНОЙ РАБОТЕ №11**

Дисциплина: «Основы алгоритмизации и программирования»

Семестр 2

Тема: Последовательные контейнеры библиотеки STL

Вариант 5.

Выполнил работу:

Студент группы РИС-22-2Б

Бехтольт Д.А.

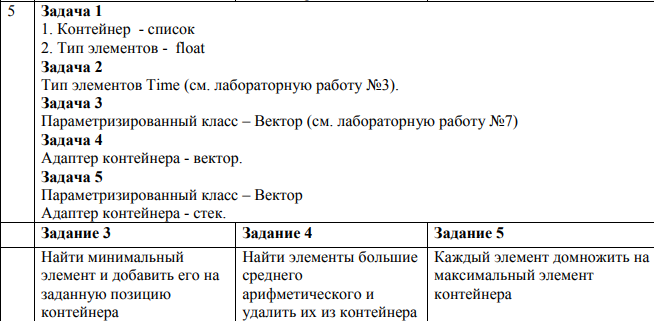
Проверил:

Доцент кафедры ИТАС

Полякова О.А.

**г. Пермь-2023**

**Постановка задачи**



**Код программы**

**Задача 1**

// Реализуем контейнер - список с элементами типа float

#include <iostream>

#include <list>

#include <algorithm>

using namespace std;

void add\_remove\_list() {

list<float> myList = { 1.5f, 2.5f, 3.5f, 4.5f, 5.5f };

myList.push\_front(0.5f); // Добавляем элемент в начало списка

myList.push\_back(6.5f); // Добавляем элемент в конец списка

// Удаляем элемент со значением 3.5f

myList.remove(3.5f);

// Выводим элементы списка

for (auto it = myList.begin(); it != myList.end(); it++) {

cout << \*it << " ";

}

cout << endl;

// Находим минимальный элемент и добавляем его на позицию 2

auto min\_it = min\_element(myList.begin(), myList.end());

myList.insert(next(myList.begin()), \*min\_it);

// Находим среднее арифметическое

float sum = 0.0f;

for (auto it = myList.begin(); it != myList.end(); it++) {

sum += \*it;

}

float average = sum / myList.size();

// Удаляем элементы, большие среднего арифметического

for (auto it = myList.begin(); it != myList.end();) {

if (\*it > average) {

it = myList.erase(it);

}

else {

it++;

}

}

// Домножаем каждый элемент на максимальный элемент

float max\_val = \*max\_element(myList.begin(), myList.end());

for (auto& x : myList) {

x \*= max\_val;

}

// Выводим измененные элементы списка

for (auto it = myList.begin(); it != myList.end(); it++) {

cout << \*it << " ";

}

cout << endl;

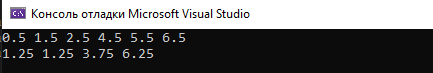
}

int main() {

add\_remove\_list();

return 0;

}



**Задача 2**

#include <iostream>

#include <vector>

#include <algorithm>

// Реализован контейнер, который хранит элементы класса Time

using namespace std;

class Time {

private:

int hours;

int minutes;

int seconds;

public:

Time(int h, int m, int s) {

hours = h;

minutes = m;

seconds = s;

}

friend bool operator<(const Time& t1, const Time& t2) {

if (t1.hours < t2.hours)

return true;

else if (t1.hours == t2.hours && t1.minutes < t2.minutes)

return true;

else if (t1.hours == t2.hours && t1.minutes == t2.minutes && t1.seconds < t2.seconds)

return true;

else

return false;

}

friend bool operator>(const Time& t1, const Time& t2) {

if (t1.hours > t2.hours)

return true;

else if (t1.hours == t2.hours && t1.minutes > t2.minutes)

return true;

else if (t1.hours == t2.hours && t1.minutes == t2.minutes && t1.seconds > t2.seconds)

return true;

else

return false;

}

friend ostream& operator<<(ostream& out, const Time& t) {

out << t.hours << ":" << t.minutes << ":" << t.seconds;

return out;

}

};

