<algorithm>

Karl Solomon

December 27, 2024

Contents

1 batchOperations

- for_each
- ranges::for_each
- for_each_n
- ranges::for_each_n

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Use std::for_each to print each element
std::for_each(numbers.begin(), numbers.end(), [](int n) { std::cout << n << " "; });

// Use std::for_each_n to print the first 3 elements
std::for_each_n(numbers.begin(), 3, [](int n) { std::cout << n << " "; });

// Use std::ranges::for_each to print each element
std::ranges::for_each(numbers, [](int n) { std::cout << n << " "; });

// Use std::ranges::for_each_n to print the first 3 elements
std::ranges::for_each_n(numbers.begin(), 3, [](int n) { std::cout << n << " "; });
```

2 Search Operations

- all_of
- any_of
- none_of

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5};
// Use std::all_of to check if all elements are positive
bool allPositive = std::all_of(numbers.begin(), numbers.end(), [](int n) { return n > 0; });
// Use std::any_of to check if any element is greater than 4
bool anyGreaterThanFour = std::any_of(numbers.begin(), numbers.end(), [](int n) { return n > 4; });
// Use std::none_of to check if no elements are negative
bool noneNegative = std::none_of(numbers.begin(), numbers.end(), [](int n) { return n < 0; });
```

- ranges::contains
- ranges::contains_subrange

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// Check if the range contains the value 5
bool containsFive = std::ranges::contains(numbers, 5);
// Check if the range contains the value 11
bool containsEleven = std::ranges::contains(numbers, 11);
// Define a subrange to check
std::vector<int> subrange = {4, 5, 6};
// Check if the range contains the subrange
```

```cpp
bool containsSubrange = std::ranges::contains_subrange(numbers,
    subrange);
// Define another subrange to check
std::vector<int> nonExistentSubrange = {7, 8, 11};
// Check if the range contains the non-existent subrange
bool containsNonExistentSubrange = std::ranges::contains_subrange(
    numbers, nonExistentSubrange);
```

- find
- find_if
- find_if_not
- ranges::find
- ranges::find_if
- ranges::find_if_not

```cpp
#include <algorithm>
#include <iostream>
#include <ranges>
#include <vector>

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

    // Use std::find to find the first occurrence of 5
    auto it = std::find(numbers.begin(), numbers.end(), 5);

    // find the first even number
    it = std::find_if(numbers.begin(), numbers.end(), [](int n) {
        return n % 2 == 0; });
    // find the first odd number
    it = std::find_if_not(numbers.begin(), numbers.end(), [](int n
        ) { return n % 2 == 0; });
    // find the first occurrence of 5
    auto range_it = std::ranges::find(numbers, 5);
    // find the first even number
    range_it = std::ranges::find_if(numbers, [](int n) { return n
        % 2 == 0; });
    // find the first odd number
    range_it = std::ranges::find_if_not(numbers, [](int n) {
        return n % 2 == 0; });
}
```

- find_last
- find_last_if
- find_last_if_not
- find_end
- ranges::find_end

```cpp
#include <algorithm>
#include <iostream>
#include <ranges>
#include <vector>

```

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 5, 6, 7};

// find the last occurrence of 5
auto lastFive = std::ranges::find_last(numbers, 5);
// find the last even number
auto lastEven = std::ranges::find_last_if(numbers, [](int n) { return n % 2 == 0; });
// find the last odd number
auto lastOdd = std::ranges::find_last_if_not(numbers, [](int n) { return n % 2 == 0; });

// Define a subrange to find
std::vector<int> subrange = {5, 6, 7};

// find the last occurrence of the subrange
auto lastSubrange = std::find_end(numbers.begin(), numbers.end(), subrange.begin(), subrange.end());

// find the last occurrence of the subrange
auto lastSubrangeRange = std::ranges::find_end(numbers, subrange);
```

- find_end
- ranges::find_end
- find_first_of
- ranges::find_first_of
- adjacent_find
- ranges::adjacent_find

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5, 3, 4, 5, 6, 7};
std::vector<int> pattern = {3, 4, 5};

// Using std::find_end to find the last occurrence of a pattern
auto it_end = std::find_end(numbers.begin(), numbers.end(), pattern.begin(), pattern.end());
// Result: it_end points to the first element of the last occurrence of {3, 4, 5}

