NOKIA

C++ Standard Library Algorithms

Piotr Nycz (Nokia)

09-03-2020

Basic information
https://en.cppreference.com/w/cpp/algorithm

<algorithm>

Non-modifying sequence
(all_of, find_if, ...)
Modifying sequence
(copy, remove, ...)

Partitioning
Sorting
Permutation
Binary Search
lower/upper_bound, ...

Set/Heap operations
Min/max
Comparison
equal
lexographical_compare, ...

<numeric>

iota
accumulate
reduce
transform_reduce
inner_product
adjacent_difference
partial_sum
inclusive_scan
exclusive_scan
transform_inclusive_scan
transform_exclusive_scan

<memory>
uninitialized_copy, ...
destroy, ...

<iterator>
Adaptors
make_move_iterator,
make_reverse_iterator, ...
back_inserter, ...
Stream iterators
istream_iterator,
ostream_iterator, ...
Operations
begin, end, rbegin, ...
size, empty, data, ...
distance, next, prev, ...

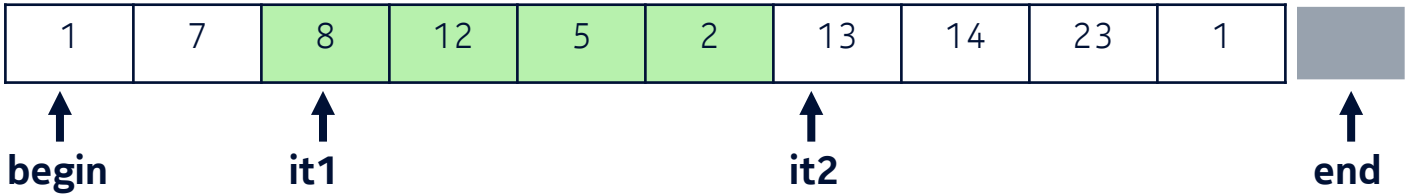
Basic information – in-class algorithms
https://en.cppreference.com/w/cpp/container

Associative containers	**Unordered associative containers**	**Most containers**
O(log N)	O(1)	erase swap
count find equal_range lower_bound upper_bound	count find	erase_if (C++20)

1. Provide better performance than standalone algorithms (like std::find (O(N)))
2. Can modify containers
   1. standalone algorithms can modify only elements
   2. compare behavior of std::remove and std::vector<T>::erase

Ranges in C++17

- Range defined by 2 iterators [it1, it2) is the basic input to all algorithms
  - Range contains all element between it1(inclusively) and it2 (exclusively!)

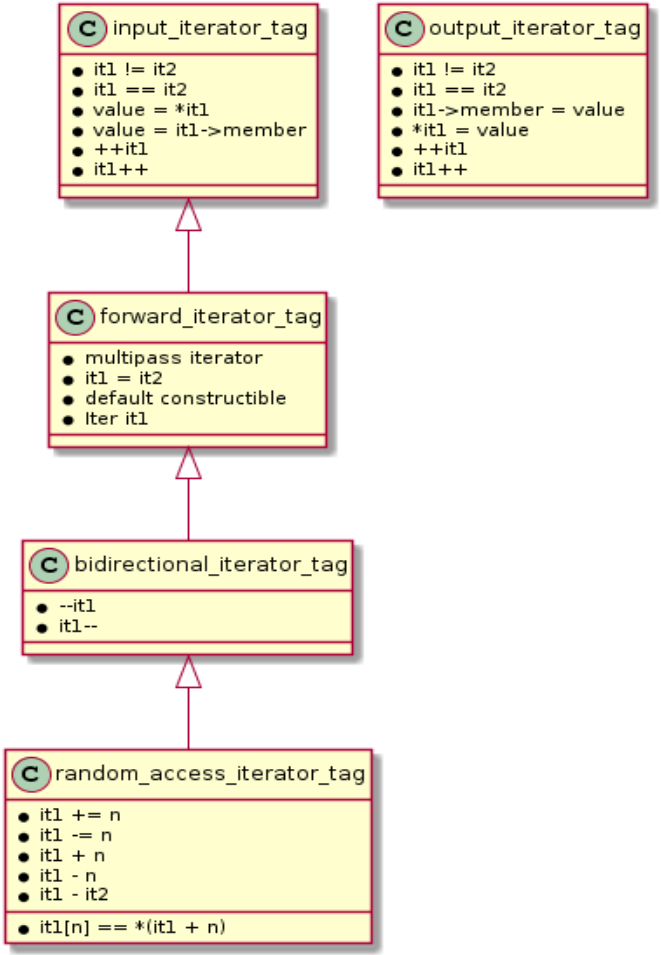


- Iterator can be anything that points to some element
  - "C" Pointer (**T***) is a valid iterator (when points to elements of an array **T[N]**)
  - All Standard Library containers have corresponding iterators (access via begin/end methods)
  - Full range for container (like **std::vector<int> a;**) is **(a.begin(), a.end())**
  - Full range for "C" array (like **char b[] = "ala ma kota";**) is **(b, b + N)**
  - Since C++11 both containers and arrays ranges can be defined by functions begin, end: **(std::begin(x), std::end(x))**
  - Iterator can be also something not related to container nor array – like stream iterators
- An example – printing all elements of a **vector<int> v;** – by copying to **std::cout**:

```
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, ""));
```

About performance

- Algorithms are written with performance in mind
  - But compiler optimization must be ON – due to big number of inline functions
- Algorithms have various versions aligned to type of iterators (see on right →). This is to achieve the best performance for the given iterator/container type. Simple example of that is **std::distance, std::advance**



About performance - multithreading
((from gcc9.1))

Some algorithms can be run in multithreads modes

- **namespace std::execution { sequenced_policy seq; }**
  - Default - operations are performed in sequence (like in default pre-C++17 mode)

- **namespace std::execution { parallel_policy par; }**
  - Operations might be performed in parallel (in different threads)
  - It requires from user to ensure no data races happen

- **namespace std::execution { parallel_unsequenced_policy par_unseq; }**
  - Operations might be performed in parallel, vectorized, migrated from thread to thread
  - It requires vectorization-safe code – e.g. no mutex allowed
    - But still no data races allowed – so it is really hard to use. But it is the most promising.

NOKIA

Copyright and confidentiality