Министерство образования и науки Российской Федерации

Федеральное государственное образовательное автономное учреждение высшего образования

"Пермский национальный исследовательский политехнический университет"

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №10

Дисциплина: Основы алгоритмизации и программирования

Тема: Последовательные контейнеры библиотеки STL.

|  |  |
| --- | --- |
| Выполнил работу | |
| Студент группы РИС-22-1б | |
| Бреднев М.П. | |
|  | |
| Проверил работу | |
| Доцент кафедры ИТАС | |
| Полякова О.А. | |
|  | |

Пермь – 2023

**Анализ предметной области**

**Постановка задачи**

Задача 1:

1. Создать последовательный контейнер.
2. Заполнить его элементами стандартного типа (тип указан в варианте).
3. Добавить элементы в соответствии с заданием.
4. Удалить элементы в соответствии с заданием.
5. Выполнить задание варианта для полученного контейнера.
6. Выполнение всех заданий оформить в виде глобальных функций.

Задача 2:

1. Создать последовательный контейнер.
2. Заполнить его элементами пользовательского типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.
3. Добавить элементы в соответствии с заданием.
4. Удалить элементы в соответствии с заданием.
5. Выполнить задание варианта для полученного контейнера.
6. Выполнение всех заданий оформить в виде глобальных функций.

Задача 3:

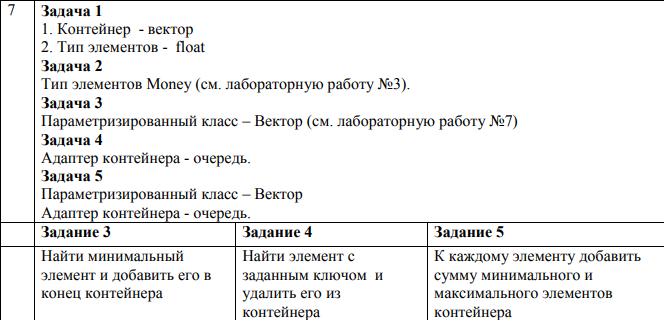
1. Создать параметризированный класс, используя в качестве контейнера последовательный контейнер.
2. Заполнить его элементами.
3. Добавить элементы в соответствии с заданием.
4. Удалить элементы в соответствии с заданием.
5. Выполнить задание варианта для полученного контейнера.
6. Выполнение всех заданий оформить в виде методов параметризированного класса.

Задача 4:

1. Создать адаптер контейнера.
2. Заполнить его элементами пользовательского типа (тип указан в варианте). Для пользовательского типа перегрузить необходимые операции.
3. Добавить элементы в соответствии с заданием.
4. Удалить элементы в соответствии с заданием.
5. Выполнить задание варианта для полученного контейнера.
6. Выполнение всех заданий оформить в виде глобальных функций.

Задача 5:

1. Создать параметризированный класс, используя в качестве контейнера адаптер контейнера.
2. Заполнить его элементами.
3. Добавить элементы в соответствии с заданием.
4. Удалить элементы в соответствии с заданием.
5. Выполнить задание варианта для полученного контейнера.
6. Выполнение всех заданий оформить в виде методов параметризированного класса.

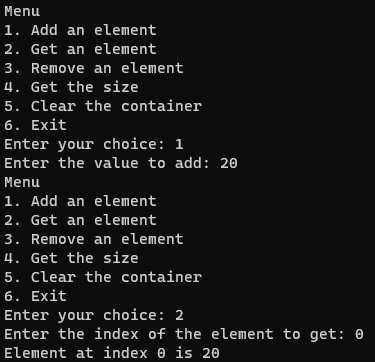
****

**Анализ задачи**

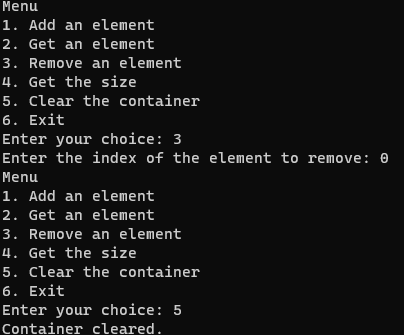
Класс Money уже реализован в 3 лабораторной работе, поэтому за основу возьмем его. У него уже все методы, которые необходимы – перегружены.