// Using std::find_first_of to find the first occurrence of any element from another range
std::vector<int> search_elements = {4, 5, 6};
auto it_first_of = std::find_first_of(numbers.begin(), numbers.end(), search_elements.begin(), search_elements.end());
// Result: it_first_of points to the first occurrence of any element from {4, 5, 6}, which is 4

// Using std::adjacent_find to find the first occurrence of two consecutive equal elements
std::vector<int> numbers_with_adjacent = {1, 2, 3, 3, 4, 5};
auto it_adjacent = std::adjacent_find(numbers_with_adjacent.begin(), numbers_with_adjacent.end());
// Result: it_adjacent points to the first element of the first pair of adjacent equal elements, which is 3
```

- count
- count_if
- ranges::count

- ranges::count_if

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// Using std::count to count occurrences of the number 5
int count_5 = std::count(numbers.begin(), numbers.end(), 5);
// Result: count_5 = 1

// Using std::count_if to count numbers greater than 5
int count_greater_than_5 = std::count_if(numbers.begin(), numbers.end(), [](int n) { return n > 5; });
// Result: count_greater_than_5 = 5

// Using std::ranges::count to count occurrences of the number 5
int ranges_count_5 = std::ranges::count(numbers, 5);
// Result: ranges_count_5 = 1

// Using std::ranges::count_if to count numbers greater than 5
int ranges_count_greater_than_5 = std::ranges::count_if(numbers, [](int n) { return n > 5; });
// Result: ranges_count_greater_than_5 = 5
```

- mismatch
- ranges::mismatch

```cpp
std::vector<int> vec1 = {1, 2, 3, 4, 5};
std::vector<int> vec2 = {1, 2, 0, 4, 5};

// Using std::mismatch to find the first position where vec1 and vec2 differ
auto mismatch_pair = std::mismatch(vec1.begin(), vec1.end(), vec2.begin());
// Result: mismatch_pair.first points to 3 in vec1, mismatch_pair.second points to 0 in vec2

// Using std::ranges::mismatch to find the first position where vec1 and vec2 differ
auto ranges_mismatch_pair = std::ranges::mismatch(vec1, vec2);
// Result: ranges_mismatch_pair.in1 points to 3 in vec1, ranges_mismatch_pair.in2 points to 0 in vec2
```

- equal
- ranges::equal

```cpp
std::vector<int> vec1 = {1, 2, 3, 4, 5};
std::vector<int> vec2 = {1, 2, 3, 4, 5};
std::vector<int> vec3 = {1, 2, 3, 0, 5};

// Using std::equal to check if vec1 and vec2 are equal
bool are_equal_1_2 = std::equal(vec1.begin(), vec1.end(), vec2.begin());
// Result: are_equal_1_2 = true

// Using std::equal to check if vec1 and vec3 are equal
bool are_equal_1_3 = std::equal(vec1.begin(), vec1.end(), vec3.begin());
// Result: are_equal_1_3 = false

// Using std::ranges::equal to check if vec1 and vec2 are equal
bool ranges_are_equal_1_2 = std::ranges::equal(vec1, vec2);
// Result: ranges_are_equal_1_2 = true

// Using std::ranges::equal to check if vec1 and vec3 are equal
bool ranges_are_equal_1_3 = std::ranges::equal(vec1, vec3);
// Result: ranges_are_equal_1_3 = false
```

- search
- search_n
- ranges::search
- ranges::search_n

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5, 3, 4, 5, 6, 7};
std::vector<int> pattern = {3, 4, 5};

// Using std::search to find the first occurrence of a subsequence
auto it_search = std::search(numbers.begin(), numbers.end(), pattern.begin(), pattern.end());
// Result: it_search points to the first element of the first occurrence of {3, 4, 5}

// Using std::search_n to find the first occurrence of three consecutive 4s
auto it_search_n = std::search_n(numbers.begin(), numbers.end(), 3, 4);
// Result: it_search_n points to numbers.end() as there are no three consecutive 4s

// Using std::ranges::search to find the first occurrence of a subsequence
auto ranges_it_search = std::ranges::search(numbers, pattern);
// Result: ranges_it_search.begin() points to the first element of the first occurrence of {3, 4, 5}

