<algorithm>

Karl Solomon

December 28, 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
```

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

- find

- find_if

- find_if_not

- ranges::find

- ranges::find_if

- ranges::find_if_not

```
1   #include <algorithm>
2   #include <iostream>
3   #include <ranges>
4   #include <vector>
5
6   int main() {
7       std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
8
9       // Use std::find to find the first occurrence of 5
10      auto it = std::find(numbers.begin(), numbers.end(), 5);
11
12      // find the first even number
13      it = std::find_if(numbers.begin(), numbers.end(), [](int n) {
           return n % 2 == 0; });
14      // find the first odd number
15      it = std::find_if_not(numbers.begin(), numbers.end(), [](int n
           ) { return n % 2 == 0; });
16      // find the first occurrence of 5
17      auto range_it = std::ranges::find(numbers, 5);
18      // find the first even number
19      range_it = std::ranges::find_if(numbers, [](int n) { return n
           % 2 == 0; });
20      // find the first odd number
21      range_it = std::ranges::find_if_not(numbers, [](int n) {
           return n % 2 == 0; });
22  }
```

- find_last

- find_last_if

- find_last_if_not

- find_end

- ranges::find_end

```
1   #include <algorithm>
2   #include <iostream>
3   #include <ranges>
4   #include <vector>
5
```

```
 6  std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 5, 6,
        7};
 7
 8  // find the last occurrence of 5
 9  auto lastFive = std::ranges::find_last(numbers, 5);
10  // find the last even number
11  auto lastEven = std::ranges::find_last_if(numbers, [](int n) {
        return n % 2 == 0; });
12  // find the last odd number
13  auto lastOdd = std::ranges::find_last_if_not(numbers, [](int n) {
        return n % 2 == 0; });
14
15  // Define a subrange to find
16  std::vector<int> subrange = {5, 6, 7};
17
18  // find the last occurrence of the subrange
19  auto lastSubrange = std::find_end(numbers.begin(), numbers.end(),
        subrange.begin(), subrange.end());
20
21  // find the last occurrence of the subrange
22  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

```
 1  std::vector<int> numbers = {1, 2, 3, 4, 5, 3, 4, 5, 6, 7};
 2  std::vector<int> pattern = {3, 4, 5};
 3
 4  // Using std::find_end to find the last occurrence of a pattern
 5  auto it_end = std::find_end(numbers.begin(), numbers.end(),
        pattern.begin(), pattern.end());
 6  // Result: it_end points to the first element of the last
        occurrence of {3, 4, 5}
 7
 8  // Using std::find_first_of to find the first occurrence of any
        element from another range
 9  std::vector<int> search_elements = {4, 5, 6};
10  auto it_first_of = std::find_first_of(numbers.begin(), numbers.end
        (), search_elements.begin(), search_elements.end());
11  // Result: it_first_of points to the first occurrence of any
        element from {4, 5, 6}, which is 4
12
13  // Using std::adjacent_find to find the first occurrence of two
        consecutive equal elements
14  std::vector<int> numbers_with_adjacent = {1, 2, 3, 3, 4, 5};
15  auto it_adjacent = std::adjacent_find(numbers_with_adjacent.begin
        (), numbers_with_adjacent.end());
16  // 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

```
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

```
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

```
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

```

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

- search

- search_n

- ranges::search

- ranges::search_n

```
1   std::vector<int> numbers = {1, 2, 3, 4, 5, 3, 4, 5, 6, 7};
2   std::vector<int> pattern = {3, 4, 5};
3
4   // Using std::search to find the first occurrence of a subsequence
5   auto it_search = std::search(numbers.begin(), numbers.end(),
        pattern.begin(), pattern.end());
6   // Result: it_search points to the first element of the first
        occurrence of {3, 4, 5}
7
8   // Using std::search_n to find the first occurrence of three
        consecutive 4s
9   auto it_search_n = std::search_n(numbers.begin(), numbers.end(),
        3, 4);
10  // Result: it_search_n points to numbers.end() as there are no
        three consecutive 4s
11
12  // Using std::ranges::search to find the first occurrence of a
        subsequence
13  auto ranges_it_search = std::ranges::search(numbers, pattern);
14  // Result: ranges_it_search.begin() points to the first element of
         the first occurrence of {3, 4, 5}
15
16  // Using std::ranges::search_n to find the first occurrence of two
         consecutive 5s
17  auto ranges_it_search_n = std::ranges::search_n(numbers, 2, 5);
18  // Result: ranges_it_search_n.begin() points to numbers.end() as
        there are no two consecutive 5s
```

- ranges::starts_with

- ranges::ends_with

