# Структуры данных в C++: стек, очередь, дек, вектор, граф, мультисписки

## Теоретическая часть

**Структура данных** – это способ организации и хранения информации в памяти компьютера для эффективного доступа и изменения. Главная цель использования структур данных – оптимизировать работу программ. Если хранить данные хаотично, поиск или изменение элементов может быть медленным. Вместо этого данные группируют и организуют так, чтобы частые операции – например, поиск нужного элемента, добавление или удаление – выполнялись быстрее. Простая аналогия: беспорядочная куча бумаг сложнее в поиске, чем аккуратно разложенные по папкам документы. Структуры данных – это как разные способы упорядочить "папки" с данными, чтобы потом быстро их найти и обработать.

Существует множество видов структур данных. Среди наиболее распространённых: **массивы**, **связанные списки**, **стеки**, **очереди**, **хеш-таблицы** (словари), **деревья** и **графы**. В рамках данного семинара мы рассмотрим следующие структуры и их применение в C++: **вектор**, **стек**, **очередь**, **дек**, **граф**, а также особый случай — **мультисписки**.

**Примечание:** В C++ есть два подхода к структурам данных: можно использовать готовые контейнеры из Стандартной библиотеки шаблонов (STL) или реализовать свои структуры самостоятельно (например, через классы и указатели). В большинстве случаев удобнее и надёжнее пользоваться готовыми контейнерами STL, так как они оптимизированы и хорошо протестированы.

### Вектор (Vector) – динамический массив

**Вектор** – это динамический массив переменного размера. Он похож на обычный массив, но умеет автоматически расширяться при добавлении новых элементов. Вы можете представить его как эластичную коробку, которая по мере наполнения может растягиваться, чтобы вместить больше вещей. Вектор хорош, когда нужно хранить список элементов, количество которых заранее не известно и может меняться.

В C++ вектор реализован классом std::vector (заголовок <vector>). Например:

#include <vector>

std::vector<int> numbers; // создаём пустой вектор целых чисел
numbers.push\_back(10); // добавляем элемент 10 в конец
numbers.push\_back(20); // добавляем элемент 20 в конец
cout << numbers[0]; // обращаемся к элементу по индексу 0 (выведет 10)

В этом примере мы создали динамический массив numbers и добавили в него два значения. Метод push\_back() вставляет элемент в конец вектора. Вектор обеспечивает быстрый доступ по индексу (как и обычный массив) и умеет эффективно добавлять элементы в конец.

### Стек (Stack) – принцип LIFO (Last In, First Out)

**Стек** – это структура данных, работающая по принципу «последним пришёл — первым вышел» (LIFO). Представьте стопку тарелок: когда мы кладём тарелку, она оказывается сверху, и брать тоже всегда будем верхнюю тарелку первым. То же самое со стеком: последний добавленный элемент доступен первым для извлечения. Стек удобен там, где нужно запоминать временные данные и потом извлекать их в обратном порядке. Примеры из жизни: история браузера (последняя открытая страница возвращается при нажатии «назад»), отмена действий (Ctrl+Z отменяет последнюю сделанную операцию) или система вызова функций в программе (последняя вызванная функция завершается первой).

В C++ стек реализован через контейнер-адаптер std::stack (заголовок <stack>). Основные операции стека: - push() – добавить элемент на верхушку стека; - top() – посмотреть элемент на вершине (верхний, последний добавленный); - pop() – убрать элемент с вершины.

Пример использования стека в C++:

#include <stack>

std::stack<std::string> pages;
pages.push("Main Page"); // кладём страницу "Main Page" на стек
pages.push("Article"); // затем "Article"
std::cout << pages.top(); // выводим верхний элемент ("Article")
pages.pop(); // удаляем верхний элемент ("Article")
std::cout << pages.top(); // теперь на вершине "Main Page"

Здесь pages имитирует историю страниц: последняя открытая страница находится сверху стека.

