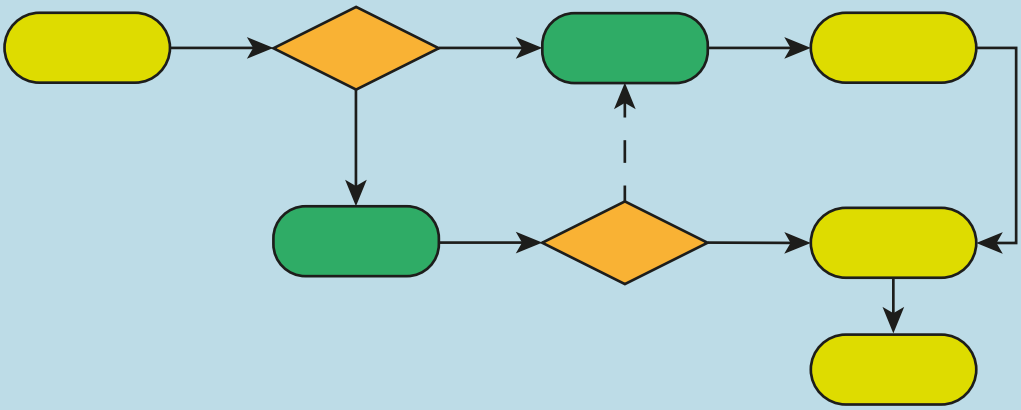
A Complete Guide to

Standard C++ Algorithms



RNDr. Šimon Tóth

A complete guide to

Standard C++ Algorithms

RNDr. Šimon Tóth

version 1.0.1
https://github.com/HappyCerberus/book-cpp-algorithms

February 3, 2023

Original copy of this book can be obtained at https://github.com/HappyCerberus/book-cpp-algorithms.

The book is also available through LeanPub where the proceeds go to Electronic Frontier Foundation (after LeanPub takes their cut) https://leanpub.com/cpp-algorithms-guide.

This copy of the book is version 1.0.1.

Preface

This book will not start with a personal story or some other flowery recollections. Instead, to protect your time, I will lay out precisely what this book is about and who am I to be qualified to write this book. Hopefully, this will help you decide whether reading this book is a good use of your time.

About this book

This book is a complete guide to the C++ standard algorithms. However, that might not mean much to you, so let me unpack this statement.

This book is a guide, as opposed to a reference, meaning that instead of describing every detail, the book concentrates on examples and pointing out notable, surprising, dangerous or interesting aspects of the different algorithms. Furthermore, unlike a reference, it is supposed to be read, for the most part, like a book in sequential order.

C++ already has one canonical reference, the C++ standard, and for quick lookup, the cppreference.com wiki is a great source.

The "complete" part of the statement refers to the width of coverage. The book covers all algorithms and relevant theory up to the C++20 standard (the C++23 standard is not finalized at the time of writing). All information is present only in sufficient depth required by the context. This depth limitation keeps the book's overall size reasonable and within the "guide" style.

About the author

I am Šimon Tóth, the sole author of this book. My primary qualification is 20 years of C++ experience, with approximately 15 of those years C++ being my primary language in professional settings.

My background is HPC, spanning academia, big tech and startup environments. I have architected, built and operated systems of all scales, from single machine hardware supported high-availability to planet-scale services.[1]

Throughout my career, my passion has always been teaching and mentoring junior engineers, which is why you are now reading this book.

[1] You can check my LinkedIn profile for a detailed view of my past career.

Feedback

Creating free educational content is very much akin to shouting into the void. There are no sales statistics to follow or milestones to hit. Therefore, please let me know if you read this book and find it helpful or hate it. It will inform my future efforts.

Why cc-by-sa-nc

This book is licensed CC-BY-SA-NC, which is both an open but, at the same time, minimal license. I aim to allow derivative works (such as translations) but not to permit commercial use, such as using the book as the basis for commercial training or selling printed copies.

Explicitly, any personal use is permitted. For example, you can read, print, or share the book with your friends.

If you want to use this content where you are unsure whether you fit within the Creative Commons commercial definition[2], feel free to contact me on Mastodon, LinkedIn or by email (my DMs are always open).

Book status

This book is currently content-complete (upto and including C++20).

To keep up with the changes, visit the hosting repository: `https://github.com/HappyCerberus/book-cpp-algorithms`.

Changelog

1.0.1 Small (mostly) formatting changes.

1.0.0 First complete release.

[2] primarily intended for or directed toward commercial advantage or monetary compensation

Contents

Chapter 1

Introduction

The C++ standard library is arguably quite limited in its functionality. However, when it comes to data and number crunching, the C++ standard library provides a versatile toolkit of algorithms.

If you are a C++ developer, good familiarity with C++ standard algorithms can save you a lot of effort and accidental bugs. Notably, whenever you see a raw loop in your code, you should question whether calling a standard algorithm wouldn't be a better solution (it usually is).

1.1 History of standard C++ algorithms

While each C++ standard introduced new algorithms or variants, there are few notable milestones in the history of C++ standard algorithms.

The C++98 standard introduced most of the algorithms. However, it was the C++11 standard with its introduction of lambdas that made algorithms worthwhile. Before lambdas, the time investment of writing a custom function object made the usefulness of algorithms dubious.

Example of `std::for_each` algorithm with a custom function object, calculating the number of elements and their sum.

```cpp
struct StatsFn {
    int cnt = 0;
    int sum = 0;
    void operator()(int v) {
        cnt++;
        sum += v;
    }
};

std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
auto result = std::for_each(data.begin(), data.end(), StatsFn{});
// result == {9, 45}
```

Open in Compiler Explorer

Example of `std::for_each` algorithm with a capturing lambda, calculating the number of elements and their sum.

```cpp
int cnt = 0, sum = 0;
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::for_each(data.begin(), data.end(), [&](int el) {
    cnt++;
        sum += el;
    });
// cnt == 9, sum == 45
```

Open in Compiler Explorer

The C++17 standard introduced parallel algorithms that provide an easy way to speed up processing with minimal effort. All you need to do is to specify the desired execution model, and the library will take care of parallelizing the execution.

Example of `std::for_each` algorithm using unsequenced parallel execution model. Note that counters are now shared state and need to be `std::atomic` or protected by a `std::mutex`.

```cpp
std::atomic<int> cnt = 0, sum = 0;
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::for_each(std::execution::par_unseq,
    data.begin(), data.end(),
    [&](int el) {
        cnt++;
        sum += el;
    });
// cnt == 9, sum == 45
```

Open in Compiler Explorer

Finally, the C++20 standard introduced a significant re-design in the form of ranges and views. Range versions of algorithms can now operate on ranges instead of `begin` and `end` iterators and views provide lazily evaluated versions of algorithms and utilities.

Example of the range version of the `std::for_each` algorithm.

```
int cnt = 0, sum = 0;
std::vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};
std::ranges::for_each(data, [&](int el) {
    cnt++;
    sum += el;
});
// cnt == 9, sum == 45
```

Open in Compiler Explorer

As of the time of writing, the C++23 standard is not finalized. However, we already know that it will introduce more range algorithms, more views and the ability to implement custom views.

1.2 Iterators and ranges

Algorithms operate on data structures, which poses an issue. How do you abstract the implementation details of a specific data structure and allow the algorithm to work with any data structure that satisfies the algorithm's requirements?

The C++ standard library solution to this problem are iterators and ranges. Iterators encapsulate implementation details of data structure traversal and simultaneously expose a set of operations possible on the given data structure in constant time and space.

A range is then denoted by a pair of iterators, or more generally, since C++20, an iterator and a sentinel. In mathematical terms, a pair of iterators `it1`, `it2` denotes a range `[it1, it2)`, that is, the range includes the element referenced by `it1` and ends before the element referenced by `it2`.

To reference the entire content of a data structure, we can use the `begin()` and `end()` methods that return an iterator to the first element and an iterator one past the last element, respectively. Hence, the range `[begin, end)` contains all data structure elements.

Example of specifying a range using two iterators.

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

auto it1 = data.begin();
auto it2 = it1 + 2;
std::for_each(it1, it2, [](int el) {
    std::cout << el << ", ";
});
// Prints: 1, 2,

auto it3 = data.begin() + 5;
auto it4 = data.end();
std::for_each(it3, it4, [](int el) {
    std::cout << el << ", ";
});
// Prints: 6, 7, 8, 9,
```

Open in Compiler Explorer

Sentinels follow the same idea. However, they do not need to be of an iterator type. Instead, they only need to be comparable to an iterator. The exclusive end of the range is then the first iterator that compares equal to the sentinel.

Example of specifying a range using an iterator and custom sentinel. The sentinel will compare `true` with iterators at least the given distance from the start iterator, therefore defining a range with the specified number of elements.

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

struct Sentinel {
    using iter_t = std::vector<int>::iterator;
    iter_t begin;
    std::iter_difference_t<iter_t> cnt;
    bool operator==(const iter_t& l) const {
        return std::distance(begin, l) >= cnt;
    }
};

auto it1 = data.begin();
std::ranges::for_each(it1, Sentinel{it1, 5}, [](int el) {
    std::cout << el << ", ";
});
// Prints: 1, 2, 3, 4, 5,
```

Open in Compiler Explorer

1.2.1 Iterator categories

The set of operations that are possible in constant time and space defines the following categories of iterators (and consequently ranges):

input/output iterator read/write each element once, advance
data streams, e.g. writing/reading data to/from a network socket

forward iterator read/write each element repeatedly, advance
singly-linked list, e.g. std::forward_list

bidirectional iterator forward iterator + move back
doubly-linked list, e.g. std::list, std::map, std::set

random access iterator bidirectional iterator + advance and move back by any integer and calculate distance between two iterators
multi-array data structures, e.g. std::deque

contiguous iterator random access iterator + the storage of elements is contiguous
arrays, e.g. std::vector

Example demonstrating the difference between a random access iterator provided by `std::vector` and a bidirectional iterator provided by `std::list`.

```
std::vector<int> arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto it1 = arr.begin();
it1 += 5; // OK, std::vector provides random access iterator
++it1; // OK, all iterators provide advance operation

ptrdiff_t dst1 = it1 - arr.begin(); // OK, random access iterator
// dst1 == 6

std::list<int> lst = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
auto it2 = lst.begin();
// it2 += 5; Would not compile.
std::advance(it2, 5); // OK, linear advance by 5 steps
++it2; // OK, all iterators provide advance operation

// it2 - lst.begin(); Would not compile
ptrdiff_t dst2 = std::distance(lst.begin(), it2); // OK, linear calc.
// dst2 == 6
```

Open in Compiler Explorer

1.2.2 Range categories

Ranges can be classified using the same categories as iterators. In this book, we will be using the range nomenclature over iterators (e.g. input range, forward range, bidirectional range, etc.).