```
1   std::vector<int> numbers = {1, 2, 3, 4, 5};
2   std::vector<int> prefix = {1, 2};
3   std::vector<int> suffix = {4, 5};
4   std::vector<int> non_prefix = {2, 3};
5   std::vector<int> non_suffix = {3, 4};
6
7   // Using std::ranges::starts_with to check if numbers starts with
        prefix
8   bool starts_with_prefix = std::ranges::starts_with(numbers, prefix
        );
9   // Result: starts_with_prefix = true
10
11  // Using std::ranges::starts_with to check if numbers starts with
        non_prefix
12  bool starts_with_non_prefix = std::ranges::starts_with(numbers,
        non_prefix);
13  // Result: starts_with_non_prefix = false
14
15  // Using std::ranges::ends_with to check if numbers ends with
        suffix
```

```
16  bool ends_with_suffix = std::ranges::ends_with(numbers, suffix);
17  // Result: ends_with_suffix = true
18
19  // Using std::ranges::ends_with to check if numbers ends with
        non_suffix
20  bool ends_with_non_suffix = std::ranges::ends_with(numbers,
        non_suffix);
21  // 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

```
1   std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2   std::vector<int> result(10);
3
4   // Using std::copy to copy all elements
5   std::copy(numbers.begin(), numbers.end(), result.begin());
6   // Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
7
8   // Using std::copy_if to copy only even numbers
9   auto it = std::copy_if(numbers.begin(), numbers.end(), result.
        begin(), [](int n) { return n % 2 == 0; });
10  // Result: result = {2, 4, 6, 8, 10, ?, ?, ?, ?, ?} (remaining
        elements are unspecified)
11
12  // Using std::ranges::copy to copy all elements
13  std::ranges::copy(numbers, result.begin());
14  // Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
15
16  // Using std::ranges::copy_if to copy only even numbers
17  it = std::ranges::copy_if(numbers, result.begin(), [](int n) {
        return n % 2 == 0; });
```

```
18  // Result: result = {2, 4, 6, 8, 10, ?, ?, ?, ?, ?} (remaining
       elements are unspecified)
19
20  // Using std::copy_n to copy the first 5 elements
21  std::copy_n(numbers.begin(), 5, result.begin());
22  // Result: result = {1, 2, 3, 4, 5, ?, ?, ?, ?, ?} (remaining
       elements are unspecified)
23
24  // Using std::ranges::copy_n to copy the first 5 elements
25  std::ranges::copy_n(numbers.begin(), 5, result.begin());
26  // Result: result = {1, 2, 3, 4, 5, ?, ?, ?, ?, ?} (remaining
       elements are unspecified)
27
28  // Using std::copy_backward to copy elements in reverse order
29  std::copy_backward(numbers.begin(), numbers.end(), result.end());
30  // Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
31
32  // Using std::ranges::copy_backward to copy elements in reverse
       order
33  std::ranges::copy_backward(numbers, result.end());
34  // Result: result = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

- move

- ranges::move

- move_backward

- ranges::move_backward

