

# **Modern C++ for Computer Vision and Image Processing**

## **Lecture 04: C++ STL Library**

**Ignacio Vizzo and Cyrill Stachniss**

# std::array

```
1 #include <array>
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5
6 int main() { ✓ Data type & size
7     std::array<float, 3> data{10.0F, 100.0F, 1000.0F}; ✓
8
9     for (const auto& elem : data) {
10         cout << elem << endl;
11     }
12
13     cout << std::boolalpha;
14     cout << "Array empty: " << data.empty() << endl;
15     cout << "Array size : " << data.size() << endl;
16 }
```

# std::array

ప్రా.యన్.దుంబె



- `#include <array>` to use `std::array`
- Store a **collection of items** of **same type**
- Create from data:  
`array<float, 3> arr = {1.0f, 2.0f, 3.0f};`
- Access items with `arr[i]`  
indexing starts with **0**
- Number of stored items: `arr.size()`
- Useful access aliases:
  - First item: `arr.front() == arr[0]`
  - Last item: `arr.back() == arr[arr.size() - 1]`

# std::vector

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 using std::cout;
5 using std::endl;
6
7 int main() {
8     std::vector<int> numbers = {1, 2, 3};
9     std::vector<std::string> names = {"Nacho", "Cyrill"};
10
11     names.emplace_back("Roberto");
12
13     cout << "First name : " << names.front() << endl;
14     cout << "Last number: " << numbers.back() << endl;
15     return 0;
16 }
```

push-back() → {x, y}

emplace-back() ←

# std::vector

- `#include <vector>` to use `std::vector`
- Vector is implemented as a **dynamic table**
- Access stored items just like in `std::array`
- Remove all elements: `vec.clear()`
- Add a new item in one of two ways:
  - `vec.emplace_back(value)` [preferred, c++11]
  - `vec.push_back(value)` [historically better known]
- **Use it! It is fast and flexible!**  
Consider it to be a default container to store collections of items of any same type

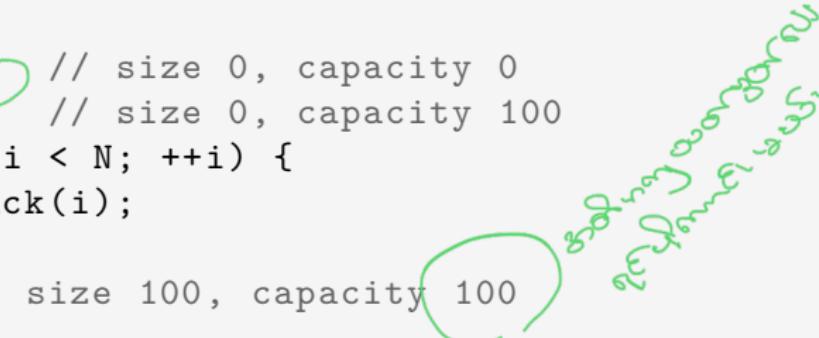
# Optimize vector resizing

ಬ್ರಿಂಗ್  
- ಸಿಡೆ ಯಾರ್ಗ್ಯಾರ್ಡ್  
+  
- ವಿನ್ಯಾಸ ಕೆಂಪಿಂಗ್  
ಅದ್

- `std::vector` size unknown.
- Therefore a `capacity` is defined.
- `size ≠ capacity`
- Many `push_back`/`emplace_back` operations force vector to change its `capacity` many times ✓ ( infinite loop ತಾಹ್ಯವಿಲ್ಲ )
- `reserve(n)` ensures that the vector has enough memory to store `n` items ಗ್ರಹಣಣಗ್ರಹಣಿ
- The parameter `n` can even be approximate
- This is a very **important optimization**

# Optimize vector resizing