Далее необходимо выполнить задач и к каждой задаче – 3 задания. Для выполнения будут использовать STL библиотеки list и queue.

**Тестирование программы**

****

*Рис. 1 – Работа программы.*

**

*Рис. 2 – Работа программы.*

**Заключение**

Была разработана программа, которая взаимодействует с контейнерами STL библиотеки и использует их для хранения и упорядочивания данных.

# Ответы на контрольные вопросы

1. Из каких частей состоит библиотека STL.

Библиотека STL (Standard Template Library) включает три части: контейнеры, алгоритмы и итераторы.

1. Какие типы контейнеров существуют в STL.

* Последовательные контейнеры: vector, deque, list, forward\_list, array
* Ассоциативные контейнеры: set, multiset, map, multimap
* Неупорядоченные ассоциативные контейнеры: unordered\_set, unordered\_multiset, unordered\_map, unordered\_multimap
* Адаптеры контейнеров: stack, queue, priority\_queue

1. Чту нужно сделать для использования контейнера STL в своей программе.

Для использования контейнеров STL в своей программе необходимо подключить заголовочный файл, соответствующий нужному контейнеру. Например, для использования вектора (vector) нужно подключить заголовочный файл <vector> с помощью директивы #include <vector>. Также, необходимо использовать пространство имен std.

1. Что представляет собой итератор?

Итератор - это объект, который позволяет последовательно перебирать элементы контейнера STL (и не только STL). Он представляет собой абстракцию, которая инкапсулирует конкретную реализацию обхода элементов контейнера. Итераторы могут быть использованы для доступа к элементам контейнера, изменения их значений и удаления. Они могут также перемещаться по контейнеру вперед и назад. Кроме того, алгоритмы STL используют итераторы для работы с контейнерами, что позволяет создавать универсальные алгоритмы, работающие с любыми контейнерами STL.

1. Какие операции можно выполнять над итераторами.

Итераторы поддерживают несколько базовых операций:

* Операция разыменования (\*) - получение значения, на которое указывает итератор.
* Операция инкремента (++) - перемещение итератора на следующий элемент в контейнере.
* Операция декремента (--) - перемещение итератора на предыдущий элемент в контейнере (не поддерживается всеми итераторами).
* Операции сравнения (==, !=, <, >, <=, >=) - сравнение итераторов между собой.
* Операция присваивания (=) - присваивание одного итератора другому.

В зависимости от типа контейнера, итератор может поддерживать дополнительные операции, например, операцию вставки (insert) или удаления (erase) элементов контейнера.

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

Для перебора контейнера с использованием итератора можно организовать цикл, используя два итератора - начальный и конечный. Начальный итератор указывает на первый элемент контейнера, а конечный итератор указывает на элемент, следующий за последним элементом контейнера. Таким образом, цикл перебирает все элементы контейнера, начиная с первого и заканчивая последним (не включая его).

vector<int> v = {1, 2, 3, 4, 5};

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

std::cout << \*it << " ";

}

В данном случае, в цикле используется auto для определения типа итератора, начальный итератор получается с помощью функции begin() контейнера, а конечный итератор - с помощью функции end(). В теле цикла происходит разыменование итератора для получения значения элемента контейнера, на который он указывает.

1. Какие типы итераторов существуют?

В STL существует пять основных типов итераторов, отличающихся своими свойствами и возможностями:

Input iterator - итератор ввода, который позволяет перебирать элементы контейнера только в одном направлении и только для чтения.

Output iterator - итератор вывода, который позволяет записывать значения в контейнер только в одном направлении.

