Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

Пермский национальный исследовательский политехнический университет

Электротехнический факультет

Кафедра информационных технологий и автоматизированных систем

**ОТЧЕТ**

**о работе по информатике**

Семестр: 2

На тему: Лабораторная работа №6: «АТД. Контейнеры».

**Вариант 13**

Выполнил студент ИВТ-22-2б:

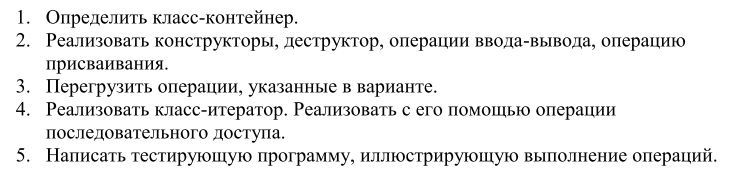
Коняев Александр Сергеевич

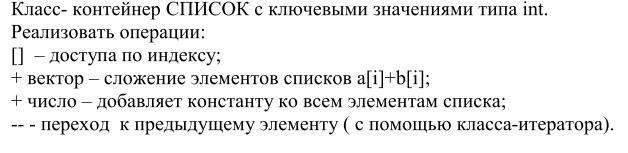
Проверил доцент кафедры ИТАС:

Полякова Ольга Андреевна

Пермь 2023

**Постановка задачи**





**Анализ задачи**

Класс Node

Класс LinkedList

* Конструктор с параметрами
* Конструктор без параметров
* Конструктор копирования
* Метод clear
* Метод push\_back
* Метод pop\_back
* Метод get\_At
* Метод insert
* Метод erase
* Метод print
* Перегруженные операторы: =, [], +, <<, >>

Класс Iterator

* Конструктор без параметров
* Конструктор копирования
* Перегруженные операторы: ==, !=, ++, --, \*()

**UML – диаграмма**

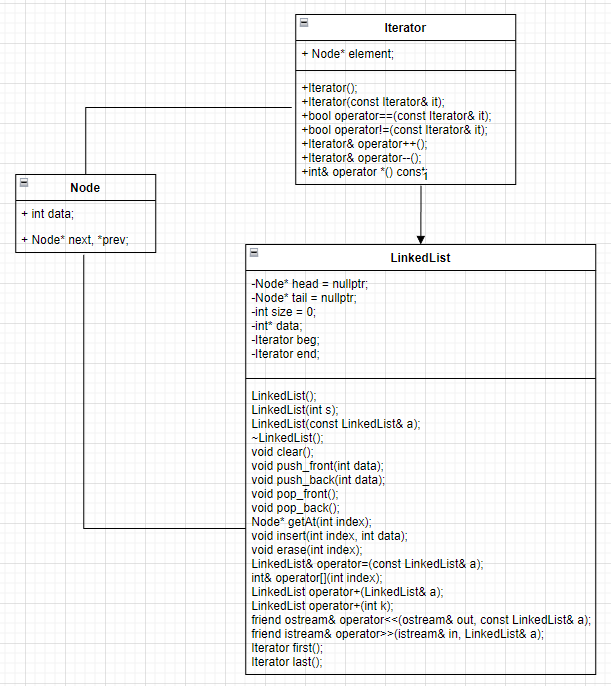


Рисунок 1 – UML-диаграмма.