1.3 Naming and common behaviour

While the naming of many algorithms is sub-optimal, there are a few common naming patterns.

1.3.1 Counted variants "_n"

Counted variants of algorithms accept the range specified using the start iterator and the number of elements (instead of begin and end). This behaviour can be a convenient alternative when working with input and output ranges, where we often do not have an explicit end iterator.

Examples: `std::for_each_n`, `std::copy_n`

Note: while `std::search_n` does follow the naming, it does not follow the same semantics. The _n here refers to the number of instances of the searched element.

1.3.2 Copy variants "_copy"

Copy variants of in-place algorithms do not write their output back to the source range. Instead, they output the result to one or more output ranges, usually defined by a single iterator denoting the first element to be written to (the number of elements is implied from the source range). The copy behaviour allows these variants to operate on immutable ranges.

Examples: `std::remove_copy`, `std::partial_sort_copy`

1.3.3 Predicate variants "_if"

Predicate variants of algorithms use a predicate to determine a "match" instead of comparing against a value. The standard also has one instance of `_if_not` variant that inverts the predicate logic (`false` is treated as a match).

Examples: `std::find_if`, `std::replace_if`

1.3.4 Restrictions on invocable

Many algorithms can be customized using an invocable. However, with a few exceptions, the invocable is not permitted to modify elements of the range or invalidate iterators. On top of that, unless explicitly noted, the algorithms do not guarantee any particular order of invocation.

These restrictions in practice mean that the passed invocable must be regular. The invocable must return the same result if invoked again with the same arguments. This definition permits accessing a global state such as a cache but does not permit invocables that change their result based on their internal state (such as generators).

1.4 A simpler mental model for iterators

It can be tricky to internalize all the rules associated with iterators when working with standard algorithms. One shorthand that can help is to think in terms of ranges instead of iterators.

Ranges passed in as arguments are usually apparent, typically specified by pair of iterators.

Example with two ranges passed in as an argument. The input range is fully specified, and the end iterator for the output range is implied from the number of elements in the input range.

```cpp
std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
std::vector<int> out(7,0);

std::copy(data.begin(), data.end(), // input range
    out.begin() // output range, end iterator is implied:
    // std::next(out.begin(),
    //        std::distance(data.begin(), data.end()));
);
```

Open in Compiler Explorer

The returned range can also be evident from the semantics of the algorithm.

Example of `std::is_sorted_until` that returns an iterator to the first out-of-order element, which can also be thought as the end iterator for a maximal sorted sub-range.

```cpp
std::vector<int> data{1, 4, 5, 7, 9, 2, 3};

// is_sorted_until returns the first out of order element.
auto result = std::is_sorted_until(data.begin(), data.end());

// [begin, result) is the maximal sorted sub-range
for (auto it = data.begin(); it != result; it++) {
    // Iterate over all elements in the sorted sub-range.
    // {1, 4, 5, 7, 9}
}
for (auto v : std::ranges::subrange(data.begin(), result)) {
    // Same, but using a range-based for loop.
}
```

Open in Compiler Explorer

The benefit of thinking about the returned value as the end iterator of a range is that it removes the potential for corner cases. For example, what if the algorithm doesn't find any element out of order? The returned value will be the end iterator of the source range, meaning that the range returned is simply the entire source range.

In some cases, a single returned iterator denotes multiple ranges.

Example of `std::lower_bound` that splits the range into two sub-ranges.

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

// lower_bound returns the first element !(el < 4)
auto lb = std::lower_bound(data.begin(), data.end(), 4);

for (auto v : std::ranges::subrange(data.begin(), lb)) {
    // lower range [begin, lb): elements < 4
}
for (auto v : std::ranges::subrange(lb, data.end())) {
    // upper range [lb, end): elements >= 4
}
```

Open in Compiler Explorer

Even when the algorithm returns an iterator to a specific element, it might be worth considering the implied range.

Example of `std::find` establishing a prefix range that doesn't contain the searched-for element.

```cpp
std::string str("Hello World!");

// Returns the iterator to the first occurence of ' '
auto it = std::find(str.begin(), str.end(), ' ');

// Range [begin, it) is the maximal prefix range
// that doesn't contain ' '
for (auto v : std::string_view(str.begin(), it)) {
    // iterate over "Hello"
}
```

Open in Compiler Explorer

Chapter 2

The algorithms

2.1 Introducing the algorithms

In this chapter, we introduce each of the standard algorithms. The groups of algorithms are arbitrary and mainly introduced for presentation clarity. Therefore, you might correctly argue that a specific algorithm would be better suited to reside in a different group.

Before we start, we will use the `std::for_each` and `std::for_each_n` algorithms to demonstrate this chapter's structure for each algorithm.

① The presentation of each algorithm will start with a short description.

② The margin will contain information about the history of the algorithm: which C++ standard introduced it and whether it has constexpr, parallel and range variants and including versions of the standard that introduced them.

③ Following that will be the description of constraints. Algorithms that write data to a distinct output range are denoted with an arrow: `input_range -> output_range`.

④ Finally, each description will conclude with one or more examples with explanations.

2.1.1 `std::for_each`

① The `std::for_each` algorithm applies the provided invocable to each element of the range in order. If the underlying range is mutable, the invocable is permitted to change the state of elements but cannot invalidate iterators.

② `std::for_each`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

③ constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_invocable`

④ The C++11 standard introduced the range-based for loop, which mostly replaced the uses of `std::for_each`.

Example of a range loop over all elements of a `std::vector`.

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
for(auto el : data) {
    sum += el;
}
// sum == 45
```

Open in Compiler Explorer

Example of a `std::for_each` loop over all elements of a `std::vector`.

```
std::vector<int> data = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
std::for_each(data.begin(), data.end(), [&sum](int el) {
    sum += el;
    });
// sum == 45
```

Open in Compiler Explorer

However, there are still a few corner cases when using `std::for_each` is preferable.

The first case is straightforward parallelization. Invoking an expensive operation for each element in parallel is trivial with `std::for_each`. As long as the operations are independent, there is no need for synchronization primitives.

Example of a parallel `std::for_each` invoking a method on each element independently in parallel.

```
struct Custom {
    void expensive_operation() {
        // ...
    }
};

std::vector<Custom> data(10);

std::for_each(std::execution::par_unseq,
    data.begin(), data.end(),
    [](Custom& el) {
        el.expensive_operation();
    });
```

Open in Compiler Explorer

Second, the range version can provide a more concise and explicit syntax in some cases because of the projection support introduced in C++20.

Example of the range version of `std::ranges::for_each` utilizing a projection to invoke the method `getValue()` (line 13) on each element and summing the resulting values using a lambda (line 12).

```
1  struct Custom {
2      explicit Custom(double value) : value_(value) {}
3      double getValue() { return value_; }
4  private:
5      double value_;
6  };
7
8  std::vector<Custom> data(10, Custom{1.0});
9
10 double sum = 0;
11 std::ranges::for_each(data,
12     [&sum](auto v) { sum += v; },
13     &Custom::getValue);
14 // sum == 10.0
```

Open in Compiler Explorer

2.1.2 `std::for_each_n`

① The `std::for_each_n` algorithm applies the provided invocable to each element of the range specified using an iterator and the number of elements. If the underlying range is mutable, the invocable is permitted to change the state of elements but cannot invalidate iterators.

② `for_each_n`	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	C++20

③ constraints		
domain	`input_iterator`	
parallel domain	`forward_iterator`	
invocable	default	custom
	N/A	`unary_invocable`

④ While `std::for_each` operates on the entire range, the interval $[begin, end)$, `std::for_each_n` operates on the range $[first, first + n)$. Importantly, because the algorithm does not have access to the end iterator of the source range, it does no out-of-bounds checking, and it is the responsibility of the caller to ensure that the range $[first, first + n)$ is valid.

Example demonstrating multiple uses of `std::for_each_n`.

```
1  std::vector<Player> final_ranking = get_rankings();
2  std::ranges::sort(final_ranking, std::greater<>(), &Player::score);
3
4  std::for_each_n(std::execution::par_unseq,
5      final_ranking.begin(),
6      std::min(MAIN_SEATS, final_ranking.size()),
7      send_invitation_to_main_tournament);
8
9  auto it = final_ranking.begin();
10 uint32_t page = 0;
11 while (it != final_ranking.end()) {
12     size_t cnt = std::min(PAGE_SIZE, size_t(final_ranking.end() -
       ↪  it));
13     std::for_each_n(it, cnt, [page](const Player& p) {
14         store_final_score(page, p.display_name, p.score);
15     });
16     page++;
17     it += cnt;
18 }
```

Open in Compiler Explorer

Sending invitations to the `MAIN_SEATS` top players is done in parallel (lines 4-7). Then all users' scores are stored in chunks of `PAGE_SIZE` records (lines 13-15). Note that calculating the remaining number of elements (line 12) and jumping ahead by the number of stored elements (line 17) requires a random access iterator (in this case, provided by `std::vector`).

2.2 Swaps

Before C++11 and the introduction of move operations, swaps were the only way objects with value semantics could exchange content without involving a deep copy.

2.2.1 `std::swap`

swap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

The non-range version of `std::swap` will swap the values of the two parameters using a three-step move-swap. Users can provide a more optimized implementation as friend functions on their type.