// Функция для добавления элементов в контейнер

void addElements(vector<Time>& times) {

times.push\_back(Time(10, 20, 30));

times.push\_back(Time(8, 30, 45));

times.push\_back(Time(12, 0, 0));

times.push\_back(Time(6, 15, 0));

}

// Функция для удаления элементов из контейнера

void deleteElements(vector<Time>& times) {

times.erase(times.begin() + 2); // Удалить элемент с индексом 2

}

// Функция для нахождения максимального и минимального времени в контейнере

void findMinMaxTime(const vector<Time>& times) {

Time minTime = \*min\_element(times.begin(), times.end());

Time maxTime = \*max\_element(times.begin(), times.end());

cout << "Min Time: " << minTime << endl;

cout << "Max Time: " << maxTime << endl;

}

int main() {

vector<Time> times;

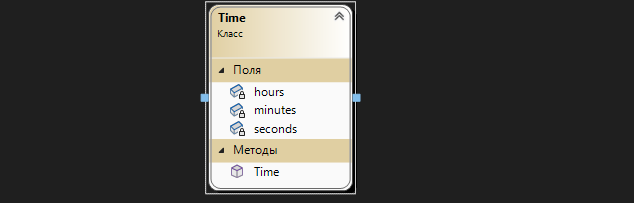
addElements(times);

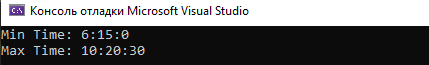
deleteElements(times);

findMinMaxTime(times);

return 0;

}

****

****

**Задача 3**

#include <iostream>

#include <vector>

#include <algorithm>

//Метод fill() заполняет вектор элементами

//метод addMinToPosition() находит минимальный элемент в векторе и добавляет его на заданную позицию

//метод removeElementsLessThan() удаляет все элементы в векторе, которые меньше заданного значения

//метод print() выводит вектор на экран.

template <typename T>

class MyVector {

private:

std::vector<T> data;

public:

MyVector() {}

void fill(int size) {

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

T value;

std::cout << "Enter element " << i + 1 << ": ";

std::cin >> value;

data.push\_back(value);

}

}

void addMinToPosition(int pos) {

T minElement = \*std::min\_element(data.begin(), data.end());

data.insert(data.begin() + pos, minElement);

}

void removeElementsLessThan(T value) {

data.erase(std::remove\_if(data.begin(), data.end(), [&](T x) { return x < value; }), data.end());

}

void print() {

for (const T& x : data) {

std::cout << x << " ";

}

std::cout << std::endl;

}

};

int main() {

MyVector<int> myVector;

myVector.fill(5);

myVector.print();

myVector.addMinToPosition(2);

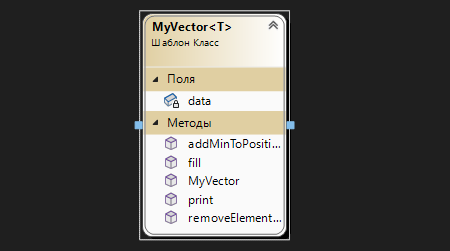
myVector.print();

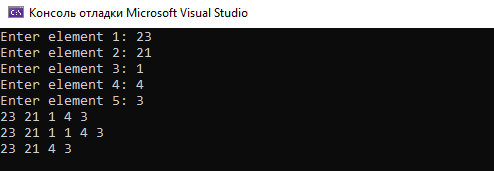
myVector.removeElementsLessThan(3);

myVector.print();

return 0;

}





**Задача 4**

#include <iostream>

#include <vector>

#include <algorithm>

// пользовательский тип MyType, который имеет единственное поле value типа int, операторы сравнения < и >, которые используются при удалении элементов вектора.

// getAverage() и removeElementsGreaterThanAverage() выполняются задачи по нахождению среднего арифметического и удалению элементов больших среднего арифметического соответственно.

//В main() создается вектор r и заполняется элементами.

//вызывается функция removeElementsGreaterThanAverage() для удаления элементов, больших среднего арифметического, и выводится измененный вектор.