**Код программы.**

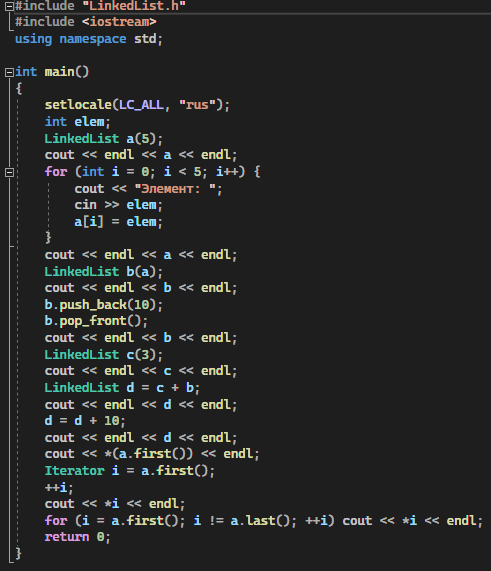


Рисунок 2 – Функция main

**Вывод программы.**

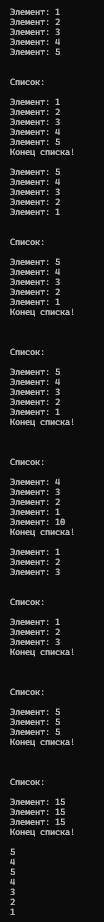
****

Рисунок 3 – Вывод программы

**Ответы на вопросы**

1. Абстрактный тип данных (АТД) — это математический конструкт, который описывает некоторый набор значений и операций, которые могут быть выполнены над этими значениями. АТД представляет собой абстракцию, которая определяет интерфейс, скрывающий реализацию. Таким образом, пользователь может использовать АТД, не зная, как он реализован.

Примеры АТД:

Список: абстрактный тип данных, представляющий собой набор упорядоченных элементов, в котором возможно добавление, удаление и получение элементов по индексу.

Очередь: АТД, реализующий очередь, где элементы добавляются в конец и удаляются из начала.

Множество: АТД, представляющий собой набор уникальных элементов, в котором возможно добавление, удаление и проверка наличия элементов.

Дерево: АТД, представляющий собой иерархическую структуру, в которой каждый элемент имеет родительский узел и ноль или более дочерних узлов.

1. Абстракции через параметризацию - это когда АТД определяет интерфейс, в котором операции и значения зависят от параметров, которые передаются в качестве аргументов. Примеры:

Обобщенный список (Generic List): АТД, который может содержать элементы любого типа, и все операции списка (добавление, удаление, получение по индексу) могут быть выполнены независимо от типа элемента.

Ассоциативный массив (Associative Array): АТД, который представляет собой коллекцию пар ключ-значение, где каждый ключ может быть любого типа, а значение зависит от этого типа.

Обобщенная очередь (Generic Queue): АТД, представляющий собой очередь, которая может содержать элементы любого типа, но операции добавления и удаления элементов будут зависеть от типа этих элементов.

Бинарное дерево поиска (Binary Search Tree): АТД, которое определяет интерфейс, зависящий от типа ключа, используемого для упорядочивания элементов в дереве. Это позволяет реализовать поиск элементов и выполнение других операций над деревом, независимо от конкретной реализации упорядочивания.

1. Абстракции через спецификацию - это когда АТД определяет интерфейс, в котором операции и значения зависят от набора определенных свойств или условий. Примеры:

Стек с ограниченной ёмкостью (Bounded Stack): АТД, который задает стек с ограниченной ёмкостью. Операции добавления и удаления элементов определены только в том случае, если стек еще не заполнен (до максимальной ёмкости). Это основывается на определенном свойстве ёмкости стека.

Очередь с приоритетом (Priority Queue): АТД, который определяет интерфейс очереди с приоритетом, в которой элементы добавляются и извлекаются с учетом их приоритета. Операции добавления и извлечения элементов определены только в том случае, если очередь не пустая. Это основывается на условии наличия элементов в очереди.

Граф с фиксированным количеством вершин (Fixed Size Graph): АТД, который определяет интерфейс графа с фиксированным количеством вершин. Операции добавления новых вершин или ребер недоступны после инициализации графа и заданного количества вершин. Это основывается на свойстве фиксированного количества вершин.