Correctly calling swap requires pulling the default std::swap version to the local scope. To read more on why this is needed check out the theory chapter of this book, specifically the section on ADL (5.1).

Example of correctly calling `std::swap`.

```
1  void some_algorithm(auto& x, auto& y) {
2      using std::swap;
3      swap(x, y);
4  }
```

Open in Compiler Explorer

The C++20 rangified version of swap removes this complexity, and it will:

- call the user-provided (found by ADL) overload of swap
- if that doesn't exist and the parameters are arrays of the same span, `std::ranges::swap` will behave as `std::ranges::swap_ranges`
- if the parameters are also not arrays, it will default to a move-swap

Example of specializing and calling `std::ranges::swap`.

```
1  namespace Library {
2  struct Storage {
3      int value;
4  };
5
6  void swap(Storage& left, Storage& right) {
7      std::ranges::swap(left.value, right.value);
8  }
9  }
10
11 int main() {
12     int a = 1, b = 2;
13     std::ranges::swap(a, b); // 3-step-swap
14
15     Library::Storage j{2}, k{3};
16     std::ranges::swap(j, k); // calls custom Library::swap()
17 }
```

Open in Compiler Explorer

2.2.2 `std::iter_swap`

The `std::iter_swap` is an indirect swap, swapping values behind two forward iterators.

constraints	
domain	(`forward_iterator`, `forward_iterator`) (non-range)

iter_swap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

Example demonstrating the use `std::iter_swap` in a generic two-way partition algorithm. The algorithm uses concepts to constrain the acceptable types of its arguments.

```
template <typename It, typename Cond>
    requires std::forward_iterator<It>
        && std::indirectly_swappable<It,It>
        && std::predicate<Cond, It>
auto partition(It first, It last, Cond cond) {
    while (first != last && cond(first)) ++first;
    if (first == last) return last;

    for (auto it = std::next(first); it != last; it++) {
        if (!cond(it)) continue;

        std::iter_swap(it, first);
        ++first;
    }
    return first;
}
```

Open in Compiler Explorer

The range version extends the functionality to other dereferenceable objects.

Example demonstrating the use of range version of `std::ranges::iter_swap` to swap the values pointed to by two instances of `std::unique_ptr`.

```
auto p1 = std::make_unique<int>(1);
auto p2 = std::make_unique<int>(2);
int* p1_pre = p1.get();
int* p2_pre = p2.get();

std::ranges::iter_swap(p1, p2);
// p1.get() == p1_pre, *p1 == 2
// p2.get() == p2_pre, *p2 == 1
```

Open in Compiler Explorer

2.2.3 `std::swap_ranges`

swap_ranges	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::swap_ranges` algorithm exchanges elements between two non-overlapping ranges (potentially from the same container).

Example of swapping the first three elements of an array with the last three elements using `std::swap_ranges`. Note the reversed order of elements due to the use of `rbegin`.

```
1 std::vector<int> data{ 1, 2, 3, 4, 5, 6, 7, 8, 9};
2 std::swap_ranges(data.begin(), data.begin()+3, data.rbegin());
3 // data = { 9, 8, 7, 4, 5, 6, 3, 2, 1 }
```

Open in Compiler Explorer

2.3 Sorting

Before we talk about sorting, we need to discuss what the standard requires for types to be comparable—specifically, the `strict_weak_ordering` required by sorting algorithms.

Implementing a `strict_weak_ordering` for a custom type at minimum requires providing an overload of `operator<` with the following behaviour:

- irreflexive $\neg f(a, a)$
- anti-symmetric $f(a, b) \Rightarrow \neg f(b, a)$
- transitive $(f(a, b) \wedge f(b, c)) \Rightarrow f(a, c)$

A good default for a `strict_weak_ordering` implementation is lexicographical ordering. Lexicographical ordering is also the ordering provided by standard containers.

Since C++20 introduced the spaceship operator, user-defined types can easily access the default version of lexicographical ordering.

Example of three approaches to implementing lexicographical comparison for a custom type.

```
1  struct Point {
2
3  int x;
4  int y;
5
6  // pre-C++20 lexicographical less-than
7  friend bool operator<(const Point& left, const Point& right) {
8      if (left.x != right.x)
9          return left.x < right.x;
10     return left.y < right.y;
11 }
12
13 // default C++20 spaceship version of lexicographical comparison
14 friend auto operator<=>(const Point&, const Point&) = default;
15
16 // manual version of lexicographical comparison using operator <=>
17 friend auto operator<=>(const Point& left, const Point& right) {
```

```
18      if (left.x != right.x)
19          return left.x <=> right.x;
20      return left.y <=> right.y;
21  }
22
23  };
```

Open in Compiler Explorer

The default lexicographical ordering (line 14) works recursively. It starts with the object's bases first, left-to-right, depth-first and then non-static members in declaration order (processing arrays element by element, left-to-right).

The type returned for the spaceship operator is the common comparison category type for the bases and members, one of:

- `std::strong_ordering`
- `std::weak_ordering`
- `std::partial_ordering`

2.3.1 `std::lexicographical_compare`

lex…compare	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

Lexicographical `strict_weak_ordering` for ranges is exposed through the `std::lexicographical_compare` algorithm.

constraints		
domain	(`input_range`, `input_range`)	
parallel domain	(`forward_range`, `forward_range`)	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `lexicographical_compare` and the built-in less than operator to compare vectors of integers.

```
1  std::vector<int> range1{1, 2, 3};
2  std::vector<int> range2{1, 3};
3  std::vector<int> range3{1, 3, 1};
4
5  bool cmp1 = std::lexicographical_compare(range1.begin(), range1.end(),
6      range2.begin(), range2.end());
7  // same as
8  bool cmp2 = range1 < range2;
9  // cmp1 == cmp2 == true
10
11 bool cmp3 = std::lexicographical_compare(range2.begin(), range2.end(),
12     range3.begin(), range3.end());
```

```cpp
// same as
bool cmp4 = range2 < range3;
// cmp3 == cmp4 == true
```

Open in Compiler Explorer

Because the standard containers already offer a built-in lexicographical comparison, the algorithm mainly finds use for comparing raw C arrays and in cases when we need to specify a custom comparator.

Example of using `lexicographical_compare` for C-style arrays and customizing the comparator.

```cpp
// for demonstration only, prefer std::array
int x[] = {1, 2, 3};
int y[] = {1, 4};

bool cmp1 = std::lexicographical_compare(&x[0], &x[3], &y[0], &y[2]);
// cmp1 == true

std::vector<std::string> names1{"Zod", "Celeste"};
std::vector<std::string> names2{"Adam", "Maria"};

bool cmp2 = std::ranges::lexicographical_compare(names1, names2,
    [](const std::string& left, const std::string& right) {
        return left.length() < right.length();
    });
// different than
bool cmp3 = names1 < names2; // Zod > Adam
// cmp2 == true, cmp3 == false
```

Open in Compiler Explorer

2.3.2 `std::lexicographical_compare_three_way`

The `std::lexicographical_compare_three_way` is the spaceship operator equivalent to `std::lexicographical_compare`. It returns one of:

- `std::strong_ordering`
- `std::weak_ordering`
- `std::partial_ordering`

The type depends on the type returned by the elements' spaceship operator.

lex…three_way	
introduced	C++20
constexpr	C++20
parallel	N/A
rangified	N/A

constraints		
domain	`(input_range, input_range)`	
invocable	default	custom
	`operator<=>`	`strong_ordering`, `weak_ordering`, `partial_ordering`

Example of using `std::lexicographical_compare_three_way`.

```
1 std::vector<int> data1 = { 1, 1, 1 };
2 std::vector<int> data2 = { 1, 2, 3 };
3
4 auto cmp = std::lexicographical_compare_three_way(
5     data1.begin(), data1.end(),
6     data2.begin(), data2.end());
7 // cmp == std::strong_ordering::less
```

Open in Compiler Explorer

2.3.3 `std::sort`

sort	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::sort` algorithm is the canonical $O(n \log n)$ sort (typically implemented as intro-sort).

constraints		
domain	`random_access_range`	
parallel domain	`random_access_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Due to the $O(n \log n)$ complexity guarantee, `std::sort` only operates on `random_access` ranges. Notably, `std::list` offers a method with an approximate $n \log n$ complexity.

Basic example of using `std::sort` and `std::list::sort`.

```
1 std::vector<int> data1 = {9, 1, 8, 2, 7, 3, 6, 4, 5};
2 std::sort(data1.begin(), data1.end());
3 // data1 == {1, 2, 3, 4, 5, 6, 7, 8, 9}
4
5 std::list<int> data2 = {9, 1, 8, 2, 7, 3, 6, 4, 5};
6 // std::sort(data.begin(), data.end()); // doesn't compile
7 data2.sort();
8 // data2 == {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Open in Compiler Explorer

With C++20, we can take advantage of projections to sort by a method or member:

Example of using a projection in conjunction with a range algorithm. The algorithm will sort the elements based on the values obtained by invoking the method `value` on each element.

```
1 struct Account {
2     double value() { return value_; }
3     double value_;
```

```cpp
};

std::vector<Account> accounts{{0.1}, {0.3}, {0.01}, {0.05}};
std::ranges::sort(accounts, std::greater<>{}, &Account::value);
// accounts = { {0.3}, {0.1}, {0.05}, {0.01} }
```

Open in Compiler Explorer

Before C++14, you would have to fully specify the type of the comparator, i.e. `std::greater<double>{}`. The type erased variant `std::greater<>{}` relies on type deduction to determine the parameter types. Projections accept an unary invocable, including pointers to members and member functions.

2.3.4 `std::stable_sort`

The `std::sort` is free to re-arrange equivalent elements, which can be undesirable when re-sorting an already sorted range. The `std::stable_sort` provides the additional guarantee of preserving the relative order of equal elements.

stable_sort	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

constraints		
domain	random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

If additional memory is available, stable_sort remains $O(n \log n)$. However, if it fails to allocate, it will degrade to an $O(n \log n \log n)$ algorithm.

Example of re-sorting a range using `std::stable_sort`, resulting in a guaranteed order of elements.

