Lecture 5: Containers

Preston Seay, Rachel Fernandez

Plan

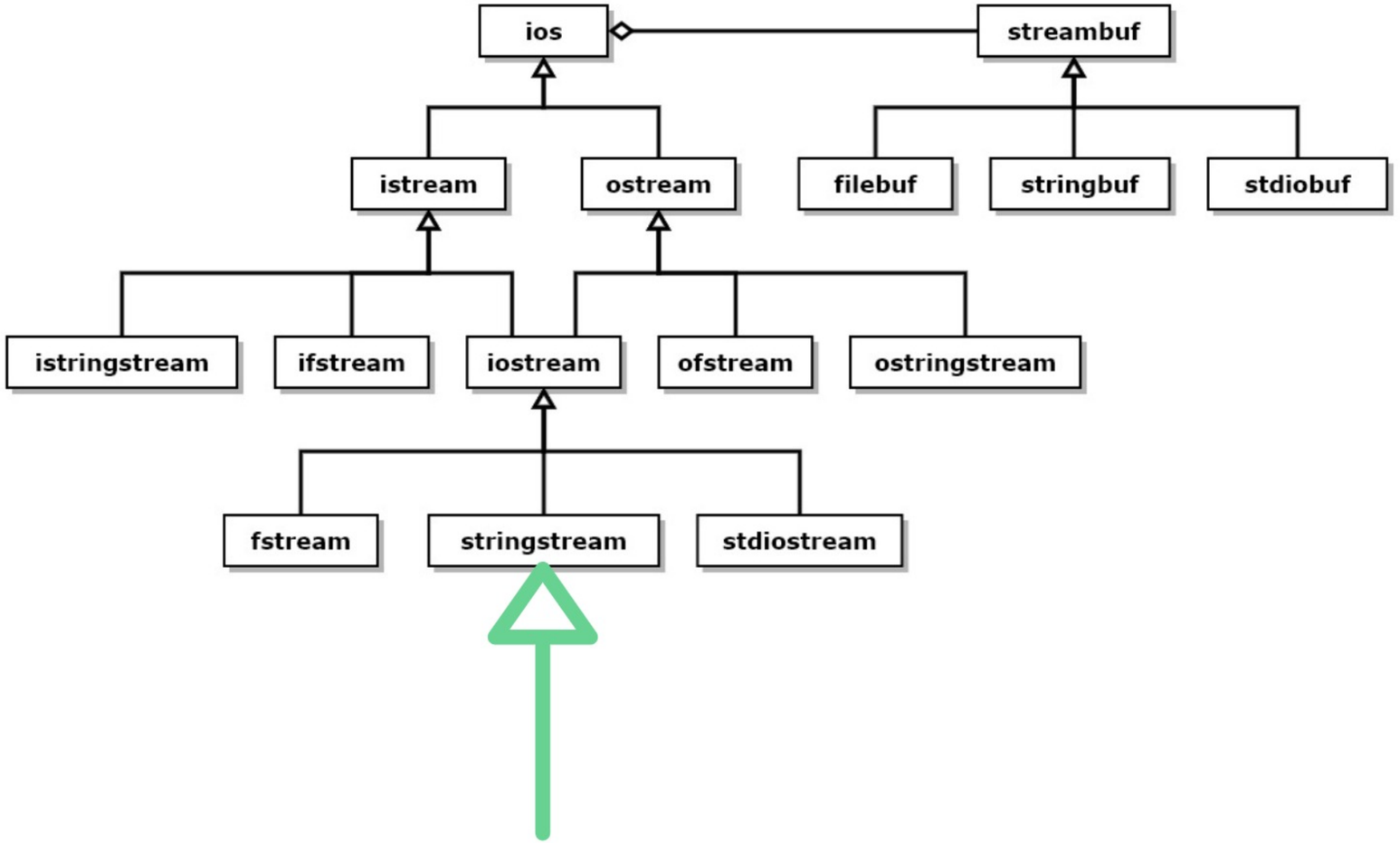
- (Recap)

1. Space-time
2. STL
3. Sequence Containers
4. Associative Containers

Recap

A `stringstream` is an...

- `istream` ✅
- `ostream` ✅



Recap

Code Console CPP Run

```cpp
#include <iostream>
#include <sstream>
#include <string>
int main() {
    std::stringstream ss;
    ss << 3.14f << ' ' << "hello"; // use as ostream

    float pi; std::string hi;
    ss >> pi >> hi; // use as istream

    std::cout << pi << '\n' << hi << std::endl;
}
```

Plan

1. **Space-time**
2. STL
3. Sequence Containers
4. Associative Containers

Task - Fetch a wrench


AHA
JEGS


AHA
HONDA

How much stuff?


A


Space helps you
see and navigate


"Space is time."
- Bjarne Stroustrup
source

Disorganized

- Space efficient
- Slow to search
- Ex: vector

Organized

- Space inefficient
- Faster to search
- Ex: map

Plan

1. **Space-time**
2. STL
3. Sequence Containers
4. Associative Containers

Announcements

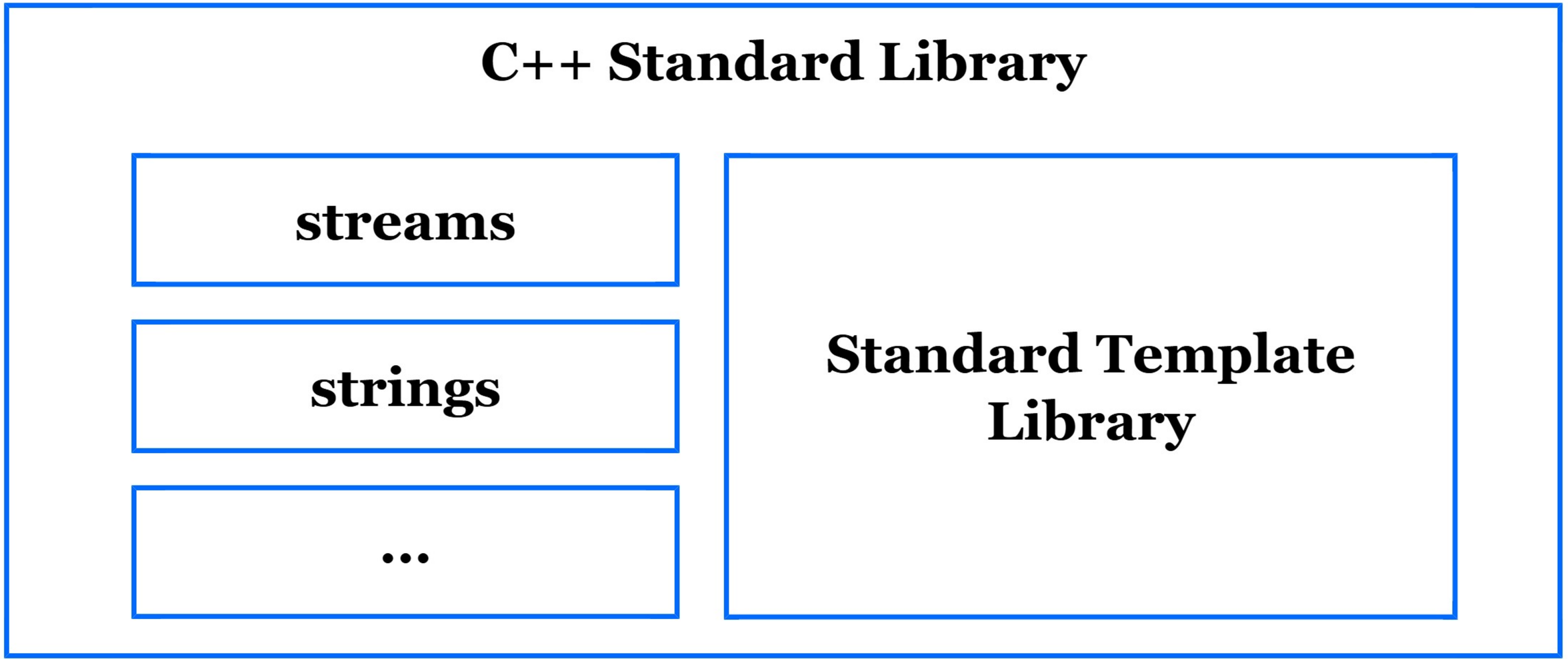
- Assignment 1 is out, due this Friday!
- Office hours start this Thursday, 4:30-5:20pm in Thornton 210.