//

// определяем пользовательский тип

struct MyType {

int value;

MyType(int value) : value(value) {}

bool operator<(const MyType& other) const {

return value < other.value;

}

bool operator>(const MyType& other) const {

return value > other.value;

}

friend std::ostream& operator<<(std::ostream& os, const MyType& obj) {

os << obj.value;

return os;

}

};

// определяем глобальные функции для выполнения задания

double getAverage(const std::vector<MyType>& data) {

double sum = 0;

for (const MyType& x : data) {

sum += x.value;

}

return sum / data.size();

}

void removeElementsGreaterThanAverage(std::vector<MyType>& data) {

double average = getAverage(data);

data.erase(std::remove\_if(data.begin(), data.end(), [&](const MyType& x) { return x > average; }), data.end());

}

int main() {

// создаем вектор и заполняем его элементами

std::vector<MyType> myVector;

myVector.push\_back(MyType(5));

myVector.push\_back(MyType(3));

myVector.push\_back(MyType(8));

myVector.push\_back(MyType(4));

myVector.push\_back(MyType(6));

// выводим на экран исходный вектор

std::cout << "Original vector: ";

for (const MyType& x : myVector) {

std::cout << x << " ";

}

std::cout << std::endl;

// удаляем элементы большие среднего арифметического

removeElementsGreaterThanAverage(myVector);

// выводим на экран измененный вектор

std::cout << "Modified vector: ";

for (const MyType& x : myVector) {

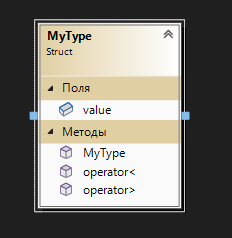
std::cout << x << " ";

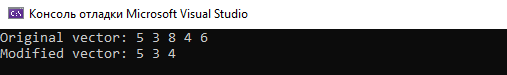
}

std::cout << std::endl;

return 0;

}





**Задача 5**

#include <iostream>

#include <vector>

#include <stack>

#include <algorithm>

template <typename T, template <typename, typename> class Container>

class Vector {

public:

Vector() {}

void add(T element) {

container.push(element);

}

void remove() {

container.pop();

}

// В методе multiply\_by\_max мы используем функцию std::max\_element, чтобы найти максимальный элемент контейнера, а затем умножаем каждый элемент на этот максимум.

//void multiply\_by\_max() {

T max\_element = \*std::max\_element(std::begin(container), std::end(container));

for (auto& element : container) {

element \*= max\_element;

}

}

void print() {

while (!container.empty()) {

std::cout << container.top() << " ";

container.pop();

}

std::cout << std::endl;

}

private:

Container<T, std::deque<T>> container;

};

int main() {

Vector<int, std::stack> v;

v.add(5);

v.add(10);

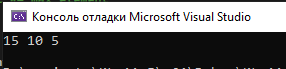
v.add(15);

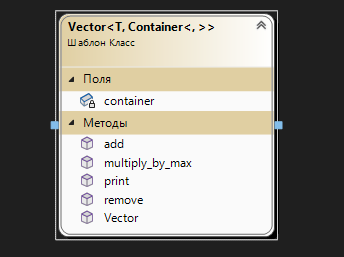
//v.multiply\_by\_max();

v.print();

return 0;

}





**Ответы на контрольные вопросы**

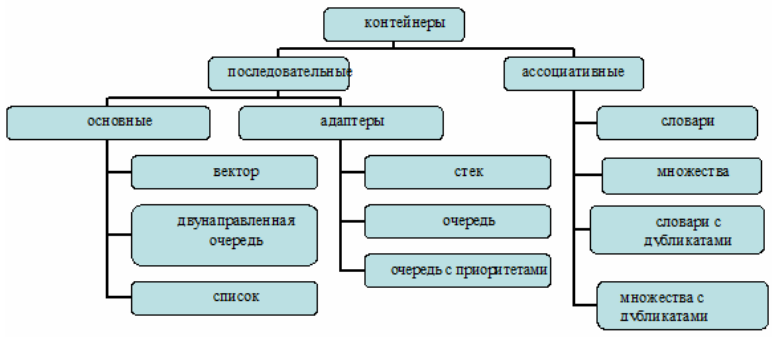
1. Из каких частей состоит библиотека STL?