### Очередь (Queue) – принцип FIFO (First In, First Out)

**Очередь** – структура данных, работающая по принципу «первым пришёл — первым вышел» (FIFO). Она напоминает обычную очередь в магазине или на остановке: кто первым встал, тот обслуживается первым. Добавляют элементы в конец очереди, а берут – из начала. Очереди полезны, когда нужно обрабатывать объекты в порядке поступления. Из жизни: очередь печати (документы печатаются в порядке отправки на принтер), очередь задач на сервере или обработка запросов пользователей в порядке их поступления.

В C++ очередь реализована через std::queue (заголовок <queue>). Основные методы: - push() – поместить элемент в конец очереди; - front() – получить элемент в начале (первый в очереди); - pop() – убрать элемент из начала (после этого следующий становится первым).

Пример использования очереди:

#include <queue>

std::queue<std::string> requests;
requests.push("Login"); // запрос "Login" встал в очередь
requests.push("QueryData"); // затем "QueryData"
requests.push("Logout"); // затем "Logout"
std::cout << requests.front(); // выводим первый запрос ("Login")
requests.pop(); // удаляем первый (обработан "Login")
std::cout << requests.front(); // следующий в очереди теперь "QueryData"

В примере выше мы имитируем очередь запросов: сначала поступил запрос "Login", потом "QueryData" и т.д. Они обрабатываются в порядке поступления.

### Дек (Deque) – двусторонняя очередь

**Дек** (двусторонняя очередь) – это структура данных, которая позволяет добавлять и удалять элементы с обоих концов. Название происходит от *double-ended queue*. Можно представить дек как очередь, в которой есть доступ не только к началу, но и к концу, или как комбинацию стека и очереди одновременно. Реальная аналогия: двухсторонняя очередь на аттракционе, где организаторы могут запускать людей и спереди, и сзади, если это нужно. Дек подходит для задач, где требуется гибкая работа с обоими концами последовательности – например, реализация кеша с выталкиванием самых старых элементов или организация задач с возможностью добавлять срочные задачи в начало.

В C++ дек представлен контейнером std::deque (заголовок <deque>). Он предоставляет функции вставки и удаления как в конец (push\_back, pop\_back), так и в начало (push\_front, pop\_front). Также, как и вектор, дек позволяет доступ по индексу к элементам.

Пример коротко: если использовать std::deque<int> dq;, то dq.push\_back(5); dq.push\_front(3); поместит 5 в конец, 3 в начало, и дек будет хранить [3, 5]. dq.pop\_back(); удалит элемент 5, а dq.pop\_front(); удалит элемент 3. Таким образом, дек очень гибок в управлении элементами с обеих сторон.

### Граф (Graph) – связи между объектами

**Граф** – это нелинейная структура данных, представляющая совокупность объектов (вершин) и связей между ними (рёбер). Графы удобны для моделирования всевозможных сетей и отношений: представьте карту дорог между городами (города – вершины, дороги – рёбра), социальную сеть (пользователи связаны дружбой), или зависимости между пакетами программ (пакеты – вершины, "зависит от" – ребро). В графе можно перемещаться от одной вершины к другой по рёбрам.

В памяти компьютера граф часто представляют с помощью **списков смежности**: для каждой вершины хранится список соседних вершин, с которыми она соединена. В C++ нет готового контейнера "граф", но удобно использовать стандартные структуры: например, динамические массивы (std::vector) или списки для хранения соседей, а также словари (std::unordered\_map) для быстрого доступа к вершине по её ключу (например, имени).

*Пример:* можно представить граф зависимостей пакетов как unordered\_map<std::string, std::vector<std::string>> dependency;, где ключ — название пакета, а значение — вектор названий пакетов, от которых он зависит. Хеш-таблица (unordered\_map) здесь играет роль картотеки или телефонного справочника: по имени пакета быстро находим список его зависимостей.

Графы используются во многих алгоритмах, например, для поиска кратчайшего пути (навигация на карте), выявления циклов зависимостей, топологической сортировки (порядок установки пакетов без конфликтов) и др.