```
1   std::vector<int> source = {1, 2, 3, 4, 5};
2   std::vector<int> destination(5);
3
4   // Using std::move to transfer elements from source to destination
5   std::move(source.begin(), source.end(), destination.begin());
6   // Result: destination = {1, 2, 3, 4, 5}
7   // Result: source = {?, ?, ?, ?, ?} (unspecified, but valid state)
8
9   // Reset source for the next example
10  source = {6, 7, 8, 9, 10};
11
12  // Using std::move_backward to transfer elements from source to
       destination in reverse order
13  std::move_backward(source.begin(), source.end(), destination.end()
       );
14  // Result: destination = {6, 7, 8, 9, 10}
15  // Result: source = {?, ?, ?, ?, ?} (unspecified, but valid state)
```

5 Swap Operations

- swap

- swap_ranges

- ranges::swap_ranges

- iter_swap

```
1   // Demonstrate std::swap
2   int a = 10, b = 20;
3   // Before swap: a = 10, b = 20
4   std::swap(a, b);
5   // After swap: a = 20, b = 10
```

```
6
7   // Demonstrate std::swap_ranges
8   std::vector<int> vec1 = {1, 2, 3, 4, 5};
9   std::vector<int> vec2 = {6, 7, 8, 9, 10};
10  // Before swap_ranges: vec1 = 1 2 3 4 5, vec2 = 6 7 8 9 10
11
12  std::swap_ranges(vec1.begin(), vec1.end(), vec2.begin());
13  // After swap_ranges: vec1 = 6 7 8 9 10, vec2 = 1 2 3 4 5
14
15  // Demonstrate std::ranges::swap_ranges
16  std::vector<int> vec3 = {11, 12, 13, 14, 15};
17  std::vector<int> vec4 = {16, 17, 18, 19, 20};
18  // Before ranges::swap_ranges: vec3 = 11 12 13 14 15, vec4 = 16 17
        18 19 20
19
20  std::ranges::swap_ranges(vec3, vec4);
21  // After ranges::swap_ranges: vec3 = 16 17 18 19 20, vec4 = 11 12
        13 14 15
22
23  // Demonstrate std::iter_swap
24  std::vector<int> vec5 = {21, 22, 23, 24, 25};
25  // Before iter_swap: vec5 = 21 22 23 24 25
26
27  std::iter_swap(vec5.begin(), vec5.begin() + 4);
28  // After iter_swap: vec5 = 25 22 23 24 21
```

6 Transform Operations

- transform

- ranges::transform

```
1   #include <algorithm>
2   #include <execution>
3   #include <iostream>
4   #include <vector>
5
6   int main() {
7       // Original vector
8       std::vector<int> l1 = {1, 2, 3, 4, 5};
9       std::vector<int> l2 = std::vector<int>(l1.size(), 0);
10      std::vector<int> l3 = std::vector<int>(l1.size(), 0);
11
12      // simple transform (1 input, 1 output)
13      std::transform(l1.begin(), l1.end(), l2.begin(), [](int a) {
            return a * 10; });
14
15      // transform (2 inputs, 1 output)
16      int multiplier = 2;
17      std::transform(std::execution::par_unseq, l1.begin(), l1.end()
            , l2.begin(), l3.begin(),
18                     [multiplier](int a, int b) { return (multiplier
                        * a) + b; });
19
20      return 0;
21  }
```

- 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;

```

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

8 Removing Operations

- remove: removes all occurrences of a specified value

- remove_if: removes all elements that satisfy a defined condition

- ranges::remove

- ranges::remove_if

- remove_copy: copies elements from the source to the result that are not equal to the value

- remove_copy_if: copies elements from the source to the result that do not satisfy the condition

