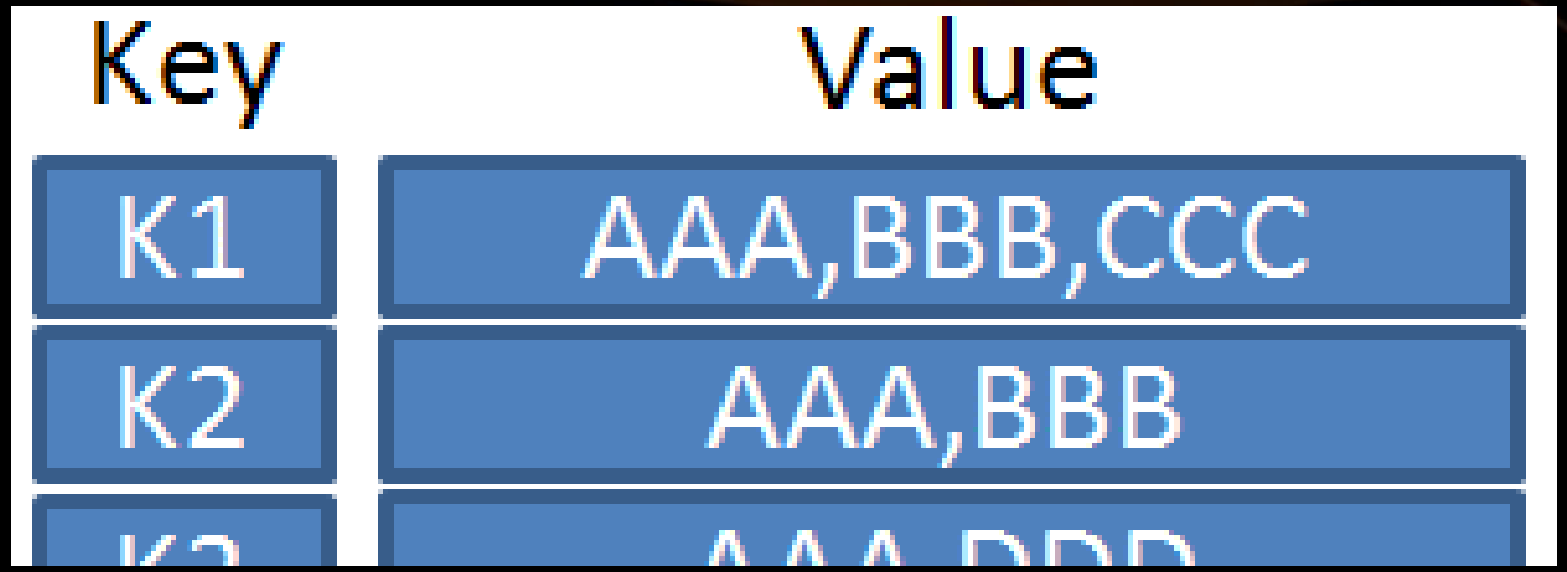
Associative Containers and STL Algorithms

Key-Value Container Concept, Maps, Sets, STL Algorithms



SoftUni Foundation

Georgi Georgiev

A guy that knows C++

Table of Contents

SoftUni Foundation

SoftUni Foundation

Key-Value Containers

Pairs of Things in Key-Value Pairs

Key-Value Containers

SoftUni Foundation

- Real-World information is often "labeled" or "named"
  - Contacts usually have names and numbers/emails:

```
{George -> +359899123123} {NSA -> 1-301-688-6524}
{Bon Jovi -> [no info]}
```

  - Locations have GPS coordinates:

```
{Great Pyramid of Gyza -> 29.9792345,31.1342019}
{SoftUni -> 42.666775,23.352277}
```

- Labels can also be created by context – this is called "mapping"
  - E.g. numeric values mapped to their names

```
{1 -> "one"} {2 -> "two"} {3 -> "three"}
```

Representing Key-Value Pairs in C++

SoftUni Foundation

- `std::pair<T1, T2>` can represent two values in one variable
  - E.g. `pair<string, int> namedNumber("five", 5);`
  - `#include<utility>`
  - `first` accesses the first value, `second` accesses the second value
  - `first` and `second` can be read and written directly, e.g.: `namedNumber.first="six"; namedNumber.second=6;`

```
pair<string, string> contact("George", "***@gmail.com");
contact.first = "George Georgiev";
cout << contact.first << " " << contact.second << endl;
```


SoftUni
Foundation
Uses of std::pair
LIVE DEMO

Key-Value Containers

SoftUni Foundation

- Computer Science calls these labeled values "**key-value** pairs"
  - A "**key**" is the label, a "**value**" is the thing we have labeled
  - Accessing the value is usually done through the key
  - *E.g. to call someone you search by their name to get their number*
- There are containers optimized for key-value operations
  - Called **associative containers**/**arrays**, **maps**, **dictionaries**, etc.
  - Fast access, insertion, and deletion by **key** – `O(log(N))` or `O(1)`

