Modern C++ for Computer Vision Lecture 04: C++ STL Library

Ignacio Vizzo, Rodrigo Marcuzzi, Cyrill Stachniss

Size of container

sizeof()

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

size()

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

Empty Container

No standard way of checking if empty

```
int empty_arr[10];
printf("Array empty: %d\n", empty_arr[0] == NULL);

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

empty()

```
std::vector<int> empty_vec_{};
cout << "Array empty: " << empty_vec_.empty() << endl;

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

Access last element

No robust way of doing it

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

back()

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

Clear elements

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

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

clear()

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

Remember std::string

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

Why containers?

- Why **Not**?
- Code readability.
- More functionalities than arrays:
  - `size()`
  - `empty()`
  - `front()`
  - `back()`
  - `swap()`
  - STL algorithms...
  - Much more!

std::array

```cpp
#include <array>
#include <iostream>
using std::cout;
using std::endl;

int main() {
  std::array<float, 3> data{10.0F, 100.0F, 1000.0F};

  for (const auto& elem : data) {
    cout << elem << endl;
  }

  cout << std::boolalpha;
  cout << "Array empty: " << data.empty() << endl;
  cout << "Array size : " << data.size() << endl;
}
```

std::array

- `#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

```cpp
#include <iostream>
#include <string>
#include <vector>
using std::cout;
using std::endl;

int main() {
  std::vector<int> numbers = {1, 2, 3};
  std::vector<std::string> names = {"Nacho", "Cyrill"};

  names.emplace_back("Roberto");

  cout << "First name : " << names.front() << endl;
  cout << "Last number: " << numbers.back() << endl;
  return 0;
}
```

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

- `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
- `reserve(n)` ensures that the vector has enough memory to store `n` items
- The parameter `n` can even be approximate
- This is a very **important optimization**

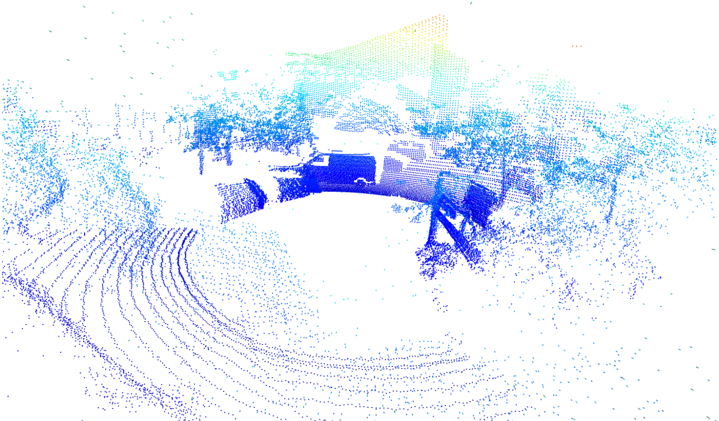
Optimize vector resizing

```cpp
int main() {
  const int N = 100;

  vector<int> vec;  // size 0, capacity 0
  vec.reserve(N);   // size 0, capacity 100
  for (int i = 0; i < N; ++i) {
    vec.emplace_back(i);
  }
  // vec ends with size 100, capacity 100

  vector<int> vec2;  // size 0, capacity 0
  for (int i = 0; i < N; ++i) {
    vec2.emplace_back(i);
  }
  // vec2 ends with size 100, capacity 128
}
```

Containers in CV

Open3D::PointCloud

```
std::vector<Eigen::Vector3d> points_;
std::vector<Eigen::Vector3d> normals_;
std::vector<Eigen::Vector3d> colors_;
```



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.

http://en.cppreference.com/w/cpp/container/map

std::map

- Create from data:

