Back To Basics
Standard Library Containers

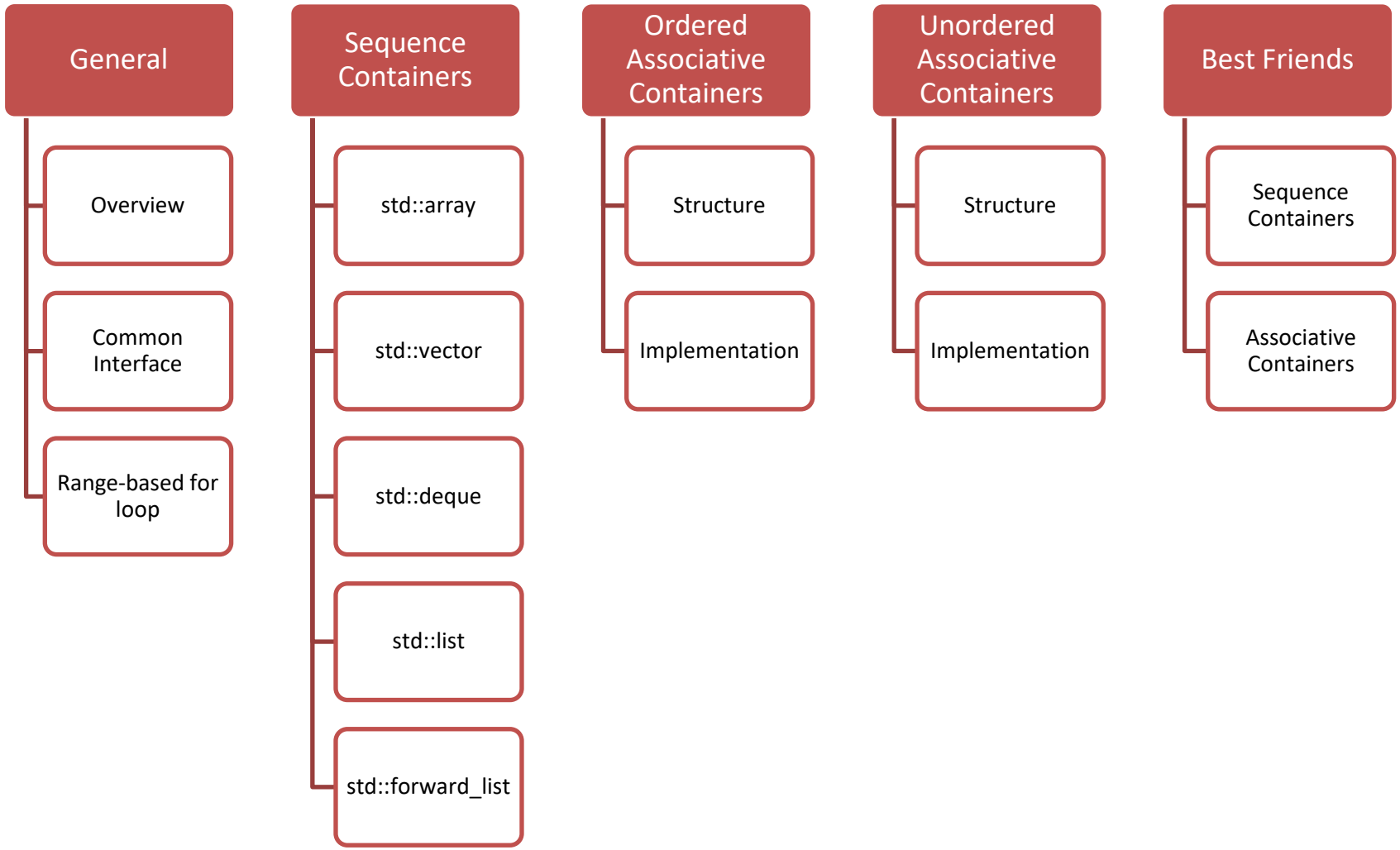
RAINER GRIMM

Cppcon
The C++ Conference

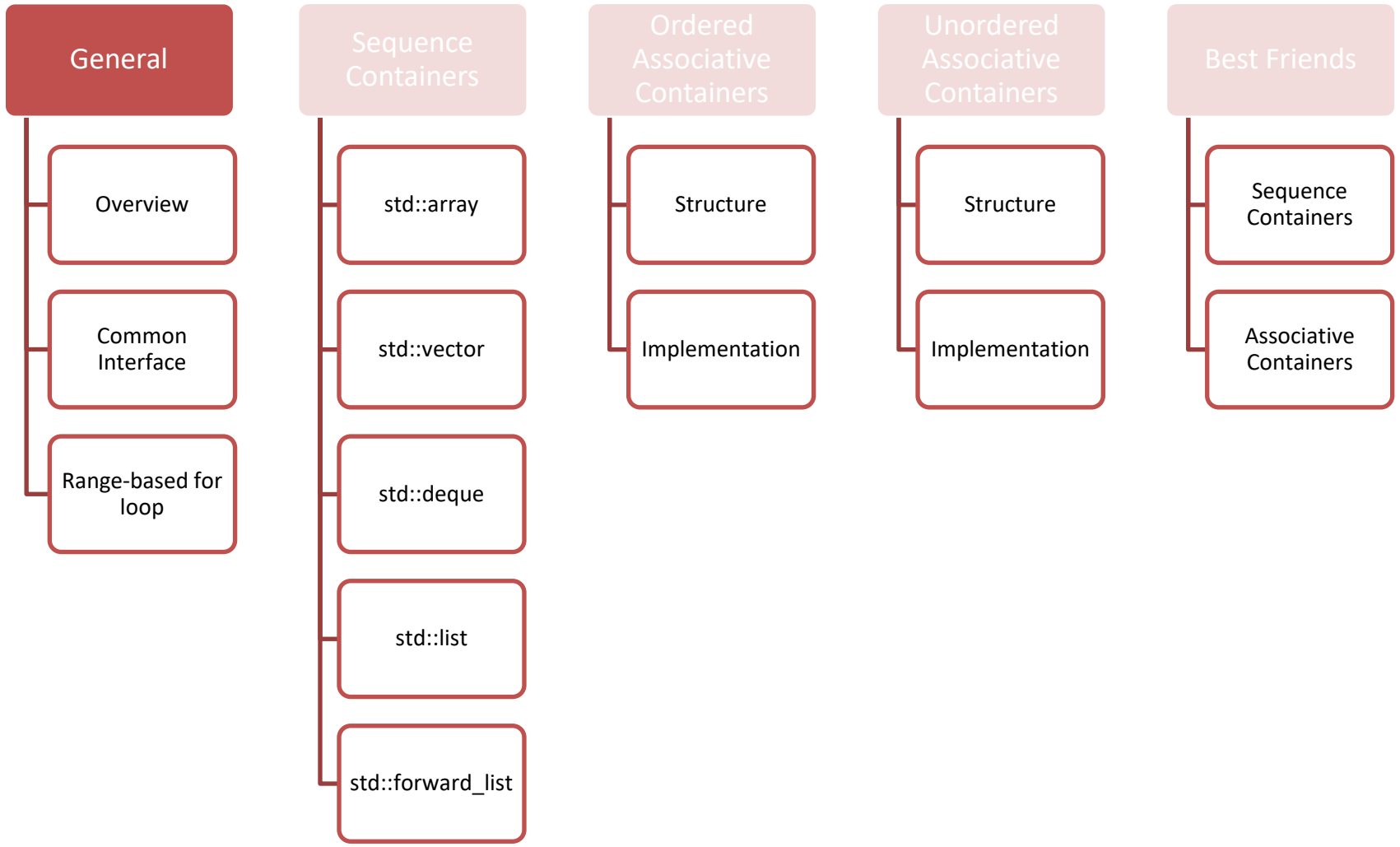
2022

September 12th-16th

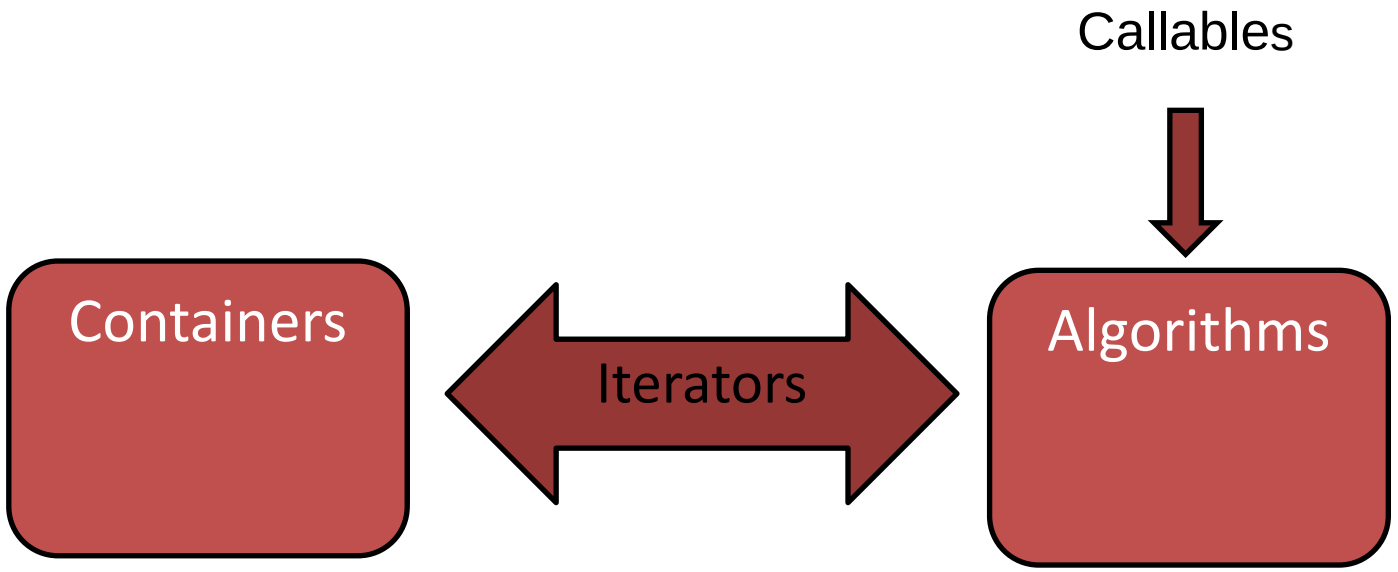
Containers of the STL



Containers of the STL



The Standard Template Library: STL



The Containers of the STL

Sequence Containers

```
std::array
std::vector
std::deque
std::list
std::forward_list
```

Ordered Associative Containers

```
std::map
std::set
std::multimap
std::multiset
```

Unordered Associative Containers

```
std::unordered_map
std::unordered_set
std::unordered_multimap
std::unordered_multiset
```

Interface of the Containers

The Containers

- have a type parameter(s) and an allocator.

```
template<typename T, typename Allocator = std::allocator<T>>
class vector;
```

- support the same basic functionality.
- provide value semantics.

Exceptions

- `std::array` has a fixed size → you cannot change its size during run time
- `std::forward_list` does not know its length

`std::string` is quite similar to `std::vector<char>`.

Create and Delete

Constructors

- Default
```
std::vector<int> first;
```
- Count
```
std::vector<int> sec(5);
```
- Range
```
std::vector<int> third(sec.begin(), sec.end());
```
- Copy
```
std::vector<int> fourth(third);
```
- Move
```
std::vector<int> fifth(std::move(fourth));
```
- Sequence
```
std::vector<int> sixth{1, 2, 3, 4, 5};