Associative Containers vs. Linear Containers

SoftUni Foundation

- **Associative containers** are arrays indexed by **keys**
  - A **key** can be anything – integer, string, char, or any other object
  - Linear containers can only have numeric indexing (array, vector)

Array or `std::vector`

key	0	1	2	3	4
value	8	-3	12	408	33

Associative array

key	value
John Smith	+1-555-8976
Lisa Smith	+1-555-1234
Sam Doe	+1-555-5030

SoftUni Foundation

C++ Associative Containers

Maps, Sets, Ordered & Unordered Variants

C++ Associative Containers

SoftUni Foundation

- Saying just "C++ Associative Container" implies "ordered"
  - `std::map`, `std::set`, `std::multimap`, `std::multiset`
  - Keep elements ordered by key – iterating gives them sorted by key
    E.g. an English-Chinese dictionary – ordered by English words
  - `find()`, `insert()`, and `erase()` are fast – `O(log(N))`
- Ordered associative containers have requirements for the key
  - By default – must support `operator<` (`int`, `double`, `string`, …)
  - Functors allow changing this (discussed later)

std::map – Initialization & Iteration

SoftUni Foundation

- Represents keys associated with values, ordered by key
  - Two type parameters – one for key, one for value
    `map<K, V>;`
  - Can be initialized like linear containers, but elements are pairs

```
map<string, int> cities = {
  pair<string, int>{"Gabrovo", 58950},
  pair<string, int>{"Sofia", 58950},
  pair<string, int>{"Melnik", 385},
};
```

- Iterating – elements are **pairs**, ordered by **pair::first**

```
for (auto i = cities.begin(); i != cities.end(); i++)
{ cout << i->first << " " << i->second << endl; }
for (pair<string, int> element : cityPopulations)
{ cout << element.first << " " << element.second << endl; }
```

std::map – Access & Search

SoftUni Foundation

- `operator[]` by key, returns direct reference to the value
  - Accesses value, if no such element, creates it: `cities["X"]++;`
    `//adds {"X", 0}, returns int& (the 0), 0++ gives 1, so {"X", 1}`
- Searching – `find()` by key, returns iterator to the pair
  - `cout << cities.find("Lom")->second` `//prints 27294`
  - `end()` if not found: `cities.find("Z") == cities.end();`

```
auto result = cities.find(searchCityName);
if (result != cities.end()) cout <<result->first<<" "<<result->second;
else cout << "No information about " << searchCityName << endl;
```

std::map – Insert & Erase

SoftUni Foundation

- `insert()` adds an element (key-value pair) into the map
  - Position is determined automatically by the map
  - `cities.insert(pair<string, int>("Melnik", 385));`
- `erase()` can remove by key or by iterator
  - `cities.erase("Melnik");` is *almost* the same as `cities.erase(cities.find("Melnik"));`
    (if `"Melnik"` key is in the map, if not – there will be a runtime error in the second case)
  - Deletion by iterator (if you have it) is a bit faster


SoftUni
Foundation
std::map
LIVE DEMO

Quick Quiz TIME:

SoftUni Foundation

- What will the following code print?

```
map<int, string> numbers { {2, "two"} };
for (int i = 0; i < numbers.size(); i++) {
   cout << numbers[i] << ",";
}
```