Forward iterator - итератор прямого доступа, который позволяет перебирать элементы контейнера только в одном направлении и изменять значения элементов.

Bidirectional iterator - итератор двунаправленного доступа, который поддерживает все операции Forward iterator и добавляет возможность перемещения итератора назад.

Random access iterator - итератор произвольного доступа, который поддерживает все операции Bidirectional iterator и добавляет возможность перемещения итератора на произвольное число шагов вперед и назад.

1. Перечислить операции и методы общие для всех контейнеров.

* begin() - метод, возвращающий итератор на первый элемент контейнера.
* end() - метод, возвращающий итератор на элемент, следующий за последним элементом контейнера.
* size() - метод, возвращающий количество элементов в контейнере.
* empty() - метод, возвращающий true, если контейнер пустой, и false в противном случае.
* clear() - метод, удаляющий все элементы из контейнера.
* insert() - метод, вставляющий новый элемент в контейнер.
* erase() - метод, удаляющий элемент из контейнера.
* find() - метод, возвращающий итератор на элемент, найденный по ключу или значению.

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

Контейнер vector реализован в виде динамического массива, поэтому для него эффективными операциями являются:

* Доступ к произвольному элементу с помощью оператора [] - O(1) по времени, так как элементы хранятся последовательно в памяти и доступ к любому элементу происходит за константное время.
* Вставка нового элемента в конец контейнера методом push\_back() - O(1) по амортизированному времени, так как вектор хранит элементы в памяти последовательно и может использовать выделенную под него память для хранения новых элементов. Если места для новых элементов недостаточно, то выделится новая область памяти и произойдет копирование всех элементов в нее, что займет O(N) времени, где N - текущее количество элементов в контейнере.
* Удаление последнего элемента из контейнера методом pop\_back() - O(1) по времени, так как это просто удаление элемента из конца массива.

Эти операции эффективны для контейнера vector, так как позволяют работать с ним за константное время.

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

Контейнер list реализован в виде двусвязного списка, поэтому для него эффективными операциями являются:

* Вставка нового элемента в произвольное место контейнера методом insert() - O(1) по времени, так как при вставке элемента в список не требуется перемещать другие элементы, достаточно просто обновить ссылки на следующий и предыдущий элементы.
* Удаление элемента из произвольного места контейнера методом erase() - O(1) по времени, так как при удалении элемента из списка не требуется перемещать другие элементы, достаточно просто обновить ссылки на следующий и предыдущий элементы.
* Обход элементов контейнера в прямом и обратном направлении с помощью итераторов - O(N) по времени, где N - количество элементов в списке. Обход списка в прямом направлении выполняется за линейное время, так как каждый элемент списка имеет ссылку на следующий элемент, и можно легко перейти от одного элемента к другому.

Эти операции эффективны для контейнера list, так как позволяют работать с ним за константное время

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

Контейнер deque (double-ended queue) представляет собой двустороннюю очередь, которая состоит из блоков фиксированного размера. Для него эффективными операциями являются:

* Вставка и удаление элементов с начала и конца очереди с помощью методов push\_front(), pop\_front(), push\_back() и pop\_back() - O(1) по времени, так как эти операции выполняются непосредственно над блоками, без необходимости переноса элементов из одного блока в другой.
* Доступ к произвольному элементу с помощью оператора [] и метода at() - O(1) по времени, так как элементы хранятся в блоках, и к каждому элементу можно получить доступ за константное время.
* Итерация с помощью итераторов - O(N) по времени, где N - количество элементов в deque. Итерация выполняется за линейное время, так как элементы хранятся последовательно в блоках.

Эти операции эффективны для контейнера deque, так как позволяют выполнять основные операции за константное время

1. Перечислите методы, которые поддерживает последовательный контейнер vector.