- ranges::remove_copy

- ranges::remove_copy_if

```
1   std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
2   std::vector<int> result(10);
3
4   // Using std::remove to remove all occurrences of 5
5   auto end_remove = std::remove(numbers.begin(), numbers.end(), 5);
6   // Result: numbers = {1, 2, 3, 4, 6, 7, 8, 9, 10, ?} (last element
         unspecified)
7
8   // Using std::remove_if to remove all even numbers
9   auto end_remove_if = std::remove_if(numbers.begin(), numbers.end()
       , [](int n) { return n % 2 == 0; });
10  // Result: numbers = {1, 3, 5, 7, 9, ?, ?, ?, ?, ?} (remaining
       elements unspecified)
11
12  // Reset numbers for the next example
13  numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
14
15  // Using std::remove_copy to copy elements except 5 to result
16  auto end_remove_copy = std::remove_copy(numbers.begin(), numbers.
       end(), result.begin(), 5);
17  // Result: result = {1, 2, 3, 4, 6, 7, 8, 9, 10, ?} (last element
       unspecified)
18
19  // Using std::remove_copy_if to copy elements except even numbers
       to result
20  auto end_remove_copy_if =
21      std::remove_copy_if(numbers.begin(), numbers.end(), result.
           begin(), [](int n) { return n % 2 == 0; });
22  // Result: result = {1, 3, 5, 7, 9, ?, ?, ?, ?, ?} (remaining
       elements unspecified)
```

- unique

- unique_copy

- ranges::unique

- `ranges::unique_copy`

```cpp
std::vector<int> numbers = {1, 2, 2, 3, 4, 4, 4, 5, 6, 6, 7};
std::vector<int> result(11);

// Using std::unique to remove consecutive duplicates in place
auto end_unique = std::unique(numbers.begin(), numbers.end());
// Result: numbers = {1, 2, 3, 4, 5, 6, 7, ?, ?, ?, ?} (remaining
    elements unspecified)

// Reset numbers for the next example
numbers = {1, 2, 2, 3, 4, 4, 4, 5, 6, 6, 7};

// Using std::unique_copy to copy unique elements to result
auto end_unique_copy = std::unique_copy(numbers.begin(), numbers.
    end(), result.begin());
// Result: result = {1, 2, 3, 4, 5, 6, 7, ?, ?, ?, ?} (remaining
    elements unspecified)
```

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`

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

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

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

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

```

```cpp
15  // Using std::rotate to rotate elements in place
16  std::rotate(numbers.begin(), numbers.begin() + 3, numbers.end());
17  // Result: numbers = {4, 5, 6, 7, 8, 9, 10, 1, 2, 3}
18
19  // Reset numbers for the next example
20  numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
21
22  // Using std::rotate_copy to copy rotated elements to result
23  std::rotate_copy(numbers.begin(), numbers.begin() + 3, numbers.end
        (), result.begin());
24  // Result: result = {4, 5, 6, 7, 8, 9, 10, 1, 2, 3}
25
26  // Using std::shift_left to shift elements to the left
27  std::shift_left(numbers.begin(), numbers.end(), 3);
28  // Result: numbers = {4, 5, 6, 7, 8, 9, 10, ?, ?, ?} (last
        elements unspecified)
29
30  // Reset numbers for the next example
31  numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
32
33  // Using std::shift_right to shift elements to the right
34  std::shift_right(numbers.begin(), numbers.end(), 3);
35  // Result: numbers = {?, ?, ?, 1, 2, 3, 4, 5, 6, 7} (first
        elements unspecified)
36
37  // Using std::shuffle to randomly shuffle elements
38  std::random_device rd;
39  std::mt19937 g(rd());
40  std::shuffle(numbers.begin(), numbers.end(), g);
41  // Result: numbers = {?, ?, ?, ?, ?, ?, ?, ?, ?, ?} (random order)
```

10 Random Number Generation

- ranges::generate_random

11 Sampling Operations

- sample

- ranges::sample

```cpp
1  #include <algorithm>  // For std::sample
2  #include <random>     // For std::random_device, std::mt19937
3  #include <ranges>     // For std::ranges::sample
4  #include <vector>
5
6  int main() {
7      std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
8      std::vector<int> result(5);
9
10     // Using std::sample to randomly select 5 elements
11     std::random_device rd;
12     std::mt19937 gen(rd());
13     std::sample(numbers.begin(), numbers.end(), result.begin(), 5,
            gen);
14     // Result: result = {?, ?, ?, ?, ?} (random selection of 5
            elements)
15
16     // Using std::ranges::sample to randomly select 5 elements
17     std::ranges::sample(numbers, result.begin(), 5, gen);
18     // Result: result = {?, ?, ?, ?, ?} (random selection of 5
            elements)
```