```
1 int main() {  
2     const int N = 100;  
3  
4     vector<int> vec; // size 0, capacity 0  
5     vec.reserve(N); // size 0, capacity 100  
6     for (int i = 0; i < N; ++i) {  
7         vec.emplace_back(i);  
8     }  
9     // vec ends with size 100, capacity 100  
10  
11    vector<int> vec2; // size 0, capacity 0  
12    for (int i = 0; i < N; ++i) {  
13        vec2.emplace_back(i);  
14    }  
15    // vec2 ends with size 100, capacity 128  
16 }
```

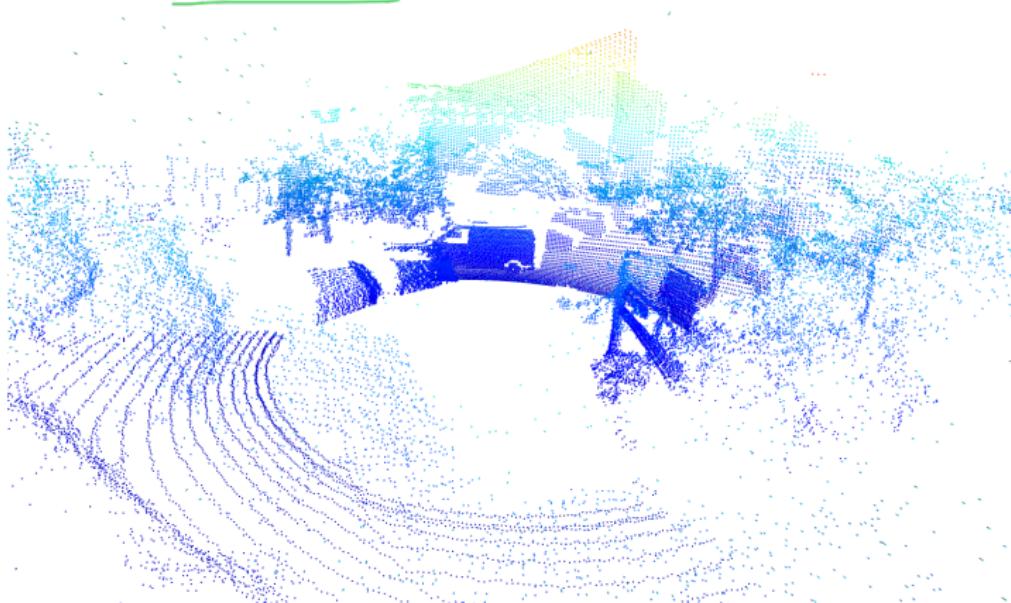


# Containers in CV

## Open3D::PointCloud

```
1 std::vector<Eigen::Vector3d> points_; ocjyž  
2 std::vector<Eigen::Vector3d> normals_;  
3 std::vector<Eigen::Vector3d> colors_;
```

*sgjyž*



# Size of container

Object size എന്നിൽ Byte  
ഇ. കോളഡ് അംഗം type ആണ്  
size ←

## sizeof()

```
1 int data[17];  
2 size_t data_size = sizeof(data) / sizeof(data[0]);  
3 printf("Size of array: %zu\n", data_size); → 17
```

68 / ↗

## size()

```
1 std::array<int, 17> data_{}; ✓  
2 cout << "Size of array: " << data_.size() << endl;
```

# Empty Container

C style

## No standard way of checking if empty

```
1 int empty_arr[10];
2 printf("Array empty: %d\n", empty_arr[0] == NULL);
3
4 int full_arr[5] = {1, 2, 3, 4, 5};
5 printf("Array empty: %d\n", full_arr[0] == NULL);
```

**empty()** ✓

C++ style

```
1 std::vector<int> empty_vec_{};
2 cout << "Array empty: " << empty_vec_.empty() << endl;
3
4 std::vector<int> full_vec_{1, 2, 3, 4, 5};
5 cout << "Array empty: " << full_vec_.empty() << endl;
```

# Access last element

No robust way of doing it

```
1 float f_arr[N] = {1.5, 2.3};  
2 // is it 3, 2 or 900?  
3 printf("Last element: %f\n", f_arr[3]);
```

C style  
out of range!  
Quarantine

## back()