// Using std::ranges::search_n to find the first occurrence of two consecutive 5s
auto ranges_it_search_n = std::ranges::search_n(numbers, 2, 5);
// Result: ranges_it_search_n.begin() points to numbers.end() as there are no two consecutive 5s
```

- ranges::starts_with
- ranges::ends_with

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<int> prefix = {1, 2};
std::vector<int> suffix = {4, 5};
std::vector<int> non_prefix = {2, 3};
std::vector<int> non_suffix = {3, 4};

// Using std::ranges::starts_with to check if numbers starts with prefix
bool starts_with_prefix = std::ranges::starts_with(numbers, prefix);
// Result: starts_with_prefix = true

// Using std::ranges::starts_with to check if numbers starts with non_prefix
bool starts_with_non_prefix = std::ranges::starts_with(numbers, non_prefix);
// Result: starts_with_non_prefix = false

// Using std::ranges::ends_with to check if numbers ends with suffix
```

```cpp
bool ends_with_suffix = std::ranges::ends_with(numbers, suffix);
// Result: ends_with_suffix = true

// Using std::ranges::ends_with to check if numbers ends with non_suffix
bool ends_with_non_suffix = std::ranges::ends_with(numbers, non_suffix);
// Result: ends_with_non_suffix = false
```

3 Fold Operations

- ranges::fold_left
- ranges::_fold_left_first
- ranges::fold_left_with_iter
- ranges::fold_left_first_with_iter
- ranges::fold_right
- ranges::fold_right_last

4 Copy Operations

- copy: Copies all elements from numbers to result
- copy_if: Copies only even numbers from numbers to result
- ranges::copy
- ranges::copy_if
- copy_n: Copies the first 5 elements from numbers to result
- ranges::copy_n: Copies elements from numbers to result in reverse order.
- copy_backwards
- ranges::copy_backwards

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

// Using std::copy to copy all elements
std::copy(numbers.begin(), numbers.end(), result.begin());
// Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// Using std::copy_if to copy only even numbers
auto it = std::copy_if(numbers.begin(), numbers.end(), result.begin(), [](int n) { return n % 2 == 0; });
// Result: result = {2, 4, 6, 8, 10, ?, ?, ?, ?, ?} (remaining elements are unspecified)

// Using std::ranges::copy to copy all elements
std::ranges::copy(numbers, result.begin());
// Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// Using std::ranges::copy_if to copy only even numbers
it = std::ranges::copy_if(numbers, result.begin(), [](int n) { return n % 2 == 0; });
// Result: result = {2, 4, 6, 8, 10, ?, ?, ?, ?, ?} (remaining elements are unspecified)

// Using std::copy_n to copy the first 5 elements
std::copy_n(numbers.begin(), 5, result.begin());
// Result: result = {1, 2, 3, 4, 5, ?, ?, ?, ?, ?} (remaining elements are unspecified)

// Using std::ranges::copy_n to copy the first 5 elements
std::ranges::copy_n(numbers.begin(), 5, result.begin());
// Result: result = {1, 2, 3, 4, 5, ?, ?, ?, ?, ?} (remaining elements are unspecified)

// Using std::copy_backward to copy elements in reverse order
std::copy_backward(numbers.begin(), numbers.end(), result.end());
// Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// Using std::ranges::copy_backward to copy elements in reverse order
std::ranges::copy_backward(numbers, result.end());
// Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

- move
- ranges::move
- move_backward
- ranges::move_backward

```
std::vector<int> source = {1, 2, 3, 4, 5};
std::vector<int> destination(5);

// Using std::move to transfer elements from source to destination
std::move(source.begin(), source.end(), destination.begin());
// Result: destination = {1, 2, 3, 4, 5}
// Result: source = {?, ?, ?, ?, ?} (unspecified, but valid state)

// Reset source for the next example
source = {6, 7, 8, 9, 10};

// Using std::move_backward to transfer elements from source to destination in reverse order
std::move_backward(source.begin(), source.end(), destination.end());
// Result: destination = {6, 7, 8, 9, 10}
// Result: source = {?, ?, ?, ?, ?} (unspecified, but valid state)
```

