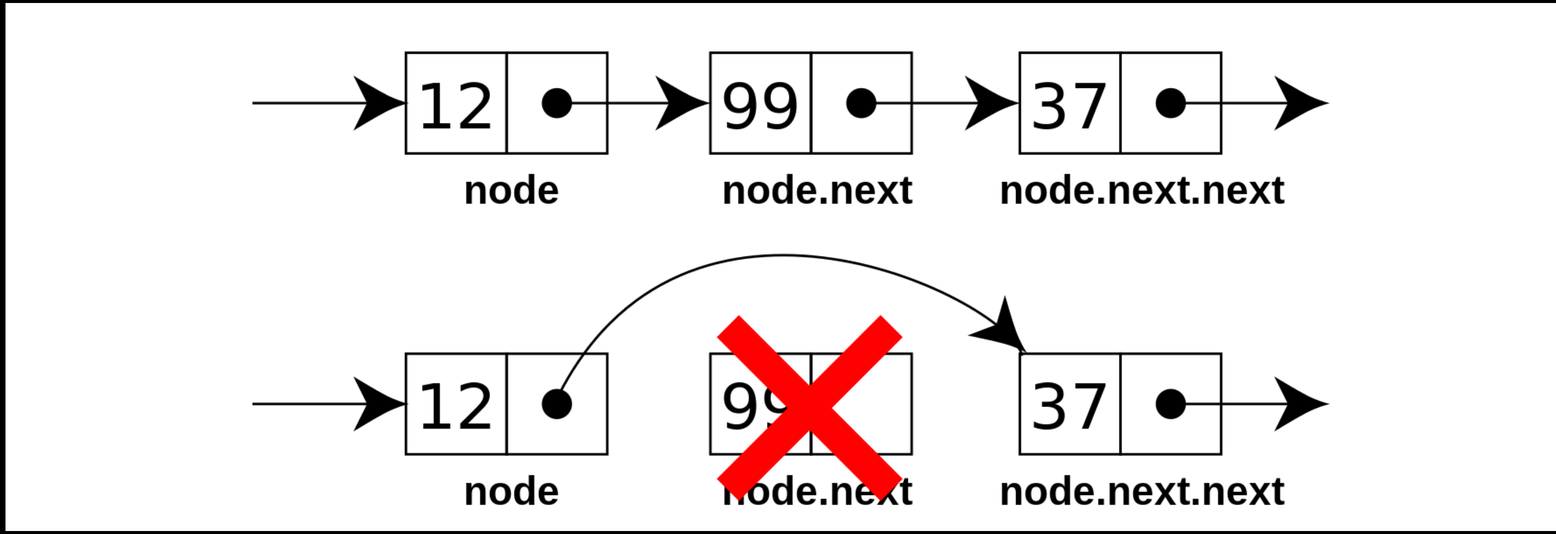
Linear Containers

Multidimensional Arrays, Lists, Queues, Stacks

SoftUni Foundation

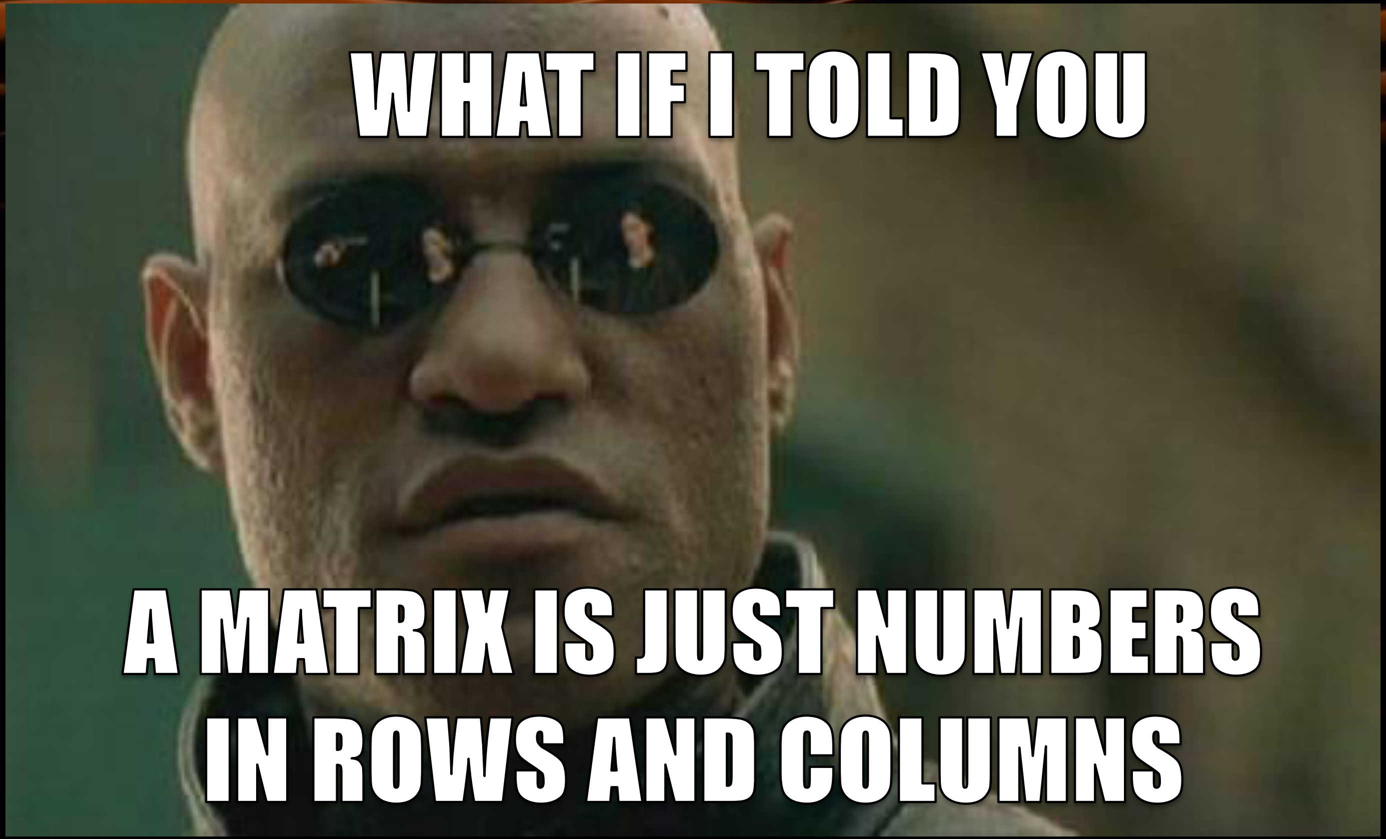


Georgi Georgiev
A guy that knows C++

Table of Contents

SoftUni Foundation

1. Multidimensional Arrays
2. Data Structures - Concept
3. STL Linear Containers
   - `std::vector`
   - Iterators
   - `std::list`
   - `std::stack` & `std::queue`



Multidimensional Arrays

Matrices and Higher Dimensions

Multidimensional Arrays

- C++ can make arrays act "as if" they have many dimensions
  - "as if" – they are just normal arrays which are indexed differently
  - Compiler enforces dimension syntax in code
- Imagine each element is actually an array
  - 2D (matrix): array of arrays (each element is a "normal" array)
  - 3D array: array of 2D arrays (each element is a matrix)
- Most-common usage: making a matrix/table