```cpp
struct Record {
    std::string label;
    int rank;
};

std::vector<Record> data {{"q", 1}, {"f", 1}, {"c", 2},
                          {"a", 1}, {"d", 3}};

std::ranges::stable_sort(data, {}, &Record::label);
std::ranges::stable_sort(data, {}, &Record::rank);
// Guarantted order: a-1, f-1, q-1, c-2, d-3
```

Open in Compiler Explorer

2.3.5 `std::is_sorted`

The `std::is_sorted` algorithm is a linear check returning a boolean denoting whether the ranges elements are in non-descending order.

is_sorted	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::less`	`strict_weak_ordering`

Example of testing a range using `std::is_sorted`.

```
1 std::vector<int> data1 = {1, 2, 3, 4, 5};
2 bool test1 = std::is_sorted(data1.begin(), data1.end());
3 // test1 == true
4
5 std::vector<int> data2 = {5, 4, 3, 2, 1};
6 bool test2 = std::ranges::is_sorted(data2);
7 // test2 == false
8 bool test3 = std::ranges::is_sorted(data2, std::greater<>{});
9 // test3 == true
```

Open in Compiler Explorer

2.3.6 `std::is_sorted_until`

is_sorted_until	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::is_sorted_until` algorithm returns the first out-of-order element in the given range, thus denoting a sorted sub-range.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::less`	`strict_weak_ordering`

Example of testing a range using `std::is_sorted_until`.

```
1 std::vector<int> data {1, 5, 9, 2, 4, 6};
2 auto it = std::is_sorted_until(data.begin(), data.end());
3 // *it == 2
```

Open in Compiler Explorer

Note that because of the behaviour of `std::is_sorted_until`, the following is always true:
`std::is_sorted(r.begin(), std::is_sorted_until(r.begin(), r.end()))`

2.3.7 `std::partial_sort`

partial_sort	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::partial_sort` algorithm reorders the range's elements such that the leading sub-range is in the same order it would when fully sorted. However, the algorithm leaves the rest of the range in an unspecified order.

constraints		
domain	(random_access_range, random_access_iterator)	
parallel domain	(random_access_range, random_access_iterator)	
invocable	default	custom
	operator<	strict_weak_ordering

The benefit of using a partial sort is faster runtime — approximately $O(n * logk)$, where k is the number of elements sorted.

Example of using `std::partial_sort` to sort the first three elements of a range.

```
1  std::vector<int> data{9, 8, 7, 6, 5, 4, 3, 2, 1};
2  std::partial_sort(data.begin(), data.begin()+3, data.end());
3  // data == {1, 2, 3, -unspecified order-}
4
5  std::ranges::partial_sort(data, data.begin()+3, std::greater<>());
6  // data == {9, 8, 7, -unspecified order-}
```

Open in Compiler Explorer

2.3.8 `std::partial_sort_copy`

The `std::partial_sort_copy` algorithm has the same behaviour as `std::partial_sort`; however, it does not operate inline. Instead, the algorithm writes the results to a second range.

partial_sort_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	input_range -> random_access_range	
parallel domain	forward_range -> random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

The consequence of writing output to a second range is that the source range does not have to be mutable nor provide random access.

Example of using `std::partial_sort_copy` to iterate over ten integers read from standard input and storing the top three values in sorted order.

```
1  // input == "0 1 2 3 4 5 6 7 8 9"
2  std::vector<int> top(3);
3
4  auto input = std::istream_iterator<int>(std::cin);
5  auto cnt = std::counted_iterator(input, 10);
6
7  std::ranges::partial_sort_copy(cnt, std::default_sentinel,
8      top.begin(), top.end(),
9      std::greater<>{});
10 // top == { 9, 8, 7 }
```

Open in Compiler Explorer

2.3.9 `qsort` - C standard library

Because the C standard library is part of the C++ standard library, we also have access to `qsort`.

Example of using `qsort` to sort an array of integers.

```cpp
int data[] = {2, 1, 9, -1, 7, -8};
int size = sizeof data / sizeof(int);

qsort(data, size, sizeof(int),
    [](const void* left, const void* right){
        int vl = *(const int*)left;
        int vr = *(const int*)right;

        if (vl < vr) return -1;
        if (vl > vr) return 1;
        return 0;
    });
// data == {-8, -1, 1, 2, 7, 9}
```

Open in Compiler Explorer

I would strongly recommend avoiding `qsort`, as `std::sort` and `std::ranges::sort` should be a better choice in every situation. Moreover, `qsort` is only valid for trivially copyable types, and those will correctly optimize to `memcpy` / `memmove` operations even when using `std::sort`.

Example of using `std::sort` to achieve the same result as in the previous example.

```cpp
int data[] = {2, 1, 9, -1, 7, -8};
int size = sizeof data / sizeof(int);

std::sort(&data[0], &data[size], std::less<>());
// data == {-8, -1, 1, 2, 7, 9}
```

Open in Compiler Explorer

2.4 Partitioning

Partition algorithms rearrange elements in the range based on a predicate, such that elements for which the predicate returns `true` precede elements for which the predicate returns `false`.

Partitioning comes up often when we need to group elements based on a particular property. You can also think of partitioning as equal to sorting if we would sort based on the values of a boolean property.

2.4.1 `std::partition`

The `std::partition` algorithm provides the basic partitioning functionality, reordering elements based on a unary predicate. The algorithm returns the partition point, an iterator to the first element for which the predicate returned `false`.

partition	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::partition` to process exam results.

```cpp
std::vector<ExamResult> results = get_results();

auto pp = std::partition(results.begin(), results.end(),
    [threshold = 49](const auto& r) {
        return r.score >= threshold;
    });

// process passing students
for (auto it = results.begin(); it != pp; ++it) {
    std::cout << "[PASS] " << it->student_name << "\n";
}
// process failed students
for (auto it = pp; it != results.end(); ++it) {
    std::cout << "[FAIL] " << it->student_name << "\n";
}
```

Open in Compiler Explorer

2.4.2 `std::stable_partition`

The `std::partition` algorithm is permitted to rearrange the elements with the only guarantee that elements for which the predicate evaluated to `true` will precede elements for which the predicate evaluated to `false`. This behaviour can be undesirable, for example, for UI elements.

The `std::stable_partition` algorithm adds the guarantee of preserving the relative order of elements in both partitions.

stable_partition	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

constraints		
domain	`bidirectional_range`	
parallel domain	`bidirectional_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::stable_partition` to move selected items to the beginning of a list.

```
auto& widget = get_widget();
std::ranges::stable_partition(widget.items, &Item::is_selected);
```

Open in Compiler Explorer

2.4.3 `std::is_partitioned`

is_partitioned	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::is_partitioned` algorithm is a linear check returning a boolean denoting whether the ranges elements are partitioned in regards to the predicate.

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

Note that a sorted range is always partitioned for any possible value (with a different partition point).

Example of using `std::is_partitioned`.

```
std::vector<int> data{2, 4, 6, 7, 9, 11};

auto is_even = [](int v) { return v % 2 == 0; };
bool test1 = std::ranges::is_partitioned(data, is_even);
// test1 == true

bool test2 = true;
for (int i = 0; i < 16; ++i) {
    test2 = test2 && std::is_partitioned(data.begin(), data.end(),
        [&i](int v) { return v < i; });
}
// test2 == true
```

Open in Compiler Explorer

2.4.4 `std::partition_copy`

partition_copy	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::partition_copy` is a variant of `std::partition` that, instead of reordering elements, will output the partitioned elements to the two output ranges denoted by two iterators.

constraints		
domain	`input_range -> (output_iterator, output_iterator)`	
parallel domain	`forward_range -> (forward_iterator, forward_iterator)`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::partition_copy` to copy even elements into one range and odd elements into another range.

```cpp
std::vector<int> data{2, 4, 6, 1, 3, 5};
auto is_even = [](int v) { return v % 2 == 0; };

std::vector<int> even, odd;
std::partition_copy(data.begin(), data.end(),
    std::back_inserter(even),
    std::back_inserter(odd),
    is_even);

// even == {2, 4, 6}
// odd == {1, 3, 5}
```

Open in Compiler Explorer

2.4.5 `std::nth_element`

The `nth_element` algorithm is a partitioning algorithm that ensures that the element in the nth position is the element that would be in this position if the range was sorted.

The nth element also partitions the range into `[begin, nth)`, `[nth, end)` such that all elements preceding the nth element are less than or equal to the rest of the range. Alternatively, for any element $i \in$ `[begin, nth)` and $j \in$ `[nth, end)` it holds that $\neg(j < i)$.

nth_element	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(random_access_range, random_access_iterator)`	
parallel domain	`(random_access_range, random_access_iterator)`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Because of its selection/partitioning nature, `std::nth_element` offers a better theoretical complexity than `std::partial_sort` - $O(n)$ vs $O(n * logk)$.

However, note that the standard only mandates average $O(n)$ complexity, and `std::nth_element` implementations can have high overhead, so always test to determine which provides better performance for your use case.

Example of using `std::nth_element`.

```cpp
std::vector<int> data{9, 1, 8, 2, 7, 3, 6, 4, 5};
std::nth_element(data.begin(), data.begin() + 4, data.end());
// data[4] == 5, data[0..3] < data[4]

std::nth_element(data.begin(), data.begin() + 7, data.end(),
    std::greater<>());
// data[7] == 2, data[0..6] > data[7]
```

Open in Compiler Explorer

2.5 Divide and conquer

Divide and conquer algorithms offer a great mix of performance and functionality.

While we can utilize hash-based containers to lookup any specific element in $O(1)$ amortized time, this approach has two drawbacks. Firstly, we can only look up a specific element; if that element is not present in the container, we get a simple lookup failure. Secondly, our type must be hashable, and the hash function must be reasonably fast.

Divide and conquer algorithms allow the lookup of bounds based on strict weak ordering and work even when the container's specific value is not present. Additionally, since we are working with a sorted container, we can easily access neighbouring values once we have determined a boundary.

2.5.1 `std::lower_bound`, `std::upper_bound`

`lower_bound`	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

`upper_bound`	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

The `std::lower_bound` and `std::upper_bound` algorithms offer boundary search with logarithmic complexity (for random access ranges).

constraints		
domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

The two algorithms differ in which bound they return:

- the `std::lower_bound` returns the first element for which `elem < value` returns false (i.e. first element for which `elem >= value`)
- the `std::upper_bound` returns the first element for which `value < elem`
- if no such element exists, both algorithms return the end iterator

Example of using `std::lower_bound` and `std::upper_bound` to divide a sorted range into three parts: lower than the bottom threshold, between the bottom and upper threshold and higher than the upper threshold.