5 Swap Operations

- swap
- swap_ranges
- ranges::swap_ranges
- iter_swap

```
// Demonstrate std::swap
int a = 10, b = 20;
// Before swap: a = 10, b = 20
std::swap(a, b);
// After swap: a = 20, b = 10

// Demonstrate std::swap_ranges
std::vector<int> vec1 = {1, 2, 3, 4, 5};
```

```
std::vector<int> vec2 = {6, 7, 8, 9, 10};
// Before swap_ranges: vec1 = 1 2 3 4 5, vec2 = 6 7 8 9 10

std::swap_ranges(vec1.begin(), vec1.end(), vec2.begin());
// After swap_ranges: vec1 = 6 7 8 9 10, vec2 = 1 2 3 4 5

// Demonstrate std::ranges::swap_ranges
std::vector<int> vec3 = {11, 12, 13, 14, 15};
std::vector<int> vec4 = {16, 17, 18, 19, 20};
// Before ranges::swap_ranges: vec3 = 11 12 13 14 15, vec4 = 16 17 18 19 20

std::ranges::swap_ranges(vec3, vec4);
// After ranges::swap_ranges: vec3 = 16 17 18 19 20, vec4 = 11 12 13 14 15

// Demonstrate std::iter_swap
std::vector<int> vec5 = {21, 22, 23, 24, 25};
// Before iter_swap: vec5 = 21 22 23 24 25

std::iter_swap(vec5.begin(), vec5.begin() + 4);
// After iter_swap: vec5 = 25 22 23 24 21
```

6 Transform Operations

- `transform`
- `ranges::transform`

```
#include <algorithm>
#include <execution>
#include <iostream>
#include <vector>

int main() {
    // Original vector
    std::vector<int> l1 = {1, 2, 3, 4, 5};
    std::vector<int> l2 = std::vector<int>(l1.size(), 0);
    std::vector<int> l3 = std::vector<int>(l1.size(), 0);

    // simple transform (1 input, 1 output)
    std::transform(l1.begin(), l1.end(), l2.begin(), [](int a) { return a * 10; });

    // transform (2 inputs, 1 output)
    int multiplier = 2;
    std::transform(std::execution::par_unseq, l1.begin(), l1.end(), l2.begin(), l3.begin(),
                   [multiplier](int a, int b) { return (multiplier * a) + b; });

    return 0;
}
```

- `replace`
- `replace_if`
- `ranges::replace`
- `ranges::replace_if`

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// Using std::replace to replace all occurrences of 5 with 50
std::replace(numbers.begin(), numbers.end(), 5, 50);
// Result: numbers = {1, 2, 3, 4, 50, 6, 7, 8, 9, 10}

// Using std::replace_if to replace all even numbers with 0
std::replace_if(numbers.begin(), numbers.end(), [](int n) { return n % 2 == 0; }, 0);
// Result: numbers = {1, 0, 3, 0, 0, 0, 7, 0, 9, 0}

// Reset the numbers vector for ranges example
numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

// Using std::ranges::replace to replace all occurrences of 5 with 50
std::ranges::replace(numbers, 5, 50);
// Result: numbers = {1, 2, 3, 4, 50, 6, 7, 8, 9, 10}

// Using std::ranges::replace_if to replace all even numbers with 0
std::ranges::replace_if(numbers, [](int n) { return n % 2 == 0; }, 0);
// Result: numbers = {1, 0, 3, 0, 0, 0, 7, 0, 9, 0}
```

- replace_copy
- ranges::replace_copy
- replace_copy_if
- ranges::replace_copy_if

```cpp
std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::vector<int> result(numbers.size());

// Using std::replace_copy to copy elements and replace 5 with 50
std::replace_copy(numbers.begin(), numbers.end(), result.begin(), 5, 50);
// Result: result = {1, 2, 3, 4, 50, 6, 7, 8, 9, 10}

// Using std::replace_copy_if to copy elements and replace even numbers with 0
std::replace_copy_if(numbers.begin(), numbers.end(), result.begin(), [](int n) { return n % 2 == 0; }, 0);
// Result: result = {1, 0, 3, 0, 5, 0, 7, 0, 9, 0}

// Using std::ranges::replace_copy to copy elements and replace 5 with 50
std::ranges::replace_copy(numbers, result.begin(), 5, 50);
// Result: result = {1, 2, 3, 4, 50, 6, 7, 8, 9, 10}

// Using std::ranges::replace_copy_if to copy elements and replace even numbers with 0
std::ranges::replace_copy_if(numbers, result.begin(), [](int n) { return n % 2 == 0; }, 0);
// Result: result = {1, 0, 3, 0, 5, 0, 7, 0, 9, 0}
```

7 Generation Operations

- fill

- fill_n
- ranges::fill
- ranges::fill_n

```cpp
std::vector<int> numbers(10);

// Using std::fill to fill the entire vector with 5
std::fill(numbers.begin(), numbers.end(), 5);
// Result: numbers = {5, 5, 5, 5, 5, 5, 5, 5, 5, 5}