a) `zero,one,two,`

b) `,,two,`

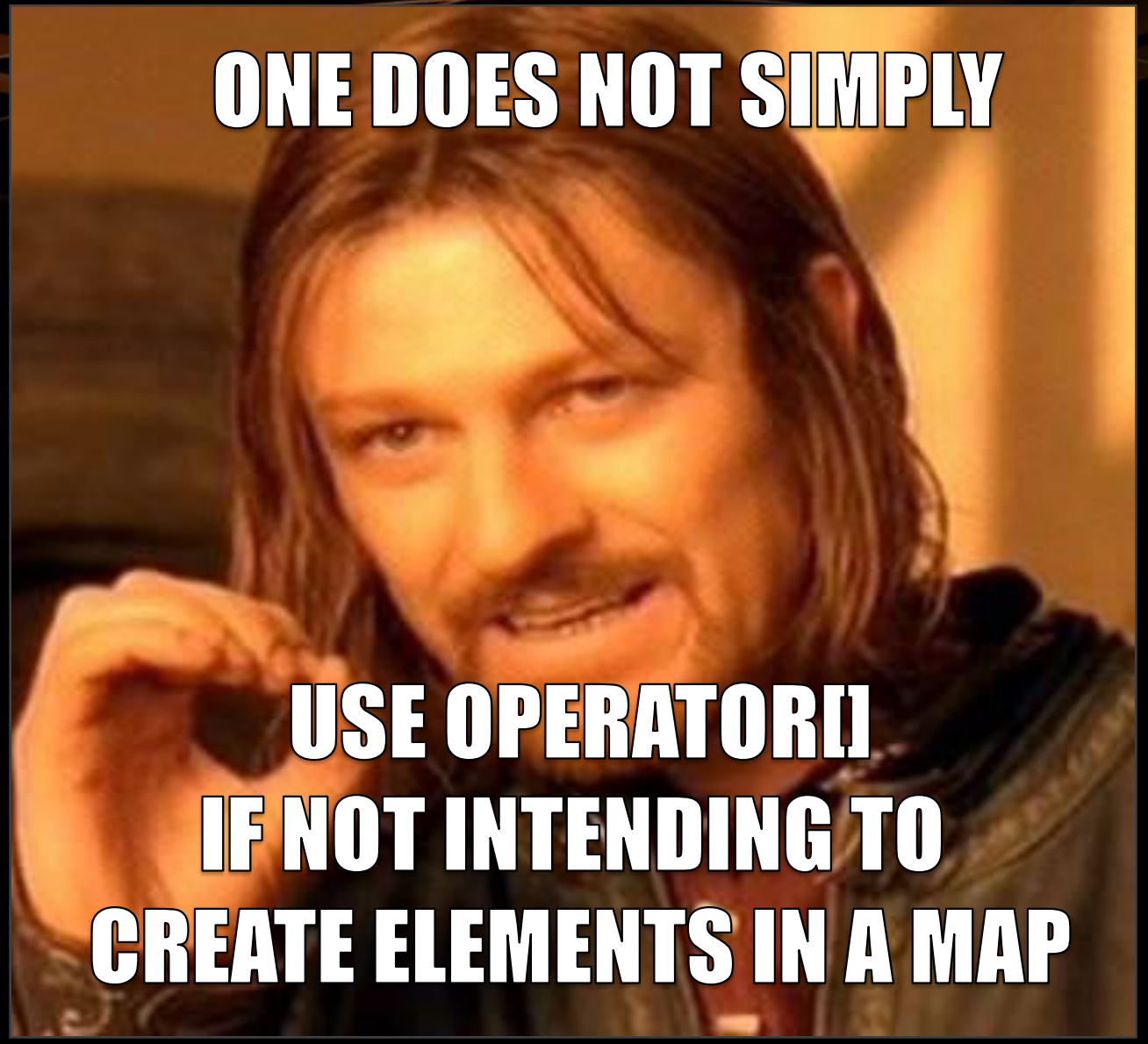
c) `two,`

d) There will be a runtime error

C++ PITFALL: MAP OPERATOR[] INSERTS NEW ELEMENT IF KEY NOT FOUND

If you use access elements with `operator[]`, without checking, in a loop, it is possible that you always add an element and increase the map's `size()`

It is safer to use `find()` if you just want to access existing elements



std::set

SoftUni Foundation

- Similar to map, but only stores keys, without values

```
set<int> nums { 4, 1, 4, 0, 6, 9, 1, 8, 6, 2, 3, 5, 6, 7 };
for (int n : nums) { cout << n << " "; } //0 1 2 3 4 5 6 7 8 9
```

  - Single type parameter – **set<K>**, no **operator[]**
  - Keys unique & sorted *(like in **map**)* – useful for removing duplicates
- Search, insertion, and deletion work the same as for **map**
  - **find()** returns iterator to key, or **end()** if not found
  - **insert()** only inserts if there is no such key
    – **nums.insert(10); nums.insert(10);** inserts **10** once


SoftUni
Foundation
std::set
LIVE DEMO

Unordered Associative Containers

SoftUni Foundation

- C++11 adds "unordered" associative containers
  - Same names but with `unordered_` prefix
  - E.g. `unordered_map`, `unordered_set`, etc.
  - Same `operator[]` (for maps), `find()`, `insert()`, `erase()`
- Faster *(usually)* – operations are `O(1)` instead of `O(log(N))`
- Elements are NOT ordered in any way
  - Iterating is same syntax but ordering is "random"

std::unordered_map

SoftUni Foundation

- Same operations, methods, initialization, etc. as `map`

```
unordered_map<string, int> cities = {
  pair<string, int>{"Gabrovo", 58950},
};
cities.insert(pair<string, int>{"Sofia", 58950));
cities["Melnik"] = 385;
cities.erase("Gabrovo");
```

- Iteration order is not defined, i.e. "random" (syntax is the same)

```
for (auto i = cities.begin(); i != cities.end(); i++)
{ cout << i->first << " " << i->second << endl; }
for (pair<string, int> element : cityPopulations)
{ cout << element.first << " " << element.second << endl; }
```