```

Create and Delete

Destructor

```
std::vector<int> first;
delete first;
```

Removing

```
std::vector<int> sixth{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
sixth.clear();

std::erase(sixth, 5);
std::erase_if(sixth, [](auto i){ return i >= 3; });
```

Determine the Size

```
std::vector<std::string> first{"one", "two", "three", "four"};
```

Empty?

```
first.empty();
```

Numbers of elements?

```
first.size();
```

Maximal size?

```
first.max_size();
```

Assignment and Swap

```
std::vector<std::string> first{"one", "two", "three", "four"};
std::vector<std::string> second{"five", "six"};
```

Assignment

```
second = first;
second = std::move(first);
second = {"seven", "eight"};
```

Swap

```
second.swap(first);
std::swap(second, first)
```

Comparison

All Containers

`==` and `!=`

Sequence and Ordered Associative Containers

`<`, `<=`, `>`, and `>=`

Rules

- The containers must have the same type.
- Two containers are equal, if they have the same elements in the same sequence (applies to sequence containers and ordered associative containers).
- The containers are compared lexicographically.

The Range-Based for loop

Syntax:

```
for (Declaration: Sequence){ . . .
```

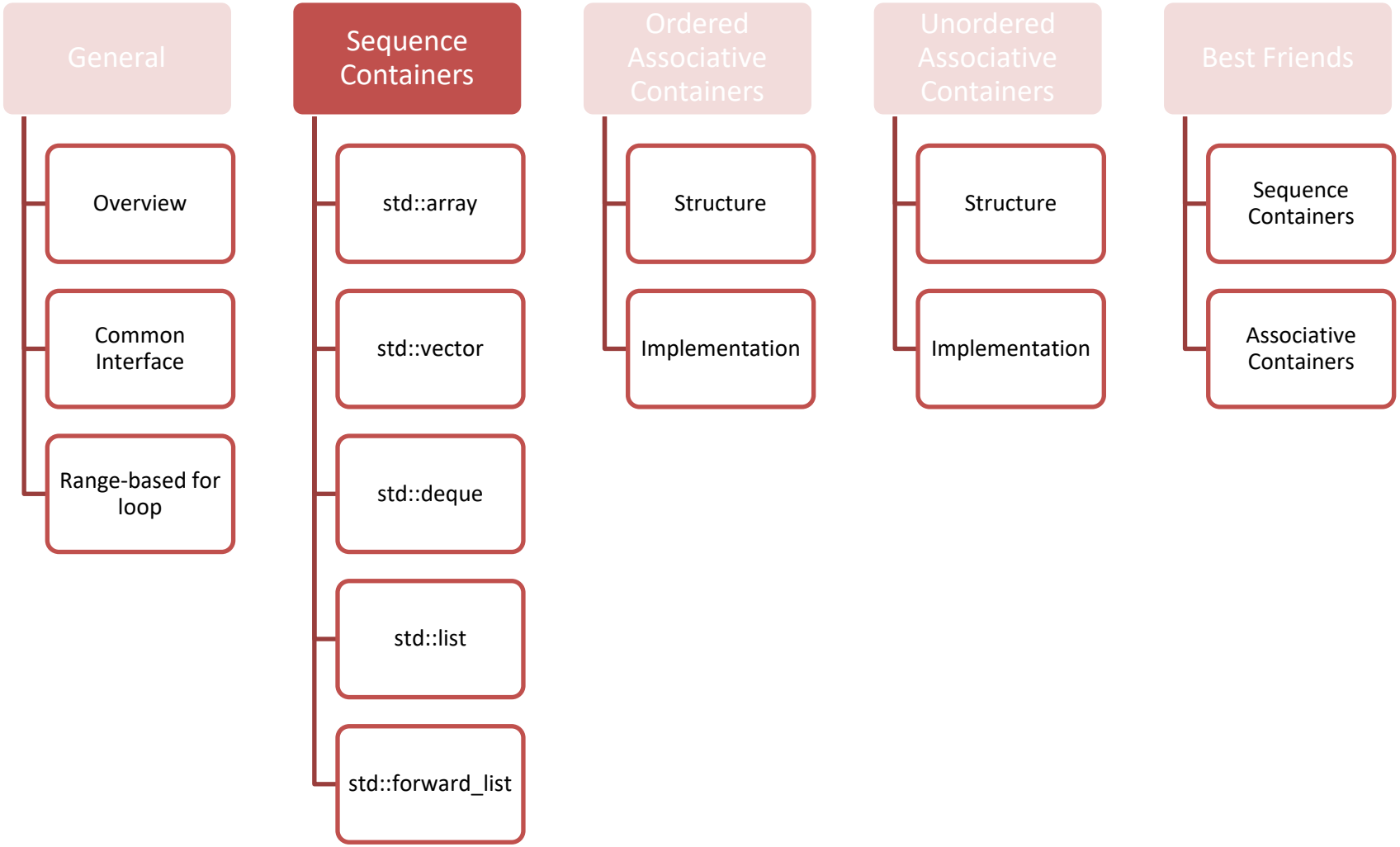
Sequence: initializer list, C array, C++ string, STL container, or range

```
std::vector<int> vec{1, 2, 3, 4, 5};
for (auto v: vec) std::cout << v << " ";        // 1 2 3 4 5
```