```
1  const std::vector<ExamResult>& results = get_results();
2
3  auto lb = std::ranges::lower_bound(results, 49, {},
4                                     &ExamResult::score);
5  // First element for which: it->score >= 49
6  auto ub = std::ranges::upper_bound(results, 99, {},
7                                     &ExamResult::score);
8  // First element for which: 99 < it->score
9
10 for (auto it = results.begin(); it != lb; it++) {
11     // Process exam fails, upto 48
```

```cpp
}

for (auto it = lb; it != ub; it++) {
    // Process exam passes, 49-99
}

for (auto it = ub; it != results.end(); it++) {
    // Process exams with honors, 100+
}
```

Open in Compiler Explorer

While the algorithms will operate on any `forward_range`, the logarithmic divide and conquer behaviour is only available for `random_access_range`. Data structures like `std::set`, `std::multiset`, `std::map` and `std::multimap` offer their $O(\log n)$ implementations of lower and upper bound as methods.

Example of using `lower_bound` and `upper_bound` methods on a `std::multiset`.

```cpp
std::multiset<int> data{1, 2, 3, 4, 5, 6, 6, 6, 7, 8, 9};

auto lb = data.lower_bound(6);
// std::distance(data.begin(), lb) == 5, *lb == 6

auto ub = data.upper_bound(6);
// std::distance(data.begin(), ub) == 8, *ub == 7
```

Open in Compiler Explorer

2.5.2 `std::equal_range`

equal_range	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

The `std::equal_range` algorithm returns both lower and upper bounds for the given value.

constraints		
domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Because the lower bound returns the first element for which `elem >= value` and the upper bound returns the first element for which `value < elem`, the result is a range `[lb, ub)` of elements equal to the value.

Example of using `std::equal_range`.

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

auto [lb, ub] = std::equal_range(data.begin(), data.end(), 6);
```

```
// std::distance(data.begin(), lb) == 5, *lb == 6
// std::distance(data.begin(), ub) == 8, *ub == 7
```

Open in Compiler Explorer

2.5.3 `std::partition_point`

partition_point	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

Despite the naming, `std:partition_point` works very similarly to `std::upper_bound`, however instead of searching for a particular value, it searches using a predicate.

constraints		
domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

`std::partition_point` will return the first element that does not satisfy the provided predicate. This algorithm only requires the range to be partitioned (with respect to the predicate).

Example of using `std::partition_point`.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto pp = std::partition_point(data.begin(), data.end(),
    [](int v) { return v < 5; });
// *pp == 5
```

Open in Compiler Explorer

2.5.4 `std::binary_search`

binary_search	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

The `std::binary_search` provides a presence check, returning a boolean indicating whether the requested value is present in the sorted range or not.

constraints		
domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Using `std::binary_search` is equivalent to calling `std::equal_range` and checking whether the returned is non-empty; however, `std::binary_search` offers a single lookup performance, where `std::equal_range` does two lookups to determine the lower and upper bounds.

Example of using `std::binary_search` with an equivalent check using `std::equal_range`.

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

bool exists = std::ranges::binary_search(data, 5);
// exists == true
auto [lb, ub] = std::ranges::equal_range(data, 5);
// lb != ub, i.e. the value is in the range
```

Open in Compiler Explorer

2.5.5 `bsearch` - C standard library

From the C standard library, C++ inherits `bsearch`. This algorithm returns one of the elements equal to the provided key, or `nullptr` if none such element is found.

Example of using `bsearch`.

```cpp
int data[] = {-2, -1, 0, 1, 2};
int size = sizeof data / sizeof(int);

auto cmp = [](const void* left, const void* right){
    int vl = *(const int*)left;
    int vr = *(const int*)right;

    if (vl < vr) return -1;
    if (vl > vr) return 1;
    return 0;
};

int value = 1;
void* el1 = bsearch(&value, data, size, sizeof(int), cmp);
// *static_cast<int*>(el1) == 1

value = 3;
void *el2 = bsearch(&value, data, size, sizeof(int), cmp);
// el2 == nullptr
```

Open in Compiler Explorer

As with `qsort`, there is effectively no reason to use `bsearch` in C++ code. Depending on the specific use case, one of the previously mentioned algorithms should be a suitable replacement.

Example demonstrating alternatives to `bsearch`.

```
int data[] = {-2, -1, 0, 1, 2};
int size = sizeof data / sizeof(int);

int value = 1;
bool exist = std::binary_search(&data[0], &data[size], value);
// exist == true

auto candidate = std::lower_bound(&data[0], &data[size], value);
if (candidate != &data[size] && *candidate == value) {
    // process element
}

auto [lb, ub] = std::equal_range(&data[0], &data[size], value);
if (lb != ub) {
    // process equal elements
}
```

Open in Compiler Explorer

2.6 Linear operations on sorted ranges

In this section, we will discuss algorithms operating on sorted ranges in linear time. The same functionality on unsorted ranges would require algorithms operating in quadratic time.

2.6.1 `std::includes`

includes	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::includes` algorithm will determine whether one range (all elements) is contained within another range.

constraints		
domain	`(input_range, input_range)`	
parallel domain	`(forward_range, forward_range)`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::includes` to check whether a string contains all English (lower-case) letters.

```
std::vector<char> letters('z'-'a'+1,'\0');
std::iota(letters.begin(), letters.end(), 'a');
std::string input = "the quick brown fox jumps over the lazy dog";
std::ranges::sort(input);

bool test = std::ranges::includes(input, letters);
// test == true
```

Open in Compiler Explorer

The example uses `std::iota` to generate the lower-case letters (line 2), which requires the destination vector to be pre-allocated (line 1).

2.6.2 `std::merge`

merge	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::merge` algorithm merges two sorted ranges, with the output written to a third range which cannot overlap with either of the input ranges.

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

The merge operation is stable. Equal elements from the first range will be ordered before equal elements from the second range.

Example of using `std::merge`.

```
1  struct LabeledValue {
2      int value;
3      std::string label;
4  };
5
6  std::vector<LabeledValue> data1{
7      {1, "first"}, {2, "first"}, {3, "first"}};
8  std::vector<LabeledValue> data2{
9      {0, "second"}, {2, "second"}, {4, "second"}};
10
11 std::vector<LabeledValue> result;
12 auto cmp = [](const auto& left, const auto& right)
13     { return left.value < right.value; };
14
15 std::ranges::merge(data1, data2, std::back_inserter(result), cmp);
16 // result == {0, second}, {1, first}, {2, first},
17 //           {2, second}, {3, first}, {4, second}
```

Open in Compiler Explorer

The parallel version requires the output to be a forward range (represented by a `forward_iterator`). Therefore, we cannot use wrappers like `std::back_inserter` and must preallocate the output range to sufficient capacity.

Example of parallel `std::merge`.

```
1 std::vector<int> data1{1, 2, 3, 4, 5, 6};
2 std::vector<int> data2{3, 4, 5, 6, 7, 8};
3
4 std::vector<int> out(data1.size()+data2.size(), 0);
5 std::merge(std::execution::par_unseq,
6     data1.begin(), data1.end(),
7     data2.begin(), data2.end(),
8     out.begin());
9 // out == {1, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 8}
```

Open in Compiler Explorer

inplace_merge	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

2.6.3 `std::inplace_merge`

The `std::inplace_merge` algorithm merges two consecutive sub-ranges.

constraints		
domain	`(bidirectional_range, bidirectional_iterator)`	
parallel domain	`(bidirectional_range, bidirectional_iterator)`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

When using the iterator-based interface, the middle iterator (i.e. the iterator to the first element of the second sub-range) is the second argument.

Example of using `std::inplace_merge`.

```
1 std::vector<int> range{1, 3, 5, 2, 4, 6};
2 std::inplace_merge(range.begin(), range.begin()+3, range.end());
3 // range == { 1, 2, 3, 4, 5, 6 }
```

Open in Compiler Explorer

2.6.4 `std::unique`, `std::unique_copy`

unique	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::unique` algorithm removes consecutive duplicate values. The typical use case is in conjunction with a sorted range. However, this is not a requirement of `std::unique`.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`binary_predicate`

Because unique works in-place and cannot resize the underlying range, it leaves the end of the range with unspecified values and returns an iterator to the beginning of this sub-range.

Example of using `std::unique`.

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

auto it = std::unique(data.begin(), data.end());
// data == {1, 2, 3, 4, 5, 6, unspec, unspec, unspec, unspec}

data.resize(std::distance(data.begin(), it));
// data == {1, 2, 3, 4, 5, 6}
```

Open in Compiler Explorer

The `std::unique_copy` is a variant of `std::unique` that outputs the unique values to a second range.

unique_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	`operator==`	`binary_predicate`

Example of using `std::unique_copy`.

```cpp
std::vector<int> data{1, 1, 2, 2, 3, 4, 5, 6, 6, 6};
std::vector<int> out;

std::ranges::unique_copy(data, std::back_inserter(out));
// out == {1, 2, 3, 4, 5, 6}
```

Open in Compiler Explorer

2.7 Set operations

The group of set algorithms simulates different set operations on two sorted ranges.

2.7.1 `std::set_difference`

The `std::set_difference` algorithm produces a range containing elements present in the first range but not in the second range.

set_difference	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::set_difference`.

