C++ Standard Library Algorithms

Piotr Nycz (Nokia)

15-03-2022

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, …

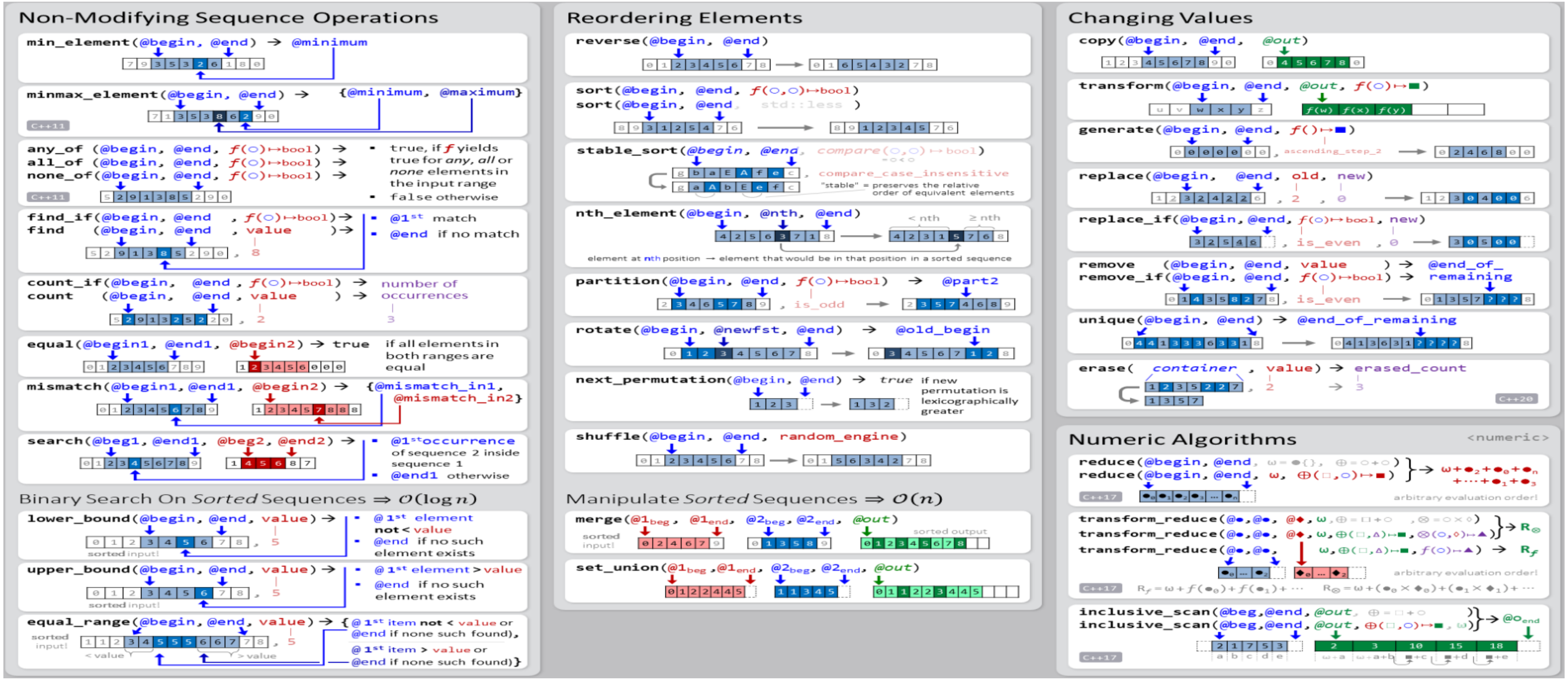
<numeric>

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, …

Other links:
https://hackingcpp.com/cpp/cheat_sheets.html



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

Associative containers

O(log N)

count
find
equal_range
lower_bound
upper_bound

Unordered associative containers

O(1)

count
find

Most containers

erase
swap

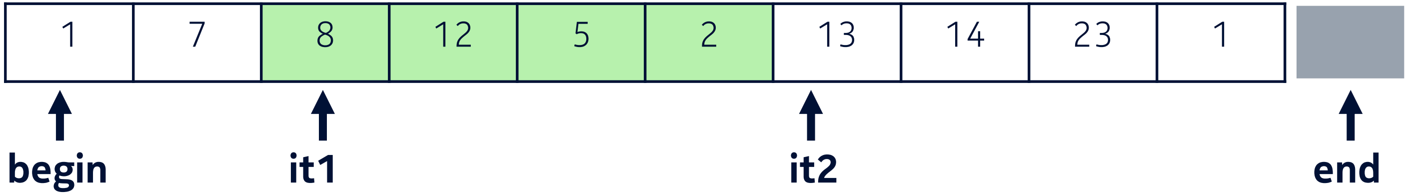
erase_if (C++20)

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

NOKIA

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!)