Plan

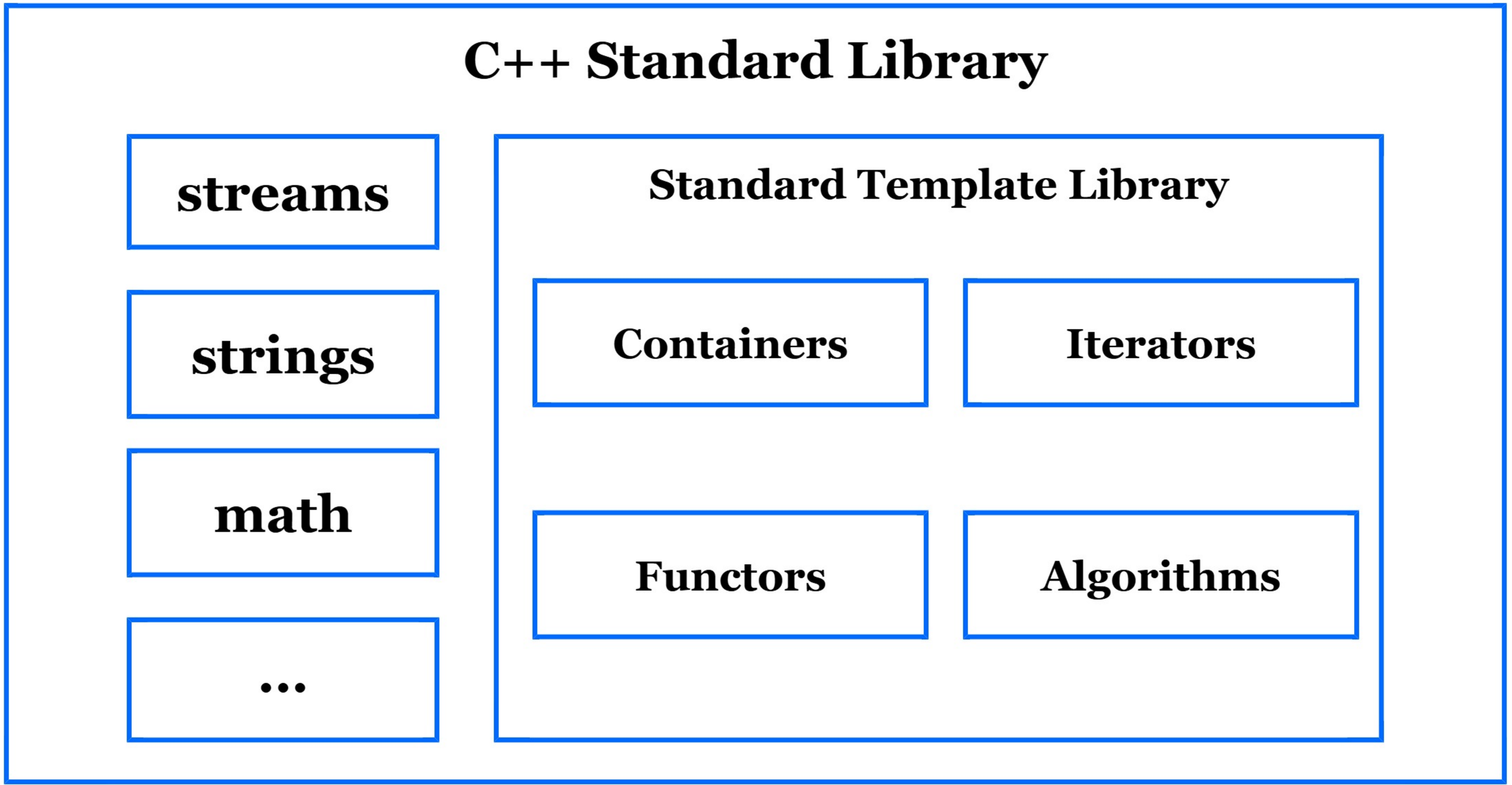
1. Space-time
2. **STL**
3. Sequence Containers
4. Associative Containers

Standard Template Library (STL)

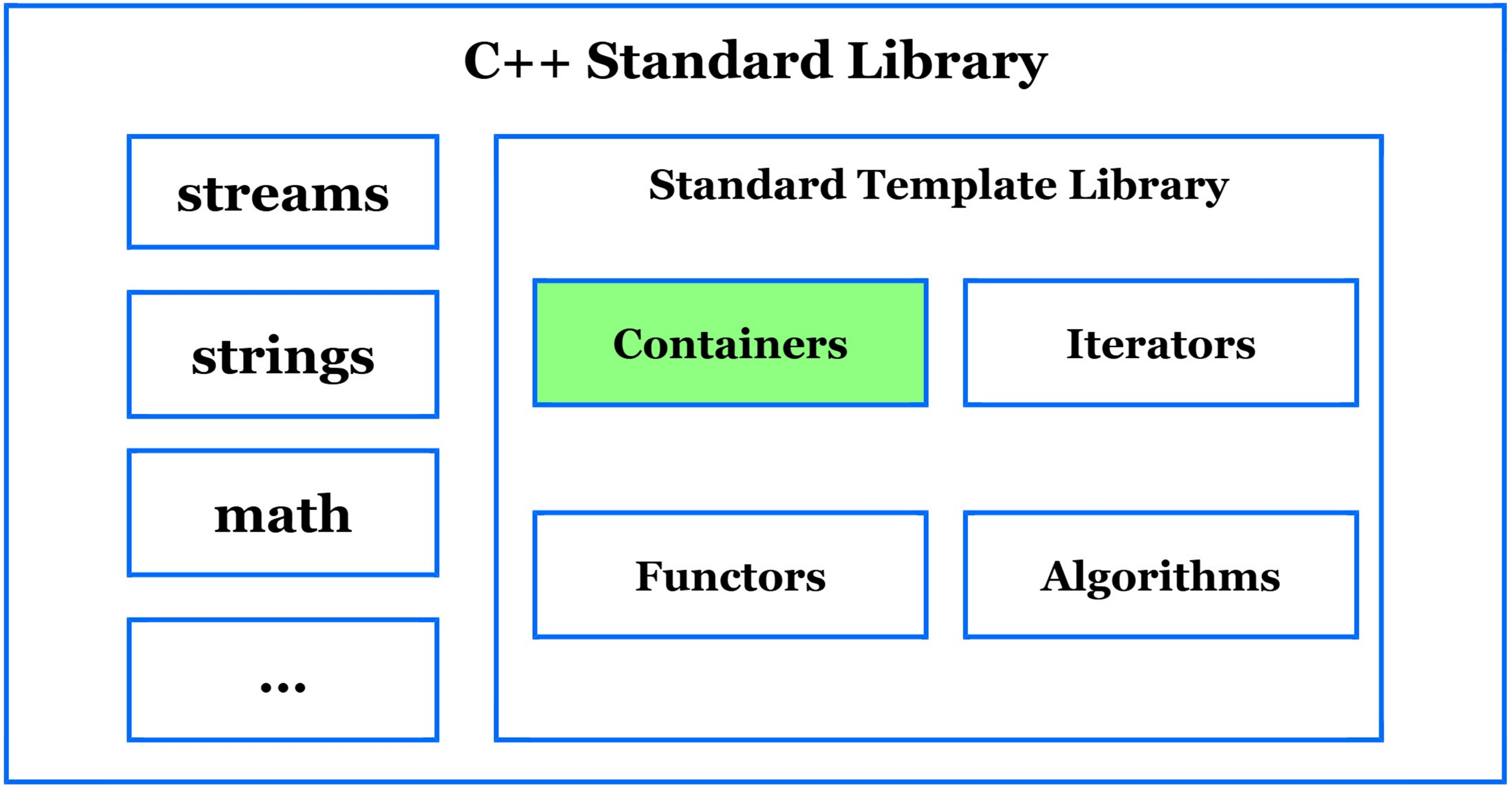
How is this different from the standard library `std`?