```
std::vector<int> data1{1, 2, 3, 4, 5, 6};
std::vector<int> data2{3, 4, 5};

std::vector<int> difference;
std::ranges::set_difference(data1, data2,
    std::back_inserter(difference));
// difference == {1, 2, 6}
```

Open in Compiler Explorer

For equivalent elements, where the first range contains M such elements and the second range contains N such elements, the result will contain the last `std::max(M-N,0)` such elements from the first range.

Example demonstrating `std::set_difference` behaviour when equivalent elements are present.

```
struct Labeled {
    std::string label;
    int value;
};

auto cmp = [](const auto& l, const auto& r) {
    return l.value < r.value;
};

std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 1},
                            {"first_c", 1}, {"first_d", 1}};
std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 1}};

std::vector<Labeled> equal_difference;
std::ranges::set_difference(equal1, equal2,
    std::back_inserter(equal_difference), cmp);
// equal_difference == { {"first_c", 1}, {"first_d", 1} }
```

Open in Compiler Explorer

2.7.2 `std::set_symmetric_difference`

set_sym…diff…	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::set_symetric_difference` algorithm produces a range containing elements only present in one of the ranges, but not both.

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::set_symmetric_difference`.

```cpp
std::vector<int> data1{1, 3, 5, 7, 9};
std::vector<int> data2{3, 4, 5, 6, 7};

std::vector<int> symmetric_difference;
std::ranges::set_symmetric_difference(data1, data2,
    std::back_inserter(symmetric_difference));
// symmetric_difference == {1, 4, 6, 9}
```

Open in Compiler Explorer

For equivalent elements, where the first range contains M such elements and the second range contains N such elements, the result will contain the last `std::abs(M-N)` such elements from the corresponding range. That is, if $M > N$, $M - N$ elements will be copied from the first range; otherwise, $N - M$ elements will be copied from the second range.

Example demonstrating `std::set_symmetric_difference` behaviour when equivalent elements are present.

```cpp
struct Labeled {
    std::string label;
    int value;
};

auto cmp = [](const auto& l, const auto& r) {
    return l.value < r.value;
};

std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 2},
                            {"first_c", 2}};
std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 1},
                            {"second_c", 2}};

std::vector<Labeled> equal_symmetric_difference;
std::ranges::set_symmetric_difference(equal1, equal2,
    std::back_inserter(equal_symmetric_difference), cmp);
// equal_symmetric_difference == { {"second_b", 1}, {"first_c", 2} }
```

Open in Compiler Explorer

2.7.3 `std::set_union`

The `std::set_union` algorithm produces a range containing elements present in either of the ranges.

set_union	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::set_union`.

```cpp
std::vector<int> data1{1, 3, 5};
std::vector<int> data2{2, 4, 6};

std::vector<int> set_union;
std::ranges::set_union(data1, data2,
    std::back_inserter(set_union));
// set_union == { 1, 2, 3, 4, 5, 6 }
```

Open in Compiler Explorer

For equivalent elements, where the first range contains M such elements and the second range contains N such elements, the result will contain M elements from the first range, followed by the last `std::max(N-M,0)` elements from the second range.

Example demonstrating `std::set_union` behaviour when equivalent elements are present.

```cpp
struct Labeled {
    std::string label;
    int value;
};

auto cmp = [](const auto& l, const auto& r) {
    return l.value < r.value;
};

std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 1},
                            {"first_c", 2}};
std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 2},
                            {"second_c", 2}};

std::vector<Labeled> equal_union;
std::ranges::set_union(equal1, equal2,
    std::back_inserter(equal_union), cmp);
// equal_union == { {"first_a", 1}, {"first_b", 1},
//                  {"first_c", 2}, {"second_c", 2} }
```

Open in Compiler Explorer

2.7.4 `std::set_intersection`

set_intersection	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::set_intersection` algorithm produces a range containing elements present in both of the ranges.

constraints		
domain	`(input_range, input_range) -> output_iterator`	
parallel domain	`(forward_range, forward_range) -> forward_iterator`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of using `std::set_intersection`.

```
std::vector<int> data1{1, 2, 3, 4, 5};
std::vector<int> data2{2, 4, 6};

std::vector<int> intersection;
std::ranges::set_intersection(data1, data2,
    std::back_inserter(intersection));
// intersection == {2, 4}
```

Open in Compiler Explorer

For equivalent elements, where the first range contains M such elements and the second range contains N such elements, the result will contain the first `std::min(M,N)` elements from the first range.

Example demonstrating `std::set_intersection` behaviour when equivalent elements are present.

```
struct Labeled {
    std::string label;
    int value;
};

auto cmp = [](const auto& l, const auto& r) {
    return l.value < r.value;
};

std::vector<Labeled> equal1{{"first_a", 1}, {"first_b", 2}};
std::vector<Labeled> equal2{{"second_a", 1}, {"second_b", 2},
                            {"second_c", 2}};

std::vector<Labeled> intersection;
std::ranges::set_intersection(equal1, equal2,
    std::back_inserter(intersection), cmp);
// intersection == { {"first_a", 1}, {"first_b", 2} }
```

Open in Compiler Explorer

2.8 Transformation algorithms

In this section, we will discuss algorithms that transform ranges by changing the values of elements and removing and re-ordering elements.

2.8.1 `std::transform`

transform	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The most straightforward transformation possible is to apply a transformation function to each element. The `std::transform` algorithm provides this functionality in unary and binary variants (input from one or two ranges).

constraints		
domain	`input_range -> output_iterator` `(input_range, input_iterator) -> output_iterator`	
parallel domain	`forward_range -> forward_iterator` `(forward_range, forward_iterator) -> forward_iterator`	
invocable	default	custom
	N/A	`unary_functor` `binary_functor`

Example of unary and binary version of `std::transform`. Note that the output iterator can be one of the input ranges' begin iterator (line 4 and 12).

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8};
2
3  std::transform(data.begin(), data.end(),
4      data.begin(),
5      [](int v) { return v*2; });
6  // data == {2, 4, 6, 8, 10, 12, 14, 16}
7
8  std::vector<int> add{8, 7, 6, 5, 4, 3, 2, 1};
9
10 std::transform(data.begin(), data.end(),
11     add.begin(),
12     data.begin(),
13     [](int left, int right) { return left+right; });
14 // data == {10, 11, 12, 13, 14, 15, 16, 17}
```

Open in Compiler Explorer

Note that `std::transform` does not guarantee strict left-to-right evaluation. If that is required, use `std::for_each` instead.

2.8.2 `std::remove`, `std::remove_if`

remove, remove_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::remove` and `std::remove_if` algorithms "remove" elements that match the given value or for which the given predicate evaluates to true.

Because the algorithms cannot resize the underlying range, the removal is achieved by moving the other elements in the range. The algorithms then return an iterator beyond the last not removed element, i.e. the new end iterator.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::remove` and `std::remove_if`.

```
1  std::vector<int> data{1, 2, 3, 4, 5};
2
3  auto it = std::remove(data.begin(), data.end(), 3);
4  // data == { 1, 2, 4, 5, ?}
5
6  data.erase(it, data.end()); // Erase sub-range
7  // data == {1, 2, 4, 5}
8
9  auto is_even = [](int v) { return v % 2 == 0; };
10 it = std::remove_if(data.begin(), data.end(), is_even);
11 // data == {1, 5, ?, ?}
12
13 data.resize(it - data.begin()); // Random Access Ranges only
14 // data = {1, 5}
```

Open in Compiler Explorer

2.8.3 `std::replace`, `std::replace_if`

replace, replace_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::replace` and `std::replace_if` algorithms replace elements that match the given value or for which the given predicate evaluates to true.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::replace` and `std::replace_if`.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
2
3  std::ranges::replace(data, 4, 0);
4  // data == {1, 2, 3, 0, 5, 6, 7}
5
6  auto is_odd = [](int v) { return v % 2 != 0; };
7  std::ranges::replace_if(data, is_odd, -1);
8  // data == {-1, 2, -1, 0, -1, 6, -1}
```

Open in Compiler Explorer

2.8.4 `std::reverse`

reverse	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::reverse` algorithm will reverse the order of elements in the range by applying `std::swap` to pairs of elements.

constraints	
domain	`bidirectional_range`
parallel domain	`bidirectional_range`

Note that using `std::reverse` is only a reasonable solution if the range must be mutated because bidirectional ranges already support reverse iteration.

Example of using `std::reverse` and reverse iteration, provided by bidirectional ranges.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
2
3  std::reverse(data.begin(), data.end());
4  // data == {7, 6, 5, 4, 3, 2, 1}
5
6  for (auto it = data.rbegin(); it != data.rend(); ++it) {
7      // iterate over: 1, 2, 3, 4, 5, 6, 7
8  }
```

Open in Compiler Explorer

C-style arrays and C-style strings can be adapted using `std::span` and `std::string_view` to allow reverse iteration.