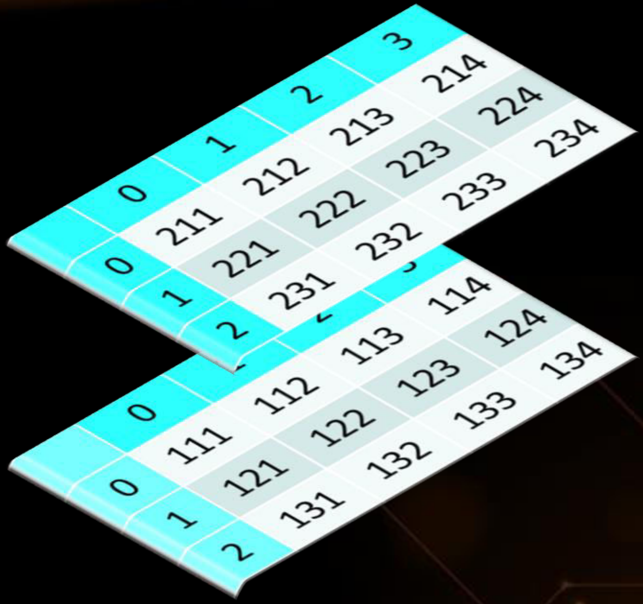
Using Multidimensional Arrays

- Accessing elements is done with one indexer per dimension
  - `matrix` 1st element (column) of 2nd row is `matrix[1][0]`
- Declaring: add a `[size]` for each additional dimension
  - First dimension can omit size
  - E.g. `int matrix2Rows3Cols[2][3]`
    or just `matrix2Rows3Cols[][3]`
    (needs initializer, unless function parameter)

Using Multidimensional Arrays

- Each dimension is an array with 1 less dimension
  - ```
    int matrix2Rows3Cols[][3] = {
       {11, 12, 13},
       {21, 22, 23}
    };
    ```
  - ```
    int cube[2][3][4] = {
       { {111, 112, 113, 114}, {121, 122, 123, 124}, {131, 132, 133, 134} },
       { {211, 212, 213, 214}, {221, 222, 223, 224}, {231, 232, 233, 234} }
    }
    ```
- If no initializer `{}` brackets, values are undefined
- If more elements than initialized, others are defaults

	0					1			
	0	1	2	3		0	1	2	3
0	111	112	113	114	0	211	212	213	214
1	121	122	123	124	1	221	222	223	224
2	131	132	133	134	2	231	232	233	234



Multidimensional Arrays

LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- What will the following code do?

a) cause a compile-time error

b) cause a runtime error due to index being out of bounds

c) set `matrix[2][0]` to `0`

d) summon demons

e) you know nothing

```
const int rows = 4;
const int cols = 3;
int matrix[rows][cols] = {
    {11, 12, 13},
    {21, 22, 23},
    {31, 32, 33},
    {41, 42, 43}
};

matrix[1][3] = 0;
```