```
19
20      return 0;
21  }
```

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`

```cpp
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}

```

```
14  // Using std::inplace_merge to merge two consecutive sorted ranges
        within a single range
15  std::vector<int> inplaceVec = {1, 3, 5, 7, 2, 4, 6, 8};
16  std::inplace_merge(inplaceVec.begin(), inplaceVec.begin() + 4,
       inplaceVec.end());
17  // Result: inplaceVec = {1, 2, 3, 4, 5, 6, 7, 8}
18
19  // Using std::ranges::inplace_merge to merge two consecutive
        sorted ranges within a single range
20  std::vector<int> inplaceVecRanges = {1, 3, 5, 7, 2, 4, 6, 8};
21  std::ranges::inplace_merge(inplaceVecRanges, inplaceVecRanges.
       begin() + 4);
22  // Result: inplaceVecRanges = {1, 2, 3, 4, 5, 6, 7, 8}
```

17 Heap Operations

- push_heap

- ranges::push_heap

- pop_heap

- ranges::pop_heap

```
1   std::vector<int> heap = {3, 1, 4, 1, 5, 9, 2, 6};
2
3   // Convert the vector into a heap
4   std::make_heap(heap.begin(), heap.end());
5   // Result: heap = {9, 6, 4, 1, 5, 3, 2, 1}
6
7   // Using std::push_heap to add a new element and maintain heap
        property
8   heap.push_back(7);
9   std::push_heap(heap.begin(), heap.end());
10  // Result: heap = {9, 7, 4, 6, 5, 3, 2, 1, 1}
11
12  // Using std::pop_heap to remove the largest element and maintain
        heap property
13  std::pop_heap(heap.begin(), heap.end());
14  heap.pop_back();
15  // Result: heap = {7, 6, 4, 1, 5, 3, 2, 1}
16
17  // Using std::ranges::push_heap to add a new element and maintain
        heap property
18  heap.push_back(8);
19  std::ranges::push_heap(heap);
20  // Result: heap = {8, 7, 4, 6, 5, 3, 2, 1, 1}
21
22  // Using std::ranges::pop_heap to remove the largest element and
        maintain heap property
23  std::ranges::pop_heap(heap);
24  heap.pop_back();
25  // 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());
```

```
28  // Result: heapEnd points to numbers.begin() + 3, where the value
        10 violates the heap property
29
30  // Re-check using std::ranges::is_heap
31  isHeapRanges = std::ranges::is_heap(numbers);
32  // Result: isHeapRanges = false
33
34  // Re-check using std::ranges::is_heap_until
35  heapEndRanges = std::ranges::is_heap_until(numbers);
36  // 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

```
1   int a = 10;
2   int b = 20;
3
4   // Using std::max to find the maximum of two values
5   int maxVal = std::max(a, b);
6   // Result: maxVal = 20
7
8   // Using std::min to find the minimum of two values
9   int minVal = std::min(a, b);
10  // Result: minVal = 10
11
12  std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
13
14  // Using std::ranges::max to find the maximum value in a range
15  int maxInRange = std::ranges::max(numbers);
16  // Result: maxInRange = 9
17
18  // Using std::ranges::min to find the minimum value in a range
19  int minInRange = std::ranges::min(numbers);
20  // Result: minInRange = 1
```

- max_element

- min_element

- ranges::max_element

- ranges::min_element

```
1   std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
2
3   // Using std::max_element to find the maximum element in a range
4   auto maxElement = std::max_element(numbers.begin(), numbers.end())
        ;
5   // Result: *maxElement = 9
6
7   // Using std::min_element to find the minimum element in a range
8   auto minElement = std::min_element(numbers.begin(), numbers.end())
        ;
9   // Result: *minElement = 1
10
```

```
11  // Using std::ranges::max_element to find the maximum element in a
        range
12  auto maxElementRanges = std::ranges::max_element(numbers);
13  // Result: *maxElementRanges = 9
14
15  // Using std::ranges::min_element to find the minimum element in a
        range
16  auto minElementRanges = std::ranges::min_element(numbers);
17  // Result: *minElementRanges = 1
```