Example of using `std::span` and `std::string_view` to addapt C-style constructs for reverse iteration.

```
1  int c_array[] = {1, 2, 3, 4, 5, 6, 7};
2  auto arr_view = std::span(c_array, sizeof(c_array)/sizeof(int));
3
4  for (auto it = arr_view.rbegin(); it != arr_view.rend(); ++it) {
5      // iterate over: {7, 6, 5, 4, 3, 2, 1}
6  }
7
8  const char* c_string = "No lemon, no melon";
9  auto str_view = std::string_view(c_string);
10
11 for (auto it = str_view.rbegin(); it != str_view.rend(); ++it) {
12     // iterate over: "nolem on ,nomel oN"
13 }
```

Open in Compiler Explorer

2.8.5 `std::rotate`

rotate	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::rotate` algorithm rearranges elements in the range from `[first, middle)`, `[middle, last)` to `[middle, last)`, `[first, middle)`.

constraints	
domain	`(forward_range, forward_iterator)`
parallel domain	`(forward_range, forward_iterator)`

Example of using `std::rotate`.

```
1 std::vector<int> data{1, 2, 3, 4, 5, 6, 7};
2 std::rotate(data.begin(), data.begin()+3, data.end());
3 // data == {4, 5, 6, 7, 1, 2, 3}
```

Open in Compiler Explorer

2.8.6 `std::shift_left`, `std::shift_right`

The `std::shift_left` and `std::shift_right` algorithms move elements in the provided range by the specified amount of positions. However, unlike std::rotate, the shift doesn't wrap around.

shift_left	
introduced	C++20
constexpr	C++20
parallel	C++20
rangified	C++20

shift_right	
introduced	C++20
constexpr	C++20
parallel	C++20
rangified	C++20

constraints	
domain	`forward_range`
parallel domain	`forward_range`

Example of using `std::shift_left` and `std::shift_right` with a trivially copyable type.

```
1 std::vector<int> data{1,2,3,4,5,6,7,8,9};
2 std::shift_left(data.begin(), data.end(), 3);
3 // data == {4, 5, 6, 7, 8, 9, 7, 8, 9}
4
5 data = {1,2,3,4,5,6,7,8,9};
6 std::shift_right(data.begin(), data.end(), 3);
7 // data == {1, 2, 3, 1, 2, 3, 4, 5, 6}
```

Open in Compiler Explorer

One way to think about these algorithms is that they "make space" for the requested number of new elements.

Example of using `std::shift_right` to make space for four new elements. Note that `'d'` wasn't moved as there is no place to move it to, and in the context of "making space" will be overwritten anyway.

```
1  struct EmptyOnMove {
2      char value;
3      EmptyOnMove(char value) : value(value) {}
4      EmptyOnMove(EmptyOnMove&& src) :
5          value(std::exchange(src.value,'-')) {}
6      EmptyOnMove& operator=(EmptyOnMove&& src) {
7          value = std::exchange(src.value, '-');
8          return *this;
9      }
10     EmptyOnMove(const EmptyOnMove&) = default;
11     EmptyOnMove& operator=(const EmptyOnMove&) = default;
```

```
};

int main() {
    std::vector<EmptyOnMove> nontrivial{
        {'a'},{'b'},{'c'},{'d'},{'e'},{'f'},{'g'}};

    std::shift_right(nontrivial.begin(), nontrivial.end(), 4);
    // nontrivial == { '-', '-', '-', 'd', 'a', 'b', 'c' }
}
```

Open in Compiler Explorer

2.8.7 `std::shuffle`

shuffle	
introduced	C++11
constexpr	N/A
parallel	N/A
rangified	C++20

The `std::shuffle` algorithm is a successor of the now-defunct (deprecated in C++14, removed in C++17) `std::random_shuffle` algorithm and relies on new random facilities added in C++11.

constraints	
domain	random_access_range

Example of using `std::shuffle` to shuffle a deck of cards.

```
struct Card {
    unsigned index;

friend std::ostream& operator << (std::ostream& s, const Card& card) {
    static constexpr std::array<const char*, 13> ranks =
        {"Ace", "Two", "Three", "Four", "Five", "Six", "Seven",
         "Eight", "Nine", "Ten", "Jack", "Queen", "King"};
    static constexpr std::array<const char*, 4> suits =
        {"Hearts", "Diamonds", "Clubs", "Spades"};

    if (card.index >= 52)
        throw std::domain_error(
            "Card index has to be in the range 0..51");

    s << ranks[card.index%13] << " of " << suits[card.index/13];
    return s;
}

};

int main() {

std::vector<Card> deck(52, Card{});
std::ranges::generate(deck, [i = 0u]() mutable { return Card{i++}; });
// deck == {Ace of Hearts, Two of Hearts, Three of Hearts, Four...}

std::random_device rd;
```

```
28 std::mt19937 gen{rd()};
29
30 std::ranges::shuffle(deck, gen);
31 // deck == { random order }
32
33 }
```

Open in Compiler Explorer

2.8.8 `std::next_permutation`, `std::prev_permutation`

The `std::next_permutation` and `std::prev_permutation` algorithms will rearrange the element so that they are in their next or previous (by lexicographical comparison) permutation. When there is no such ordering, both algorithms will wrap around, but will also return `false` to notify the caller.

next_permutation	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

prev_permutation	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

constraints		
domain	`bidirectional_range`	
invocable	default	custom
	`operator <`	`strict_weak_ordering`

Example of using `std::next_permutation` to iterate over all permutations of three unique elements.

```
1 std::vector<int> data{1, 2, 3};
2 do {
3     // iterate over:
4     // 1, 2, 3
5     // 1, 3, 2
6     // 2, 1, 3
7     // 2, 3, 1
8     // 3, 1, 2
9     // 3, 2, 1
10 } while (std::next_permutation(data.begin(), data.end()));
11 // data == {1, 2, 3}
```

Open in Compiler Explorer

2.8.9 `std::is_permutation`

The `std::is_permutation` algorithm is an excellent tool for checking whether two ranges have the same content but not necessarily the same order of elements.

is_permutation	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`(forward_range, forward_range)`	
parallel domain	`(forward_range, forward_range)`	
invocable	default	custom
	`operator==`	`binary_predicate`

Example of using `std::is_permutation` to validate a simple sort implementation.

```
std::vector<int> data = { 8, 1, 7, 3, 4, 6, 2, 5};
for (size_t i = 0; i < data.size()-1; ++i)
    for (size_t j = i+1; j < data.size(); ++j)
        if (data[i] > data[j])
            std::swap(data[i], data[j]);

bool is_sorted = std::ranges::is_sorted(data);
// is_sorted == true

bool is_permutation = std::ranges::is_permutation(data,
    std::vector<int>{1, 2, 3, 4, 5, 6, 7, 8});
// is_permutation == true
```

Open in Compiler Explorer

2.9 Left folds

Left folds are one of the two big groups of numerical algorithms. Folds operate in a strict order, "folding in" one element at a time by evaluating `acc=fold_op(acc,el)` for each element. For left folds, the direction of operation is left to right.

Because of the strictly linear operation, none of the left-fold algorithms supports a parallel version.

When using numerical algorithms, it's worth understanding the behaviour of numerical types in C++, notably the interactions between silent implicit conversions and template type deduction rules. You can read more on this in the theory chapter in the section 5.2.

accumulate	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	N/A

2.9.1 `std::accumulate`

The `std::accumulate` algorithm is the single-range left-fold.

constraints		
domain	`input_range`	
invocable	default	custom
	`operator +`	`binary_functor`

Example of using `std::accumulate`.

```
std::vector<int> data{1, 2, 3, 4, 5};
auto sum = std::accumulate(data.begin(), data.end(), 0);
// sum == 15

auto product = std::accumulate(data.begin(), data.end(), 1,
                               std::multiplies<>{});
// product == 120
```

Open in Compiler Explorer

While left folds operate strictly left-to-right, we can mitigate this limitation by utilizing reverse iterators—note, of course, that this requires at least a bidirectional range.

Example of using `std::accumulate` as a right fold.

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

auto left_fold = std::accumulate(data.begin(), data.end(), 0,
    [](int acc, int el) {
        return acc / 2 + el;
    });
// left_fold == 8

auto right_fold = std::accumulate(data.rbegin(), data.rend(), 0,
    [](int acc, int el) {
        return acc / 2 + el;
    });
// right_fold == 3
```

Open in Compiler Explorer

2.9.2 `std::inner_product`

inner_product	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	N/A

The `std::inner_product` algorithm is a left fold over two ranges. The pairs of elements are first reduced and then accumulated.

constraints		
domain	`(input_range, input_iterator)`	
invocable	default	custom
	`operator *, operator +`	`(binary_functor, binary_functor)`

The default version uses `operator*` for the reduction and `operator+` for the fold operation.

Example of using `std::inner_product`.

```cpp
std::vector<int> heights{1, 2, 3, 4, 5};
std::vector<int> widths{2, 3, 4, 5, 6};

auto total_area = std::inner_product(heights.begin(), heights.end(),
    widths.begin(), 0);
// total_area == 70
```

Open in Compiler Explorer

Because the algorithm only uses the ranges as input, there is no issue with overlapping ranges.

Example of using `std::inner_product` on a single range to calculate sum of absolute differences between elements.

```
1  std::vector<int> data{6, 4, 3, 7, 2, 1};
2  auto sum_of_diffs = std::inner_product(
3      data.begin(), std::prev(data.end()),
4      std::next(data.begin()), 0,
5      std::plus<>{},
6      [](int left, int right) { return std::abs(left-right); }
7  );
8  // sum_of_diffs == 13
```

Open in Compiler Explorer

2.9.3 `std::partial_sum`

partial_sum	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	N/A

The `std::partial_sum` algorithm operates as a left fold. However, it doesn't reduce the range into a single value. Instead, the algorithm writes each partial result to the output range.

constraints		
domain	`input_range -> output_iterator`	
invocable	default	custom
	`operator+`	`binary_functor`

The output iterator is permitted to be the input ranges' begin iterator.

Example of using `std::partial_sum`.

```
1  std::vector<int> data(6, 1);
2  // data == {1, 1, 1, 1, 1, 1}
3
4  std::partial_sum(data.begin(), data.end(), data.begin());
5  // data == {1, 2, 3, 4, 5, 6}
6
7  std::vector<int> out;
8  std::partial_sum(data.begin(), data.end(),
9      std::back_inserter(out), std::multiplies<>{});
10 // out == {1, 2, 6, 24, 120, 720}
```

Open in Compiler Explorer

2.9.4 `std::adjacent_difference`

The `std::adjacent_difference` is a numerical algorithm, which is the odd one out as it provides both a strict left-to-right variant, but at the same time also supports parallel execution, where it behaves like a generalized reduction (we will talk about those in the next section).

The algorithm operates similarly to the `std::transform` algorithm, operating on each pair of consecutive elements in the range. However, unlike `std::transform`, it does guarantee left-to-right execution.

The algorithm also supports input ranges, which is achieved by internally storing the copy of the last right operand to be reused as the left operand for the subsequent reduction step.

adjacent_difference	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	N/A

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	`operator -`	`binary_functor`

Example of the default version of `std::adjacent_difference`, which will calculate the difference of adjacent elements, with the first element copied.