### Мультисписки (Multilist) – один элемент в нескольких списках

**Мультисписок** – это структура данных, в которой один элемент одновременно может принадлежать нескольким спискам. Иначе говоря, элемент содержит несколько ссылок (указателей) на другие элементы, позволяя организовать несколько цепочек связей параллельно. Это более продвинутая концепция, но её можно пояснить на примере систем контроля версий, таких как Git.

Представьте историю коммитов в Git: обычно каждый коммит имеет одного родителя (предыдущий коммит), образуя простой связный список "цепочку истории". Но при слиянии веток один коммит может иметь **два родителя** (от каждой ветки). Такой коммит одновременно находится сразу в двух "списках" истории – это и есть простой пример мультисписка. Один и тот же узел (коммит) связан с двумя разными предшественниками, образуя разветвлённую структуру.

Для реализации мультисписка в коде можно использовать структуру, в которой у элемента несколько указателей на следующий элемент. Либо, как вариант, хранить для каждого узла список ссылок на связанные узлы. В примере с коммитами можно завести структуру:

struct Commit {
 std::string hash;
 std::string message;
 std::vector<std::string> parentHashes; // список родителей (может быть 1 или больше)
};

Здесь один коммит имеет вектор parentHashes – это по сути и есть мультисписок связей с предыдущими коммитами. Мультисписки пригодятся, когда одна и та же информация естественно вписывается в несколько различных списков одновременно.

## Практические задания

Ниже 5 задач. Каждая команда выбирает любые задачи и набирает баллы. **Цель: 10 баллов** любой комбинацией. Все решения — на **C++**, с компиляцией и запуском на тестах из условий.

### Задание 1 — Стек: валидатор HTML-тегов (2 балла)

**Что сделать.** По последовательности тегов <tag> / </tag> проверить корректность вложенности.
**Структуры.** stack<string>.
**Пример входа.** <div> <p> </p> </div> → корректно; <div> <p> </div> </p> → некорректно.
**Ожидаемый вывод.** Сообщение о корректности (например, YES/NO).

### Задание 2 — Очередь: обработка запросов (2 балла)

**Что сделать.** Смоделировать FIFO-обработку запросов (строки/пары «id, имя»).
**Структуры.** queue<...>.
**Пример входа.** Login, QueryData, Logout.
**Ожидаемый вывод.**

Processing: Login

Processing: QueryData

Processing: Logout

### Задание 3 — Дек: LRU-кэш (4 балла)

**Что сделать.** Реализовать LRU-кэш фиксированного размера: при обращении к ключу делать его «самым свежим», при переполнении вытеснять «самый старый».
**Структуры.** deque<...> (допустимо без unordered\_map).
**Пример входа.** Размер=3; обращения: A B C A D.
**Ожидаемый вывод (состояние после каждого запроса).**

A

B A

C B A

A C B

D A C

### Задание 4 — Граф + очередь: топологическая сортировка (6 баллов)

**Что сделать.** По зависимостям вида A зависит от B (ориентированный ацикличный граф) вывести корректный порядок выполнения (топосорт) или сообщить о цикле.
**Структуры.** unordered\_map<string, vector<string>>, queue<string>, счётчики входящих степеней.
**Пример входа.** A B, B C, D C.
**Ожидаемый вывод.** Один из корректных порядков, например: C B A D (или другой валидный).

### Задание 5 — Граф + стек: обход коммитов (DFS) (3 балла)

**Что сделать.** Дан граф коммитов: у узла может быть 0, 1 или несколько родителей (merge). Выполнить DFS от указанного «последнего» коммита; печатать порядок с отступами по глубине.
**Структуры.** stack<pair<string,int>>, unordered\_map<string, vector<string>>, unordered\_set<string>.
**Пример входа.**

C1: []

C2: [C1]

C3: [C1]

C4: [C2, C3]

START C4

**Ожидаемый вывод (вариант).**

Commit: C4

 Commit: C2

 Commit: C1

 Commit: C3

## Шаблон кода

Используйте следующий шаблон C++ для начала работы над любым из вышеописанных заданий:

#include <iostream>
using namespace std;

int main() {
 // Ваш код здесь
 return 0;
}