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

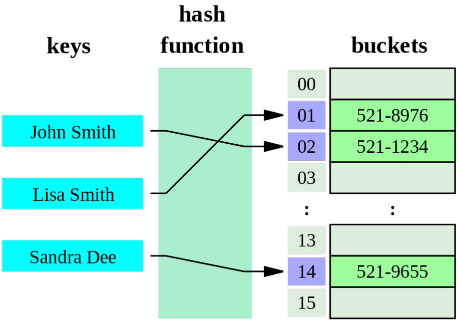
- Check size: `m.size();`
- Add item to map: `m.emplace(key, value);`
- Modify or add item: `m[key] = value;`
- Get (const) ref to an item: `m.at(key);`
- Check if key present: `m.count(key) > 0;`
  - Starting in `C++20`:
  - Check if key present: `m.contains(key)` [bool]

http://en.cppreference.com/w/cpp/container/map

```cpp
#include <iostream>
#include <map>
using namespace std;

int main() {
  std::map<int, string> cpp_students;

  // Inserting data in the students map
  cpp_students.emplace(1509, "Nacho");    // [1]
  cpp_students.emplace(1040, "Pepe");     // [0]
  cpp_students.emplace(8820, "Marcelo");  // [2]

  for (const auto& [id, name] : cpp_students) {
    cout << "id: " << id << ", " << name << endl;
  }

  return 0;
}
```

std::unordered_map

- Serves same purpose as `std::map`
- Implemented as a **hash table**
- Key type has to be hashable



http://en.cppreference.com/w/cpp/container/unordered_map

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`

http://en.cppreference.com/w/cpp/container/unordered_map

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;

int main() {
  using StudentList = std::unordered_map<int, string>;
  StudentList cpp_students;

  // Inserting data in the students map
  cpp_students.emplace(1509, "Nacho");    // [2]
  cpp_students.emplace(1040, "Pepe");     // [1]
  cpp_students.emplace(8820, "Marcelo");  // [0]

  for (const auto& [id, name] : cpp_students) {
    cout << "id: " << id << ", " << name << endl;
  }

  return 0;
}
```

```cpp
#include <functional>
template<> struct hash<bool>;
template<> struct hash<char>;
template<> struct hash<signed char>;
template<> struct hash<unsigned char>;
template<> struct hash<char8_t>;        // C++20
template<> struct hash<char16_t>;
template<> struct hash<char32_t>;
template<> struct hash<wchar_t>;
template<> struct hash<short>;
template<> struct hash<unsigned short>;
template<> struct hash<int>;
template<> struct hash<unsigned int>;
template<> struct hash<long>;
template<> struct hash<long long>;
template<> struct hash<unsigned long>;
template<> struct hash<unsigned long long>;
template<> struct hash<float>;
template<> struct hash<double>;
template<> struct hash<long double>;
template<> struct hash<std::nullptr_t>; // C++17
```

Iterating over maps

```cpp
for (const auto& kv : m) {
  const auto& key = kv.first;
  const auto& value = kv.second;
  // Do important work.
}
```

New in `C++ 17`

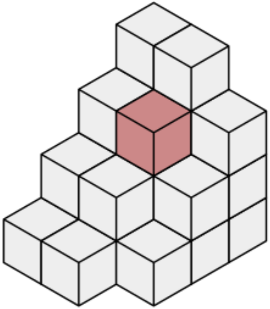
```cpp
std::map<char, int> my_map{{'a', 27}, {'b', 3}};
for (const auto& [key, value] : my_map) {
  cout << key << " has value " << value << endl;
```

- Every stored element is a pair
- `map` has keys **sorted**
- `unordered_map` has keys in **random** order

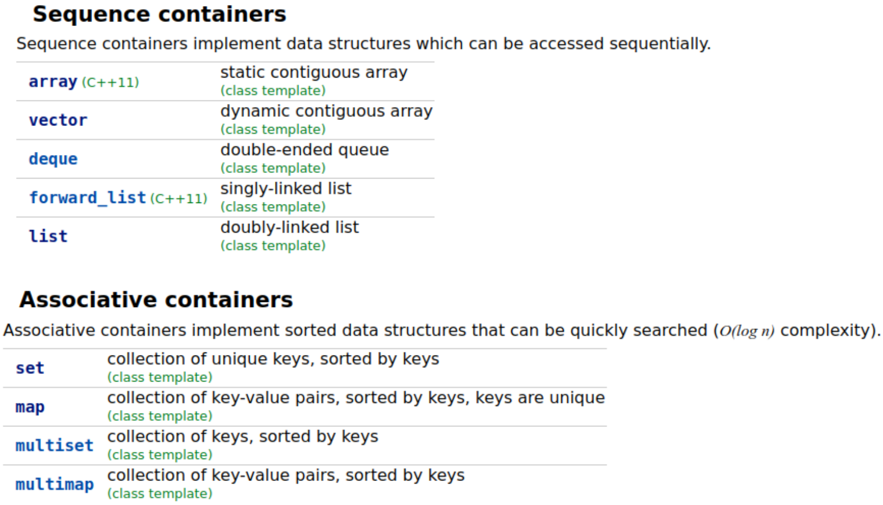
Associative Containers in CV

Open3D::VoxelGrid