```
1 std::array<float, 2> f_arr_{1.5, 2.3};  
2 cout << "Last Element: " << f_arr_.back() << endl;
```

safe

06

# Clear elements

External function call, doesn't always work with floating points

```
1 char letters[5] = {'n', 'a', 'c', 'h', 'o'};  
2 memset(letters, 0, sizeof(letters));
```

Old style  
✓

clear()

fill with capacity  
(convert to vector)

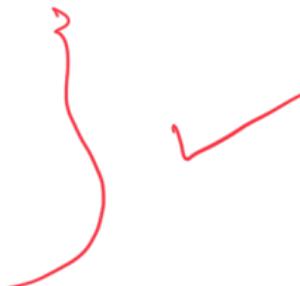
```
1 std::vector<char> letters_ = {'n', 'a', 'c', 'h', 'o'};  
2 letters_.clear();
```

Remember std::string

```
1 std::string letters_right_{"nacho"};  
2 letters_right_.clear();
```

New style

# Why containers?

- Why **Not**?
  - Same speed as C-style arrays but safer.
  - Code readability. ✓
  - More functionality provided than a plain C-style array:
    - `size()`
    - `empty()`
    - `front()`
    - `back()`
    - `swap()`
    - STL algorithms...
    - Much more!
- 
- A hand-drawn red curly brace is drawn from the word "array:" in the fourth bullet point to the word "algorithms..." in the fifth bullet point. A red checkmark is placed next to the word "algorithms...".

# Much more...

## **More information about std::vector**

<https://en.cppreference.com/w/cpp/container/vector>

---

## **More information about std::array**

<https://en.cppreference.com/w/cpp/container/arra>

---

# std::map

- **sorted** associative container.
- Contains **key-value** pairs.
- **keys** are unique.
- **keys** are stored using the `<` operator.
  - Your **keys** should be comparable.
  - built-in types always work, eg: `int`, `float`, etc
  - We will learn how to make your own types “comparable”.
- **value** can be any type, you name it.
- This are called dictionaries `dict` in Python.

# std::map

- Create from data:

```
1 std::map<KeyT, ValueT> m{{key1, value1}, {..}};
```

- Check size: `m.size();` ✓
- Add item to map: `m.emplace(key, value);` ✓
- Modify or add item: `m[key] = value;` ✓
- Get (const) ref to an item: `m.at(key);` ✓
- Check if key present: `m.count(key) > 0;`
  - Starting in C++20:
  - Check if key present: `m.contains(key)` [bool]

```

1 #include <iostream>
2 #include <map>
3 using namespace std;
4
5 int main() {
6     using StudentList = std::map<int, string>;
7     StudentList cpp_students;
8
9     // Inserting data in the students dictionary
10    cpp_students.emplace(1509, "Nacho");      // [1]
11    cpp_students.emplace(1040, "Pepe");        // [0]
12    cpp_students.emplace(8820, "Marcelo");     // [2]
13
14    for (const auto& [id, name] : cpp_students) {
15        cout << "id: " << id << ", " << name << endl;
16    }
17
18    return 0;
19 }

```

emplace → [ (x, "yy") ]  
By Reference

# `std::unordered_map`

- Serves same purpose as `std::map`
- Implemented as a **hash table**
- Key type has to be hashable
- Typically used with `int`, `string` as a key
- Exactly same interface as `std::map`
- Faster to use than `std::map`

```
1 #include <iostream>
2 #include <unordered_map>
3 using namespace std;
4
5 int main() {
6     using StudentList = std::unordered_map<int, string>;
7     StudentList cpp_students;
8
9     // Inserting data in the students dictionary
10    cpp_students.emplace(1509, "Nacho");      // [2]
11    cpp_students.emplace(1040, "Pepe");        // [1]
12    cpp_students.emplace(8820, "Marcelo");     // [0]
13
14    for (const auto& [id, name] : cpp_students) {
15        cout << "id: " << id << ", " << name << endl;
16    }
17
18    return 0;
19 }
```

*onpke*

[ (1509, "Nacho") ]

Template

Lecture 9