You must take the arguments by reference to modify them.

```
int array[5] = {1, 2, 3, 4, 5};
for (auto& a: array) a *=2;
for (auto a: array) std::cout << a << " ";        // 2 4 6 8 10
```

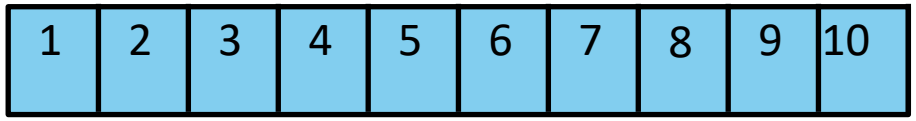
Containers of the STL



Sequence Containers: Overview

Characteristic	`std::array`	`std::vector`	`std::deque`	`std::list`	`std::forward_list`
Size	static	dynamic	dynamic	dynamic	dynamic
Implementation	static array	dynamic array	sequence of arrays	doubly linked list	singly linked list
Access	random access	random access	random access	for- and backward	forward
Optimized for		end O(1)	begin and end O(1)	begin and end O(1)	begin O(1)
Memory reservation		yes	no	no	no
Memory release		`shrink_to_fit()`	`shrink_to_fit()`	always	always
Iterator invalidation		yes	yes	no	no
Strength	▪ no memory allocation ▪ minimal memory requirements	95% solution	insert and delete at the begin and end	insert and delete at each position	▪ fast insertion and deletion ▪ minimal memory requirements
Weakness	no dynamic memory allocation	insertion and deletion at arbitrary positions O(n)	insertion and deletion at arbitrary positions 0(n)	no random access	no random access

std::array




```
std::array<int,10> myArr{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

std::array

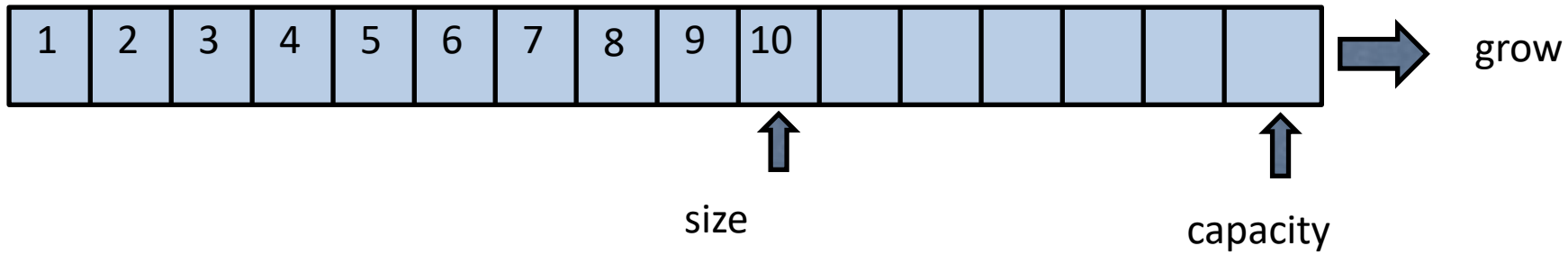
- is a homogeneous container of fixed length.
- combines the memory and performance characteristics of the C array with the interface of a C++ vector.
- knows it length.

std::array

Aggregat Initialization

- `std::array<int, 10> arr;` Elements are not initialized
- `std::array<int, 10> arr{};` Elements are default initialized
- `std::array<int, 10> arr{1, 2, 3, 4, 5};` Remaining elements are default initialized

std::vector




```
std::vector<int> myInt{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

std::vector

- it's a homogenous container of variable length.
- manages automatically its memory.
- stores it elements continuously. → Supports the index operator
- reserves more memory than needed. → reduces expensive memory allocation

std::vector

Special elements

`vec.front()` ⇒ first element (not checked)

`vec.back()` ⇒ last element (not checked)

Index access

`vec[n]` ⇒ vector boundaries are not checked

`vec.at(n)` ⇒ vector boundaries are checked (`std::out_of_range` exception)

Pointer arithmetic

`&vec[i]` ≡ `&vec[0] + i`

std::vector

Elements

- Assign

  `vec.assign( ... )`

- Insert

  `vec.insert(pos, ... ), vec.push_back(elem)`

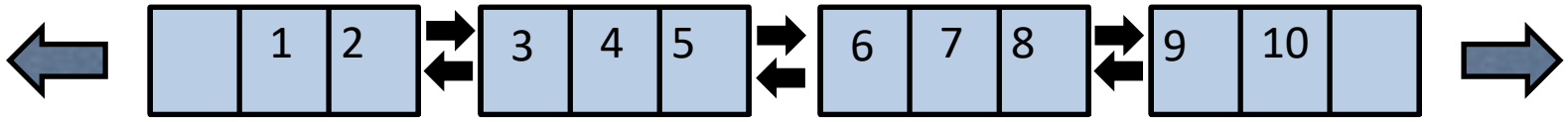
- In-place creation

  `vec.emplace(pos, args ...), vec.emplace_back(args ...)`

- Clear

  `vec.pop_back(), vec.erase(...), vec.clear()`

std::deque

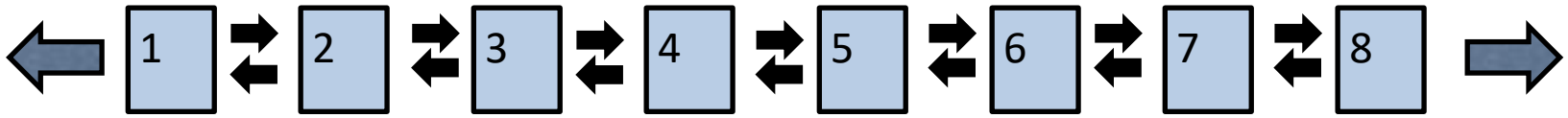



```
std::deque<int> deq{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

`std::deque` (**d**ouble **e**nded **que**ue)

- Extends the interface of `std::vector`
  - `deq.push_front(elem)`, `deq.pop_front()`, and `deq.emplace_front(args ...)`

std::list



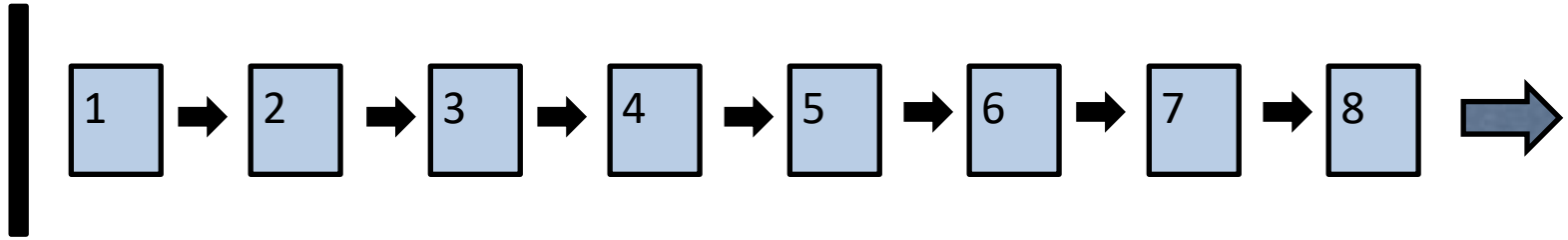