SoftUni Foundation

std::unordered_map

LIVE DEMO

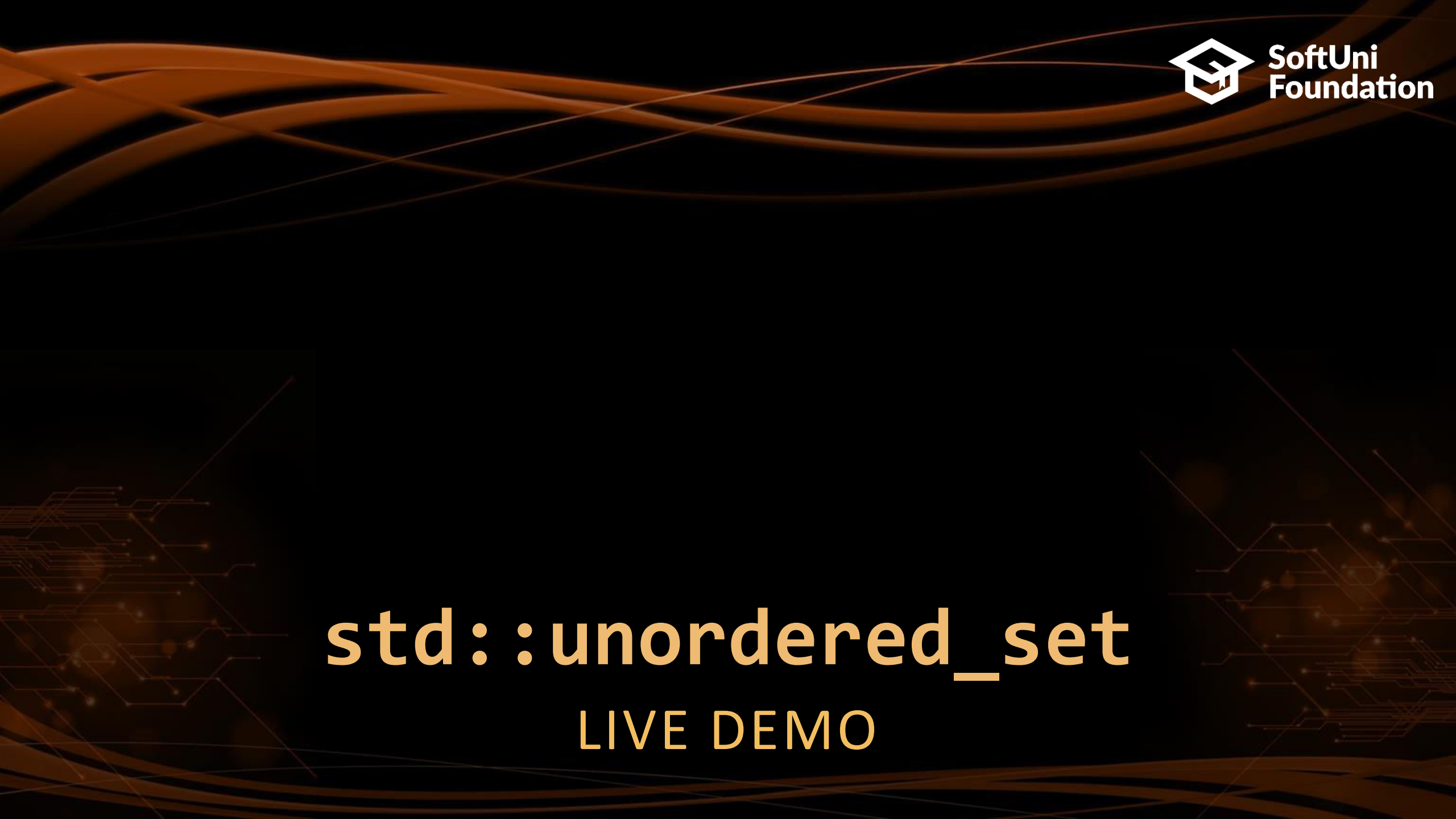
std::unordered_set

SoftUni Foundation

- Same as **set**, but no order for the keys

```
unordered_set<int> nums {
   4, 1, 4, 0, 6, 9, 1, 8, 6, 2, 3, 5, 6, 7
};
for (int n : nums) { cout << n << " "; }
// prints the numbers 0 1 2 3 4 5 6 7 8 9, but the order is unknown
```

- Useful when existence of elements needs to be checked
  - i.e. cases when no order information is needed
  - or cases where output order will not match "natural" order


SoftUni
Foundation
std::unordered_set
LIVE DEMO

Multiple Values with Same Key

SoftUni Foundation

- A common case is keeping multiple values having the same key
  - E.g. multiple phone numbers/emails for a person
  - E.g. multiple names for a number
- One approach is a map of vectors (or other linear container)
  - The key points to a list/vector/… of items
    e.g. `map<string, vector<int> > studentGrades;`
- Another approach (less common) – multimap/multiset
  - Allow duplicate keys & have operations for multiple equal keys