* push\_back() - добавляет элемент в конец вектора
* pop\_back() - удаляет элемент с конца вектора
* size() - возвращает количество элементов в векторе
* capacity() - возвращает количество элементов, которые могут быть сохранены в векторе без перевыделения памяти
* resize() - изменяет размер вектора, при необходимости добавляя новые элементы или удаляя существующие
* reserve() - изменяет емкость вектора, выделяя достаточно памяти для хранения указанного количества элементов
* clear() - удаляет все элементы из вектора
* empty() - проверяет, является ли вектор пустым
* operator[] - обращение к элементам вектора по индексу
* at() - обращение к элементам вектора по индексу с проверкой на выход за пределы
* front() - возвращает первый элемент вектора
* back() - возвращает последний элемент вектора
* data() - возвращает указатель на начало вектора для чтения и записи
* begin() - возвращает итератор, указывающий на первый элемент вектора
* end() - возвращает итератор, указывающий на элемент, следующий за последним элементом вектора.

1. Перечислите методы, которые поддерживает последовательный контейнер list.

* push\_front() - добавляет элемент в начало списка
* push\_back() - добавляет элемент в конец списка
* pop\_front() - удаляет первый элемент списка
* pop\_back() - удаляет последний элемент списка
* size() - возвращает количество элементов в списке
* empty() - проверяет, является ли список пустым
* clear() - удаляет все элементы из списка
* front() - возвращает первый элемент списка
* back() - возвращает последний элемент списка
* insert() - добавляет элементы в список по указанной позиции
* erase() - удаляет элементы из списка по указанной позиции или диапазону
* remove() - удаляет все элементы списка, равные заданному значению
* reverse() - изменяет порядок элементов в списке на обратный
* sort() - сортирует элементы списка по возрастанию (по умолчанию) или по заданному компаратору
* merge() - объединяет два отсортированных списка в один отсортированный список

1. Перечислите методы, которые поддерживает последовательный контейнер deque.

* push\_front() - добавляет элемент в начало очереди
* push\_back() - добавляет элемент в конец очереди
* pop\_front() - удаляет первый элемент очереди
* pop\_back() - удаляет последний элемент очереди
* size() - возвращает количество элементов в очереди
* empty() - проверяет, является ли очередь пустой
* clear() - удаляет все элементы из очереди
* front() - возвращает первый элемент очереди
* back() - возвращает последний элемент очереди
* at() - возвращает элемент очереди по заданному индексу
* insert() - добавляет элементы в очередь по указанной позиции
* erase() - удаляет элементы из очереди по указанной позиции или диапазону
* resize() - изменяет размер очереди
* swap() - обменивает содержимое двух очередей

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

vector<int> myVector = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto it = myVector.begin() + 1; // Итератор, указывающий на второй элемент

auto endIt = myVector.begin() + 5; // Итератор, указывающий на шестой элемент myVector.erase(it, endIt);

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

vector<int> myVector = {1, 2, 3, 4, 5};

myVector.pop\_back(); // Удаляем последний элемент

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

list<int> myList = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto start = std::next(myList.begin(), 1); // итератор на второй элемент

auto end = std::next(myList.begin(), 5); // итератор на шестой элемент

myList.erase(start, end); // удаляем элементы со второго по пятый

1. Задан контейнер list. Как удалить из него элементы последний элемент?

list<int> myList = {1, 2, 3, 4, 5};

myList.pop\_back(); // удаляем последний элемент

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

deque<int> myDeque = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// итераторы передаем сразу в метод

myDeque.erase(myDeque.begin() + 1, myDeque.begin() + 5);

1. Задан контейнер deque. Как удалить из него элементы последний элемент?

deque<int> myDeque = {1, 2, 3, 4, 5};

myDeque.pop\_back();

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

template<typename T>

void print\_container(const T& container) {

// Проходим по контейнеру с помощью итераторов

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

std::cout << \*it << " "; //Выводим текущий элемент, разыменовывая его

}

}

1. Что представляют собой адаптеры контейнеров?