```
std::unordered_map<Eigen::Vector3i,
                   Voxel,
                   hash_eigen::hash<Eigen::Vector3i>>
    voxels_;
```



Much more



Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

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

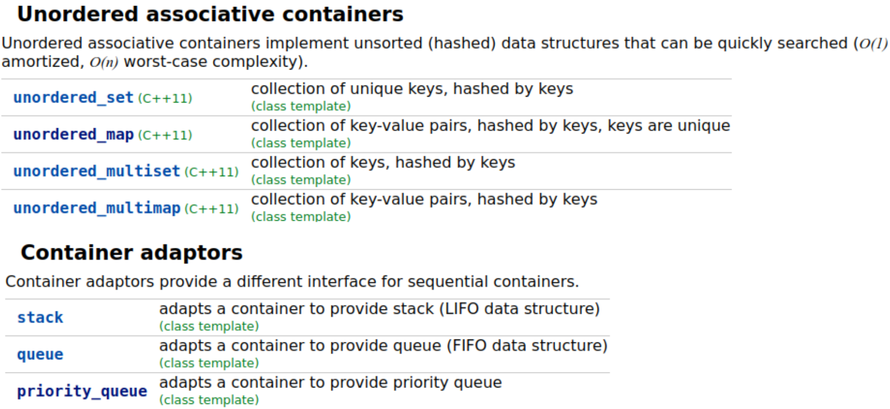
Associative containers

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)

http://en.cppreference.com/w/cpp/container

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)

http://en.cppreference.com/w/cpp/container

Print example

- Print the content of vectors and arrays.
- Need a print() implementation for each type and overload it

```
void print(const std::vector<std::string>& vec){
    for(const auto& v:vec){
        std::cout << v << " ";
    }
    std::cout << std::endl;
}

void print(const std::array<int, 10>& arr){
    for(const auto& a:arr){
        std::cout << a << " ";
    }
    std::cout << std::endl;
}
```

Print example

- Print the content of vectors and arrays.
- Need a print() implementation for each type and overload it

```cpp
int main() {
  std::array<int, 10> arr = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};
  std::vector<std::string> vec  = {"a", "u", "o", "i", "e"};

  std::cout << "arr: ";
  print(arr);
  std::cout << "vec: ";
  print(vec);

  return 0;
}
```

Print example

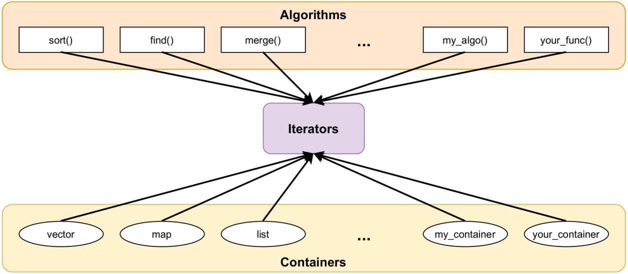
- Print the content of vectors and arrays.
- Need a print() implementation for each type and overload it
- We want to use a single print() function
- **Use iterators as interface between containers and the print() function**
- Acces elements of the container in a generic way

Print example: iterators