Standard Template Library (STL)

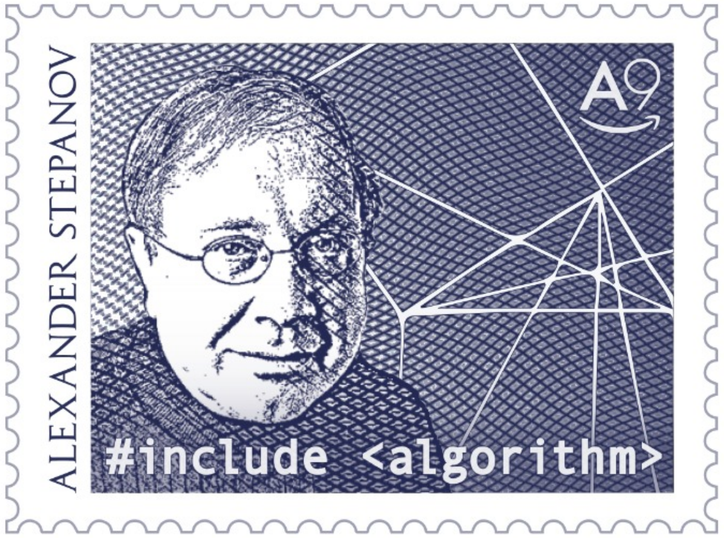


Standard Template Library (STL)



Standard Template Library (STL)

- **Made by Alexander Stepanov**



Standard Template Library

Containers

Iterators

Functors

Algorithms

What does template mean?

```
class IntList {
  ...
};

int main() {
  IntList ints(
  ints.add(10);
  ints.add(20);

  assert(ints.g
}
```

```
class StringList {
  ...
};

int main() {
  StringList strs();
  strs.add("1");
  strs.add("2");

  assert(strs.get(0) == "1");
}
```

```
ibleList {

) {
.ist dbls();
ld(1.0);
ld(2.0);

dbls.get(0) == 1.0);
```

```
class FloatList {
  ...
};
```

```
class BoolList {
  ...
};
```

```
class CharList {
  ...
};
```

What does template mean?

```
1 class IntList {
2     ...
3 };
```

```
1 class DoubleList {
2     ...
3 };
```

```
1 class StringList {
2     ...
3 };
```

```
1  // You'll learn more wk 5!
2  template <typename T>
3  class Vector<T> {
4      ...
5  }
```

```
1  #include <vector>
2
3  int main() {
4      std::vector<int> ints;
5      std::vector<double> dbls;
6      std::vector<std::string> strs;
7
8      ints.push_back(1);
9      dbls.push_back(5.4);
10     strs.push_back("hi");
11     std::cout << ints[0] + dbls[0];
12 }
```

