

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

Non-Modifying Sequence Operations

`min_element(@begin, @end) → @minimum`

7 9 3 5 3 2 6 1 8 0

`minmax_element(@begin, @end) → { @minimum, @maximum }`

7 9 3 5 3 2 6 1 8 0

`C++11`
`any_of (@begin, @end, f(○)→bool) →` true, if `f` yields true for any, all or none elements in the input range
`all_of (@begin, @end, f(○)→bool) →`
`none_of (@begin, @end, f(○)→bool) →` false otherwise

`C++11`
`find_if (@begin, @end, f(○)→bool) →` 1st match
`find (@begin, @end, value) →` @end if no match

5 2 9 1 3 8 5 2 0 0

`count_if (@begin, @end, f(○)→bool) →` number of occurrences
`count (@begin, @end, value) →`

5 2 9 1 3 8 5 2 0 0

`equal (@begin1, @end1, @begin2) → true` if all elements in both ranges are equal

0 1 2 3 4 5 6 7 8 0

`mismatch (@begin1, @end1, @begin2) → { @mismatch_in1, @mismatch_in2 }`

0 1 2 3 4 5 6 7 8 0

`search (@beg1, @end1, @beg2, @end2) →` 1st occurrence of sequence 2 in sorted sequence 1
@end1 otherwise

0 1 2 3 4 5 6 7 8 0

Binary Search On Sorted Sequences ⇒ $O(\log n)$

`lower_bound (@begin, @end, value) →` 1st element not < value
@end if no such element exists

0 1 2 3 4 5 6 7 8 0

`upper_bound (@begin, @end, value) →` 1st element > value
@end if no such element exists

0 1 2 3 4 5 6 7 8 0

`equal_range (@begin, @end, value) → { @1st item not < value or @end if none such found, @1st item > value or @end if none such found }`

1 1 2 3 4 5 5 6 6 7 8 0

Reordering Elements

`reverse (@begin, @end)`

0 1 2 3 4 5 6 7 8 → 8 7 6 5 4 3 2 1 0

`sort (@begin, @end, f(○,○)→bool)`

`sort (@begin, @end, std::less)`

8 0 3 1 2 5 4 7 6 → 8 0 1 2 3 4 5 7 6

`stable_sort (@begin, @end, compare(○,○)→bool)`
"stable" = preserves the relative order of equivalent elements

`nth_element (@begin, @nth, @end)`
element at `nth` position → element that would be in that position in a sorted sequence

`partition (@begin, @end, f(○)→bool) → @part2`
2 4 6 5 7 8 0, is_odd → 2 3 5 7 4 6 8 0

`rotate (@begin, @newfst, @end) → @old_begin`
0 1 2 3 4 5 6 7 8 → 0 3 4 5 6 7 1 2 8

`next_permutation (@begin, @end) → true` if new permutation is lexicographically greater
1 2 3 → 1 3 2

`shuffle (@begin, @end, random_engine)`
0 1 2 3 4 5 6 7 8 → 0 1 5 6 3 4 2 7 8

Manipulate Sorted Sequences ⇒ $O(n)$

`merge (@1beg, @1end, @2beg, @2end, @out)`
sorted input → sorted output

`set_union (@1beg, @1end, @2beg, @2end, @out)`
0 1 2 2 4 4 5 → 1 1 3 4 5

Changing Values

`copy (@begin, @end, @out)`

1 2 3 4 5 6 7 8 0 0 → 0 4 5 6 7 8 0

`transform (@begin, @end, @out, f(○)→■)`

u v w x y z → f(u) f(x) f(y)

`generate (@begin, @end, f(○)→■)`

0 0 0 0 0 0, ascending_step_2 → 0 2 4 6 8 0

`replace (@begin, @end, old, new)`

1 2 3 2 4 2 2 0, 2, 0 → 1 2 3 0 4 0 0 0

`replace_if (@begin, @end, f(○)→bool, new)`

3 2 5 4 6, is_even, 0 → 3 0 5 0 6

`remove (@begin, @end, value) → @end_of`

`remove_if (@begin, @end, f(○)→bool) → remaining`

0 1 4 3 5 8 2 7 8, is_even → 0 1 3 5 7 7 7 8

`unique (@begin, @end) → @end_of_remaining`

0 4 4 3 3 3 3 3 3 3 → 0 4 1 3 6 3 1 7 7 7 8

`erase (container, value) → erased_count`

1 2 3 5 7 7 7, 2 → 3

Numeric Algorithms

`reduce (@begin, @end, w = 0, @ = ○ + ○)`
`reduce (@begin, @end, w, @ (○, ○)→■)` → $w + \oplus_1 + \oplus_2 + \dots + \oplus_n$
arbitrary evaluation order!

`transform_reduce (@begin, @end, @out, w, @ = ○ + ○, @ = ○ × ○)`
`transform_reduce (@begin, @end, @out, w, @ (○, ○)→■, @ (○, ○)→■)` → R_\oplus
`transform_reduce (@begin, @end, @out, w, @ (○, ○)→■, @ (○, ○)→■)` → R_\otimes
arbitrary evaluation order!