// Using std::fill_n to fill the first 5 elements with 3
std::fill_n(numbers.begin(), 5, 3);
// Result: numbers = {3, 3, 3, 3, 3, 5, 5, 5, 5, 5}

// Using std::ranges::fill to fill the entire vector with 7
std::ranges::fill(numbers, 7);
// Result: numbers = {7, 7, 7, 7, 7, 7, 7, 7, 7, 7}

// Using std::ranges::fill_n to fill the first 5 elements with 9
std::ranges::fill_n(numbers.begin(), 5, 9);
// Result: numbers = {9, 9, 9, 9, 9, 7, 7, 7, 7, 7}
```

- generate
- generate_n
- ranges::generate
- ranges::generate_n

```cpp
std::vector<int> numbers(10);
int value = 0;

// Using std::generate to fill the vector with incrementing values starting from 1
std::generate(numbers.begin(), numbers.end(), [&value]() { return ++value; });
// Result: numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// Reset value for the next example
value = 0;

// Using std::generate_n to fill the first 5 elements with incrementing values starting from 1
std::generate_n(numbers.begin(), 5, [&value]() { return ++value; });
// Result: numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// Reset value for the next example
value = 0;

// Using std::ranges::generate to fill the vector with incrementing values starting from 1
std::ranges::generate(numbers, [&value]() { return ++value; });
// Result: numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

// Reset value for the next example
value = 0;

// Using std::ranges::generate_n to fill the first 5 elements with incrementing values starting from 1
std::ranges::generate_n(numbers.begin(), 5, [&value]() { return ++value; });
```

```
// Result: numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

8 Removing Operations

- remove
- remove_if
- ranges::remove
- ranges::remove_if
- remove_copy
- remove_copy_if
- ranges::remove_copy
- ranges::remove_copy_if
- unique
- unique_copy
- ranges::unique
- ranges::unique_copy

9 Order-Changing Operations

- reverse
- ranges::reverse
- reverse_copy
- ranges::reverse_copy
- rotate
- rotate_copy
- ranges::rotate
- ranges::rotate_copy
- shift_left
- shift_right
- ranges::shift_left
- ranges::shift_right
- shuffle
- random_shuffle
- ranges::shuffle

10 Sampling Operations

- sample
- ranges::sample

11 Random Number Generation

- ranges::generate_random

12 Partitioning Operations

- is_partitioned
- ranges::is_partitioned
- partition
- ranges::partition
- partition_copy
- ranges::partition_copy
- stable_partition
- ranges::stable_partition
- partition_point
- ranges::partition_point

13 Sorting Operations

- sort
- ranges::sort
- stable_sort
- ranges::stable_sort
- partial_sort
- ranges::partial_sort
- partial_sort_copy
- ranges::partial_sort_copy
- is_sorted
- ranges::is_sorted
- is_sorted_until
- ranges::is_sorted_until
- nth_element
- ranges::nth_element

14 Binary Search Operations (on partitioned ranges)

- lower_bound
- ranges::lower_bound
- upper_bound
- ranges::upper_bound
- equal_range
- ranges::equal_range
- binary_search
- ranges::binary_search

15 Set Operation (on sorted ranges)

- includes
- ranges::includes
- set_union
- ranges::set_union
- set_intersection
- ranges::set_intersection
- set_difference
- ranges::set_difference
- set_symmetric_difference
- ranges::set_symmetric_difference

16 Merge Operations (on sorted ranges)

- merge
- ranges::merge
- inplace_merge
- ranges::inplace_merge

```
std::vector<int> vec1 = {1, 3, 5, 7};
std::vector<int> vec2 = {2, 4, 6, 8};
std::vector<int> merged(vec1.size() + vec2.size());

// Using std::merge to merge two sorted ranges into a new range
std::merge(vec1.begin(), vec1.end(), vec2.begin(), vec2.end(),
    merged.begin());
// Result: merged = {1, 2, 3, 4, 5, 6, 7, 8}

// Using std::ranges::merge to merge two sorted ranges into a new
    range
std::vector<int> mergedRanges(vec1.size() + vec2.size());
std::ranges::merge(vec1, vec2, mergedRanges.begin());
// Result: mergedRanges = {1, 2, 3, 4, 5, 6, 7, 8}