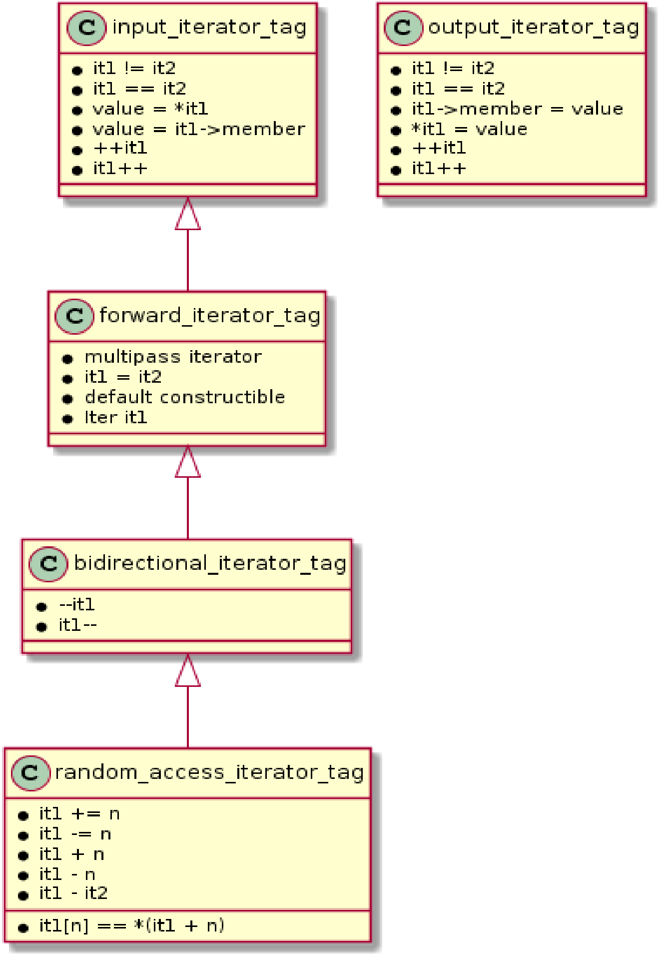


- Iterator can be anything that points to some element
  - “C” Pointer (**T***) is a valid iterator (when points to elements of an array **T[N]**)
  - All Standard Library containers have corresponding iterators (access via begin/end methods)
  - Full range for container (like **std::vector<int> a;**) is **(a.begin(), a.end())**
  - Full range for “C” array (like **char b[] = “ala ma kota”;**) is **(b, b + sizeof(b)/sizeof(b[0]))**
  - 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!
- 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**



C++ idioms: erase/remove

1. std::remove_if works on iterators
2. We need to use container method (like vector::erase) to actually removes the elements
3. C++20 fixes that by adding erase_if to containers

```
std::vector<int> vv{1,2,3,4,5,6};
print(vv);
auto it = std::remove_if(begin(vv), end(vv), [](auto v) { return v & 1; });
print(vv);
print(begin(vv), it);
vv.erase(it, end(vv));
print(vv);
```

```
1 2 3 4 5 6
2 4 6 4 5 6
2 4 6
2 4 6
2 4 6
```

C++20 Introduced std::erase/erase_if

```
std::vector<int> vv{1,2,3,4,5,6};
std::erase_if(vv, [](auto v) { return v & 1; });
print(vv);
```

Exercise: implement erase/remove for std::map (and std::set)

1. Remove erase_if for associative containers – associative_erase_if(container)
2. Of course – for pre C++20 code
3. And note that std::remove_if is not working on such containers (cannot move elements)

Exercise: Implement C++20 equalUnordered

1. Shall work as std::equal – but:
   1. It shall return true also for ranges that are equal after sorting.
   2. It shall not sort input range – i.e. after execution of the algorithm – input shall be unchanged!
2. Examples:

```
const std::vector<int> a{1,12,23,34};
const std::array<int, 4> b{1,12,23,34};
const bool equal = equalUnordered(begin(a), end(a), begin(b));
assert(equal);
```

```
const std::vector<int> a{21,1,34,23};
const std::array<int, 4> b{1,21,23,34};
const bool equal = equalUnordered(begin(a), end(a), begin(b));
assert(equal);
```

```
const std::vector<int> a{1,2,4,3};
const std::array<int, 4> b{1,2,3,3};
const bool equal = equalUnordered(begin(a), end(a), begin(b));
assert(!equal);
```