```
std::list<int> lis{1, 2, 3, 4, 5, 6, 7, 8};
```

`std::list`

- is pretty different to `std::array`, `std::vector`, and `std::deque`.
- supports fast access at the front and end of the list.

`std::list` has special member functions optimized for pointer manipulation.

std::forward_list



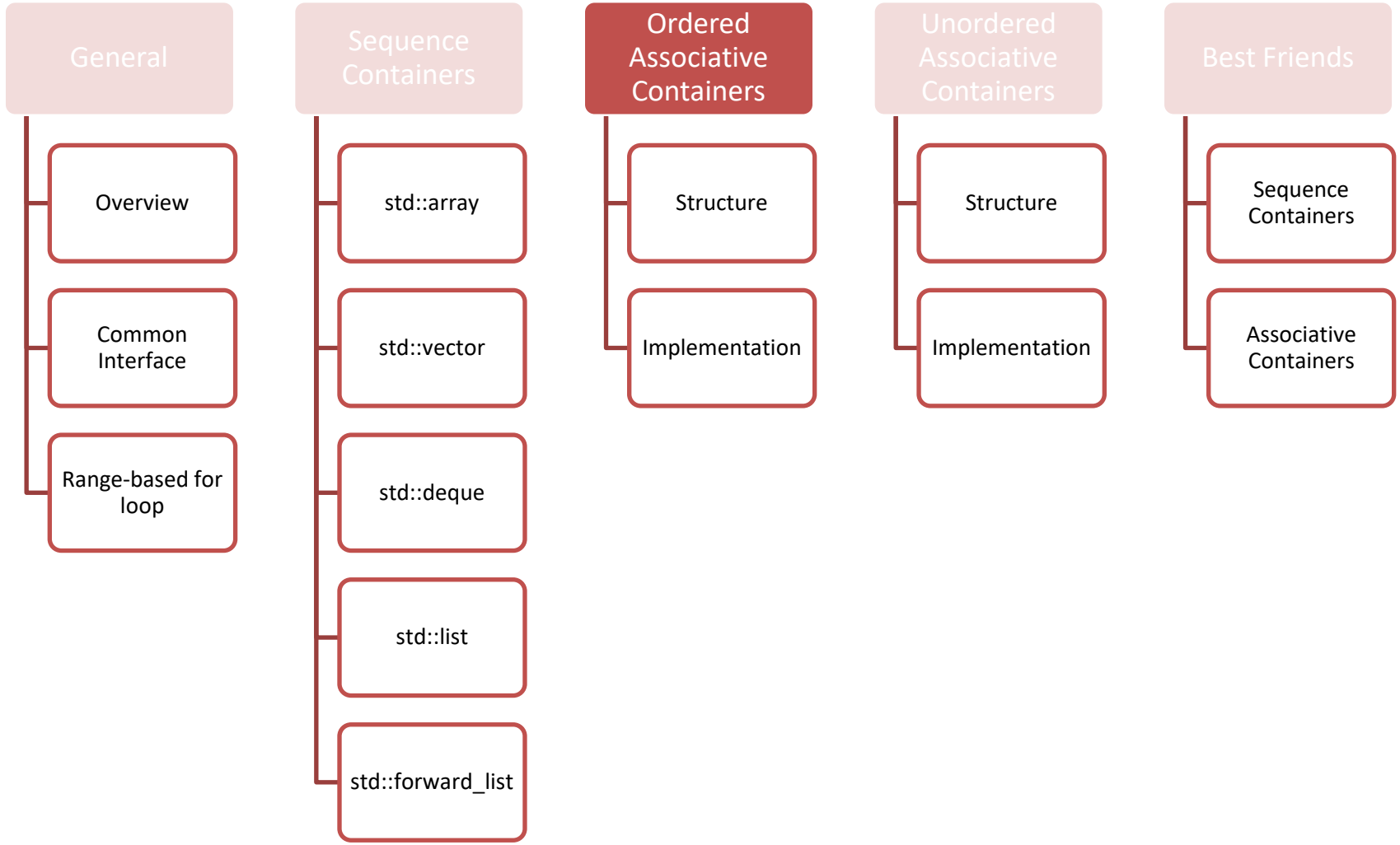

```
std::forward_list<int> for{1, 2, 3, 4, 5, 6, 7, 8};
```

`std::forward_list`

- is a single linked list.
- is similar to `std::list`, but with a restricted interface.
- is optimized for minimal memory requirements.

`std::forward_list` is designed for the special use case.

Containers of the STL



Ordered Associative Containers

The ordered associative containers

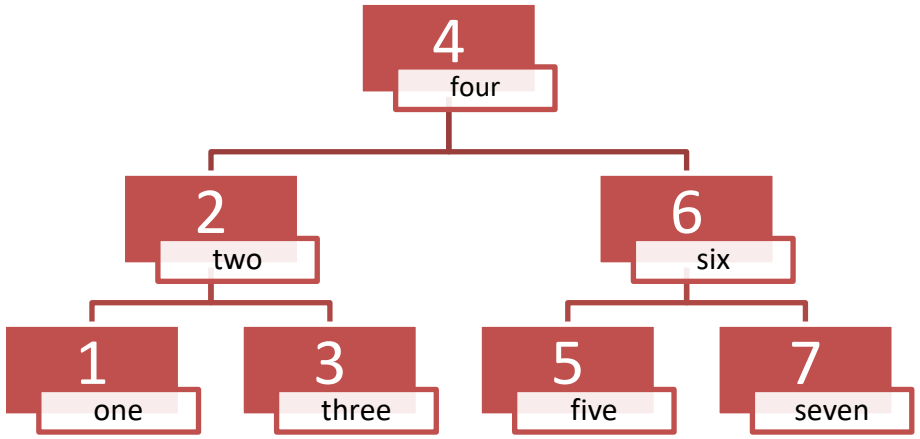
- are *analogous phone books*.

Ordered Associative Containers	Value Associated	More Identical Keys	Header
`std::set`	no	no	`<set>`
`std::multiset`	no	yes	`<set>`
`std::map`	yes	no	`<map>`
`std::multimap`	yes	yes	`<map>`

- have a type(s), a comparison function, and an allocator.

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

Ordered Associative Containers