```cpp
template<typename Iterator>
void print_it(Iterator begin, Iterator end){
    for (Iterator it = begin; it != end; it++){
        std::cout << *it << " ";
    }
    std::cout << std::endl;
}
```

Iterators

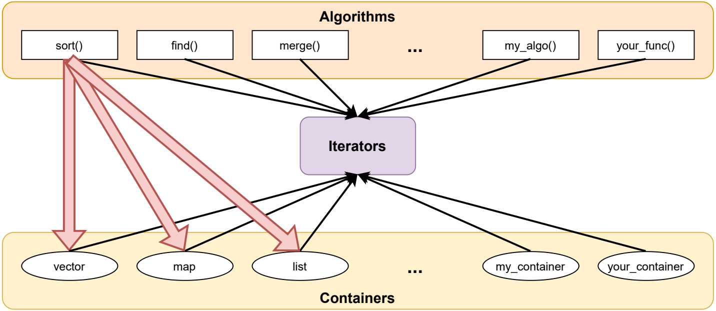
"Iterators are the **glue** that ties standard-library algorithms to their data. Iterators are the mechanism used to **minimize an algorithm's dependence** on the data structures on which it operates"



The C++ Programing Language, 4th edition, Chapter 33

Sort different containers

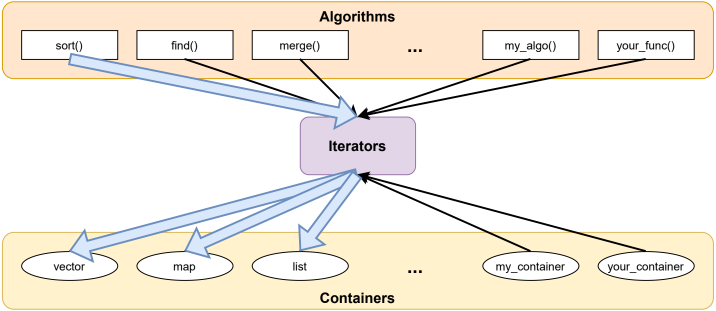
- Create several sort() functions



Algorithms in standard library: `http://en.cppreference.com/w/cpp/algorithm`

Sort different containers

- Create sort() function for iterators



Algorithms in standard library: `http://en.cppreference.com/w/cpp/algorithm`

Iterators

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 `==`, `!=`, `<`

Range Access Iterators

- `begin`, `cbegin` :
  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`:
  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:

```
#include <array>
#include <deque>
#include <forward_list>
#include <iterator>
#include <list>
#include <map>
#include <regex>
#include <set>
#include <span>
#include <string>
#include <string_view>
#include <unordered_map>
#include <unordered_set>
```

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).

Algorithms in standard library: `http://en.cppreference.com/w/cpp/algorithm`

Don’t reinvent the wheel

- Before writting your own `sort` function : `http://en.cppreference.com/w/cpp/algorithm`

- When using `std::vector`, `std::array`, etc. try to avoid writing your own algorithms.

- If you are not using STL containers, provide implementations for the standard iterators. **gives you acess to all the algorithms**

- There is a lot of functions in `std` which are at least as fast as hand-written ones.

std::sort

```cpp
int main() {
  array<int, 10> s = {5, 7, 4, 2, 8, 6, 1, 9, 0, 3};

  cout << "Before sorting: ";
  Print(s);

  std::sort(s.begin(), s.end());
  cout << "After  sorting: ";
  Print(s);

  return 0;
}
```

Output:

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

std::find

```cpp
int main() {
  const int n1 = 3;
  std::vector<int> v{0, 1, 2, 3, 4};

  auto result1 = std::find(v.begin(), v.end(), n1);

  if (result1 != std::end(v)) {
    cout << "v contains: " << n1 << endl;
  } else {
    cout << "v does not contain: " << n1 << endl;
  }
}
```

Output:

```
v contains: 3
```

std::fill

```cpp
int main() {
  std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

  std::fill(v.begin(), v.end(), -1);

  Print(v);
}
```

Output:

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

std::count

```cpp
int main() {
  std::vector<int> v{1, 2, 3, 4, 4, 3, 7, 8, 9, 10};

  const int n1 = 3;
  const int n2 = 5;
  int num_items1 = std::count(v.begin(), v.end(), n1);
  int num_items2 = std::count(v.begin(), v.end(), n2);
  cout << n1 << " count: " << num_items1 << endl;
  cout << n2 << " count: " << num_items2 << endl;

  return 0;
}
```

Output:

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

std::count_if