```
1 #include <functional>
2 template<> struct hash<bool>;
3 template<> struct hash<char>;
4 template<> struct hash<signed char>;
5 template<> struct hash<unsigned char>;
6 template<> struct hash<char8_t>; // C++20
7 template<> struct hash<char16_t>;
8 template<> struct hash<char32_t>;
9 template<> struct hash<wchar_t>;
10 template<> struct hash<short>;
11 template<> struct hash<unsigned short>;
12 template<> struct hash<int>;
13 template<> struct hash<unsigned int>;
14 template<> struct hash<long>;
15 template<> struct hash<long long>;
16 template<> struct hash<unsigned long>;
17 template<> struct hash<unsigned long long>;
18 template<> struct hash<float>;
19 template<> struct hash<double>;
20 template<> struct hash<long double>;
21 template<> struct hash<std::nullptr_t>; // C++17
```

# Iterating over maps

```
1 for (const auto& kv : m) {  
2     const auto& key = kv.first;  
3     const auto& value = kv.second;  
4     // Do important work.  
5 }
```

## New in C++17

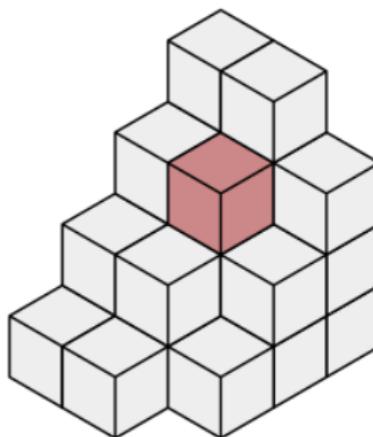
```
1 std::map<char, int> my_dict{{'a', 27}, {'b', 3}};  
2 for (const auto& [key, value] : my_dict) {  
3     cout << key << " has value " << value << endl;
```

- Every stored element is a pair
- map has keys sorted
- unordered\_map has keys in random order

# Associative Containers in CV

## Open3D::VoxelGrid