```
std::map<int,std::string> int2String{{3, "three"}, {2, "two"},
                                     {1, "one"}, {5, "five"},
                                     {6, "six"}, {4, "four"},
                                     {7, "seven"} };
```

Ordered associative Containers are

- binary, balanced search trees.
- sorted in ascending order (by default).

Ordered Associative Containers

`std::map`

- is the most popular ordered associative container.
- supports the index operator [ ].

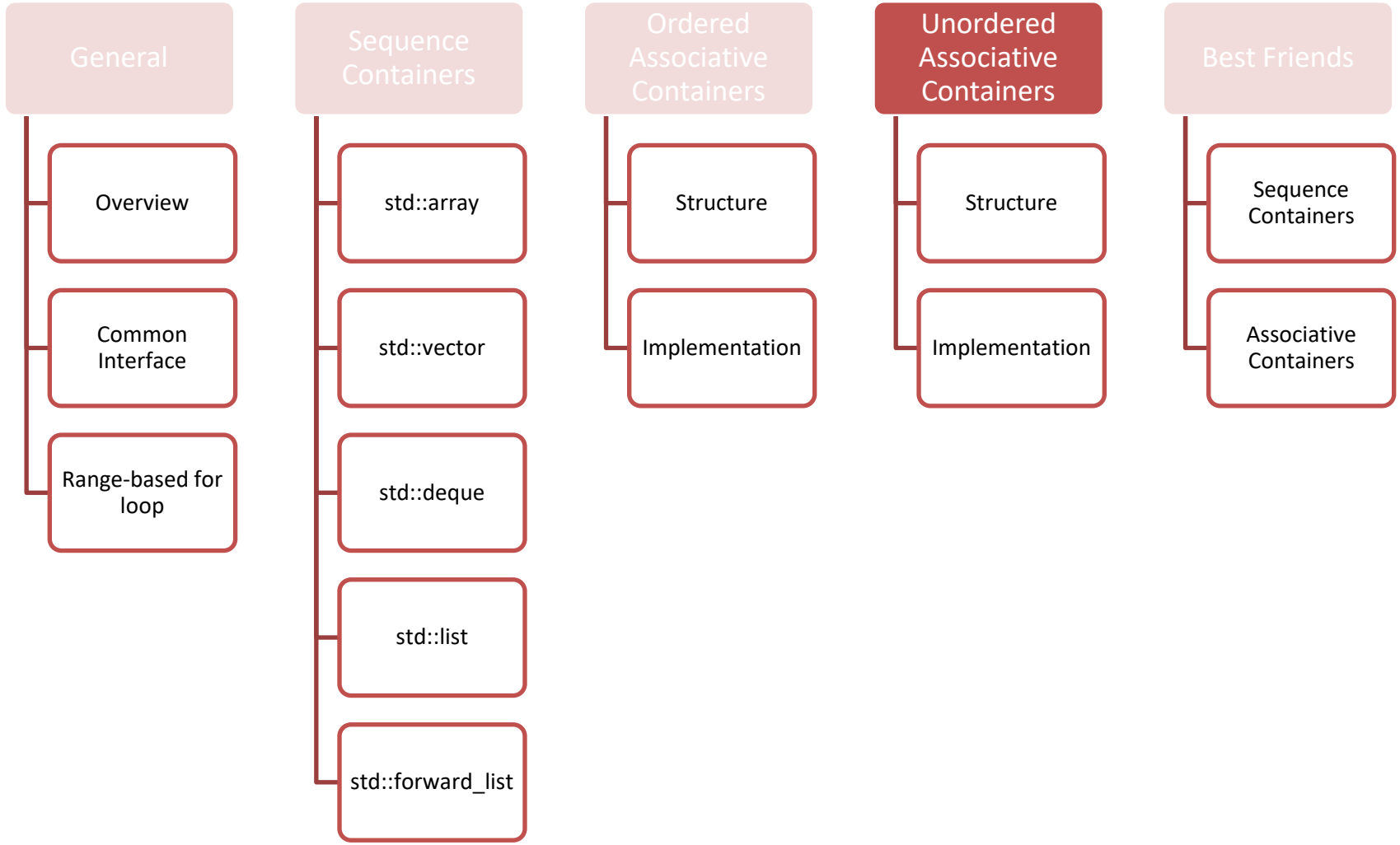
Index operator [ ]

- enables the reading and writing access.
- Creates a new key/value pair if the key is not available,
  - ➡ invokes the default constructor for the value.

`at` operator

- Allows to read a key without creating the value
  - ➡ `std::out_of_range` exception

Containers of the STL



Unordered associative Containers

The unordered associative Containers

- are *digital phone books.*

Unordered Associative Containers	Value Available	More Identical Keys	Header
`std::unordered_set`	no	no	`<unordered_set>`
`std::unordered_multiset`	no	yes	`<unordered_set>`
`std::unordered_map`	yes	no	`<unordered_map>`
`std::unordered_multimap`	yes	yes	`<unordered_map>`

- are also known as dictionary, associative arrays, or hash tables.
- extend the interface of the ordered associative containers.

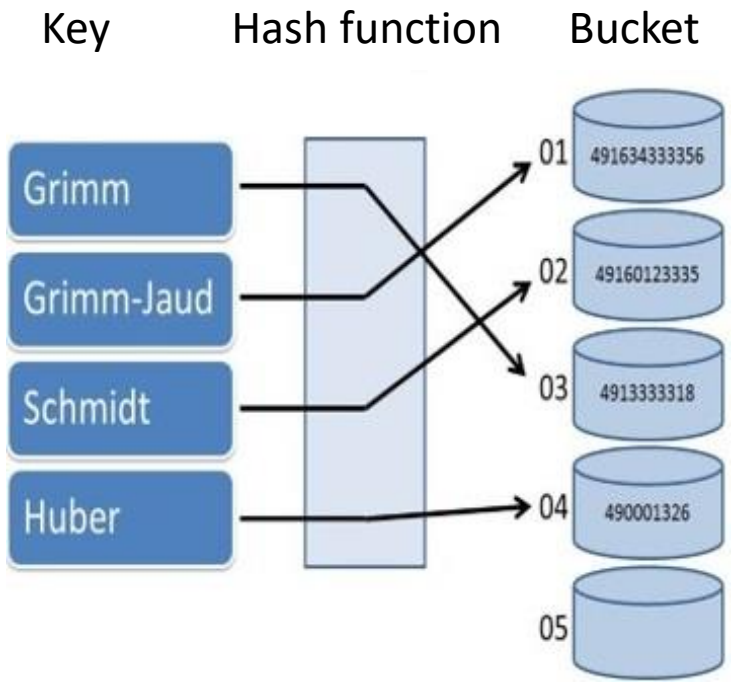
Unordered associative Containers

The unordered associative Containers

- have a data type(s), a hash function, a equal function, and an allocator.

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

Unordered Associative Containers