```

```cpp
// Using std::inplace_merge to merge two consecutive sorted ranges within a single range
std::vector<int> inplaceVec = {1, 3, 5, 7, 2, 4, 6, 8};
std::inplace_merge(inplaceVec.begin(), inplaceVec.begin() + 4, inplaceVec.end());
// Result: inplaceVec = {1, 2, 3, 4, 5, 6, 7, 8}

// Using std::ranges::inplace_merge to merge two consecutive sorted ranges within a single range
std::vector<int> inplaceVecRanges = {1, 3, 5, 7, 2, 4, 6, 8};
std::ranges::inplace_merge(inplaceVecRanges, inplaceVecRanges.begin() + 4);
// Result: inplaceVecRanges = {1, 2, 3, 4, 5, 6, 7, 8}
```

17 Heap Operations

- push_heap
- ranges::push_heap
- pop_heap
- ranges::pop_heap

```cpp
std::vector<int> heap = {3, 1, 4, 1, 5, 9, 2, 6};

// Convert the vector into a heap
std::make_heap(heap.begin(), heap.end());
// Result: heap = {9, 6, 4, 1, 5, 3, 2, 1}

// Using std::push_heap to add a new element and maintain heap property
heap.push_back(7);
std::push_heap(heap.begin(), heap.end());
// Result: heap = {9, 7, 4, 6, 5, 3, 2, 1, 1}

// Using std::pop_heap to remove the largest element and maintain heap property
std::pop_heap(heap.begin(), heap.end());
heap.pop_back();
// Result: heap = {7, 6, 4, 1, 5, 3, 2, 1}

// Using std::ranges::push_heap to add a new element and maintain heap property
heap.push_back(8);
std::ranges::push_heap(heap);
// Result: heap = {8, 7, 4, 6, 5, 3, 2, 1, 1}

// Using std::ranges::pop_heap to remove the largest element and maintain heap property
std::ranges::pop_heap(heap);
heap.pop_back();
// Result: heap = {7, 6, 4, 1, 5, 3, 2, 1}
```

- make_heap
- ranges::make_heap
- sort_heap
- ranges::sort_heap

```cpp
std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6};

// Using std::make_heap to create a max-heap from the numbers vector
std::make_heap(numbers.begin(), numbers.end());
// Result: numbers = {9, 6, 4, 1, 5, 1, 2, 3}

// Using std::sort_heap to sort the heap
std::sort_heap(numbers.begin(), numbers.end());
// Result: numbers = {1, 1, 2, 3, 4, 5, 6, 9}

// Reset the numbers vector for ranges example
numbers = {3, 1, 4, 1, 5, 9, 2, 6};

// Using std::ranges::make_heap to create a max-heap from the numbers vector
std::ranges::make_heap(numbers);
// Result: numbers = {9, 6, 4, 1, 5, 1, 2, 3}

// Using std::ranges::sort_heap to sort the heap
std::ranges::sort_heap(numbers);
// Result: numbers = {1, 1, 2, 3, 4, 5, 6, 9}
```

- is_heap
- ranges::is_heap
- is_heap_until
- ranges::is_heap_until

```cpp
std::vector<int> numbers = {9, 6, 4, 1, 5, 1, 2, 3};

// Using std::is_heap to check if the numbers vector is a heap
bool isHeap = std::is_heap(numbers.begin(), numbers.end());
// Result: isHeap = true

// Using std::is_heap_until to find the first position where the heap property is violated
auto heapEnd = std::is_heap_until(numbers.begin(), numbers.end());
// Result: heapEnd points to numbers.end(), indicating the entire range is a heap

// Using std::ranges::is_heap to check if the numbers vector is a heap
bool isHeapRanges = std::ranges::is_heap(numbers);
// Result: isHeapRanges = true

// Using std::ranges::is_heap_until to find the first position where the heap property is violated
auto heapEndRanges = std::ranges::is_heap_until(numbers);
// Result: heapEndRanges points to numbers.end(), indicating the entire range is a heap

// Modify the vector to violate the heap property
numbers = {9, 6, 4, 10, 5, 1, 2, 3};

// Re-check using std::is_heap
isHeap = std::is_heap(numbers.begin(), numbers.end());
// Result: isHeap = false

// Re-check using std::is_heap_until
heapEnd = std::is_heap_until(numbers.begin(), numbers.end());
```

```
// Result: heapEnd points to numbers.begin() + 3, where the value 10 violates the heap property

// Re-check using std::ranges::is_heap
isHeapRanges = std::ranges::is_heap(numbers);
// Result: isHeapRanges = false

// Re-check using std::ranges::is_heap_until
heapEndRanges = std::ranges::is_heap_until(numbers);
// Result: heapEndRanges points to numbers.begin() + 3, where the value 10 violates the heap property
```