STL – Standard Template Library, стандартная библиотека шаблонов

состоит из двух основных частей: набора контейнерных классов и набора обобщенных алгоритмов. Контейнеры — это объекты, содержащие другие однотипные объекты. Обобщенные алгоритмы реализуют большое количество процедур, применимых к контейнерам: поиск, сортировку, слияние и т. п.

2. Какие типы контейнеров существуют в STL?

Последовательные контейнеры (векторы (vector), списки (list) и двусторонние очереди (deque)), ассоциативные контейнеры (словари (mар), словари с дубликатами (multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset)), есть еще специализированные контейнеры (или адаптеры контейнеров), реализованные на основе базовых — стеки (stack), очереди (queue) и очереди с приоритетами (priority\_queue)



3. Что нужно сделать для использования контейнера STL в своей программе?

Для использования контейнера в программе необходимо включить в нее соответствующий заголовочный файл. Тип объектов, сохраняемых в контейнере, задается с помощью аргумента шаблона, например:

#include <vector>

#include <list>

4. Что представляет собой итератор?

Итераторы (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива.

5. Какие операции можно выполнять над итераторами?

С итераторами можно работать так же, как с указателями. К ним можно применить операции \*, инкремента, декремента. Присваивание одного итератора другому. Сравнение итераторов на равенство и неравенство (== и !=).

6. Каким образом можно организовать цикл для перебора контейнера с использованием итератора?

for (iterator it = v.begin(); it != v.end(), ++it) cout << v[it] << endl; //vector

7. Какие типы итераторов существуют?

Существует пять типов итераторов:

* Итераторы ввода (input iterator) поддерживают операции равенства, разыменования и инкремента. ==, !=, \*i, ++i, i++, \*i++
* Итераторы вывода (output iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента. ++i, i++, \*i = t, \*i++ = t
* Однонаправленные итераторы (forward iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание. ==, !=, =, \*i, ++i, i++, \*i++
* Двунаправленные итераторы (bidirectional iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (--i, i--, \*i--), что позволяет им проходить контейнер в обоих направлениях.
* Итераторы произвольного доступа (random access iterator) обладают всеми свойствами bidirectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу. i += n, i + n, i -= n, i - n, i1 - i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2

В STL также поддерживаются обратные итераторы (reverse iterators). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

8. Перечислить операции и методы общие для всех контейнеров.

Операции равенства (==), неравенства (!=), операция присваивания (=), clear, insert, erase, size\_type size() const, size\_type max\_size() const, bool empty() const, iterator begin(), iterator end(), reverse\_iterator begin(), reverse\_iterator end()

9. Какие операции являются эффективными для контейнера vector? Почему?

Контейнер вектор является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at. Происходит это потому, что массив - это последовательно занятая память, так что доступ к любому элементу происходит быстро.

10. Какие операции являются эффективными для контейнера list? Почему?

Контейнер список организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции элемента в списке, поскольку не требуется заново перевыделять память, достаточно переобозначить связи с предыдущим и последующим элементом.

11. Какие операции являются эффективными для контейнера deque? Почему?

Контейнер двусторонняя очередь во многом аналогичен вектору, элементы хранятся в непрерывной области памяти. Но в отличие от вектора двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего). Дек является сочетанием вектора и списка, поэтому при добавлении элементов в начало создается новый массив, конец которого по принципа списка указывает на изначальный массив. Аналогично с добавлением элементов в конец.