```
std::unorderet_map<std::string, int> {{"Grimm", 4916343333},
                                      {"Grimm-Jaud", 491601233},
                                      {"Schmidt", 49133318},
                                      {"Huber", 4900013}};
```

The hash function maps the key in constant time to its index.

Unordered Associative Containers

Collision:

- Unordered associative containers store their keys in the buckets.
- Different keys with the same hash value can be stored in the same bucket.
- The access time of the bucket is constant, the search in the bucket is linear.

Capacity:

- Number of buckets

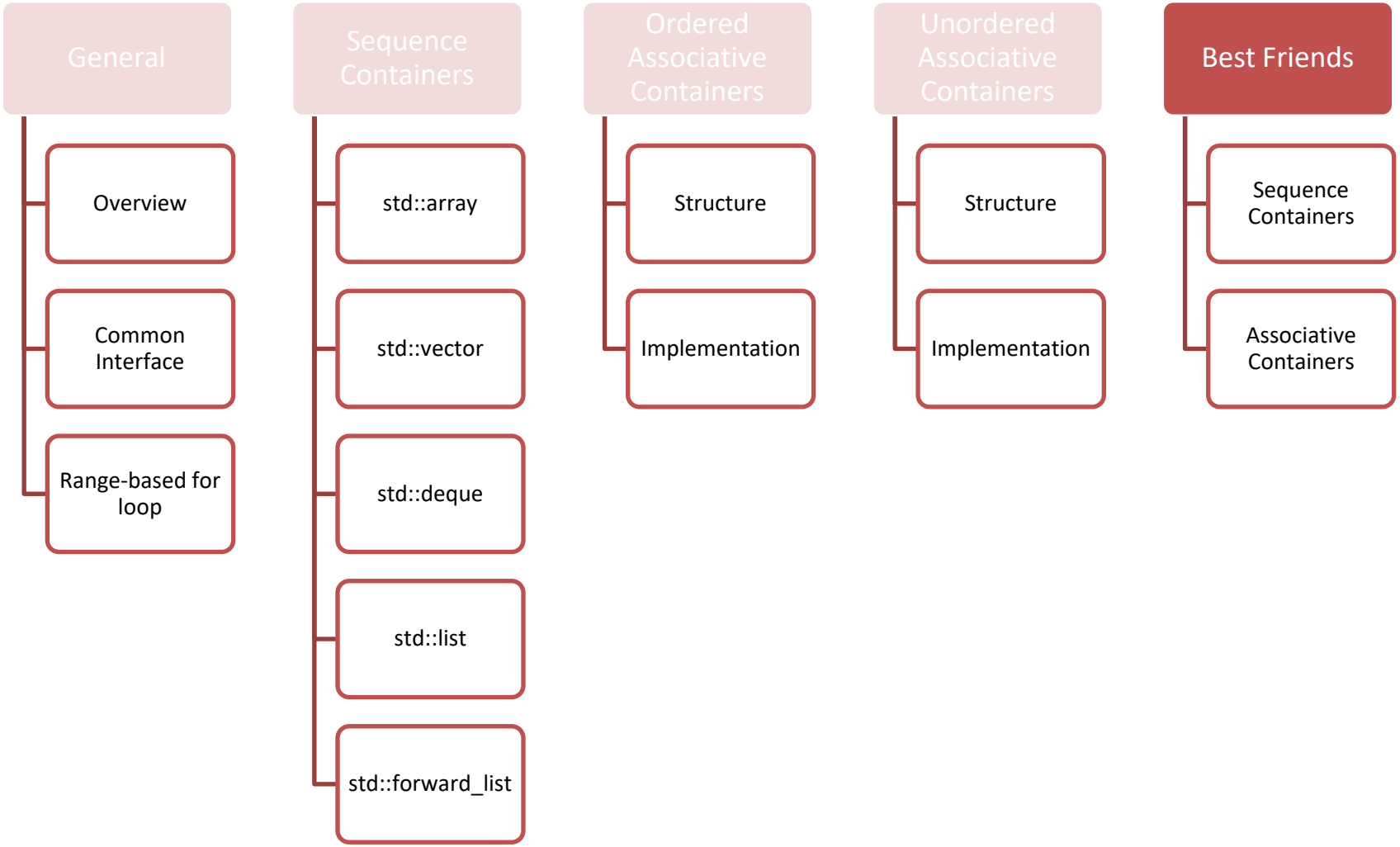
Load factor:

- Average number of elements of each bucket.

Rehashing:

- New buckets are created per default, if the load factor is bigger than 1.

Containers of the STL



`std::array` and `std::vector`

Prefer `std::array` and `std::vector` to a C-array