`C++17`
 $R_\oplus = w + f(\oplus_1) + f(\oplus_2) + \dots$
 $R_\otimes = w + (\oplus_1 \times \oplus_2) + (\oplus_2 \times \oplus_3) + \dots$

`inclusive_scan (@begin, @end, @out, @ = ○ + ○)`

`inclusive_scan (@begin, @end, @out, @ (○, ○)→■, w)` → @out

2 1 7 5 3 → 2 3 10 15 18
a b c d e → w+a w+a+b w+a+b+c w+a+b+c+d w+a+b+c+d+e

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

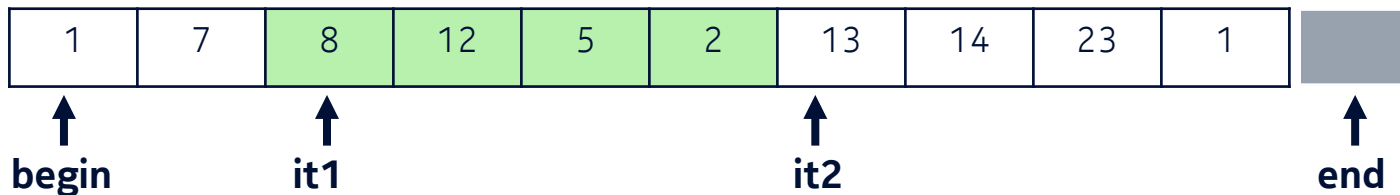
erase
swap

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`

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!)

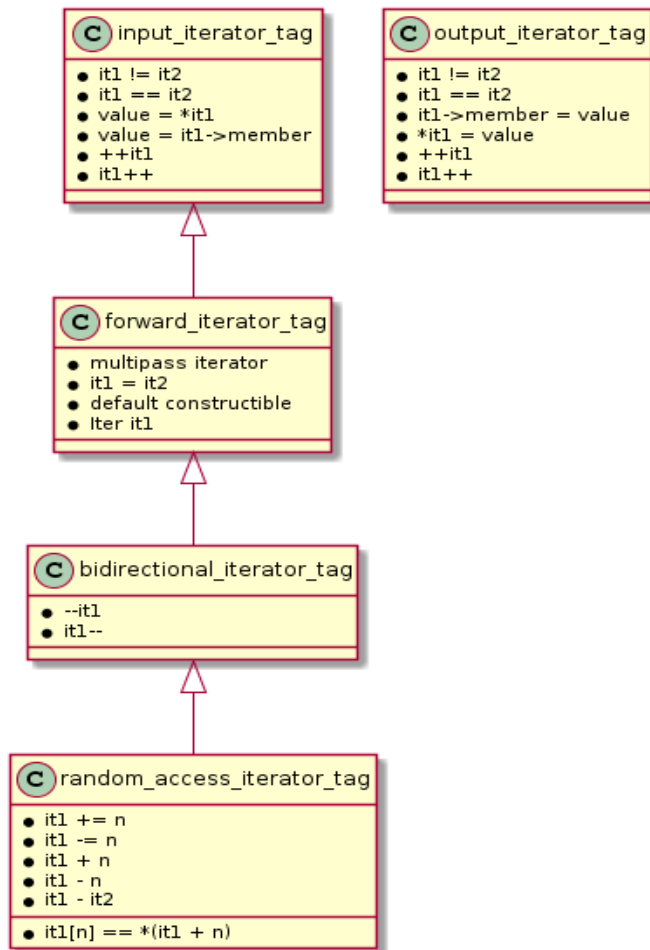


- Iterator can be anything that points to some element
 - “C” Pointer (**T***) is a valid iterator (when points to elements of an array **T[N]**)
 - All Standard Library containers have corresponding iterators (access via begin/end methods)
 - Full range for container (like **std::vector<int> a;**) is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

The contents of this document are proprietary and confidential property of Nokia. This document is provided subject to confidentiality obligations of the applicable agreement(s).

This document is intended for use of Nokia's customers and collaborators only for the purpose for which this document is submitted by Nokia. No part of this document may be reproduced or made available to the public or to any third party in any form or means without the prior written permission of Nokia. This document is to be used by properly trained professional personnel. Any use of the contents in this document is limited strictly to the use(s) specifically created in the applicable agreement(s) under which the document is submitted. The user of this document may voluntarily provide suggestions, comments or other feedback to Nokia in respect of the contents of this document ("Feedback").

Such Feedback may be used in Nokia products and related specifications or other documentation. Accordingly, if the user of this document gives Nokia Feedback on the contents of this document, Nokia may freely use, disclose, reproduce, license, distribute and otherwise commercialize the feedback in any Nokia product, technology, service, specification or other documentation.

Nokia operates a policy of ongoing development. Nokia reserves the right to make changes and improvements to any of the products and/or services described in this document or withdraw this document at any time without prior notice.

The contents of this document are provided "as is". Except as required by applicable law, no warranties of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose,

are made in relation to the accuracy, reliability or contents of this document. NOKIA SHALL NOT BE RESPONSIBLE IN ANY EVENT FOR ERRORS IN THIS DOCUMENT or for any loss of data or income or any special, incidental, consequential, indirect or direct damages howsoever caused, that might arise from the use of this document or any contents of this document.

This document and the product(s) it describes are protected by copyright according to the applicable laws.

Nokia is a registered trademark of Nokia Corporation. Other product and company names mentioned herein may be trademarks or trade names of their respective owners.