Код для реализации примера "Стек с ограниченной ёмкостью":

class BoundedStack<T> {

constructor(capacity) {

this.capacity = capacity;

this.array = new Array(capacity);

this.pointer = -1;

}

push(element) {

if (this.pointer < this.capacity - 1) {

this.pointer++;

this.array[this.pointer] = element;

return true;

} else {

return false;

}

}

pop() {

if (this.pointer >= 0) {

const element = this.array[this.pointer];

this.pointer--;

return element;

} else {

return null;

}

}

top() {

if (this.pointer >= 0) {

return this.array[this.pointer];

} else {

return null;

}

}

}

1. Контейнер - это объект, который содержит и управляет хранением других объектов. В C++ существует множество различных контейнеров. Некоторые примеры:

std::vector - динамический массив;

std::list - двусвязный список;

std::set - множество уникальных элементов;

std::map - контейнер, хранящий пары ключ-значение;

std::queue - очередь;

std::stack - стек;

std::unordered\_map - неупорядоченный контейнер, хранящий пары ключ-значение;

std::unordered\_set - неупорядоченное множество уникальных элементов.

Пример создания std::vector, заполнения его значением и вывода на экран:

#include <iostream>

#include <vector>

int main() {

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

for (int i : v) {

std::cout << i << " ";

}

return 0;

}

1. В контейнерах можно выделить три группы операций:

Операции добавления и удаления элементов:

push\_back(value) - добавление элемента в конец контейнера (для std::vector и std::deque);

push\_front(value) - добавление элемента в начало контейнера (для std::list и std::deque);

insert(pos, value) - добавление элемента на позицию pos;

emplace(аргументы) - добавление элемента с помощью конструктора;

erase(pos) - удаление элемента из позиции pos;

pop\_back() - удаление последнего элемента из контейнера (для std::vector и std::deque);

pop\_front() - удаление первого элемента из контейнера (для std::list и std::deque);

clear() - удаление всех элементов из контейнера.

Операции доступа к элементам:

operator - доступ к элементу по индексу (для std::vector и std::array);

at(index) - доступ к элементу по индексу с генерацией исключения при выходе индекса за пределы контейнера;

front() - доступ к первому элементу (для std::list и std::deque);

back() - доступ к последнему элементу (для std::list и std::deque).

Операции информирования о размере и свойствах контейнера:

size() - количество элементов в контейнере;

empty() - проверка, является ли контейнер пустым;

max\_size() - максимально возможное количество элементов в контейнере;

resize(size) - изменение размера контейнера;

shrink\_to\_fit() - освобождение неиспользуемой памяти, занимаемой контейнером (только для std::vector и std::string).

1. Существует несколько видов доступа к элементам контейнера:

Доступ к элементу по индексу: осуществляется с помощью оператора [], который принимает в качестве аргумента индекс элемента, к которому нужно обратиться. Например:

std::vector<int> vec = {1, 2, 3};

int x = vec[1]; // x = 2

Доступ к элементу с помощью метода at(index): также осуществляется по индексу элемента, но при выходе за пределы контейнера генерируется исключение. Например:

std::vector<int> vec = {1, 2, 3};

int x = vec.at(4); // генерируется исключение std::out\_of\_range

Доступ к первому и последнему элементам контейнера: осуществляется с помощью методов front() и back(). Например:

std::vector<int> vec = {1, 2, 3};

int first = vec.front(); // first = 1

int last = vec.back(); // last = 3

1. Итератор в C++ представляет собой объект, который позволяет перебирать элементы контейнера (например, вектора или списка) в определенном порядке. Итератор позволяет получить доступ к элементам контейнера в любом порядке, заменять их значения и удалять элементы. Итераторы делятся на две основные категории:

Константные итераторы (const\_iterator): используются для доступа к элементам контейнера без возможности изменения значений.

Неконстантные итераторы (iterator): используются для доступа к элементам контейнера, а также для изменения их значений.

Для использования итераторов в C++ существует несколько стандартных функций, таких как begin() и end() для получения итератора, указывающего на первый и последний элементы контейнера, соответственно, а также различные функции для получения других типов итераторов в зависимости от нужд разработчика. Итераторы позволяют реализовывать многие стандартные алгоритмы работы со структурами данных, такие как сортировка, обход и поиск элементов, а также являются важным средством для операций на контейнерных данных.