Exercise: Implement C++20 equalUnordered

1. Algorithm:
   1. Copy iterator ranges (begin1, end1) and (begin2, …) to local containers (vectors)
   2. Sort these iterator ranges with comparing values pointed by iterators
   3. Do std::equal on these local ranges, comparing values pointed by iterators (not iterators)
2. At first implement this signature:

```
template <typename Iter1, typename Iter2>
bool equalUnordered(Iter1 begin1, Iter1 end1, Iter2 begin2);
```

Exercise: Implement C++20 equalUnordered

1. Algorithm:
   1. **Copy iterator ranges (begin1, end1) and (begin2, …) to local containers (vectors)**
   2. Sort these iterator ranges with comparing values pointed by iterators
   3. Do std::equal on these local ranges, comparing values pointed by iterators (not iterators)

```
template <typename Iter1, typename Iter2>
bool equalUnordered(Iter1 begin1, Iter1 end1, Iter2 begin2)
{
  const auto size = std::distance(begin1, end1);
  std::vector<Iter1> it1(size); std::vector<Iter2> it2(size);
  std::generate_n(it1.begin(), size, [begin1]() mutable { return begin1++; });
  std::generate_n(it2.begin(), size, [begin2]() mutable { return begin2++; });
...
```


Exercise: Implement C++20 equalUnordered

1. Algorithm:
   1. Copy iterator ranges (begin1, end1) and (begin2, …) to local containers (vectors)
   2. **Sort these iterator ranges with comparing values pointed by iterators**
   3. Do std::equal on these local ranges, comparing values pointed by iterators (not iterators)

```
template <typename Iter1, typename Iter2>
bool equalUnordered(Iter1 begin1, Iter1 end1, Iter2 begin2)
{
  const auto size = std::distance(begin1, end1);
  std::vector<Iter1> it1(size); std::vector<Iter2> it2(size);
  std::generate_n(it1.begin(), size, [begin1]() mutable { return begin1++; });
  std::generate_n(it2.begin(), size, [begin2]() mutable { return begin2++; });
  std::sort(it1.begin(), it1.end(), [](Iter1 i1, Iter1 i2) { return *i1 < *i2; });
  std::sort(it2.begin(), it2.end(), [](Iter2 i1, Iter2 i2) { return *i1 < *i2; });
...
```


Exercise: Implement C++20 equalUnordered

1. Algorithm:
   1. Copy iterator ranges (begin1, end1) and (begin2, …) to local containers (vectors)
   2. Sort these iterator ranges with comparing values pointed by iterators
   3. **Do std::equal on these local ranges, comparing values pointed by iterators (not iterators)**

```
template <typename Iter1, typename Iter2>
bool equalUnordered(Iter1 begin1, Iter1 end1, Iter2 begin2)
{
  const auto size = std::distance(begin1, end1);
  std::vector<Iter1> it1(size); std::vector<Iter2> it2(size);
  std::generate_n(it1.begin(), size, [begin1]() mutable { return begin1++; });
  std::generate_n(it2.begin(), size, [begin2]() mutable { return begin2++; });
  std::sort(it1.begin(), it1.end(), [](Iter1 i1, Iter1 i2) { return *i1 < *i2; });
  std::sort(it2.begin(), it2.end(), [](Iter2 i1, Iter2 i2) { return *i1 < *i2; });
  return std::equal(it1.begin(), it1.end(), it2.begin(), [](Iter1 i1, Iter2 i2) { return *i1 == *i2; });
}
```


Homework: Implement C++20 equalUnordered with custom comparator

1. What to do when there is no way to sort (no sorting comparator)?
   1. Prepare version when no std::sort is used
   2. Show in comments what is the algortihms computational complexity (big O notation)

```
template <typename Iter1, typename Iter2, typename Comparator>
bool equalUnordered(Iter1 begin1, Iter1 end1, Iter2 begin2, Comparator comp);
```

```
template <typename Iter1, typename Iter2, typename Comparator, typename SortingComparator>
bool equalUnordered(Iter1 begin1, Iter1 end1, Iter2 begin2, Comparator comp, SortingComparator sortComp);
```


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