Адаптеры контейнеров - это шаблонные классы, которые позволяют изменять поведение существующих контейнеров STL без изменения их основной структуры. Они обычно представляют собой обертки над существующими контейнерами, добавляющие новые функциональные возможности или изменяющие существующие.

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

Объявление stack<int> s создает стек, который использует vector<int> в качестве контейнера-адаптера, тогда как объявление stack<int, list<int>> s создает стек, который использует list<int> в качестве контейнера-адаптера. Таким образом, различие между ними заключается в типе контейнера, используемого для хранения элементов стека.

1. Перечислите методы, которые поддерживает контейнер stack.

* Push(): добавляет элемент на вершину стека.
* Pop(): удаляет элемент с вершины стека.
* Top(): возвращает элемент, находящийся на вершине стека.
* Empty(): возвращает true, если стек пуст, и false в противном случае.
* Size(): возвращает количество элементов в стеке.

1. Перечислите методы, которые поддерживает контейнер queue.

* Push(): добавляет элемент в конец очереди.
* Pop(): удаляет элемент из начала очереди.
* Front(): возвращает элемент, находящийся в начале очереди.
* Back(): возвращает элемент, находящийся в конце очереди.
* Empty(): возвращает true, если очередь пуста, и false в противном случае.
* Size(): возвращает количество элементов в очереди.

1. Чем отличается друг от друга контейнеры queue и priority\_queue?

Контейнеры queue и priority\_queue относятся к адаптерам контейнеров и представляют собой структуры данных для хранения элементов в определенном порядке. Основное отличие между ними заключается в том, как элементы извлекаются из контейнера:

queue - используется для хранения элементов в порядке, в котором они были добавлены, и извлекает элементы по принципу "первым пришел - первым вышел".

priority\_queue - используется для хранения элементов в отсортированном порядке, и извлекает элементы с наибольшим приоритетом (в соответствии с заданной функцией сравнения).

Таким образом, queue поддерживает методы push(), pop() и front(), а priority\_queue - методы push(), pop() и top().

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

template <class T>

void del(stack<T>& st, int index)

{

stack<T> temp;

for (int i = 0; i < index; i++)

{

temp.push(st.top());

st.pop();

}ss

st.pop();

while (temp.size())

{

st.push(temp.top());

temp.pop();

}

}

Но так делать не надо, потому что stack не предназначен для того, чтобы из него удалялись элементы не по порядку. В данном случае будет гораздо логичнее применить структуру данных список или любой контейнер STL.

1. Задан контейнер deque. Как удалить из него элемент с заданным номером?

std::deque<int> dq{1, 2, 3, 4, 5};

int index = 2;

auto it = std::next(dq.begin(), index);

dq.erase(it);

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

Это невозможно, так как у контейнера stack даже итераторов нет.

Вот пример как можно посмотреть все, что лежит в stack, но так делать тоже не надо, потому что это ломает весь смысл работы с данным контейнером, гораздо логичнее будет взять любой другой контейнер, например отлично подойдет однонаправленный список.

template <class T>

void printStack(stack<T> s)

{

while (!s.empty()) {

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

s.pop();

}}

1. Написать функцию для печати контейнера deque с использованием итератора.

template <class T>

void printDeque(deque<T> dq)

{

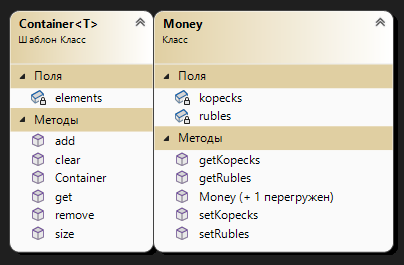
for (auto i = dq.begin(); i != dq.end(); ++i)

cout << \*i << endl;

}

**Приложения**

UML-диаграмма:



Приложение Б – код программы

Main.cpp:

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

#include "Money1.h"

#include "Container1.h"

using namespace std;