1. Итератор может быть реализован в C++ при помощи определения класса, который будет представлять собой итератор. В этом классе может быть определено несколько методов для доступа к элементам контейнера. Общий шаблон реализации итератора в С++ выглядит следующим образом:

class Iterator {

public:

// Конструкторы

Iterator();

Iterator(const Iterator&);

// Операторы

Iterator& operator=(const Iterator&);

Iterator& operator++(); // префиксный инкремент

Iterator operator++(int); // постфиксный инкремент

bool operator==(const Iterator&) const;

bool operator!=(const Iterator&) const;

T& operator\*();

// Методы

void someMethod();

private:

// Данные

T\* ptr;

};

T - это тип элемента, который мы будем перебирать.

someMethod() - это просто пример метода, который вы можете определить в вашем итераторе для какой-то дополнительной логики.

Реализовать эти методы по каждому конкретному типу контейнера, который вы хотите перебирать.

1. В C++ объединить контейнеры можно, используя стандартные алгоритмы и функции, такие как std::merge, std::inplace\_merge, std::set\_union, std::merge и другие.

std::merge отсортирует и объединит два отсортированных контейнера в один отсортированный контейнер. Например:

std::vector<int> vec1 = {1, 3, 5, 7};

std::vector<int> vec2 = {2, 4, 6, 8};

std::vector<int> merged\_vec(vec1.size() + vec2.size());

std::merge(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), merged\_vec.begin());

// merged\_vec = {1, 2, 3, 4, 5, 6, 7, 8}

std::set\_union создаст новый контейнер, содержащий уникальные элементы из двух исходных контейнеров. Например:

std::vector<int> vec1 = {1, 2, 3, 4};

std::vector<int> vec2 = {3, 4, 5, 6};

std::vector<int> union\_vec(vec1.size() + vec2.size());

std::set\_union(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(), union\_vec.begin());

// union\_vec = {1, 2, 3, 4, 5, 6}

У каждой из этих функций есть свои особенности и ограничения, поэтому выбор метода зависит от конкретной задачи.

1. Контейнеры, состоящие из элементов "ключ-значение", в С++ это std::map или std::unordered\_map (в зависимости от выбора пользователем).

В std::map доступ к элементам осуществляется через ключ, то есть пользователь может использовать значение ключа для получения соответствующего значения.

В std::unordered\_map элементы распределяются по хеш-таблице, поэтому доступ к элементам осуществляется с помощью ключа, который хешируется и используется для быстрого поиска элемента в таблице.

Оба контейнера предоставляют доступ к элементам через итераторы.

1. Контейнер, в котором вставка и удаление элементов выполняется на одном конце контейнера, называется стек (stack) в C++.
2. Объект d. int mas[100] является контейнером, так как это массив, который может содержать множество элементов одного типа и поддерживает операции вставки и удаления элементов на одном из концов (стековые операции).
3. Объект d не является контейнером, так как он представляет простую переменную типа int, которая может содержать только одно значение. Она не имеет возможности вставки и удаления элементов.
4. Доступ к элементам контейнера будет осуществляться по индексу. Индексация элементов начинается с 0.

Пример доступа к элементам контейнера:

// объявление и заполнение контейнера

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

// доступ к элементу контейнера по индексу

int element = myVector[2]; // получим значение 3

1. Доступ к элементам контейнера, реализованного в виде линейного списка, также будет осуществляться по индексу. Однако, поскольку каждый элемент списка не обязательно расположен в памяти последовательно, доступ к элементам по индексу может быть немного медленнее, чем в случае со стандартными контейнерами, реализованными в виде массивов. Для обеспечения безопасного доступа к элементам можно использовать методы, предусмотренные в интерфейсе контейнера, например, метод at().