Отлично! Разберем каждую линейную структуру данных с точки зрения того, что действительно нужно знать разработчику на C++.

# 1. Массивы (Arrays)

## Что это:

- **Статические**: Фиксированного размера, создаются на стеке
- **Динамические**: Размер определяется в runtime, создаются в куче

## Что нужно знать:

### Теория:

- **Непрерывность памяти** - элементы расположены друг за другом
- **Быстрый доступ по индексу** за O(1) - адрес вычисляется:
  `base_address + index * sizeof(type)`
- **Фиксированный размер** (для статических) или ручное управление памятью (для динамических)

### Практика в C++:

```cpp
// Статический массив
int static_arr[5] = {1, 2, 3, 4, 5};

// Динамический массив
int size = 10;
int* dynamic_arr = new int[size];
// ... использование ...
delete[] dynamic_arr; // Не забыть!

// std::array (рекомендуется)
#include <array>
std::array<int, 5> std_arr = {1, 2, 3, 4, 5};
```

### Ключевые операции:

- Доступ: `arr[i]` - O(1)
- Поиск: O(n) - линейный поиск
- Вставка/удаление в середину: O(n) - нужно сдвигать элементы

**Когда использовать:** Когда нужен быстрый произвольный доступ и размер известен заранее.

## 2. Связные списки (Linked Lists)

### Что это:

- **Односвязные**: Каждый элемент указывает только на следующий
- **Двусвязные**: Элементы указывают на предыдущий и следующий

### Что нужно знать:

**Теория:**

- **Элементы разбросаны в памяти** (не непрерывны)
- Каждый узел содержит данные и указатель(ли) на соседей
- **Быстрая вставка/удаление** если есть указатель на место операции

**Практика в C++:**

```cpp
#include <list>
// Двусвязный список (по умолчанию)
std::list<int> my_list = {1, 2, 3};

// Односвязный список
#include <forward_list>
std::forward_list<int> forward_lst = {1, 2, 3};
```

**Ключевые операции:**

- Вставка/удаление в известной позиции: O(1)

- Доступ по индексу: O(n) - нужно пройти от начала
- Поиск: O(n)

**Преимущества перед массивом:**

- Динамический размер
- Быстрая вставка/удаление в любом месте
- Не нужно перевыделять память при росте

**Когда использовать:** Когда частые вставки/удаления в середине списка.

## 3. Стеки (Stack) - LIFO

### Что это:

- **LIFO** (Last-In, First-Out) - последний пришел, первый ушел
- Как стопка тарелок - можно работать только с верхним элементом

### Что нужно знать:

**Теория:**

- Ограниченный интерфейс - только операции с вершиной
- Можно реализовать на массиве или списке

**Практика в C++:**

```cpp
#include <stack>
std::stack<int> st;

// Основные операции
st.push(10);    // Добавить на вершину
st.top();       // Посмотреть вершину (не удаляя)
st.pop();       // Удалить вершину
st.empty();     // Проверить на пустоту
```

**Ключевые операции** (все O(1)):

- `push()` - добавление
- `pop()` - удаление
- `top()` - просмотр вершины

**Применение:**

- Отмена операций (Undo)
- Парсинг выражений
- Обход в глубину (DFS)
- Вызов функций (стек вызовов)

## 4. Очереди (Queue) - FIFO

### Что это:

- **FIFO** (First-In, First-Out) - первый пришел, первый ушел
- Как очередь в магазине

### Что нужно знать:

**Теория:**

- Элементы добавляются в конец, извлекаются из начала
- Обычно реализуется на связном списке или массиве (кольцевом буфере)

**Практика в C++:**

```cpp
#include <queue>
std::queue<int> q;

q.push(1);      // Добавить в конец
q.front();      // Посмотреть первый элемент
```

```
q.pop();        // Удалить первый элемент
q.back();       // Посмотреть последний элемент
```

**Ключевые операции** (все O(1)):

- `push()` - в конец
- `pop()` - из начала
- `front()` - просмотр начала

**Применение:**

- Очереди задач
- Буферизация данных
- Поиск в ширину (BFS)
- Многопоточное программирование

## 5. Дек (Deque) - Двусторонняя очередь

### Что это:

- Гибрид стека и очереди
- Можно добавлять/удалять с обоих концов

### Что нужно знать:

**Теория:**

- Обычно реализуется как массив массивов (сегментированная структура)
- Позволяет эффективные операции с обоими концами

**Практика в C++:**

```
cpp

#include <deque>
std::deque<int> dq;
```

```
// Операции с обоих концов
dq.push_front(1);   // В начало
dq.push_back(2);    // В конец
dq.pop_front();     // Из начала
dq.pop_back();      // Из конца
dq[0];              // Произвольный доступ!
```

**Ключевые операции:**

- Операции с концами: O(1)
- Произвольный доступ: O(1) - в отличие от очереди!
- Вставка/удаление в середине: O(n)

**Преимущества:**

- Гибкость стека + очереди
- Быстрый произвольный доступ
- Динамическое расширение

**Когда использовать:** Когда нужны операции с обоими концами + произвольный доступ.

## Сравнительная таблица

| Структура | Доступ | Вставка | Удаление | Особенн |
|---|---|---|---|---|
| **Массив** | O(1) | O(n) | O(n) | Быстрый |
| **Список** | O(n) | O(1)* | O(1)* | Быстрая |
| **Стек** | O(1) | O(1) | O(1) | Только LI |
| **Очередь** | O(1) | O(1) | O(1) | Только F |
| **Дек** | O(1) | O(1) | O(1) | Гибкость |

*При наличии итератора на место вставки/удаления

## Главное для запоминания:

1. **Массив** - когда важен быстрый доступ по индексу
2. **Список** - когда много вставок/удалений в середине
3. **Стек** - для алгоритмов с возвратом (LIFO)
4. **Очередь** - для обработки в порядке поступления (FIFO)
5. **Дек** - универсальный выбор, когда нужна гибкость

Выбор структуры зависит от того, какие операции будут выполняться чаще