SoftUni
Foundation
Map of Vectors
LIVE DEMO

SoftUni Foundation

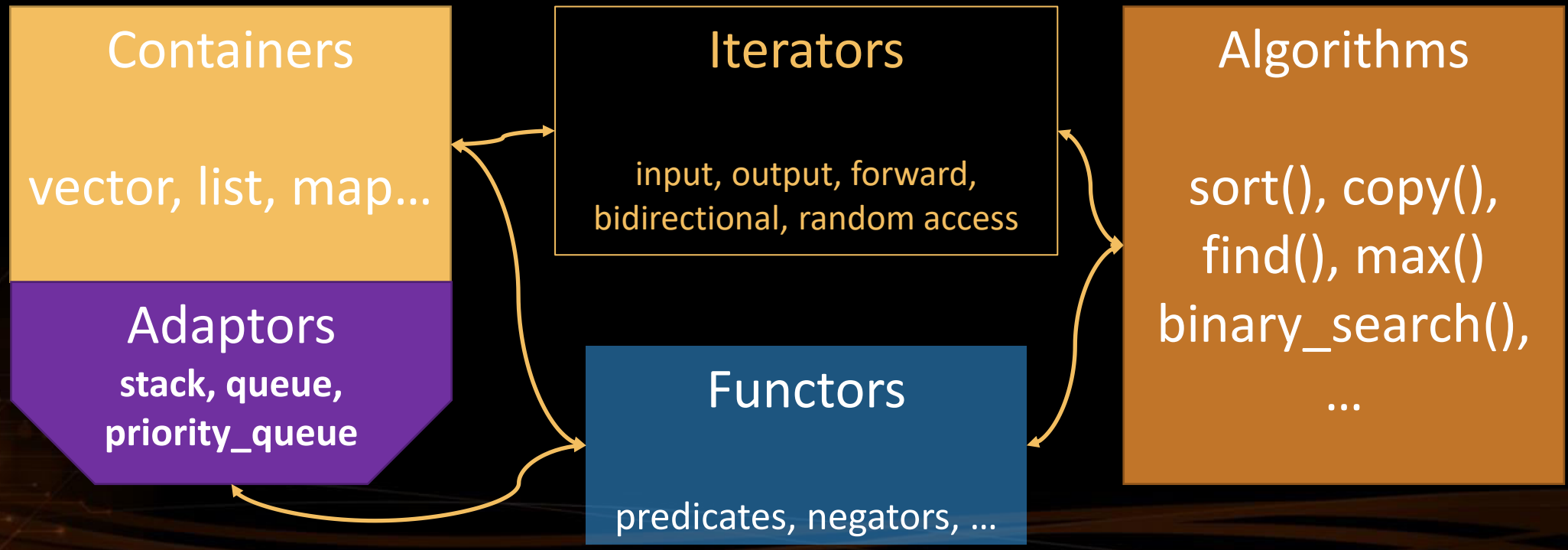
STL Algorithms

Sorting, Searching, Copying

STL Algorithms (#include<algorithm>)

SoftUni Foundation

- STL Provides common Computer Science algorithms
- Iterators define where to act (e.g. from `begin()` to `end()`)
- Functors define how to act (e.g. how to compare values)



Array Iterators

SoftUni Foundation

- Normal arrays can also be used in STL algorithms
  - The array's **name** acts is it's **begin()** iterator
  - Array iterators are random-access iterators
  - So, **array name** + **array size** = **array end()** iterator

```
string wordsArray[4] { "whales", "cats", "dogs", "fish" };
auto begin = wordsArray;
auto end = wordsArray + 4;
```