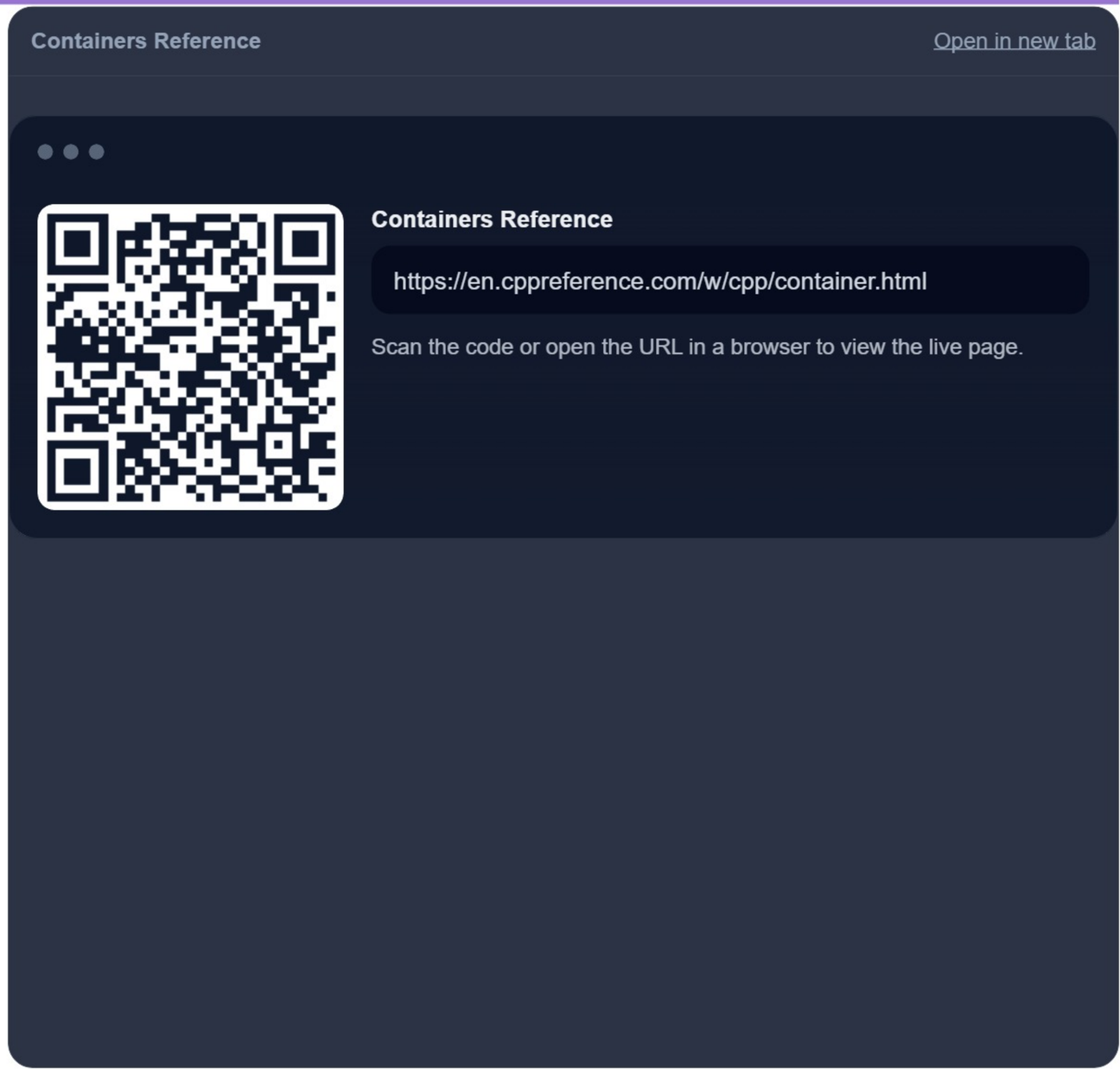
Plan

1. Space-time
2. STL
3. **Sequence Containers**
4. Associative Containers

Sequence Containers

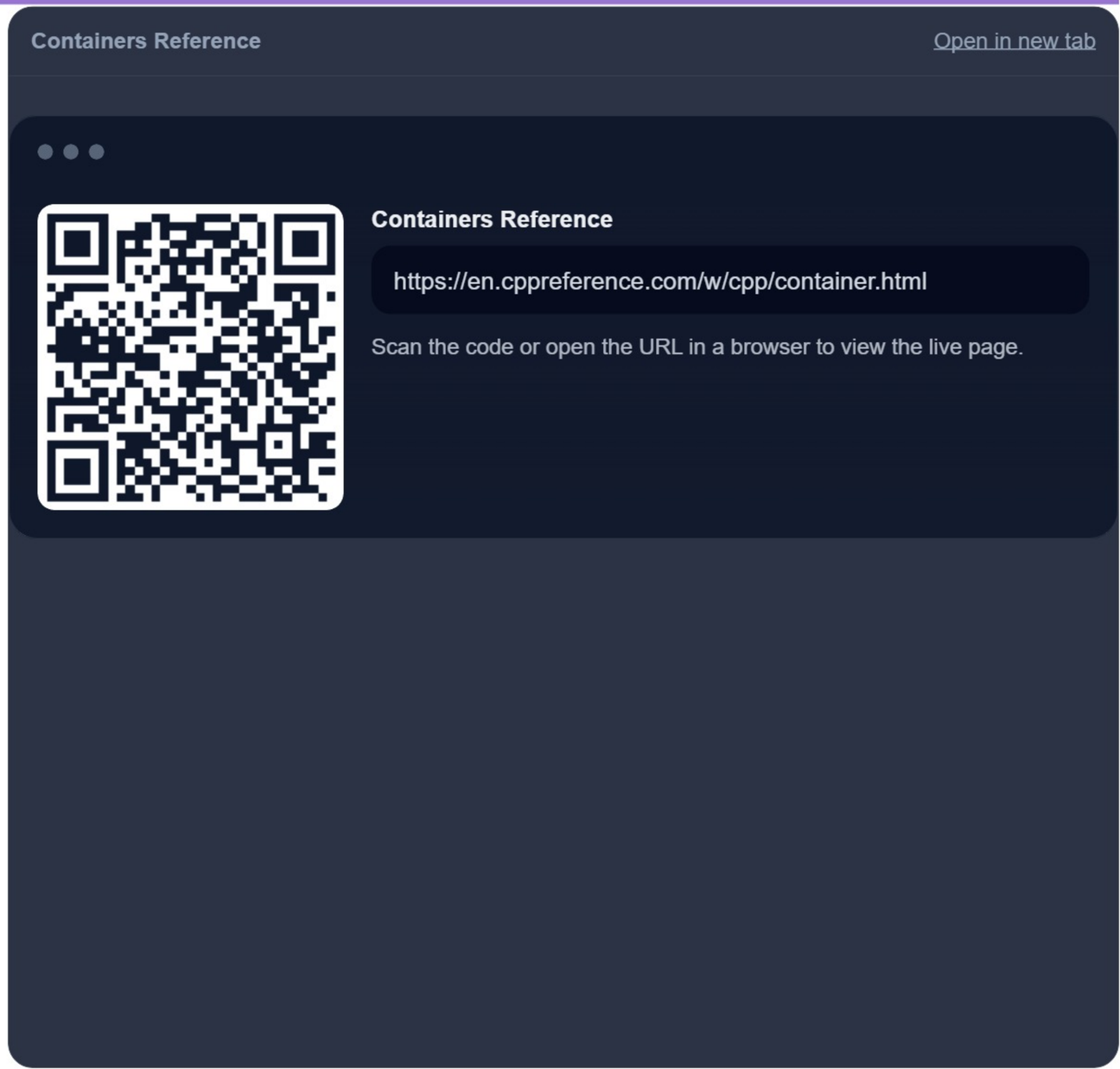
"Sequence containers implement data structures which can be accessed sequentially."

In other words, they contain sequences.



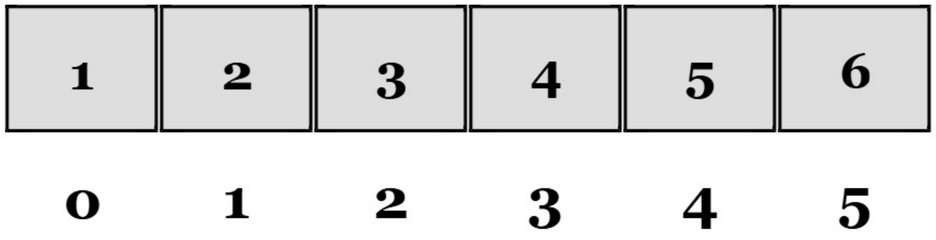
Sequence Containers

- **Vector**
- **Deque**
- **Array**
- **List**



Vectors

"A resizable contiguous array"



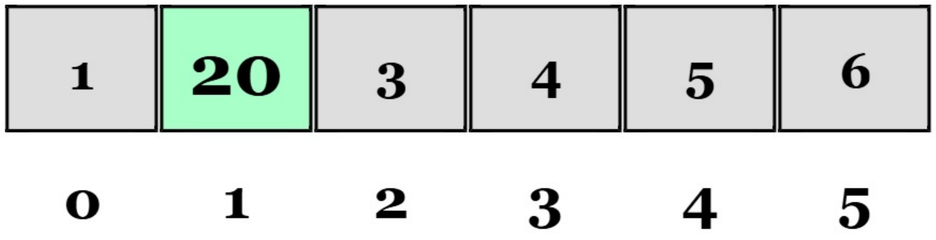
Code Console CPP Run

```cpp
#include <vector>
#include <iostream>
int main() {
    std::vector<int> vec { 1, 2, 3, 4 };
    vec.push_back(5);
    vec.push_back(6);
    vec[1] = 20;

    for (size_t i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << " ";
    }
}
```

Vectors

"A resizable contiguous array"



Code Console CPP Run

```cpp
#include <vector>
#include <iostream>
int main() {
    std::vector<int> vec { 1, 2, 3, 4 };
    vec.push_back(5);
    vec.push_back(6);
    vec[1] = 20;

    for (size_t i = 0; i < vec.size(); i++) {
        std::cout << vec[i] << " ";
    }
}
```

Index Checking?