- minmax

- ranges::minmax

- minmax_element

- ranges::minmax_element

```
1   int a = 10;
2   int b = 20;
3
4   // Using std::minmax to find the minimum and maximum of two values
5   auto minmaxPair = std::minmax(a, b);
6   // Result: minmaxPair.first = 10, minmaxPair.second = 20
7
8   std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};
9
10  // Using std::minmax_element to find the minimum and maximum
        elements in a range
11  auto minmaxElements = std::minmax_element(numbers.begin(), numbers
        .end());
12  // Result: *minmaxElements.first = 1, *minmaxElements.second = 9
13
14  // Using std::ranges::minmax to find the minimum and maximum in a
        range
15  auto minmaxRange = std::ranges::minmax(numbers);
16  // Result: minmaxRange.min = 1, minmaxRange.max = 9
17
18  // Using std::ranges::minmax_element to find the minimum and
        maximum elements in a range
19  auto minmaxRangeElements = std::ranges::minmax_element(numbers);
20  // Result: *minmaxRangeElements.min = 1, *minmaxRangeElements.max
        = 9
21  //
22  // TODO (ksolomon): make sure usage is correct. why does minmax
        return a pair, whereas ranges::minmax return a tuple?
23  //
```

- clamp

- ranges::clamp

```
1   int value = 15;
2   int lowerBound = 10;
3   int upperBound = 20;
4
5   // Using std::clamp to constrain the value within the range [
        lowerBound, upperBound]
6   int clampedValue = std::clamp(value, lowerBound, upperBound);
7   // Result: clampedValue = 15
8
9   // Using std::clamp to constrain a value below the lower bound
10  int belowLower = 5;
11  int clampedBelow = std::clamp(belowLower, lowerBound, upperBound);
```

```
12  // Result: clampedBelow = 10
13
14  // Using std::clamp to constrain a value above the upper bound
15  int aboveUpper = 25;
16  int clampedAbove = std::clamp(aboveUpper, lowerBound, upperBound);
17  // Result: clampedAbove = 20
18
19  // Using std::ranges::clamp to constrain the value within the
       range [lowerBound, upperBound]
20  int clampedValueRanges = std::ranges::clamp(value, lowerBound,
       upperBound);
21  // Result: clampedValueRanges = 15
22
23  // Using std::ranges::clamp to constrain a value below the lower
       bound
24  int clampedBelowRanges = std::ranges::clamp(belowLower, lowerBound
       , upperBound);
25  // Result: clampedBelowRanges = 10
26
27  // Using std::ranges::clamp to constrain a value above the upper
       bound
28  int clampedAboveRanges = std::ranges::clamp(aboveUpper, lowerBound
       , upperBound);
29  // 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

```
1  std::vector<int> numbers = {1, 2, 3};
2  std::vector<int> otherNumbers = {3, 2, 1};
3
4  // Using std::next_permutation to get the next lexicographical
       permutation
5  std::next_permutation(numbers.begin(), numbers.end());
6  // Result: {1, 3, 2}
7
8  // Using std::ranges::next_permutation to get the next
       lexicographical permutation
9  std::ranges::next_permutation(numbers);
10  // Result: {2, 1, 3}
11
12  // Using std::previous_permutation to get the previous
       lexicographical permutation
13  std::previous_permutation(numbers.begin(), numbers.end());
```