Sorting Array-Like Containers

SoftUni Foundation

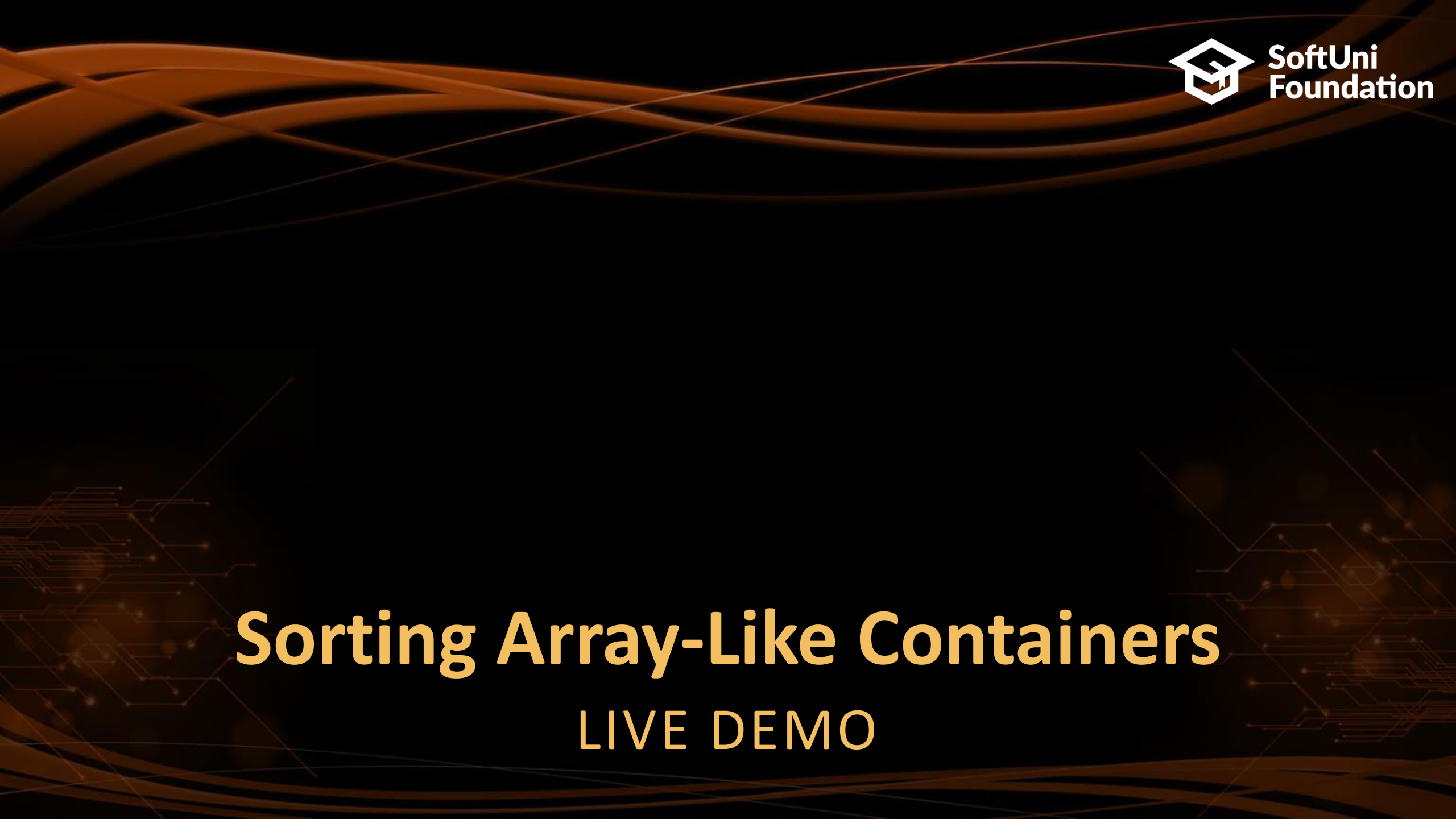
- **std::sort(begin, end)**
  - Sorts the range **[begin, end)**, data must have **operator<**
  - Requires random-access iterators (**array**, **vector**, **deque**)

```
vector<int> numsVect { 61, 41, 231, 764, 45 };
sort(numsVect.begin(), numsVect.end());

string wordsArr[4] { "whales", "cats", "dogs", "fish" };
sort(wordsArr, wordsArr + 4);
```

- **std::greater<T>** additional parameter for descending sort

```
sort(numsVect.begin(), numsVect.end(), greater<int>());
```


SoftUni
Foundation
Sorting Array-Like Containers
LIVE DEMO

Sorting Linked-Lists

SoftUni Foundation

- `std::list` is not random-access
  - `std::sort` requires random-access iterators
- So lists have their own `sort` version
  - Called directly on a list, i.e. `someList.sort();`

```
list<int> nums { 61, 41, 231, 764, 45 };
nums.sort();
```

  - List sort can also be told to sort from greater to lesser values

```
nums.sort(std::greater<int>());
```


SoftUni
Foundation
Sorting Linked Lists
LIVE DEMO

Searching – find

- `std::find(begin, end, value)`
  - Searches `[begin, end)` for `value`
  - Returns iterator to `value`, or `end` if `value` isn't found
  - If searching a `vector`/array, can subtract `begin()` to get index

```
vector<int> nums { 61, 41, 231, 764, 45 };
auto it = find(nums.begin(), nums.end(), 41);
if (it != nums.end()) {
  cout << "found " << *it << " at " << it - nums.begin() << endl;
} else {
  cout << "not found" << endl;
}
```

Searching – min_element & max_element

SoftUni Foundation

- `std::min_element(begin, end)`
  - Searches `[begin, end)` for the minimum element
  - Returns iterator if range is not empty, `end` otherwise
  - Data must have `operator<`
  - `std::max_element` does the same for the maximum element

```
vector<int> nums { 61, 41, 231, 764, 45 };

cout << min_element(nums.begin(), nums.end()) << endl; // 41

cout << max_element(nums.begin(), nums.end()) << endl; // 764
```

Quick Quiz TIME:

SoftUni Foundation

- What will the following code print?

a) `764 at 3`

b) `45 at 4`

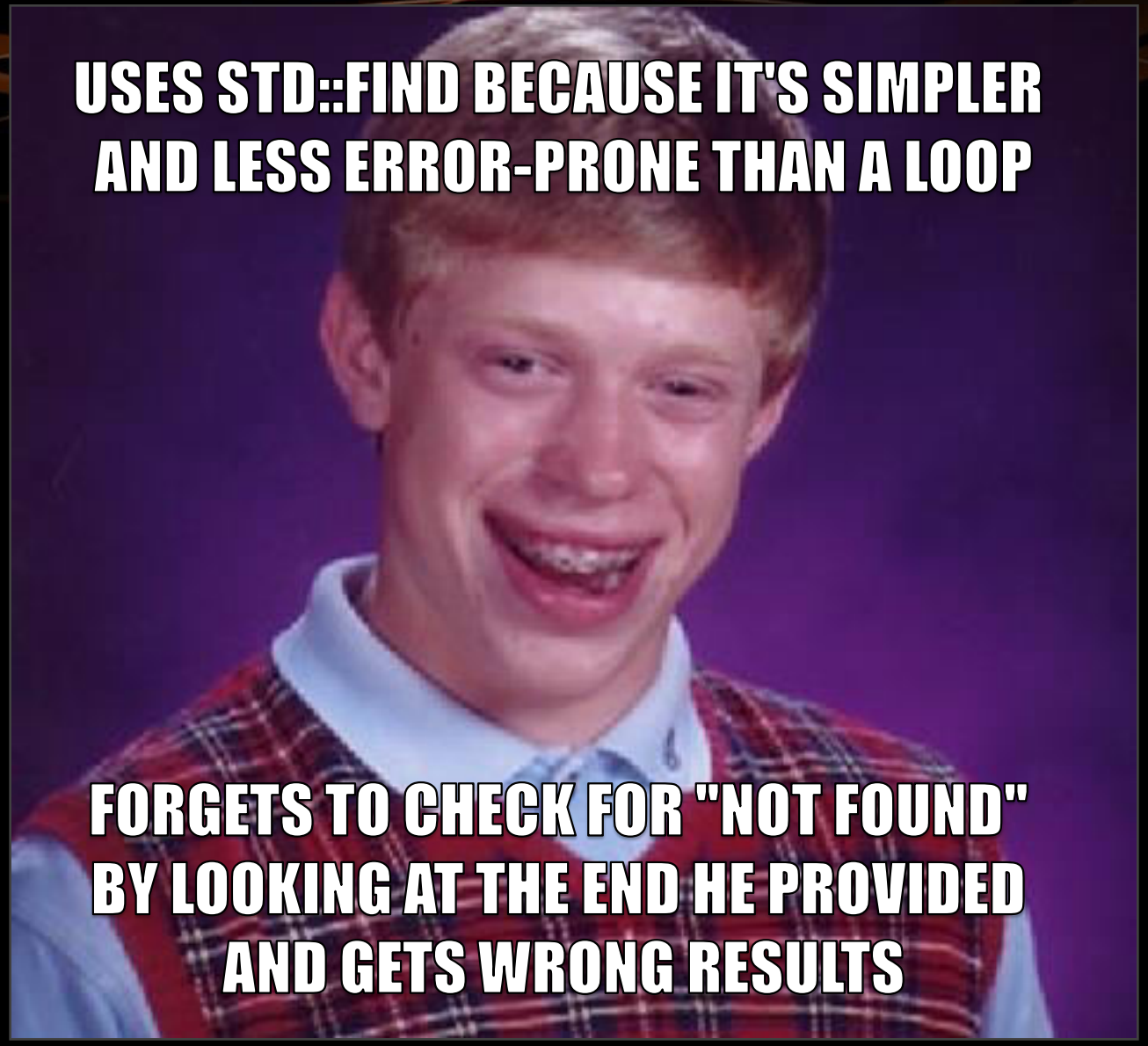
c) `not found`

d) There will be a runtime error

```
vector<int> nums { 61, 41, 231, 764, 45 };
auto it = find(nums.begin(), nums.begin() + 3, 45);
if (it != nums.end()) {
  cout << *it << " at "
       << it - nums.begin() << endl;
} else {
  cout << "not found" << endl;
}
```

C++ PITFALL: WRONG "NOT FOUND" CHECK (DIFFERENCE IN END ITERATOR)