- Be careful with indices
- [] doesn't check
- .at() does

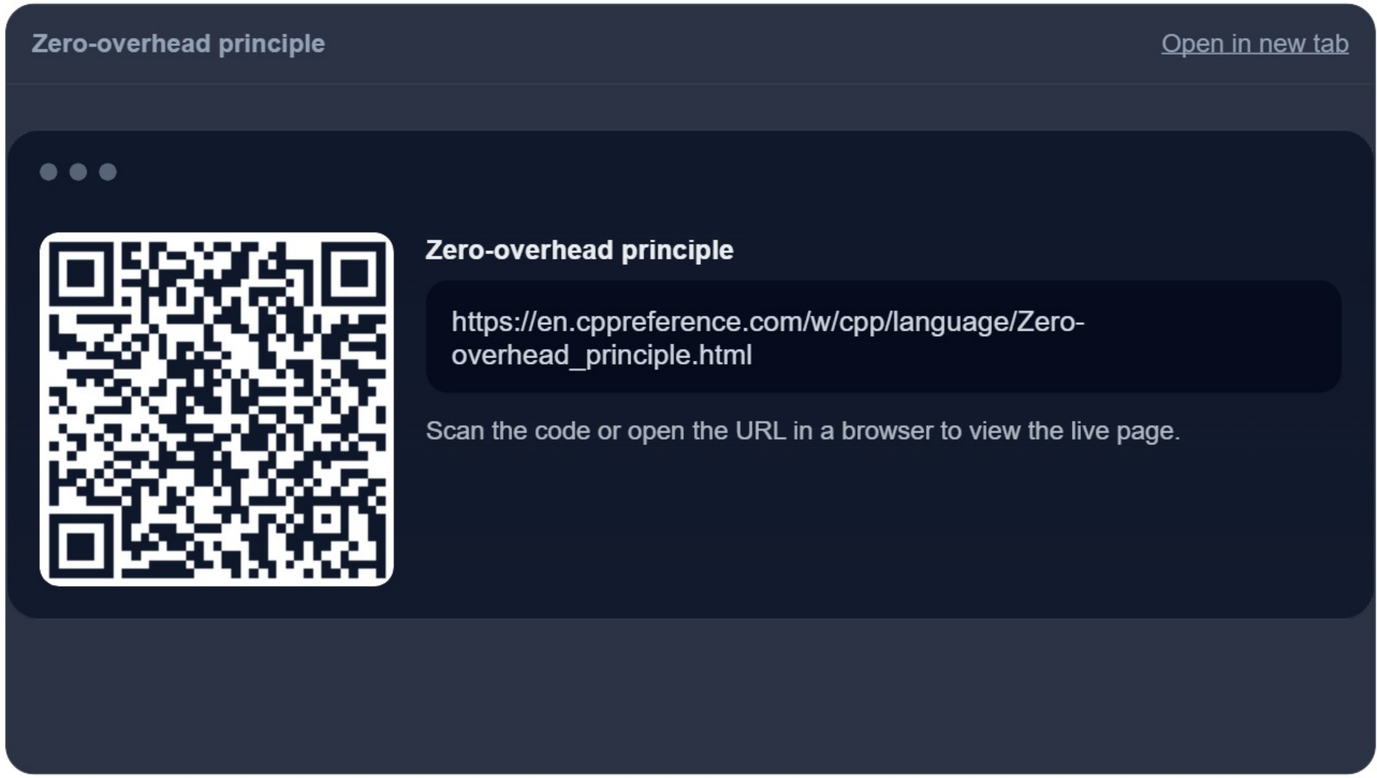
Code Console CPP Run

```cpp
#include <vector>
#include <iostream>
int main() {
    std::vector<int> vec { 1, 2, 3, 4 };
    vec.push_back(5);
    vec.push_back(6);
    vec[1] = 20;

    for (size_t i = 0; i < 10; i++) {
        std::cout << vec[i] << " ";
    }
}
```

Index Checking?

1. You don't pay for what you don't use.
2. What you do use is just as efficient as what you could reasonably write by hand.



https://en.cppreference.com/w/cpp/language/Zero-overhead_principle.html

How do Vectors Resize?

Resizing vectors copies over the whole array into a bigger array - Dynamic reallocation

Open in new tab

Resizing vectors copies over the whole array into a bigger array - Dynamic reallocation

https://web.stanford.edu/~pseay/cs106l/vectors

Scan the code or open the URL in a browser to view the live page.

106B vs. Standard Vectors

What you want to do?	Stanford Vector<int>	std::vector<int>
Create an empty vector	`Vector<int> v;`	`std::vector<int> v;`
Create a vector with **n** copies of **0**	`Vector<int> v(n);`	`std::vector<int> v(n);`
Create a vector with **n** copies of value **k**	`Vector<int> v(n, k);`	`std::vector<int> v(n, k);`
Add **k** to the end of the vector	`v.add(k);`	`v.push_back(k);`
Clear vector	`v.clear();`	`v.clear();`
Check if **v** is empty	`if (v.isEmpty())`	`if (v.empty())`
Get the element at index **i**	`int v = v.get(i);` `int k = v[i];`	`int k = v.at(i);` `int k = v[i];`
Replace the element at index **i**	`v.get(i) = k;` `v[i] = k;`	`v.at(i) = k;` `v[i] = k;`

Practice with Vectors

Task:
Write a C++ function to calculate the maximum value of a vector, using 4 different vector methods.

https://www.online-ide.com/6lQX5aeEn9

Walkthrough

Code Console CPP Run

```cpp
#include <iostream>
#include <vector>

/* Returns highest temperature,
   or -1 if no temperatures given. */
int findPeakHeat(const std::vector<int>& temps) {
    // Your implementation here
}

int main() {
    // High temperatures forecast for the next 7 days.
    std::vector<int> weeklyForecast = {82, 95, 102, 99, 88, 79, 81};

    std::cout << "--- Weather Report ---" << std::endl;
    std::cout << "Max temp this week will be " << findPeakHeat(weeklyForecast) <<
    std::endl;
    return 0;
}
```

Vectors Don't Reorder Well

Inserting has to shift over elements

Open in new tab

Inserting has to shift over elements

https://web.stanford.edu/~pseay/cs106l/vectors-insert

Scan the code or open the URL in a browser to view the live page.

Deques

Double-**e**nded **que**ue (pronounced "deck")

Like a `vector`, with:

- `push_back`
- `pop_back`

But also has:

- `push_front`
- `pop_front`

Deques

Ex: Maintaining a list of the last 10,000 prices

```cpp
#include <deque>

void receivePrice(deque<double>& prices, double price)
{
    prices.push_front(price);   // Super fast
    if (prices.size() > 10000)
        prices.pop_back();  // Remove last price
                            // so we don't exceed 10k
}
```

Deques Behind the Scenes

How deques are stored behind the scenes

Open in new tab

How deques are stored behind the scenes

https://web.stanford.edu/~pseay/cs106l/deque

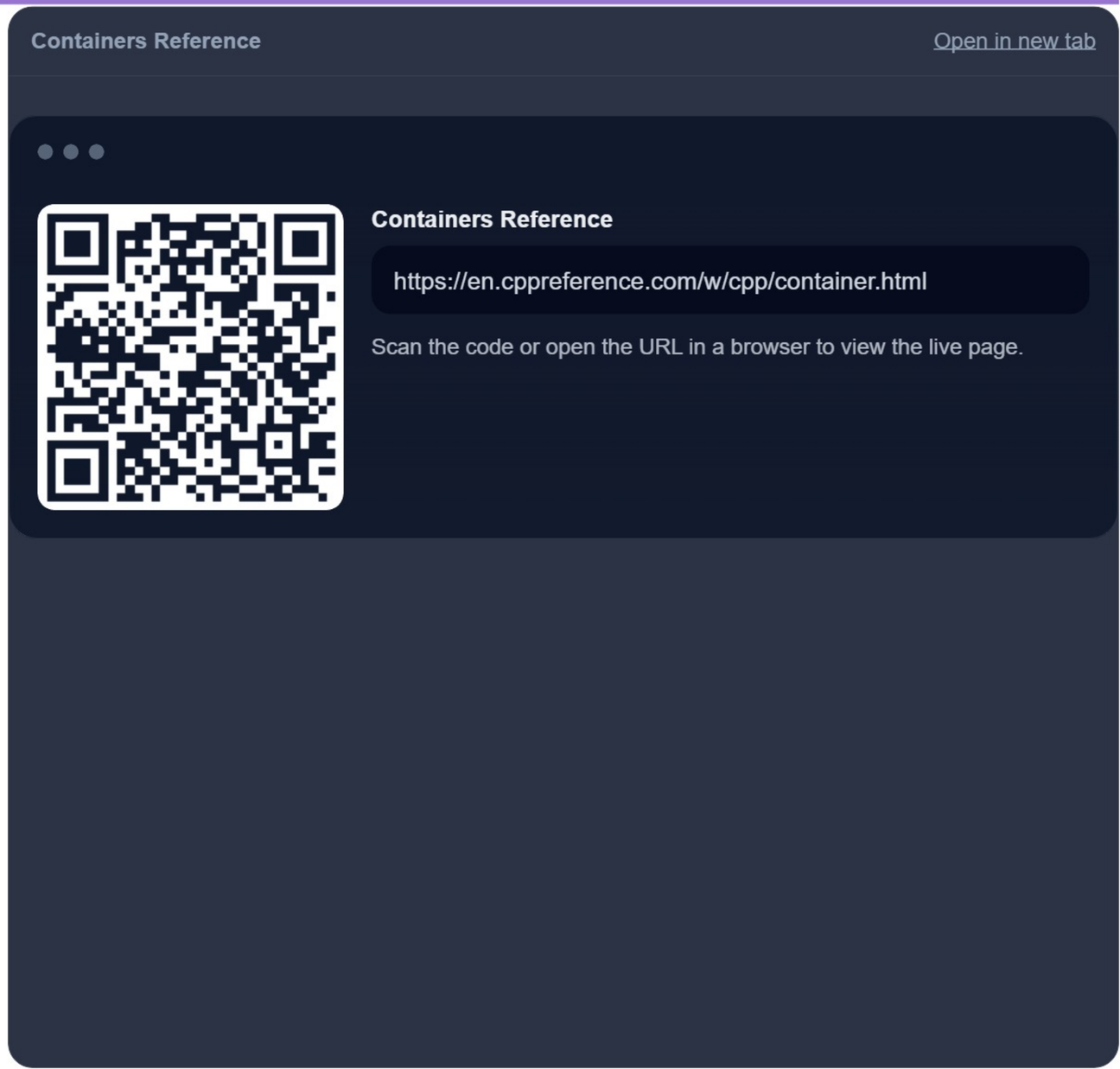
Scan the code or open the URL in a browser to view the live page.