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

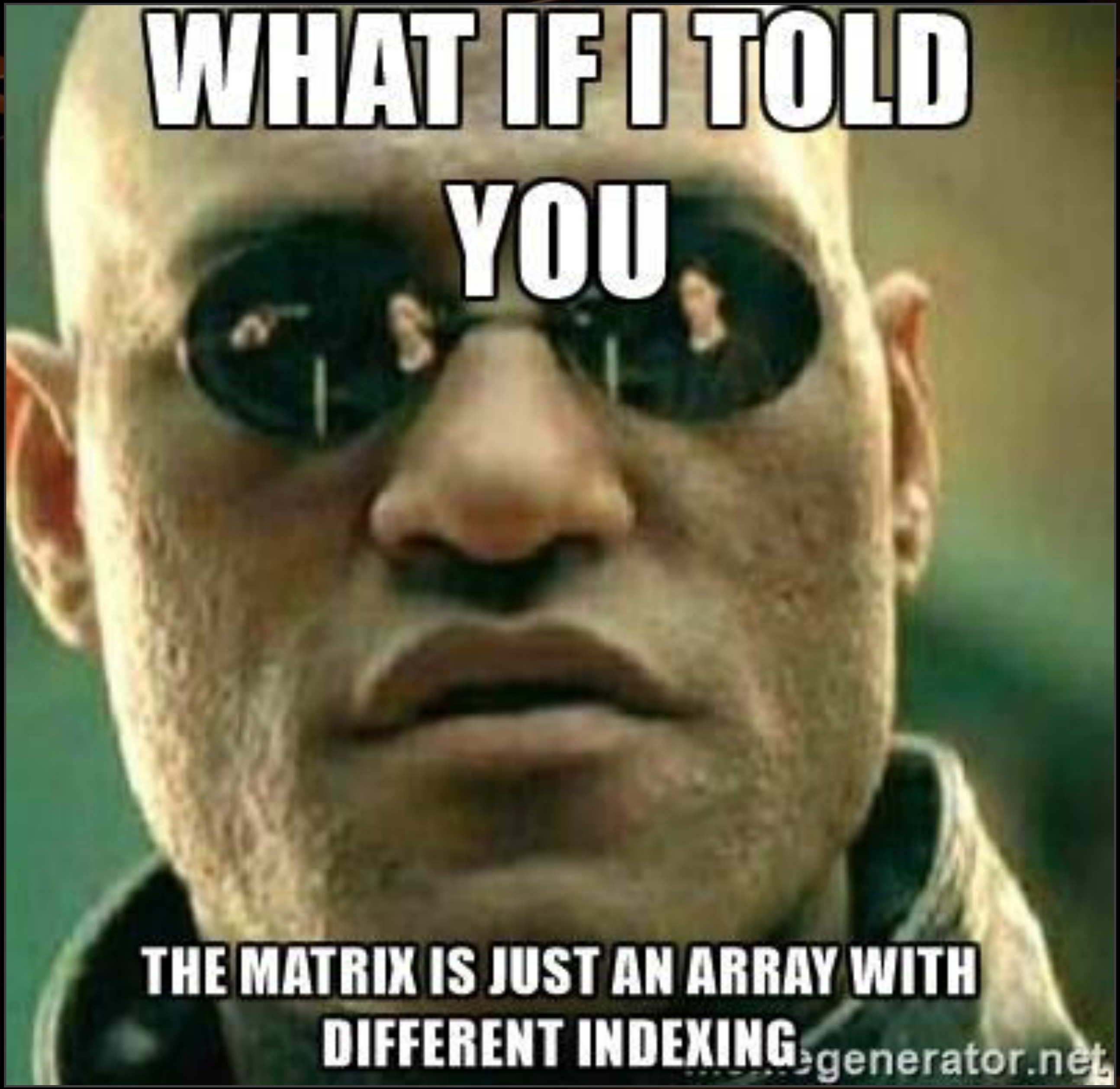
0,2	1,0	1,1	1,2	2,0

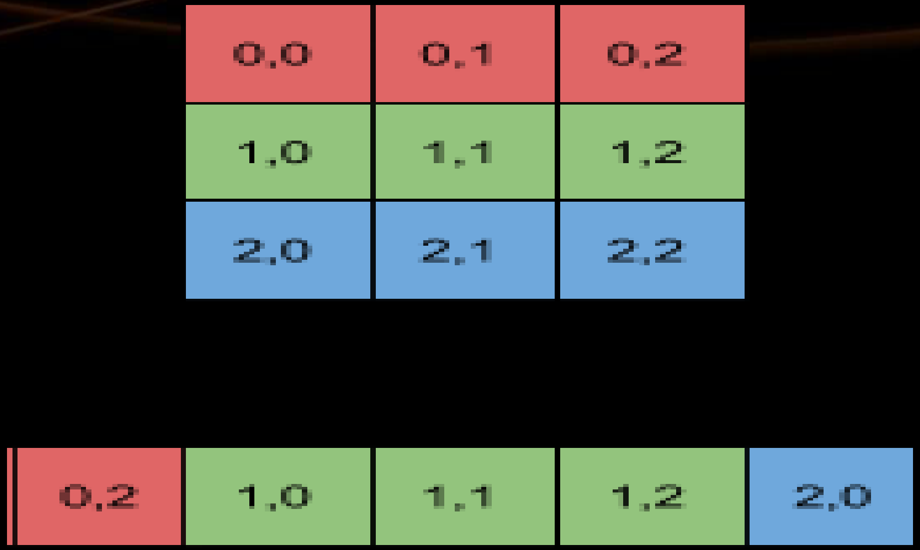
C++ PITFALL: “OUT OF BOUNDS INSIDE” MULTIDIMENSIONAL ARRAYS

C++ (C actually) stores multidimensional arrays as 1D, by joining up together 1st dimension elements, e.g. for 2D arrays – joining up rows into a 1D array.