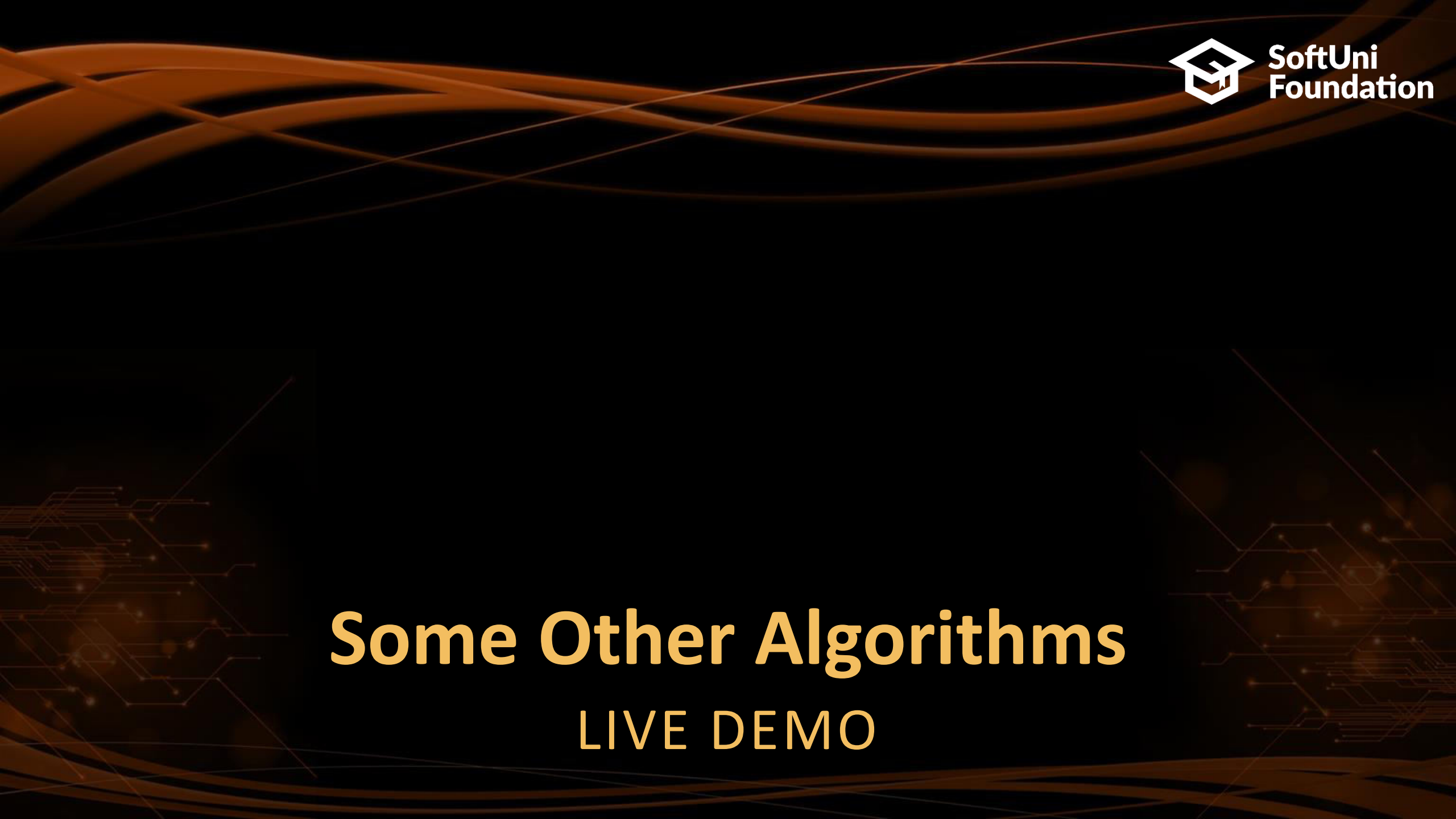
Be careful with `find(begin, end, value)` – it returns whatever `end` you gave it, if it doesn't find the value. If you're looking in part of a vector, it will return an iterator to the end of that part – not to the end of the vector – if it doesn't find `value`



Some Other Algorithms

SoftUni Foundation

- `std::lower_bound(begin, end, value)`
  - Requires `[begin, end)` to be sorted
  - Returns where `value` is, if it exists in `[begin, end)`
  - Returns where `value` should be if it doesn't exist
  - Fast – `O(log(N))`, vs. `O(N)` for `find()`
- There are many other algorithms
  - `upper_bound`, `copy`, `replace`, `remove`, `count`, `random_shuffle`, …
  - Look them up at http://en.cppreference.com/w/cpp/algorithm


SoftUni
Foundation
Some Other Algorithms
LIVE DEMO

Summary

- Associative containers map keys to values
  - Fast access/lookup/insertion/removal by key
- Maps contain key-value `pairs`
  - `map`, `unordered_map`, `multimap`, `unordered_multimap`
  - Unordered versions are usually faster… but have no ordering
- Sets (`set`, `unordered_set`) contain only keys
  - Good for extracting unique elements
- The `<algorithm>` library provides many common algorithms

Associative Containers and STL Algorithms

SoftUni Foundation

Questions?

https://softuni.bg/courses/