Plan

1. Space-time
2. STL
3. Sequence Containers
4. **Associative Containers**

Associative Containers

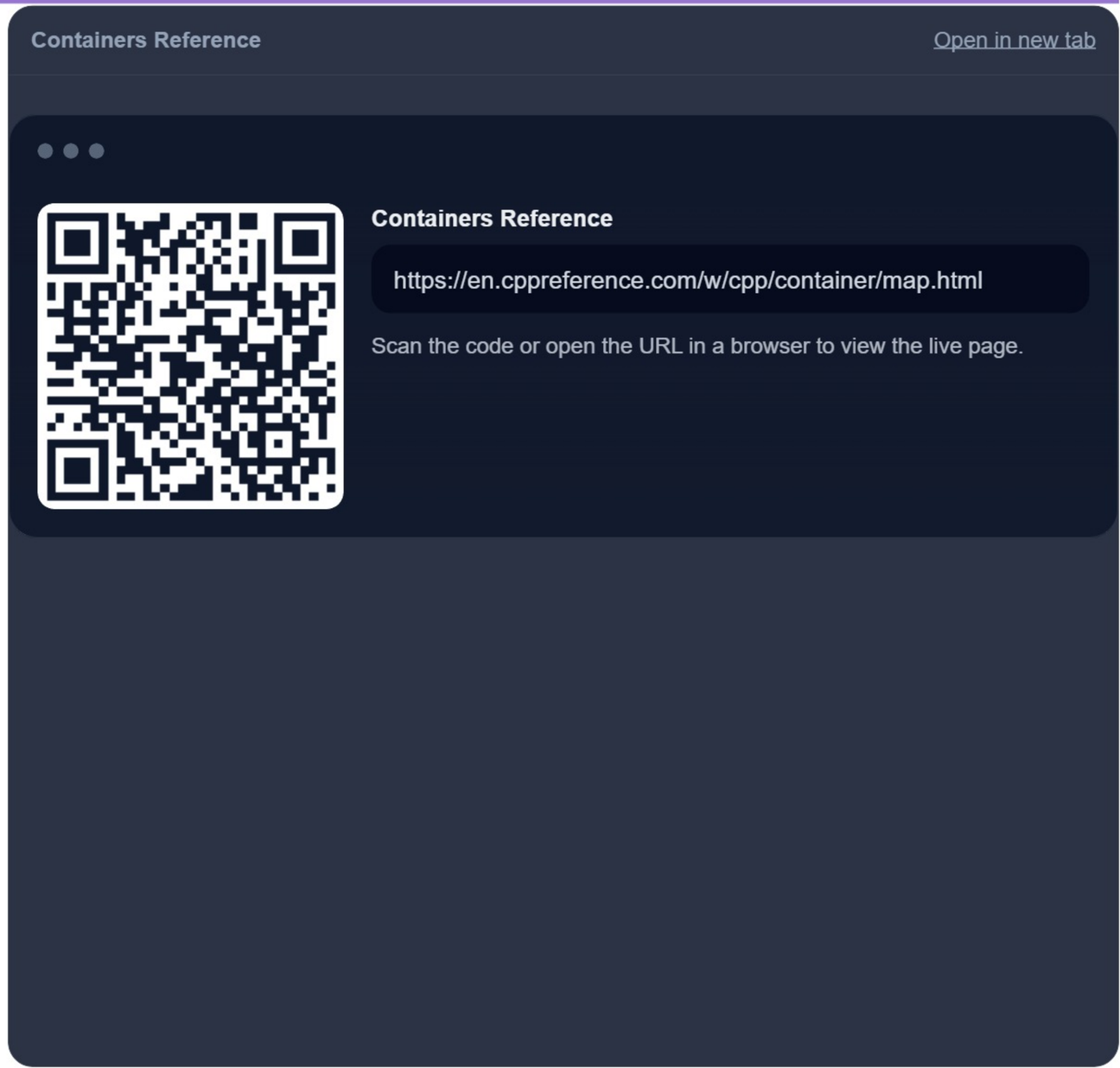
"Associative containers implement **sorted** data structures that can be **quickly searched**."



Maps

A map "contains key-value pairs with unique keys."

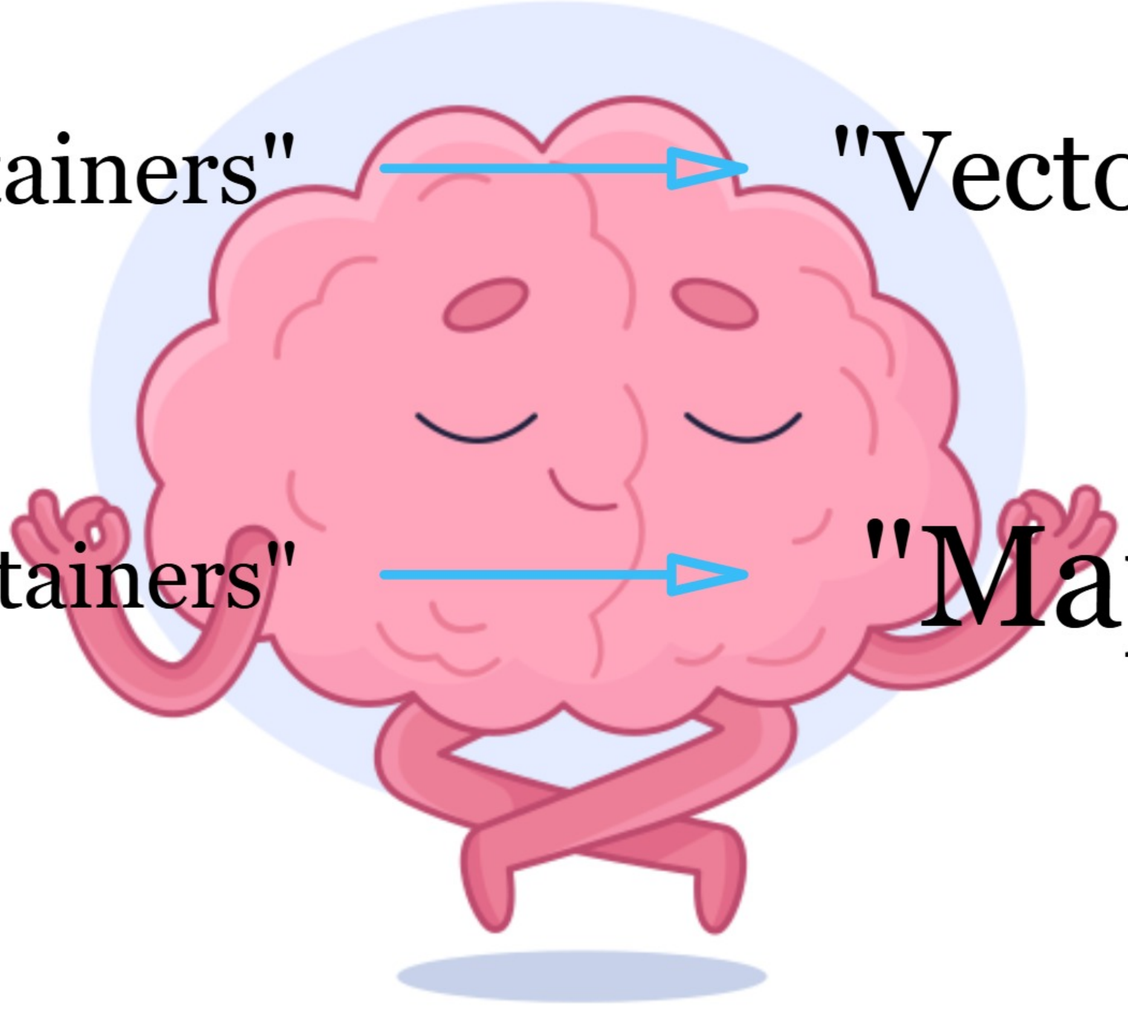
Python calls them "dictionaries"