This is called “row-major order”

E.g. for a `matrix[rows][cols]` accessing `[r][c]` just means `[r * cols + c]` in the actual array





Row-Major Order in Multidimensional Arrays

LIVE DEMO

"Multidimensional" Containers

- We know `std::vector` can contain any type
  - … actually any type with a default constructor
  - `int`, `double`, `char`, `string`, even another `std::vector`, etc.
- Often containers (e.g. `vector`s) will contain other containers
- E.g. a vector of vectors (2D), a vector of vector of vectors (3D)
  - Element access is the same code as with multidimensional arrays
  - Note: no row-major order (not contiguous in memory)

"Multidimensional" Containers

LIVE DEMO

Data Structures

Classifying Data Containers by Operation

Data Structures

SoftUni Foundation

- Data Structures organize data for efficient access
  - Different data structures are efficient for different use-cases
  - Essentially – a data container + algorithms for access
- Some of the common data structures in computer science:
  - Arrays – fast access by index, constant/dynamic size
  - Linked-list – fast add/remove at any position, no index access
  - Map/Dictionary – contains key/value pairs, fast access by key

Complexity 101

- Complexity in Computer Science describes performance
  - How fast an algorithm runs & How much memory it consumes
  - Based on the size of the input data – usually denoted as N
  - We usually care about the worst-case performance
- How do we measure complexity?
  - time = number of basic steps, memory = number of elements
- Complexity is usually denoted by the Big-O notation
  - How much the number of steps grows compared to input size

Complexity 101

SoftUni Foundation

- We usually care about **X** orders of magnitude, not **+X** or ***X**
  - `O(N+3) == O(2N) == O(N)`, i.e. we care about the **N** parts
    - *If something takes 1 million or 2 million years, it's the "million" that bothers you, not the "1" or the "2"*
- `O(1)` – "constant" time/memory – input size has no effect
- `O(log(N))` – logarithmic – complexity grows as log(input) grows
- `O(N)` – linear – complexity grows as input grows
- $O(N^2)$, $O(N^3)$, **…** – quadratic, cubic, … – complexity grows with square/cube/etc. of input size
- $O(2^N)$, $O(3^N)$, **…** – exponential – this is a monster

Data Structure Performance 101

- If **N** is the number of elements in the container (the `.size()`):

	vector	list	map, set	unordered_map, unordered_set
access i^{th}	O(1)	O(i)	O(i)	---
find(V)	O(N)	O(N)	O(log(N))	O(1)
insert(V)	O(1) at end (usually), O(N) otherwise	O(1)	O(log(N))	O(1)
erase(V)	O(1) at end (usually), O(N) otherwise	O(1)	O(log(N))	O(1)
Getting a sorted sequence	O(N*log(N)) (using std::sort algorithm)	O(N + N*log(N)) (using .sort() method)	O(N) (by just iterating)	---

STL Linear Containers

Vectors, Lists, Iterators, Container Adapters

`std::vector`

- Represents an array, has all array operations (i.e. `operator[]`)
- Changes size automatically when elements added
- `push_back()` complexity is *amortized* `O(1)`
  - i.e. usually takes `O(1)` time, occasionally takes `O(N)` time
  - i.e. slow ~10 times out of ~1000, ~32 times out of ~4 billion, etc.
- Fast access `O(1)` to any element (random index access)
  - `arr[0] = 69; arr[15] = 42;`

std::vector

LIVE DEMO

size_t and size_type

- Alias of one of the integer types
  - E.g. **unsigned long int** or **unsigned long long int**
  - Able to represent the size of any object in bytes
  - E.g. **sizeof()** returns **size_t**
- Each STL container offers a similar **::size_type**
  - A good practice is to use it instead of **int** for sizes, positions, etc.

```
for (vector<int>::size_type i = 0; i < nums.size(); i++) {
  cout << nums[i] << endl;
}
```

Container Iterators

- STL Iterators are things that know how to traverse a container
  - **`operator++`** - moves iterator to the next element
  - **`operator*`** - accesses the element
  - **`operator->`** - same as dot **`operator.`** on the element
- Each container has an iterator (e.g. `std::vector<T>::iterator`)
- Each container has **`begin()`** and **`end()`** iterators
  - **`begin()`** points to first element, **`end()`** to AFTER last
  - Range-based **`for`** loop uses those to work on ANY container