// Task 1

void add\_min\_to\_end(vector<float>& vec) {

float min\_val = \*min\_element(vec.begin(), vec.end());

vec.push\_back(min\_val);

}

void remove\_value(vector<float>& vec, float val) {

vec.erase(remove(vec.begin(), vec.end(), val), vec.end());

}

void add\_sum\_of\_min\_max(vector<float>& vec) {

float min\_val = \*min\_element(vec.begin(), vec.end());

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

float sum = min\_val + max\_val;

for (float& elem : vec) {

elem += sum;

}

}

void add\_min\_value(deque<Money>& v) {

Money minVal;

bool first = true;

for (const Money& m : v) {

if (first || m.getRubles() < minVal.getRubles() ||

(m.getRubles() == minVal.getRubles() && m.getKopecks() < minVal.getKopecks())) {

minVal = m;

first = false;

}

}

v.push\_back(minVal);

}

void remove\_money(queue<Money>& q, long rubles, int kopecks) {

queue<Money> temp;

while (!q.empty()) {

if (!(q.front().getRubles() == rubles && q.front().getKopecks() == kopecks)) {

temp.push(q.front());

}

q.pop();

}

q = temp;

}

void add\_sum\_of\_min\_max(deque<Money>& q) {

Money minVal;

Money maxVal;

bool first = true;

for (const Money& m : q) {

if (first || m.getRubles() < minVal.getRubles() ||

(m.getRubles() == minVal.getRubles() && m.getKopecks() < minVal.getKopecks())) {

minVal = m;

}

if (first || m.getRubles() > maxVal.getRubles() ||

(m.getRubles() == maxVal.getRubles() && m.getKopecks() > maxVal.getKopecks())) {

maxVal = m;

}

first = false;

}

for (Money& m : q) {

m.setRubles(m.getRubles() + minVal.getRubles() + maxVal.getRubles());

m.setKopecks(m.getKopecks() + minVal.getKopecks() + maxVal.getKopecks());

}

}

int main()

{

Container<int> container;

int choice;

int value;

int index;

do

{

cout << "Menu" << endl;

cout << "1. Add an element" << endl;

cout << "2. Get an element" << endl;

cout << "3. Remove an element" << endl;

cout << "4. Get the size" << endl;

cout << "5. Clear the container" << endl;

cout << "6. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter the value to add: ";

cin >> value;

container.add(value);

break;

case 2:

cout << "Enter the index of the element to get: ";

cin >> index;

cout << "Element at index " << index << " is " << container.get(index) << endl;

break;

case 3:

cout << "Enter the index of the element to remove: ";

cin >> index;

container.remove(index);

break;

case 4:

cout << "Size of container is " << container.size() << endl;

break;

case 5:

container.clear();

cout << "Container cleared." << endl;

break;

case 6:

cout << "Exiting..." << endl;

break;

default:

cout << "Invalid choice. Please enter a valid choice." << endl;

break;

}

} while (choice != 6);

return 0;

}

Container1.h:

#pragma once

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

template<typename T>

class Container

{

public:

Container() {}

void add(T element) {

elements.push\_back(element);

}

T get(int index) {

return elements.at(index);

}

void remove(int index) {

elements.erase(elements.begin() + index);

}

int size() {

return elements.size();

}

void clear() {

elements.clear();

}

private:

vector<T> elements;

};

Money.h

#pragma once

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

// task 2

class Money {

public:

Money() : rubles(0), kopecks(0) {}

Money(long r, int k) : rubles(r), kopecks(k) {}

long getRubles() const { return rubles; }

int getKopecks() const { return kopecks; }

void setRubles(long r) { rubles = r; }

void setKopecks(int k) { kopecks = k; }

friend bool operator==(const Money& lhs, const Money& rhs) {

return lhs.rubles == rhs.rubles && lhs.kopecks == rhs.kopecks;

}

private:

long rubles;

int kopecks;

};