Maps

"Sequence Containers" → "Vectors & Deques"

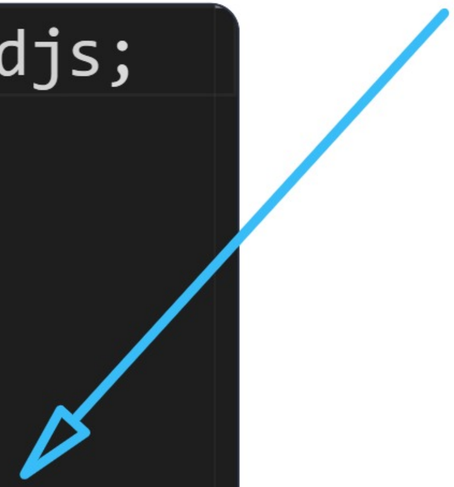
"Associative Containers" → "Maps & Sets"



Maps

We can do this:

```
1 std::map<int, char*> adjs;
2 adjs[106] = "awesome";
3 adjs[103] = "mathy";
4 adjs[107] = "deep";
5
6 std::cout << adjs[106];
```



How is it efficient?

It sorts the pairs by their keys...

Maps

Code Console CPP Run

```cpp
#include <map>
#include <iostream>
int main () {
    std::map<int, char> preston {
        {16, 'p'}, {18, 'r'}, {5, 'e'},
        {19, 's'}, {20, 't'}, {15, 'o'}, {14, 'n'}
    };
    for (const auto& pair : preston) {
        std::cout << pair.first << ' '
                  << pair.second << std::endl;
    }
}
```

A map is a collection of pairs, so here is uniform initialization of pairs.

We loop over each pair.

The pairs are sorted by key.

Maps

A `std::map<K,V>` is a collection of `std::pair<const K, V>`

Code Console CPP Run

```cpp
for (const auto& pair : myMap) {
    auto key = pair.first;
    auto value = pair.second;
}
```

Structured binding 👍

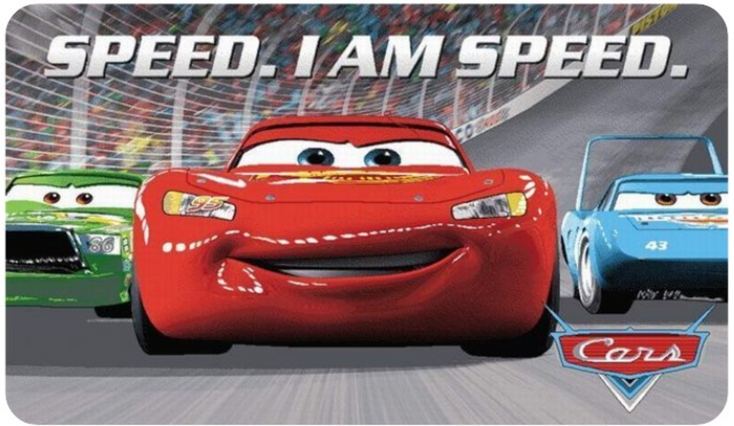
Code Console CPP Run

```cpp
for (const auto& [key, value] : myMap) {

}
```

Maps Storage - Red Black Trees

- Stores pairs in a **binary search tree** (BST)
- Specifically, it uses a **Red-Black Tree**, guaranteeing maximum depth of 2 log (n)
- This makes search **O(log(n))**



Open in new tab

https://web.stanford.edu/~pseay/cs106l/maps-rbt

Scan the code or open the URL in a browser to view the live page.

Maps - Auto Insertion

We can do this:

```
1 std::map<std::string, int> fav_num;
2 fav_num["Preston"] = 2;
3
4 std:: cout << "Preston's is " << fav_num["Preston"] <<
5     " and Rachel's is " << fav_num["Rachel"] << '\n';
```

"Ummmm.... we never defined that???"

The map automatically inserts the default value of the object.

(It actually does this every time, even on `line 2`)

Sets

Amoral maps

Maps without values

Unique Objects

Sets visualizer

Open in new tab

Sets visualizer

https://web.stanford.edu/~pseay/cs106l/sets-rbt

Scan the code or open the URL in a browser to view the live page.

Map Syntax

What you want to do?	Stanford Map<char, int>	std::map<char, int>
Create an empty map	Map<char, int> m;	std::map<char, int> m;
Add key **k** with value **v** into the map	m.put(k, v); m[k] = v;	m.insert({k, v}); m[k] = v;
Remove key **k** from the map	m.remove(k);	m.erase(k);
Check if **k** is in the map ***(* C++20)***	if (m.containsKey(k))	if (m.count(k)) if (m.contains(k)) ***(*)***
Check if the map is empty	if (m.isEmpty())	if (m.empty())
Retrieve or overwrite value associated with key k *(auto-insert default if doesn't exist)*	int i = m[k]; m[k] = i;	int i = m[k]; m[k] = i;

Set Syntax

What you want to do?	Stanford Set<char>	std::set<char>
Create an empty set	Set<char> s;	std::set<char> s;
Add k to the set	s.add(k);	s.insert(k);
Remove k from the set	s.remove(k);	s.erase(k);
Check if k is in the set *(* C++20)*	if (s.contains(k))	if (s.count(k)) if (s.contains(k)) *(*)*
Check if the set is empty	if (s.isEmpty())	if (s.empty())

Practice

Task:
Find the double agents - those who are in multiple departments.

https://www.online-ide.com/A4IvoxQ8Pr

Practice

Code Console CPP Run

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <set>

std::set<std::string> findDoubleAgents(std::map<std::string, std::set<std::string>> departments) {
    std::set<std::string> seen, doubleAgents;

    // Your code here

    return doubleAgents;
}

int main() {
    std::map<std::string, std::set<std::string>> company = {
```

There are also versions with non-unique keys...

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

`set`	collection of unique keys, sorted by keys (class template)
`map`	collection of key-value pairs, sorted by keys, keys are unique (class template)
`multiset`	collection of keys, sorted by keys (class template)
`multimap`	collection of key-value pairs, sorted by keys (class template)

The Caveat of Associative Containers

"Wait... how do they get sorted?"

std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```

This is how... they must be comparable, so it uses the "less than" comparator by default.

"How do we make memory for it?"

The Caveat of Associative Containers

Not all elements are comparable by default...

`int`, `double`, `string`

`ifstream`, `Course`

Plan

1. Space-time
2. STL
3. Sequence Containers
4. **Associative Containers**

Plan

1. Space-time
2. STL
3. Sequence Containers
4. Associative Containers
   - **Bonus - Unordered Associative Containers**

Containers library

The Containers library is a generic collection of clas
implement common data structures like queues, lis
containers:

- sequence containers,
- associative containers,
- unordered associative containers, (since C++11)

Unordered Associative Containers

`map` → `unordered_map`

`set` → `unordered_set`

Essentially optimized versions of map and set.

Unordered Associative Containers