Using Iterators with Vectors

SoftUni Foundation

- Using iterators on **vectors** is almost the same as using indexes
- To walk over a vector:
  - Start from **begin()**, move with **++** until you reach **end()**
  - Access the current element with *****

```
vector<int> nums {42, 13, 69};
for (vector<int>::iterator i = nums.begin(); i != nums.end(); i++) {
  cout << *i << endl;
}
// Equivalent code
for (vector<int>::size_type i = 0; i < nums.size(); i++) {
  cout << nums[i] << endl;
}
```

Using Iterators

- Example: Change each element in the vector by dividing it by 2

```
vector<int> numbers {42, 13, 69};
for (vector<int>::iterator i = numbers.begin(); i != numbers.end(); i++) {
  *i /= 2;
}
// Equivalent code
for (int i = 0; i < numbers.size(); i++) {
  numbers[i] /= 2;
}
```

- Example: Print each string element and its length

```
vector<string> words {"the", "quick", "purple", "fox"};
for (vector<string>::iterator i = words.begin(); i != words.end(); i++) {
  cout << *i << ": " << i->size() << endl;
}
// Equivalent code
for (int i = 0; i < words.size(); i++) {
  cout << words[i] << ": " << words[i].size() << endl;
}
```

Using Iterators with Vectors

LIVE DEMO

Why Use Iterators?

- Vectors may not need iterators, because they have indexes
  - i.e. they have sequential elements accessible by `operator[]`
- Not all containers have indexes
  - Only `std::array`, `std::vector` & `std::deque` have indexes
  - The other containers don't offer access by index
- Iterators work on all containers, abstract-away container details
  - Doesn't matter what container you iterate, code is the same

`std::list`

- Represents elements connected to each other in a sequence
  - `std::list<int> values; std::list<string> names;`
  - Each element connects to the previous and next element.
    Like Christmas lights
- All element access is done with iterators
- Can add or remove elements anywhere in `O(1)` time
  - Requires iterator to where an element should be added/removed
- `push_back()`, `push_front()`, `insert()`, `size()`, …

std::list

LIVE DEMO

Container Adaptors

- Wrap a container (e.g. `vector`) with a different interface
- Allow you to express intentions better
- Make code more abstract and focused on the task, not the code
- STL Adapters for common Computer Science data structures:
  - `std::stack` is a first-in, last-out (FIFO) data structure
  - `std::queue` is a first-in, first-out (FIFO) data structure
  - `std::priority_queue` is a data structure that gives quick access to the "highest priority item"

`std::stack`

- Represents a container (a **deque** by default) working like a stack
- A stack is a "first-in, last-out structure" (FILO)
- *Imagine a pile of bricks*
  – the last brick you put is the first you can remove
- Access to elements other than **`top()`** is not provided
- **`top()`** gets the top, **`pop()`** removes it, **`push(T)`** adds to top

`std::queue`

- Represents a container (**deque** by default) working like a queue
- A queue is a "first-in, first-out" structure (FIFO)
- *Imagine a line at a store*
  – first person in the line is the first to get out
- Access to elements other than **`front()`** is not provided
- **`front()`** gets first, **`pop()`** removes it, **`push(T)`** adds to back

`std::priority_queue`

- Represents a queue, but elements are ordered by priority
  - By default, "larger" elements have higher priority
- *Imagine a queue at a hospital's emergency room – Patients with more serious cases treated BEFORE those with less serious ones*
- Higher priority elements move in front of lower priority ones
  - Getting top-priority element is `O(1)`, insertion is `log(N)`
  - To get top-priority element use `top()` (instead of `front()`)

Container Adaptors

LIVE DEMO

Summary

- C++ Multidimensional arrays are just normal arrays
  - … with indexing for each dimension
- We usually measure performance based on input
  - We care how quickly much performance degrades based on input size. We use Big-O notation to denote that
- C++ offers several linear data structures and adaptors
  - **`vector`**, **`list`**, **`stack`**, **`queue`**, **`priority_queue`**
  - Each is efficient for certain use-cases

Linear Containers

Questions?

https://softuni.bg/courses/