```
14  // Result: {1, 3, 2}
15
16  // Using std::ranges::previous_permutation to get the previous
       lexicographical permutation
17  std::ranges::previous_permutation(numbers);
18  // Result: {1, 2, 3}
19
20  // Using std::is_permutation to check if two sequences are
       permutations of each other
21  bool isPermutation = std::is_permutation(numbers.begin(), numbers.
       end(), otherNumbers.begin());
22  // Result: true
23
24  // Using std::ranges::is_permutation to check if two sequences are
        permutations of each other
25  bool isPermutationRanges = std::ranges::is_permutation(numbers,
       otherNumbers);
26  // Result: true
```

21 Numeric Operations

- `iota`

- `ranges::iota`

```
1  // Using std::iota to fill a vector with sequential values
2  std::vector<int> numbers(10);
3  std::iota(numbers.begin(), numbers.end(), 1);  // Fills with
      values starting from 1
4
5  // Using std::ranges::iota to fill another vector with sequential
      values
6  std::vector<int> moreNumbers(10);
7  std::ranges::iota(moreNumbers, 11);  // Fills with values starting
       from 11
```

- `accumulate`

- `reduce`

- `transform_reduce`

```
1  std::vector<int> numbers = {1, 2, 3, 4, 5};
2
3  // Using std::accumulate to sum the elements
4  int sum = std::accumulate(numbers.begin(), numbers.end(), 0);
5
6  // Using std::reduce to sum the elements (C++17)
7  int sumReduce = std::reduce(std::execution::seq, numbers.begin(),
      numbers.end(), 0);
8
9  // Using std::transform_reduce to compute the sum of squares
10 int sumOfSquares = std::transform_reduce(numbers.begin(), numbers.
      end(), 0, std::plus<>(), [](int n) { return n * n; });
```

- `inner_product`

```
1  std::vector<int> vector1 = {1, 2, 3};
2  std::vector<int> vector2 = {4, 5, 6};
3
4  // Using std::inner_product to compute the inner product of
      vector1 and vector2
```

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

- adjacent_difference

```
1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 std::vector<int> partialSums(numbers.size());
3
4 // Using std::partial_sum to compute the partial sums of the
    numbers vector
5 std::partial_sum(numbers.begin(), numbers.end(), partialSums.begin
    ());
6 // partialSums: [1, 3, 6, 10, 15]
```

- partial_sum

```
1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 std::vector<int> partialSums(numbers.size());
3
4 // Using std::partial_sum to compute the partial sums of the
    numbers vector
5 std::partial_sum(numbers.begin(), numbers.end(), partialSums.begin
    ());
6 // partialSums = [1,3,6,10,15]
```

- exclusive_scan

- inclusive_scan

- transform_exclusive_scan

- transform_inclusive_scan

```
1 std::vector<int> numbers = {1, 2, 3, 4, 5};
2 std::vector<int> exclusiveScanResult(numbers.size());
3 std::vector<int> inclusiveScanResult(numbers.size());
4 std::vector<int> transformExclusiveScanResult(numbers.size());
5 std::vector<int> transformInclusiveScanResult(numbers.size());
6
7 // Using std::exclusive_scan to compute exclusive prefix sums
8 std::exclusive_scan(numbers.begin(), numbers.end(),
    exclusiveScanResult.begin(), 0);
9 // Result: {0, 1, 3, 6, 10}
10
11 // Using std::inclusive_scan to compute inclusive prefix sums
12 std::inclusive_scan(numbers.begin(), numbers.end(),
    inclusiveScanResult.begin());
13 // Result: {1, 3, 6, 10, 15}
14
15 // Using std::transform_exclusive_scan to compute exclusive prefix
     sums of squares
16 std::transform_exclusive_scan(numbers.begin(), numbers.end(),
    transformExclusiveScanResult.begin(), 0, std::plus<>(),
17                               [](int n) { return n * n; });
18 // Result: {0, 1, 5, 14, 30}
19
20 // Using std::transform_inclusive_scan to compute inclusive prefix
     sums of squares
21 std::transform_inclusive_scan(numbers.begin(), numbers.end(),
    transformInclusiveScanResult.begin(), std::plus<>(),
22                               [](int n) { return n * n; });
23 // Result: {1, 5, 14, 30, 55}
```

22 Uninitialized Memory Operations