12. Перечислить методы, которые поддерживает последовательный контейнер vector.

push\_back(), pop\_back(), insert, erase, [], at, swap, clear()

13. Перечислить методы, которые поддерживает последовательный контейнер list.

push\_back(), pop\_back(), push\_front(), pop\_back(), insert(), erase, swap, clear(), splice

14. Перечислить методы, которые поддерживает последовательный контейнер deque.

push\_back(), push\_front(), pop\_back, pop\_front, insert, erase, [] ,at

15. Задан контейнер vector. Как удалить из него элементы со 2 по 5?

for (int i = 2; i <= 5; ++i) v.erase(v.begin()+i);

16. Задан контейнер vector. Как удалить из него последний элемент?

v.erase(--v.end()); //итератор end() указывает на ячейку памяти после последнего элемента

17. Задан контейнер list. Как удалить из него элементы со 2 по 5?

list<int>::iterator beg = l.begin(); advance(beg, 2);

list<int>::iterator end = l.begin(); advance(end, 6); //удаляем включительно

l.erase(beg, end);

18. Задан контейнер list. Как удалить из него последний элемент?

l.erase(--l.end());

19. Задан контейнер deque. Как удалить из него элементы со 2 по 5?

auto beg = q.begin(); advance(beg, 2);

auto end = q.begin(); advance(end, 6);

q.erase(beg, end);

20. Задан контейнер deque. Как удалить из него последний элемент?

q.erase(--q.end());

21. Написать функцию для печати последовательного контейнера с использованием итератора.

void print(deque<int>q) {

for (deque<int>::iterator elem = q.begin(); elem != q.end(); ++elem)

cout << \*elem << " ";

}

22. Что представляют собой адаптеры контейнеров?

Специализированные последовательные контейнеры — стек, очередь и очередь с приоритетами — не являются самостоятельными контейнерными классами, а реализованы на основе рассмотренных выше классов, поэтому они называются адаптерами контейнеров.

23. Чем отличаются друг от друга объявления stack<int> s и stack<int, list<int> > s?

По умолчанию для стека прототипом является класс deque.

Объявление stack<int> s создает стек на базе двусторонней очереди (по

умолчанию). Если по каким-то причинам нас это не устраивает, и мы хотим создать стек на базе списка, то объявление будет выглядеть следующим образом: stack<int, list<int> > s;

24. Перечислить методы, которые поддерживает контейнер stack.

push () - добавление в конец; pop () - удаление из конца; top () - получение текущего элемента стека; empty() - проверка пустой стек или нет; size () – получение размера стека.

25. Перечислить методы, которые поддерживает контейнер queue.

push () – добавление в конец очереди; pop () – удаление из начала очереди; front () – получение первого элемента очереди; back()- получение последнего элемента очереди; empty () – проверка пустая очередь или нет; size() – получение размера очереди.

26. Чем отличаются друг от друга контейнеры queue и priority\_queue?

Шаблонный класс priority\_queue (заголовочный файл <queue>)

поддерживает такие же операции, как и класс queue, но реализация класса возможна либо на основе вектора (реализация по умолчанию), либо на основе списка. Очередь с приоритетами отличается от обычной очереди тем, что для извлечения выбирается максимальный элемент из хранимых в контейнере. Поэтому после каждого изменения состояния очереди максимальный элемент из оставшихся сдвигается в начало контейнера.

27. Задан контейнер stack. Как удалить из него элемент с заданным номером?

stack<int> copy;

int counter = 0;

while (counter != 2) {

copy.push(s.top());

s.pop();

++counter;

}

while (counter != 6) { //удаление не включительно

s.pop();

++counter;

}

while (!copy.empty()) {

s.push(copy.top());

copy.pop();

}

28. Задан контейнер queue. Как удалить из него элемент с заданным номером?

//перемещение всех элементов до удаляемого в конец

for (int i = 1; i < num; i++) {

push(qwe, qwe->head->key);

pop(qwe); }

pop(qwe); //удаление элемента

//возвращение очереди в прежнее состояние

//первый элемент снова ставится в начало

for (int i = 0; i < (qwe->size) - (num + 2 \* k); i++) {

push(qwe, qwe->head->key);

pop(qwe); }

29. Написать функцию для печати контейнера stack с использованием итератора.

void print(stack<int> s) {

int size = s.size();

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

cout << s.top() << " ";

s.pop(); }

}

30. Написать функцию для печати контейнера queue с использованием итератора.

void print(queue<int> s) {

int size = s.size();

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

cout << s.front() << " ";

s.pop(); }

}