```cpp
inline bool div_by_3(int i) { return i % 3 == 0; }

int main() {
  std::vector<int> v{1, 2, 3, 3, 4, 3, 7, 8, 9, 10};

  int n3 = std::count_if(v.begin(), v.end(), div_by_3);
  cout << "# divisible by 3: " << n3 << endl;
}
```

Output:

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

std::for_each

```cpp
int main() {
  std::vector<int> nums{3, 4, 2, 8, 15, 267};

  // lambda expression, lecture_9
  auto print = [](const int& n) { cout << " " << n; };

  cout << "Numbers:";
  std::for_each(nums.cbegin(), nums.cend(), print);
  cout << endl;

  return 0;
}
```

Output:

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

std::all_off

```
inline bool even(int i) { return i % 2 == 0; };
int main() {
  std::vector<int> v(10, 2);
  std::partial_sum(v.cbegin(), v.cend(), v.begin());
  Print(v);

  bool all_even = all_of(v.cbegin(), v.cend(), even);
  if (all_even) {
    cout << "All numbers are even" << endl;
  }
}
```

Output:

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

std::rotate

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

  std::rotate(v.begin(), v.begin() + 2, v.end());
  cout << "after  rotate: ";
  Print(v);
}
```

Output:

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

std::transform

```cpp
auto UpperCase(char c) { return std::toupper(c); }
int main() {
  const std::string s("hello");
  std::string S{s};
  std::transform(s.begin(),
                 s.end(),
                 S.begin(),
                 UpperCase);

  cout << s << endl;
  cout << S << endl;
}
```

Output:

```
hello
HELLO
```

std::accumulate

```cpp
int main() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  int sum = std::accumulate(v.begin(), v.end(), 0);

  int product = std::accumulate(v.begin(),
                                v.end(),
                                1,
                                std::multiplies());

  cout << "Sum    : " << sum << endl;
  cout << "Product: " << product << endl;
}
```

Output:

```
Sum    : 55
Product: 3628800
```

std::max

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

Output:

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

std::min_element

```cpp
int main() {
  std::vector<int> v{3, 1, 4, 1, 0, 5, 9};

  auto result = std::min_element(v.begin(), v.end());
  auto min_location = std::distance(v.begin(), result);
  cout << "min at: " << min_location << endl;
}
```

Output:

```
min at: 4
```

std::minmax_element

```cpp
int main() {
  using std::minmax_element;

  auto v = {3, 9, 1, 4, 2, 5, 9};
  auto [min, max] = minmax_element(begin(v), end(v));

  cout << "min = " << *min << endl;
  cout << "max = " << *max << endl;
}
```

Output:

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

std::clamp

```cpp
int main() {
  // value should be between [kMin,kMax]
  const double kMax = 1.0F;
  const double kMin = 0.0F;

  cout << std::clamp(0.5, kMin, kMax) << endl;
  cout << std::clamp(1.1, kMin, kMax) << endl;
  cout << std::clamp(0.1, kMin, kMax) << endl;
  cout << std::clamp(-2.1, kMin, kMax) << endl;
}
```

Output:

```
0.5
1
0.1
0
```

std::sample

```cpp
int main() {
  std::string in = "C++ is cool", out;
  auto rnd_dev = std::mt19937{random_device{}()};
  const int kNLetters = 5;
  std::sample(in.begin(),
              in.end(),
              std::back_inserter(out),
              kNLetters,
              rnd_dev);

  cout << "from   : " << in << endl;
  cout << "sample: " << out << endl;
}
```

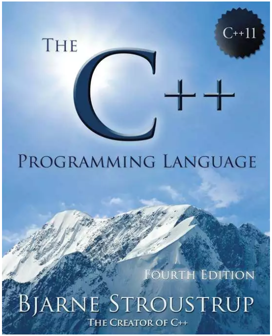
Output:

```
from   : C++ is cool
sample: C++cl
```

Suggested Video

https://youtu.be/bFSnXNIsK4A

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/algorithm