```
1  std::map<int, std::string> courses{
2      {103, "Bailey/Aiken"},
3      {107, "Cain"},
4      {109, "Gregg"}
5  };
```

Drop-in replacement

```
1  std::unordered_map<int, std::string> courses{
2      {103, "Bailey/Aiken"},
3      {107, "Cain"},
4      {109, "Gregg"}
5  };
```

The Caveat - Hashing

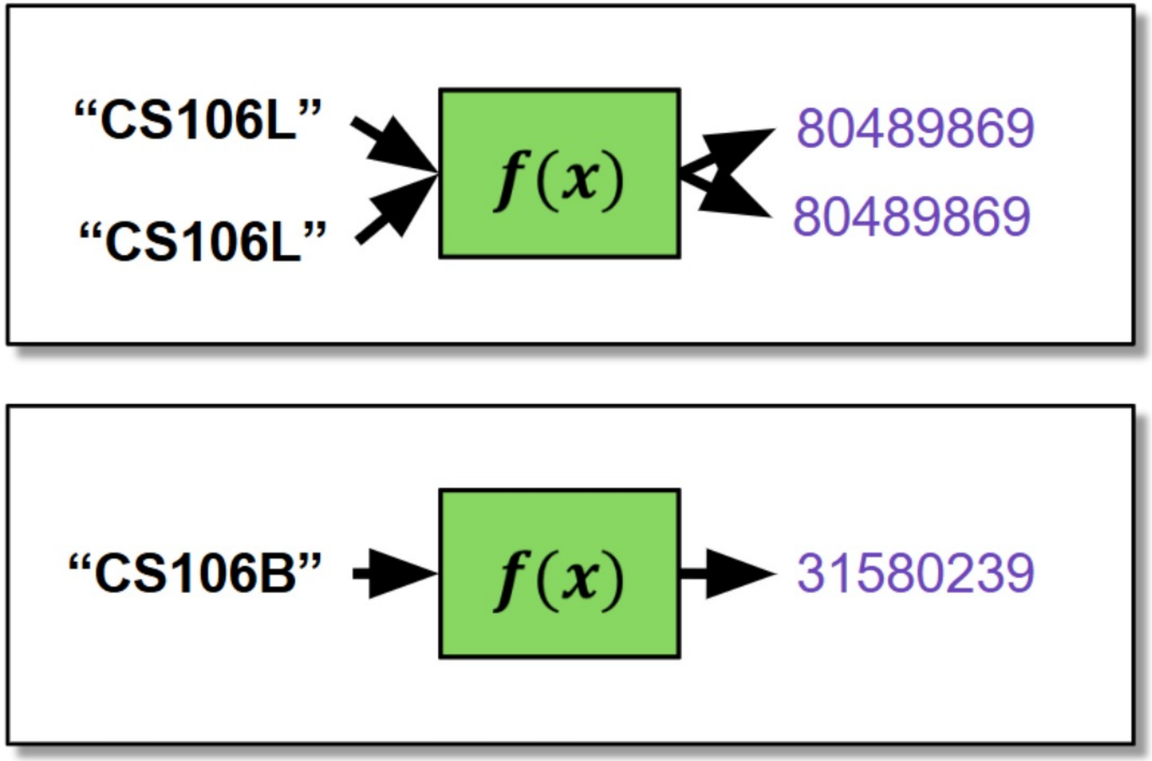
std::unordered_map

Defined in header <unordered_map>

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class unordered_map;
```

What is a hash function?

- "Scrambles" a key into a size_t (64 bit)
- Small changes in the input should produce large changes in the output



Unordered Associative Containers

```
1  std::map<int, std::string> courses{
2      {103, "Bailey/Aiken"},
3      {107, "Cain"},
4      {109, "Gregg"}
5  };
```

Drop-in replacement

```
1  std::unordered_map<int, std::string> courses{
2      {103, "Bailey/Aiken"},
3      {107, "Cain"},
4      {109, "Gregg"}
5  };
```

Unordered Map Implementation

Hash Table

Visualization

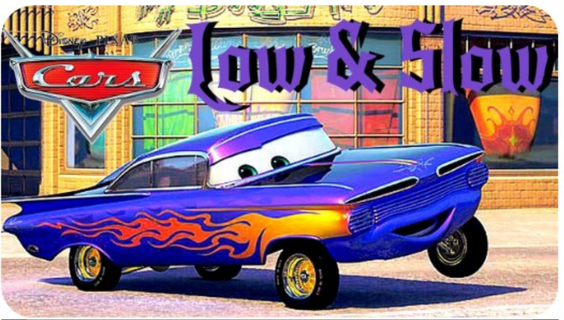
Open in new tab

Visualization

https://web.stanford.edu/~pseay/cs106l/hash-map

Scan the code or open the URL in a browser to view the live page.

Speed



Vectors

$O(n)$

Maps

$O(log(n))$

Unordered Maps

$O(1)$

Recap

Summary of Data Structures

	i^{th} element	Search	Insertion	Erase
std::vector	Very Fast	Slow	Slow	Slow
std::deque	Fast	Slow	Fast (front/back) Slow (all others)	Fast (front/back) Slow (all others)
std::set	Slow	Fast	Fast	Fast
std::map	Slow	Fast	Fast	Fast
std::unordered_set	N/A	Very Fast	Very Fast	Very Fast
std::unordered_map	N/A	Very Fast	Very Fast	Very Fast

More Data Structures = More Fun

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

`array` (C++11)	fixed-sized inplace contiguous array (class template)
`vector`	resizable contiguous array (class template)
`inplace_vector` (C++26)	resizable, fixed capacity, inplace contiguous array (class template)
`hive` (C++26)	collection that reuses erased elements' memory (class template)
`deque`	double-ended queue (class template)
`forward_list` (C++11)	singly-linked list (class template)
`list`	doubly-linked list (class template)

Container adaptors

Container adaptors provide a different interface for sequential containers.

`stack`	adapts a container to provide stack (LIFO data structure) (class template)
`queue`	adapts a container to provide queue (FIFO data structure) (class template)
`priority_queue`	adapts a container to provide priority queue (class template)
`flat_set` (C++23)	adapts a container to provide a collection of unique keys, sorted by keys (class template)
`flat_map` (C++23)	adapts two containers to provide a collection of key-value pairs, sorted by unique keys (class template)
`flat_multiset` (C++23)	adapts a container to provide a collection of keys, sorted by keys (class template)
`flat_multimap` (C++23)	adapts two containers to provide a collection of key-value pairs, sorted by keys (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

`set`	collection of unique keys, sorted by keys (class template)
`map`	collection of key-value pairs, sorted by keys, keys are unique (class template)
`multiset`	collection of keys, sorted by keys (class template)
`multimap`	collection of key-value pairs, sorted by keys (class template)

Unordered associative containers (since C++11)

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ($O(1)$ average, $O(n)$ worst-case complexity).

`unordered_set` (C++11)	collection of unique keys, hashed by keys (class template)
`unordered_map` (C++11)	collection of key-value pairs, hashed by keys, keys are unique (class template)
`unordered_multiset` (C++11)	collection of keys, hashed by keys (class template)
`unordered_multimap` (C++11)	collection of key-value pairs, hashed by keys (class template)

cppreference.com

Summary

1. **Space-time**
    - Tradeoffs
2. **STL**
    - Standard Template Library
3. **Sequence Containers**
    - Ex: `vector`, `deque`, ...
4. **(Unordered) Associative Containers**
    - Ex: (`unordered_`) `map`, `set`, ...