```
1 std::unordered_map<Eigen::Vector3i ,  
2           Voxel ,  
3           hash_eigen::hash<Eigen::Vector3i>>  
4 voxels_;
```



first  
second  
Voxel

# Much more

## Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

<code>array</code> (C++11)	<u>static</u> contiguous array (class template)
<code>vector</code>	<u>dynamic</u> contiguous array (class template)
<code>deque</code>	double-ended queue (class template)
<code>forward_list</code> (C++11)	singly-linked list (class template)
<code>list</code>	doubly-linked list (class template)

## Associative containers

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

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

SL book

# Much more

## Unordered associative containers

Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched ( $O(1)$  amortized,  $O(n)$  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)

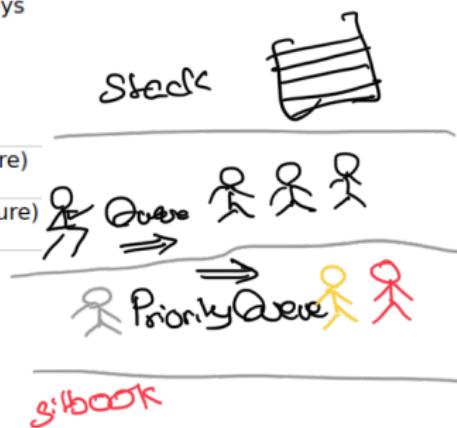
## Container adaptors

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)

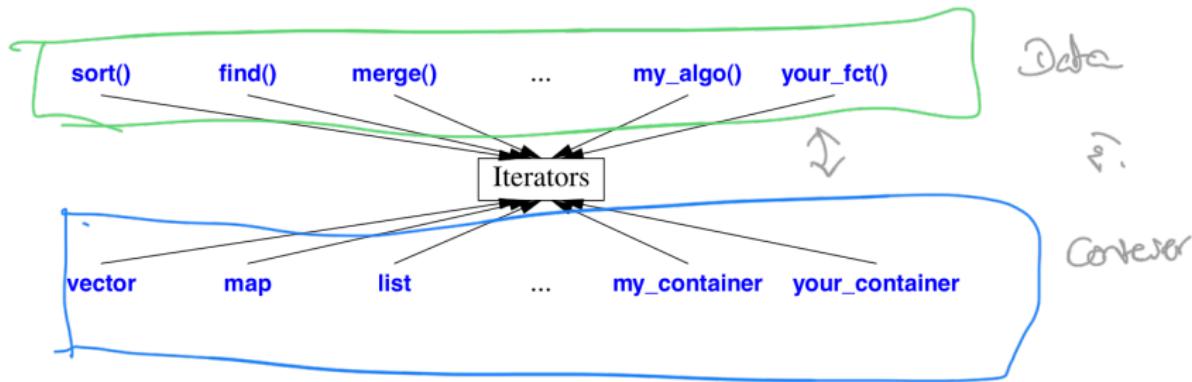
`priority_queue` adapts a container to provide priority queue  
(class template)



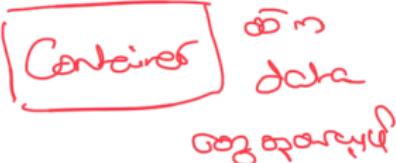
# Iterators

"Iterators are the glue that ties standard-library algorithms to their data.

Iterators are the mechanism used to **minimize an algorithm's dependence** on the data structures on which it operates"



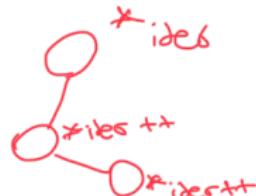
# Iterators

Iterator =  A red hand-drawn style diagram showing a rectangular box labeled "Container" with a curved arrow pointing from it to the word "data". Below the box is the word "organization".

STL uses iterators to access data in containers

- Iterators are similar to pointers
- Allow quick navigation through containers
- Most algorithms in STL use iterators
- Defined for all using STL containers

# Iterators



STL uses iterators to access data in containers

- Access current element with `*iter`
- Accepts `->` alike to pointers
- Move to next element in container `iter++`
- Prefer range-based for loops
- Compare iterators with `==, !=, <` It's Ok

# Range Access Iterators

c = constant

r = reverse  
iterator

- begin, cbegin:  
*constant*  
returns an iterator to the beginning of a container or array
- end, cend:  
returns an iterator to the end of a container or array
- rbegin, crbegin:  
*constant*  
returns a reverse iterator to a container or array
- rend, crend:  
returns a reverse end iterator for a container or array

# Range Access Iterators

**Defined for all STL containers:**

```
1 #include <array>
2 #include <deque>
3 #include <forward_list>
4 #include <iterator>
5 #include <list>
6 #include <map>
7 #include <regex>
8 #include <set>
9 #include <span>
10 #include <string>
11 #include <string_view>
12 #include <unordered_map>
13 #include <unordered_set>
```

အောင်လုပ်ခန္ဓာကိုယ်



```
1 int main() {
2     vector<double> x{1, 2, 3};
3     for (auto it = x.begin(); it != x.end(); ++it) {
4         cout << *it << endl;
5     }
6     // Map iterators
7     map<int, string> m = {{1, "hello"}, {2, "world"}};
8     map<int, string>::iterator m_it = m.find(1);
9     cout << m_it->first << ":" << m_it->second << endl;
10
11    auto m_it2 = m.find(1); // same thing
12    cout << m_it2->first << ":" << m_it2->second << endl;
13
14    if (m.find(3) == m.end()) {
15        cout << "Key 3 was not found\n";
16    }
17    return 0;
18 }
```



# STL Algorithms

- About 80 standard algorithms.
- Defined in `#include <algorithm>`
- They operate on sequences defined by a pair of iterators (for inputs) or a single iterator (for outputs).

80 Algorithms

# Don't reinvent the wheel



- Before writing your own `sort` function :  
<http://en.cppreference.com/w/cpp/algorithms>
- When using `std::vector`, `std::array`, etc.  
try to avoid writing your own algorithms.
- If you are not using STL containers, then proving implementations for the standard iterators will give you access to all the algorithms for free.
- There is a lot of functions in `std` which are at least as fast as hand-written ones.

## std::sort