- The container size is know at compile time and small → `std::array`
- The container size is not known at compile time or big → `std::vector`

- `std::vector` and `std::array`
  - know it's size.
  - automatically manage its memory (RAII).
  - allow the protected element access via the `at`-operator.
  - have an ideal memory layout.

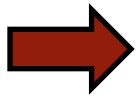
`std::array` and `std::vector` should be your first choice for a sequence container.

Sequence Containers

Performance Comparison

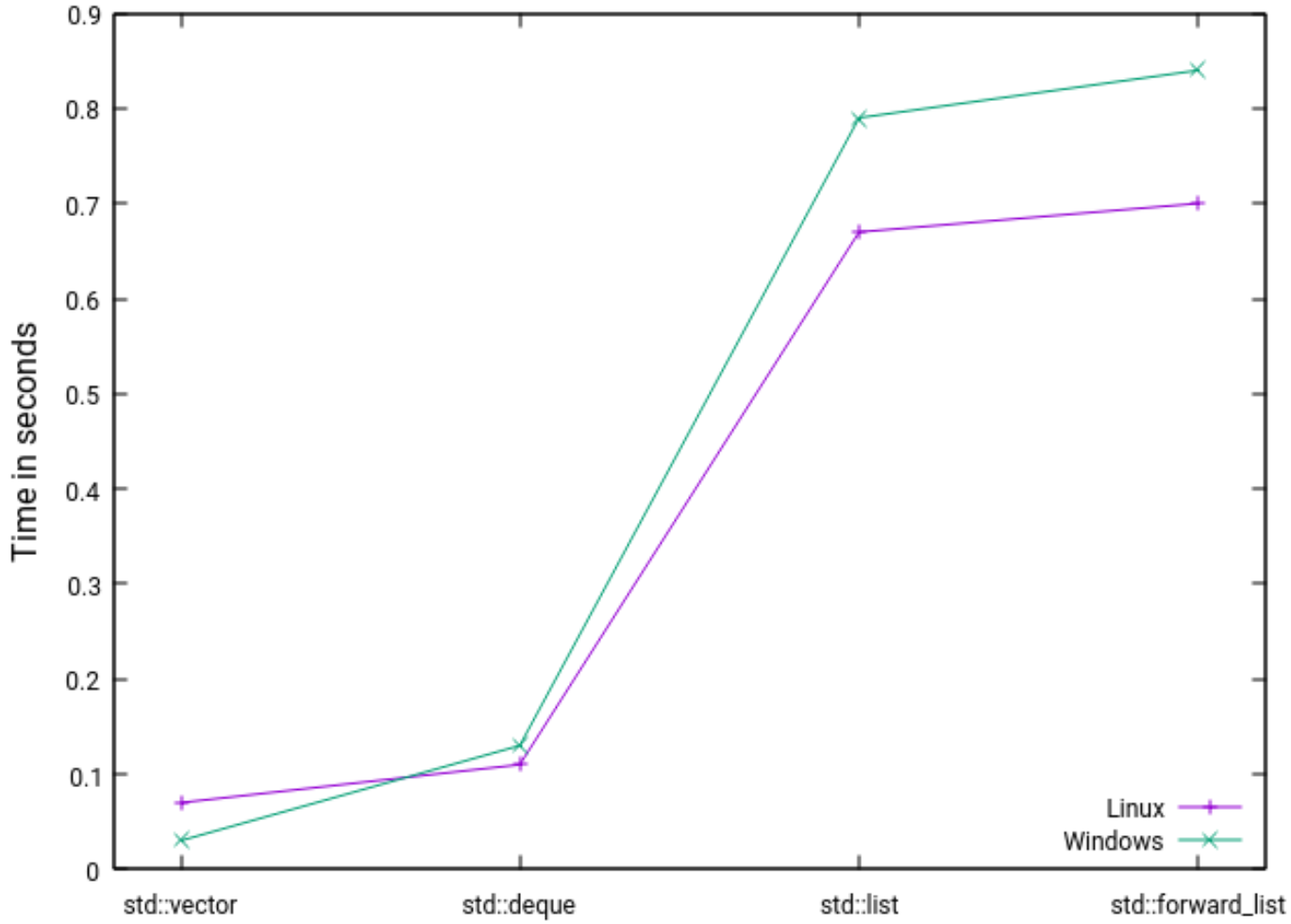
- Sum up all 100‘000‘000 values of various sequence containters using `std::accumulate`
- Used sequence containers: `std::vector`, `std::deque`, `std::list`, and `std::forward_list`

Computer architectures are optimized for the reading of contigious memory blocks.

 memory predictability

Sequence Containers

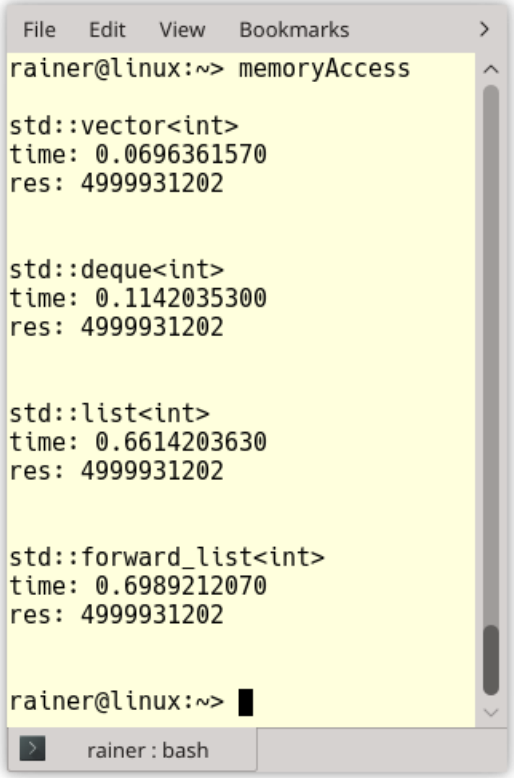