18 Min/Max Operations

- max
- min
- ranges::max
- ranges::min

```
int a = 10;
int b = 20;

// Using std::max to find the maximum of two values
int maxVal = std::max(a, b);
// Result: maxVal = 20

// Using std::min to find the minimum of two values
int minVal = std::min(a, b);
// Result: minVal = 10

std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

// Using std::ranges::max to find the maximum value in a range
int maxInRange = std::ranges::max(numbers);
// Result: maxInRange = 9

// Using std::ranges::min to find the minimum value in a range
int minInRange = std::ranges::min(numbers);
// Result: minInRange = 1
```

- max_element
- min_element
- ranges::max_element
- ranges::min_element

```
std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

// Using std::max_element to find the maximum element in a range
auto maxElement = std::max_element(numbers.begin(), numbers.end());
// Result: *maxElement = 9

// Using std::min_element to find the minimum element in a range
auto minElement = std::min_element(numbers.begin(), numbers.end());
// Result: *minElement = 1

```

```cpp
// Using std::ranges::max_element to find the maximum element in a range
auto maxElementRanges = std::ranges::max_element(numbers);
// Result: *maxElementRanges = 9

// Using std::ranges::min_element to find the minimum element in a range
auto minElementRanges = std::ranges::min_element(numbers);
// Result: *minElementRanges = 1
```

- minmax
- ranges::minmax
- minmax_element
- ranges::minmax_element

```cpp
int a = 10;
int b = 20;

// Using std::minmax to find the minimum and maximum of two values
auto minmaxPair = std::minmax(a, b);
// Result: minmaxPair.first = 10, minmaxPair.second = 20

std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

// Using std::minmax_element to find the minimum and maximum elements in a range
auto minmaxElements = std::minmax_element(numbers.begin(), numbers.end());
// Result: *minmaxElements.first = 1, *minmaxElements.second = 9

// Using std::ranges::minmax to find the minimum and maximum in a range
auto minmaxRange = std::ranges::minmax(numbers);
// Result: minmaxRange.min = 1, minmaxRange.max = 9

// Using std::ranges::minmax_element to find the minimum and maximum elements in a range
auto minmaxRangeElements = std::ranges::minmax_element(numbers);
// Result: *minmaxRangeElements.min = 1, *minmaxRangeElements.max = 9
//
// TODO (ksolomon): make sure usage is correct. why does minmax return a pair, whereas ranges::minmax return a tuple?
//
```

- clamp
- ranges::clamp

```cpp
int value = 15;
int lowerBound = 10;
int upperBound = 20;

// Using std::clamp to constrain the value within the range [lowerBound, upperBound]
int clampedValue = std::clamp(value, lowerBound, upperBound);
// Result: clampedValue = 15

// Using std::clamp to constrain a value below the lower bound
int belowLower = 5;
int clampedBelow = std::clamp(belowLower, lowerBound, upperBound);
```

```cpp
// Result: clampedBelow = 10

// Using std::clamp to constrain a value above the upper bound
int aboveUpper = 25;
int clampedAbove = std::clamp(aboveUpper, lowerBound, upperBound);
// Result: clampedAbove = 20

// Using std::ranges::clamp to constrain the value within the range [lowerBound, upperBound]
int clampedValueRanges = std::ranges::clamp(value, lowerBound, upperBound);
// Result: clampedValueRanges = 15

// Using std::ranges::clamp to constrain a value below the lower bound
int clampedBelowRanges = std::ranges::clamp(belowLower, lowerBound, upperBound);
// Result: clampedBelowRanges = 10

// Using std::ranges::clamp to constrain a value above the upper bound
int clampedAboveRanges = std::ranges::clamp(aboveUpper, lowerBound, upperBound);
// Result: clampedAboveRanges = 20
```

19 Lexicographical Operations

- lexicographical_compare
- ranges::lexicographical_compare
- lexicographical_compare_three_way