```
1 int main() {
2     array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
3
4     cout << "Before sorting: ";
5     Print(s);
6
7     std::sort(s.begin(), s.end());
8     cout << "After sorting: ";
9     Print(s);
10
11    return 0;
12 }
```

## Output:

```
1 Before sorting: 5 7 4 2 8 6 1 9 0 3
2 After sorting: 0 1 2 3 4 5 6 7 8 9
```

## std::find

```
1 int main() {
2     const int n1 = 3;
3     std::vector<int> v{0, 1, 2, 3, 4};
4
5     auto result1 = std::find(v.begin(), v.end(), n1);
6
7     if (result1 != std::end(v)) {
8         cout << "v contains: " << n1 << endl;
9     } else {
10        cout << "v does not contain: " << n1 << endl;
11    }
12 }
```

### Output:

```
1 v contains: 3
```

## std::fill

```
1 int main() {  
2     std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};  
3  
4     std::fill(v.begin(), v.end(), -1);  
5  
6     Print(v);  
7 }
```

## Output:

```
1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

## std::count

```
1 int main() {
2     std::vector<int> v{1, 2, 3, 4, 4, 3, 7, 8, 9, 10};
3
4     const int n1 = 3;
5     const int n2 = 5;
6     int num_items1 = std::count(v.begin(), v.end(), n1);
7     int num_items2 = std::count(v.begin(), v.end(), n2);
8     cout << n1 << " count: " << num_items1 << endl;
9     cout << n2 << " count: " << num_items2 << endl;
10
11    return 0;
12 }
```

### Output:

```
1 3 count: 2
2 5 count: 0
```

## std::count\_if

```
1 inline bool div_by_3(int i) { return i % 3 == 0; }
2
3 int main() {
4     std::vector<int> v{1, 2, 3, 3, 4, 3, 7, 8, 9, 10};
5
6     int n3 = std::count_if(v.begin(), v.end(), div_by_3);
7     cout << "# divisible by 3: " << n3 << endl;
8 }
```

## Output:

```
1 # divisible by 3: 4
```

## std::for\_each



```
1 int main() {  
2     std::vector<int> nums{3, 4, 2, 8, 15, 267};  
3  
4     // lambda expression, lecture_9  
5     auto print = [](const int& n) { cout << " " << n; };  
6  
7     cout << "Numbers:";  
8     std::for_each(nums.cbegin(), nums.cend(), print);  
9     cout << endl;  
10  
11    return 0;  
12 }
```

### Output:

```
1 Numbers: 3 4 2 8 15 267
```

## std::all\_of

សម្រាប់អ្ន?

```
1 inline bool even(int i) { return i % 2 == 0; }
2 int main() {
3     std::vector<int> v(10, 2);
4     std::partial_sum(v.cbegin(), v.cend(), v.begin());
5     Print(v);
6
7     bool all_even = all_of(v.cbegin(), v.cend(), even);
8     if (all_even) {
9         cout << "All numbers are even" << endl;
10    }
11 }
```

### Output:

```
1 Among the numbers: 2 4 6 8 10 12 14 16 18 20
2 All numbers are even
```

## std::rotate

```
1 int main() {  
2     std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
3     cout << "before rotate: ";  
4     Print(v);  
5  
6     std::rotate(v.begin(), v.begin() + 2, v.end());  
7     cout << "after rotate: ";  
8     Print(v);  
9 }
```



## Output:

```
1 before rotate: 1 2 3 4 5 6 7 8 9 10  
2 after rotate: 3 4 5 6 7 8 9 10 1 2
```

## std::transform

```
1 auto Uppercase(char c) { return std::toupper(c); }
2 int main() {
3     const std::string s("hello");
4     std::string S{s};           ← Copy Constructor
5     std::transform(s.begin(),
6                     s.end(),
7                     S.begin(),
8                     Uppercase);
9
10    cout << s << endl;
11    cout << S << endl;
12 }
```

class line<sup>is</sup>  
line (const line &obj);

## Output:

```
1 hello
2 HELLO
```

## std::accumulate

ഫുଳി.സം.