Relative Performance



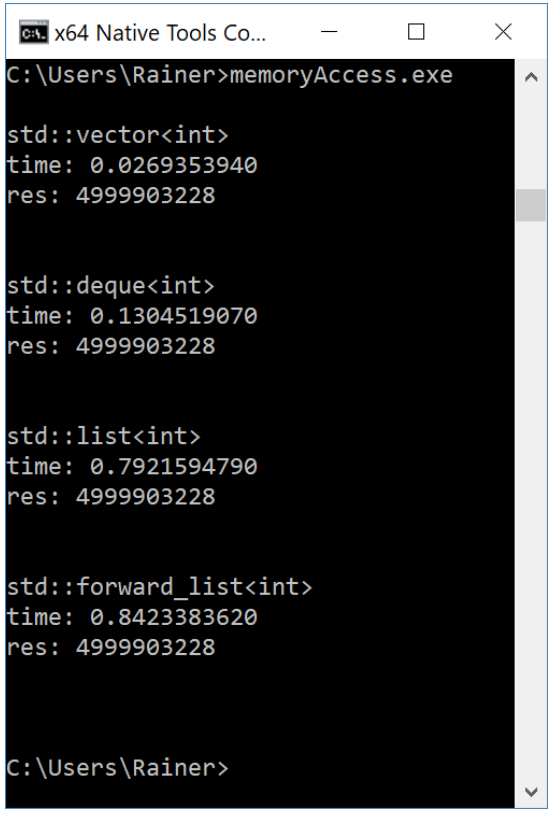
Sequence Containers

Absolute Performance

Linux



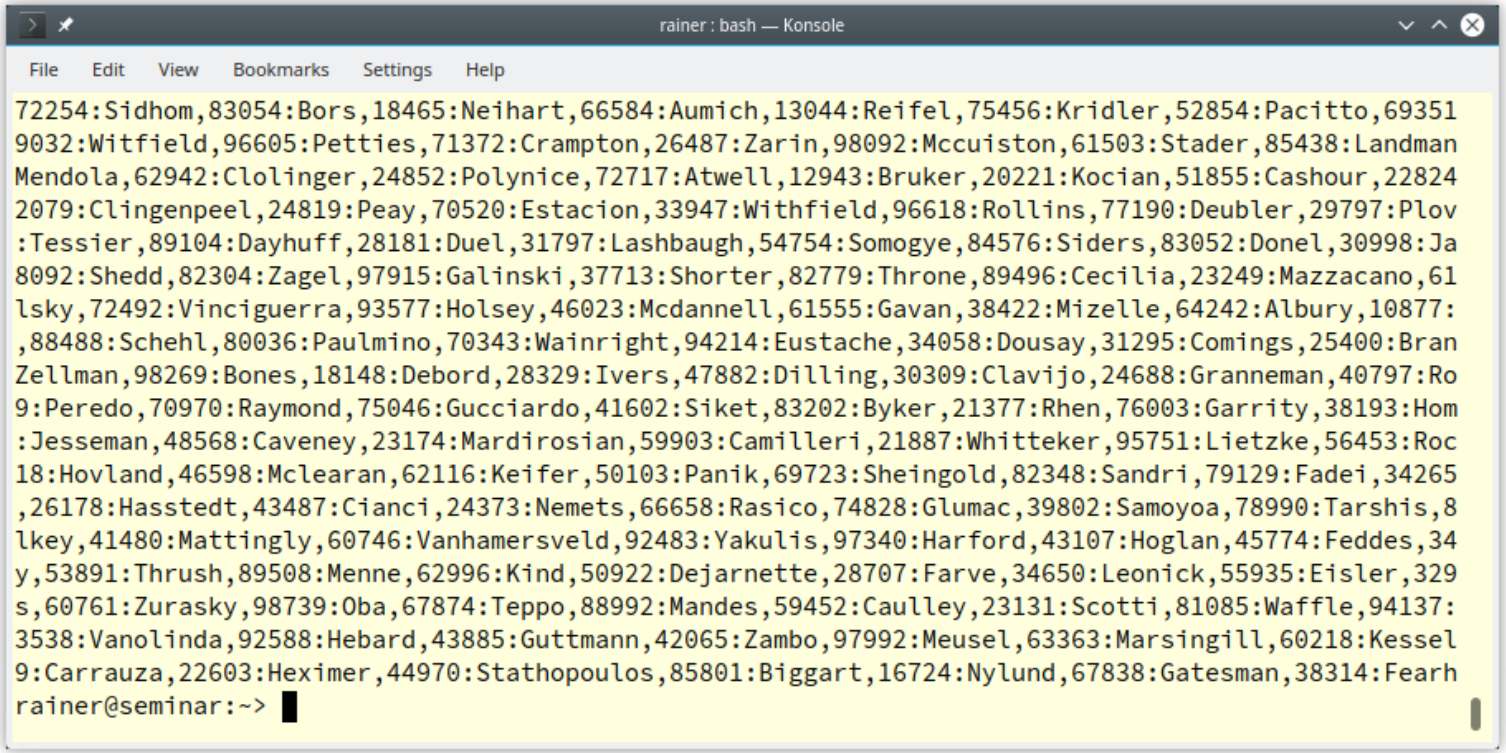
Windows



`std::map` or `std::unordered_map`

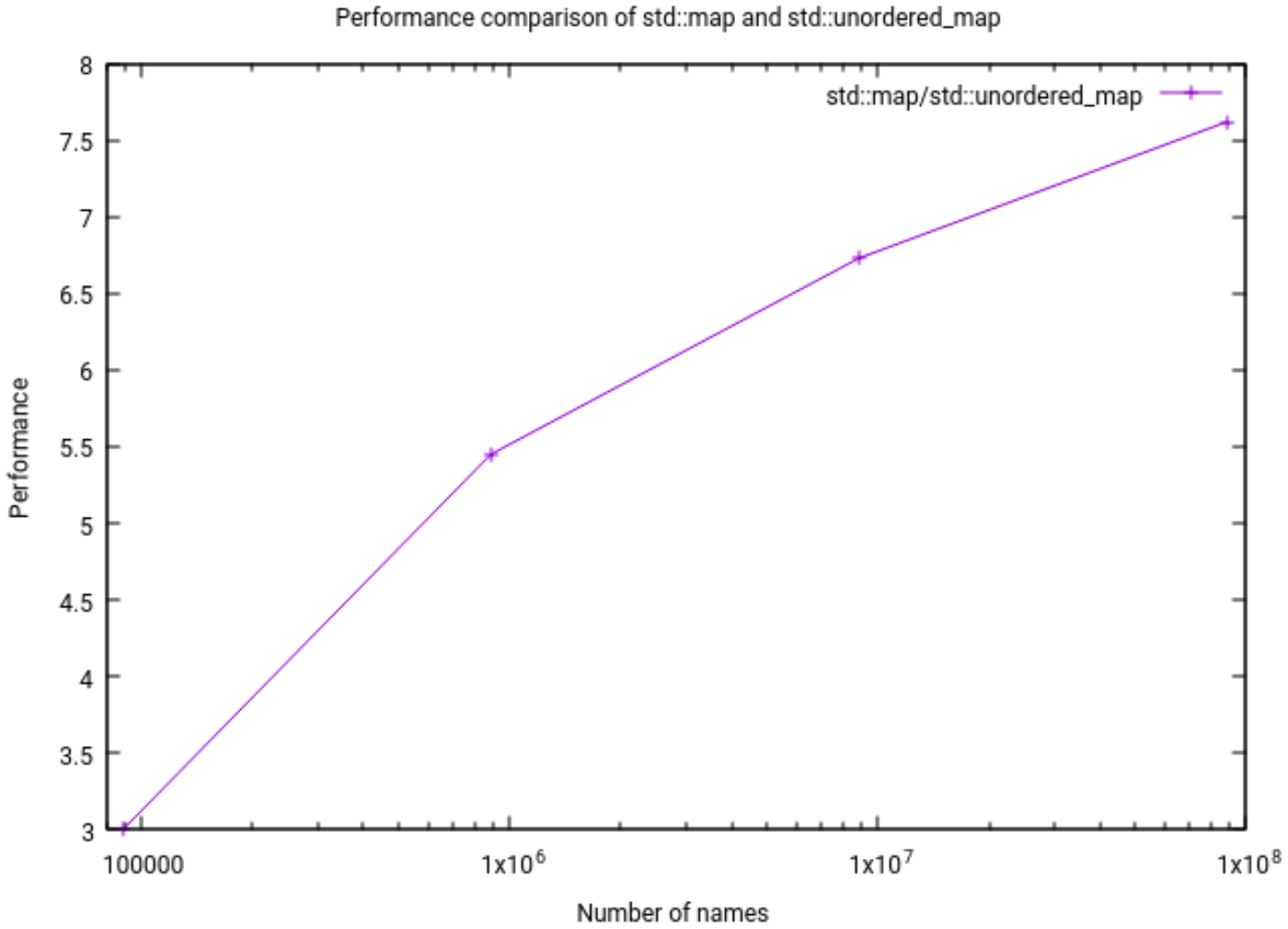
Performance Comparison:

- Reading all keys of a `std::map` and a `std::unordered_map`
- Container size: Roughly 90’000 – 90’000’000 entries



`std::map` or `std::unordered_map`

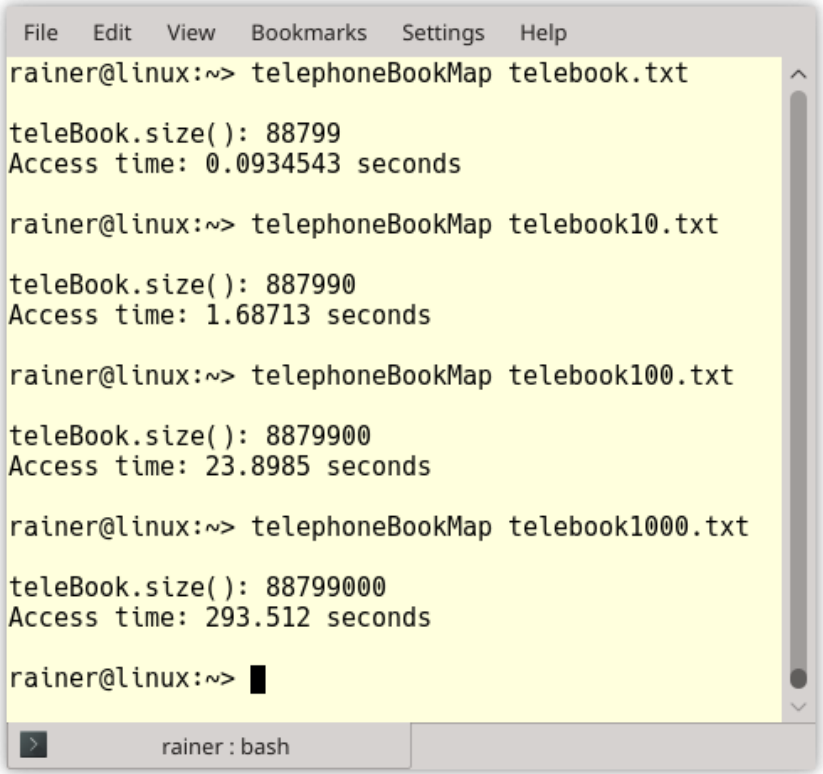
Relative Performance



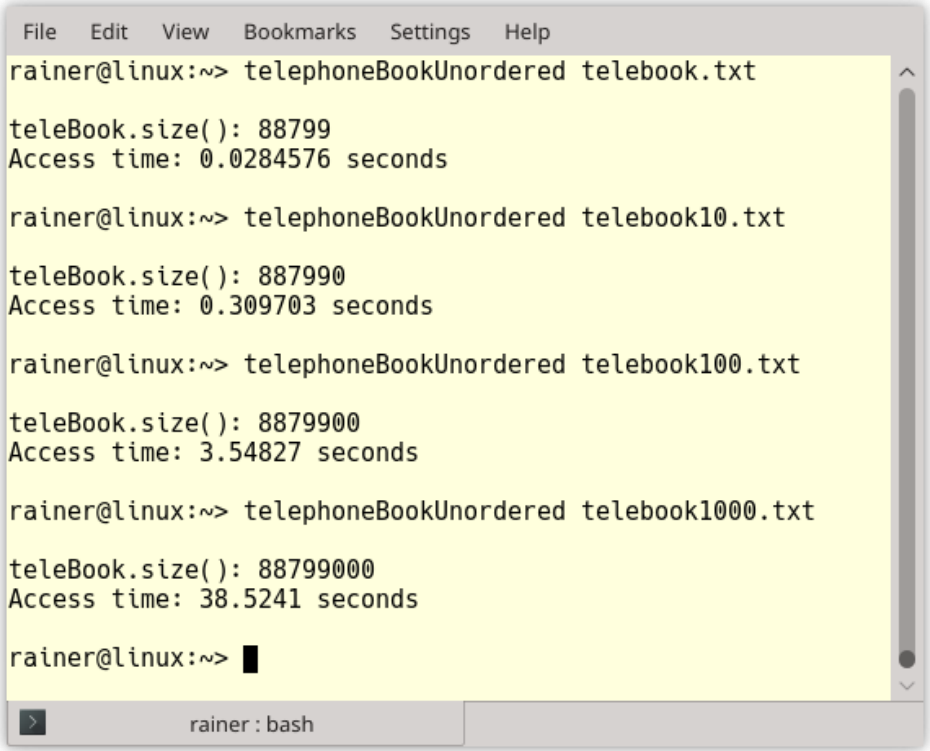
std::map or std::unordered_map

Absolute Performance

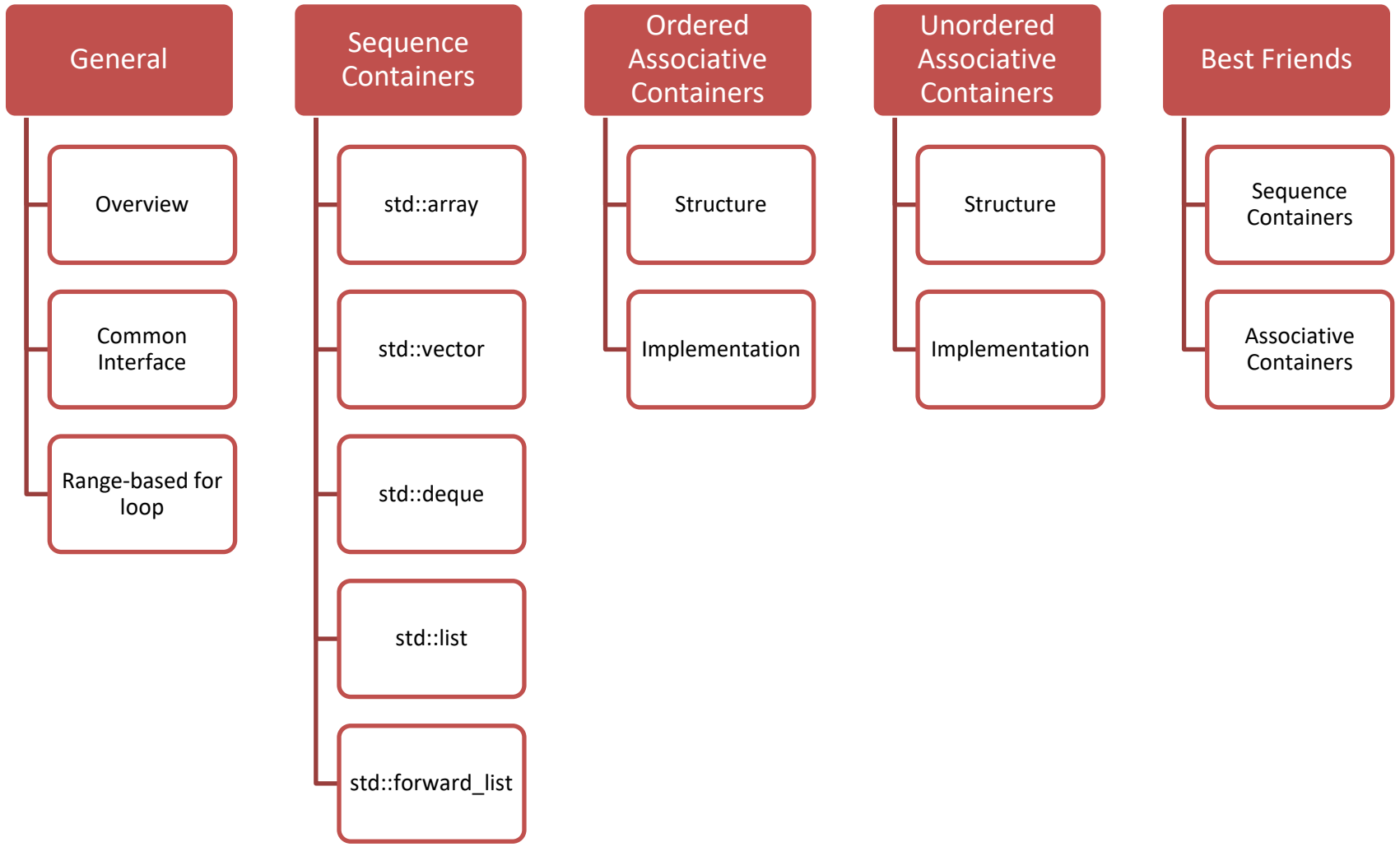
std::map



std::unordered_map



Containers of the STL



Further Information

[The C++ Standard Library]

[Posts on Modernes C++]

[Modernes C++ Training]

[Modernes C++ Mentoring]

Back To Basics
Standard Library Containers

RAINER GRIMM

Cppcon
The C++ Conference

2022

September 12th-16th