20 Permutation Operations

- next_permutation
- ranges::next_permutation
- previous_permutation
- ranges::previous_permutation
- is_permutation
- ranges::is_permutation

```cpp
std::vector<int> numbers = {1, 2, 3};
std::vector<int> otherNumbers = {3, 2, 1};

// Using std::next_permutation to get the next lexicographical permutation
std::next_permutation(numbers.begin(), numbers.end());
// Result: {1, 3, 2}

// Using std::ranges::next_permutation to get the next lexicographical permutation
std::ranges::next_permutation(numbers);
// Result: {2, 1, 3}

// Using std::previous_permutation to get the previous lexicographical permutation
std::previous_permutation(numbers.begin(), numbers.end());
```

```
// Result: {1, 3, 2}

// Using std::ranges::previous_permutation to get the previous lexicographical permutation
std::ranges::previous_permutation(numbers);
// Result: {1, 2, 3}

// Using std::is_permutation to check if two sequences are permutations of each other
bool isPermutation = std::is_permutation(numbers.begin(), numbers.end(), otherNumbers.begin());
// Result: true

// Using std::ranges::is_permutation to check if two sequences are permutations of each other
bool isPermutationRanges = std::ranges::is_permutation(numbers, otherNumbers);
// Result: true
```

21 Numeric Operations

- iota
- ranges::iota

```
// Using std::iota to fill a vector with sequential values
std::vector<int> numbers(10);
std::iota(numbers.begin(), numbers.end(), 1);  // Fills with values starting from 1

// Using std::ranges::iota to fill another vector with sequential values
std::vector<int> moreNumbers(10);
std::ranges::iota(moreNumbers, 11);  // Fills with values starting from 11
```

- accumulate
- reduce
- transform_reduce

```
std::vector<int> numbers = {1, 2, 3, 4, 5};

// Using std::accumulate to sum the elements
int sum = std::accumulate(numbers.begin(), numbers.end(), 0);

// Using std::reduce to sum the elements (C++17)
int sumReduce = std::reduce(std::execution::seq, numbers.begin(), numbers.end(), 0);

// Using std::transform_reduce to compute the sum of squares
int sumOfSquares = std::transform_reduce(numbers.begin(), numbers.end(), 0, std::plus<>(), [](int n) { return n * n; });
```

- inner_product

```
std::vector<int> vector1 = {1, 2, 3};
std::vector<int> vector2 = {4, 5, 6};

// Using std::inner_product to compute the inner product of vector1 and vector2
```

```
int result = std::inner_product(vector1.begin(), vector1.end(), vector2.begin(), 0);
// result = 32
```

- adjacent_difference

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<int> partialSums(numbers.size());

// Using std::partial_sum to compute the partial sums of the numbers vector
std::partial_sum(numbers.begin(), numbers.end(), partialSums.begin());
// partialSums: [1, 3, 6, 10, 15]
```

- partial_sum

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<int> partialSums(numbers.size());

// Using std::partial_sum to compute the partial sums of the numbers vector
std::partial_sum(numbers.begin(), numbers.end(), partialSums.begin());
// partialSums = [1,3,6,10,15]
```

- exclusive_scan
- inclusive_scan
- transform_exclusive_scan
- transform_inclusive_scan

```
std::vector<int> numbers = {1, 2, 3, 4, 5};
std::vector<int> exclusiveScanResult(numbers.size());
std::vector<int> inclusiveScanResult(numbers.size());
std::vector<int> transformExclusiveScanResult(numbers.size());
std::vector<int> transformInclusiveScanResult(numbers.size());

// Using std::exclusive_scan to compute exclusive prefix sums
std::exclusive_scan(numbers.begin(), numbers.end(), exclusiveScanResult.begin(), 0);
// Result: {0, 1, 3, 6, 10}

// Using std::inclusive_scan to compute inclusive prefix sums
std::inclusive_scan(numbers.begin(), numbers.end(), inclusiveScanResult.begin());
// Result: {1, 3, 6, 10, 15}

// Using std::transform_exclusive_scan to compute exclusive prefix sums of squares
std::transform_exclusive_scan(numbers.begin(), numbers.end(), transformExclusiveScanResult.begin(), 0, std::plus<>(),
                                  [](int n) { return n * n; });
// Result: {0, 1, 5, 14, 30}

// Using std::transform_inclusive_scan to compute inclusive prefix sums of squares
std::transform_inclusive_scan(numbers.begin(), numbers.end(), transformInclusiveScanResult.begin(), std::plus<>(),
                                  [](int n) { return n * n; });
// Result: {1, 5, 14, 30, 55}
```

22 Uninitialized Memory Operations