വൈദികി. എക്സാമീൻ. മീറ്റി

O(n) വാ

```
1 int main() {  
2     std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
3  
4     int sum = std::accumulate(v.begin(), v.end(), 0);  
5  
6     int product = std::accumulate(v.begin(),  
7                                     v.end(),  
8                                     1, 1 ഫേംഫോർമേറ്റ്  
9                                     std::multiplies());  
10  
11    cout << "Sum : " << sum << endl;  
12    cout << "Product: " << product << endl;  
13 }
```

1 - 2 - 3 - ....

Output:

1 Sum : 55  
2 Product: 3628800

അലൈ.സി ടെക്നോളജിസ്.ഇം

## std::max

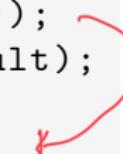
```
1 int main() {  
2     using std::max;  
3     cout << "max(1, 9999) : " << max(1, 9999) << endl;  
4     cout << "max('a', 'b') : " << max('a', 'b') << endl;  
5 }
```

### Output:

```
1 max(1, 9999) : 9999  
2 max('a', 'b') : b
```

## std::min\_element

```
1 int main() {  
2     std::vector<int> v{3, 1, 4, 1, 0, 5, 9};  
3  
4     auto result = std::min_element(v.begin(), v.end());  
5     auto min_location = std::distance(v.begin(), result);  
6     cout << "min at: " << min_location << endl;  
7 }
```



## Output:

```
1 min at: 4
```

## std::minmax\_element

```
1 int main() {
2     using std::minmax_element;
3
4     auto v = {3, 9, 1, 4, 2, 5, 9};
5     auto [min, max] = minmax_element(begin(v), end(v));
6
7     cout << "min = " << *min << endl;
8     cout << "max = " << *max << endl;
9 }
```

### Output:

```
1 min = 1
2 max = 9
```

සියලුම Pointers යොමු කළ නො පෙන්වන!  
Dynamic Memory Allocation සැපයී ඇත්තේ.

# std::clamp

ವರ್ಗ ಕಾರ್ಯ ಯಾತ್ರೆ.

```
1 int main() {  
2     // value should be between [kMin, kMax]  
3     const double kMax = 1.0F;  
4     const double kMin = 0.0F;  
5  
6     cout << std::clamp(0.5, kMin, kMax) << endl;  
7     cout << std::clamp(1.1, kMin, kMax) << endl;  
8     cout << std::clamp(0.1, kMin, kMax) << endl;  
9     cout << std::clamp(-2.1, kMin, kMax) << endl;  
10 }
```

Output:

```
1 0.5  
2 1  
3 0.1  
4 0
```

*kMin*

## స్టాండర్డ్ కంటెనర్లు std::cong

std:: pair

std:: vector

std:: list

std:: deque

std:: queue

std:: stack

std:: priority\_queue

std:: set

std:: multiset

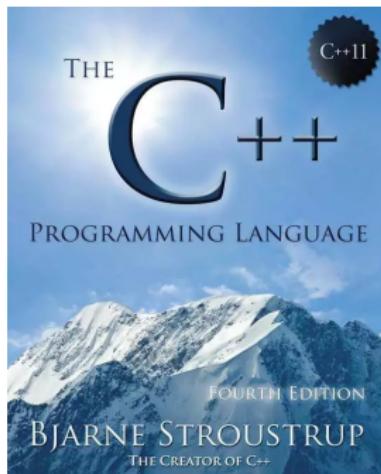
std:: unordered\_set

std:: map

std:: unordered\_map

std:: multimap

# References



## ■ Website:

<http://www.stroustrup.com/4th.html>

# References

- **Containers Library**

<https://en.cppreference.com/w/cpp/container>

---

- **Iterators**

<https://en.cppreference.com/w/cpp/iterators>

---

- **STL Algorithms**

<https://en.cppreference.com/w/cpp/algorith>

---