```
1 std::vector<int> data{2, 3, 5, 7, 11, 13};
2 std::adjacent_difference(data.begin(), data.end(), data.begin());
3 // data == {2, 1, 2, 2, 4, 2}
```

Open in Compiler Explorer

The left-to-right operation can be exploited for generative use cases.

Example of more inventive use of `std::adjacent_difference` to generate the Fibonacci sequence.

```
1 std::vector<int> data(10, 1);
2 // data == {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
3
4 std::adjacent_difference(data.begin(), std::prev(data.end()),
5     std::next(data.begin()), std::plus<int>());
6 // Fibonacci sequence:
7 // data == {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Open in Compiler Explorer

For parallel overloads, the input and output ranges cannot overlap.

Example of the parallel `std::adjacent_difference` overload.

```
1 std::vector<int> data{2, 3, 5, 7, 11, 13};
2 std::vector<int> out(data.size());
3
4 std::adjacent_difference(std::execution::par_unseq,
5     data.begin(), data.end(), out.begin());
6 // out == {2, 1, 2, 2, 4, 2}
```

Open in Compiler Explorer

2.10 General reductions

Left folds offer a great guarantee of strict left-to-right evaluation, allowing high flexibility for the fold operation.

However, when we work with operations that are associative $op(a, op(b, c)) == op(op(a, b), c)$ and commutative $op(a, b) == op(b, a)$, it really doesn't matter what permutation of elements and order of operations we evaluate, we will always arrive at the same result.

This is why with the parallel support in C++17, we also received a batch of generalised reduction algorithms that reduce elements in unspecified order and permutation.

2.10.1 `std::reduce`

`reduce`	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

The `std::reduce` algorithm is a generalised version of `std::accumulate`. That is, it reduces a range by applying the provided accumulation operation to the elements in unspecified order and permutation.

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`std::plus<>()`	`binary_functor`

Note that while we have access to a sequenced execution policy (i.e. `std::execution::seq`), this does not make `std::reduce` sequenced in a left-fold sense.

Example of using `std::reduce` with and without an execution policy.

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

auto sum = std::reduce(data.begin(), data.end(), 0);
// sum == 15

sum = std::reduce(std::execution::par_unseq,
    data.begin(), data.end(), 0);
// sum == 15

auto product = std::reduce(data.begin(), data.end(), 1,
    std::multiplies<>{});
// product == 120

product = std::reduce(std::execution::par_unseq,
    data.begin(), data.end(), 1, std::multiplies<>{});
// product == 120
```

Open in Compiler Explorer

On top of the `std::accumulate` equivalents, we get one more overload that does away with the initial accumulator value, which removes the potential for using the wrong literal. Instead, the accumulator will be of the type of the ranges' elements and will be value initialised.

Example of using `std::reduce` without specifying the initial value of the accumulator.

```
1  struct Duck {
2      std::string sound = "Quack";
3      Duck operator+(const Duck& right) const {
4          return {sound+right.sound};
5      }
6  };
7
8  std::vector<Duck> data(2, Duck{});
9  Duck final_duck = std::reduce(data.begin(), data.end());
10 // final_duck.sound == "QuackQuackQuack"
```

Open in Compiler Explorer

The initial value of the accumulator will be "Quack" (line 2). Adding the other two ducks (line 8), we end up with "QuackQuackQuack".

2.10.2 `std::transform_reduce`

The `std::transform_reduce` algorithm is the generalised counterpart to `std::inner_product`. On top of the two-range variant, the algorithm also provides a unary overload.

transform_reduce	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`(binary_functor, unary_functor)`

constraints		
domain	`(input_range, input_iterator)`	
parallel domain	`(forward_range, forward_iterator)`	
invocable	default	custom
	`(std::plus<>(), std::multiplies<>())`	`(binary_functor, binary_functor)`

Example of using the unary version of `std::transform_reduce` to calculate a sum of squares and the binary version to calculate a sum of elements multiplied by coefficients.

```
1  std::vector<int> data{1, 2, 3, 4, 5};
2  auto sum_of_squares = std::transform_reduce(data.begin(), data.end(),
3      0, std::plus<>{}, [](int v) { return v*v; });
```

```cpp
// sum_of_squares == 55

std::vector<int> coef{1, -1, 1, -1, 1};
auto result = std::transform_reduce(data.begin(), data.end(),
    coef.begin(), 0);
// result == 1*1 + 2*(-1) + 3*1 + 4*(-1) + 5*1 == 3
```

Open in Compiler Explorer

2.10.3 `std::inclusive_scan`, `std::exclusive_scan`

The `std::inclusive_scan` is a generalised version of `std::partial_sum`. On top of that, we also have access to `std::exclusive_scan`.

For `std::inclusive_scan`, the nth generated element is the sum of the first n source elements. For `std::exclusive_scan`, the nth element is the sum of the first n-1 source elements.

inclusive_scan	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

exclusive_scan	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	`std::plus<>()`	`binary_functor`

Example of using `std::inclusive_scan`.

```cpp
std::vector<int> src{1, 2, 3, 4, 5, 6};
std::vector<int> out;

std::inclusive_scan(src.begin(), src.end(),
    std::back_inserter(out));
// out == {1, 3, 6, 10, 15, 21}

std::inclusive_scan(src.begin(), src.end(),
    out.begin(), std::multiplies<>{}, 1);
// out == {1, 2, 6, 24, 120, 720}
```

Open in Compiler Explorer

Consequently, because the first element generated by `std::exclusive_scan` is the sum of zero elements, we must specify an initial value of the accumulator, which will be the value of the first generated element.

Example of using `std::exclusive_scan`.

```
std::vector<int> src{1, 2, 3, 4, 5, 6};
std::vector<int> out;

std::exclusive_scan(src.begin(), src.end(),
    std::back_inserter(out), 0);
// out == {0, 1, 3, 6, 10, 15}

std::exclusive_scan(src.begin(), src.end(),
    out.begin(), 1, std::multiplies<>{});
// out == {1, 1, 2, 6, 24, 120}
```

Open in Compiler Explorer

2.10.4 `std::transform_inclusive_scan`, `std::transform_exclusive_scan`

The `std::transform_inclusive_scan` and `std::transform_exclusive_scan` algorithms are variants of `std::inclusive_scan` and `std::exclusive_scan` that apply a unary transformation function to each element before the reduction operation.

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	N/A	`(binary_functor, unary_functor)`

`transform_inclu…`	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

`transform_exclu…`	
introduced	C++17
constexpr	C++20
parallel	C++17
rangified	N/A

Example of using `std::transform_inclusive_scan` to accumulate absolute values of elements.

```
std::vector<int> data{-10, 3, -2, 5, 6};

std::vector<int> out1;
std::inclusive_scan(data.begin(), data.end(),
    std::back_inserter(out1), std::plus<>{});
// out1 == {-10, -7, -9, -4, 2}

std::vector<int> out2;
std::transform_inclusive_scan(data.begin(), data.end(),
    std::back_inserter(out2), std::plus<>{},
    [](int v) { return std::abs(v); });
// out2 == {10, 13, 15, 20, 26}
```

Open in Compiler Explorer

2.11 Boolean reductions

When reducing boolean expressions, we can take advantage of the early termination offered by boolean logic. The standard offers a set of three boolean reduction algorithms.

2.11.1 `std::all_of`, `std::any_of`, `std::none_of`

all_of	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

any_of	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

none_of	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The algorithms either require the elements to be convertible to bool or a predicate to be specified.

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`unary_predicate`

The algorithms follow their naming in their behaviour. The `std::all_of` algorithm only returns true if no elements that evaluate to false are present. The `std::any_of` algorithm returns true if at least one element evaluates to true. Finally, `std::none_of` only returns true if no elements that evaluate true are present.

elements / algorithm	all `true`	all `false`	mixed	empty
`all_of`	`true`	`false`	`false`	`true`
`any_of`	`true`	`false`	`true`	`false`
`none_of`	`false`	`true`	`false`	`true`

Example demonstrating all three boolean reduction algorithms.

```cpp
std::vector<int> data{-2, 0, 2, 4, 6, 8};

bool all_even = std::ranges::all_of(data,
    [](int v) { return v % 2 == 0; });
// all_even == true

bool one_negative = std::ranges::any_of(data,
    [](int v) { return std::signbit(v); });
// one_negative == true

bool none_odd = std::ranges::none_of(data,
    [](int v) { return v % 2 != 0; });
// none_odd == true
```

Open in Compiler Explorer

2.12 Generators

The C++ standard offers three types of generators: fill with copies of a value, fill with results of invoking a generator functor and fill with sequentially increasing values.

2.12.1 `std::fill`, `std::generate`

The `std::fill` algorithm fills a range by consecutively assigning the given value to each element. The `std::generate` algorithm fills a range by consecutively assigning the result of the provided generator.

`fill`, `generate`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	N/A	`generator`

The generator provided to `std::generate` can be a non-regular function since `std::generate` guarantees strict left-to-right evaluation.

Example of using `std::fill` and `std::generate`.

```cpp
std::vector<int> data(5, 0);
// data == {0, 0, 0, 0, 0}

std::fill(data.begin(), data.end(), 11);
// data == {11, 11, 11, 11, 11}

std::ranges::generate(data, []() { return 5; });
// data == {5, 5, 5, 5, 5}

// iota-like
std::ranges::generate(data, [i = 0]() mutable { return i++; });
// data == {0, 1, 2, 3, 4}
```

Open in Compiler Explorer

2.12.2 `std::fill_n`, `std::generate_n`

The `std::fill_n` and `std::generate_n` are variants of `std::fill` and `std::generate` that operate on ranges specified using the start iterator and number of elements. This behaviour allows the algorithms to be used with iterator adapters, such as `std::back_inserter`.

`fill_n`, `generate_n`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

constraints		
domain	`output_iterator`	
parallel domain	`forward_iterator`	
invocable	default	custom
	N/A	`generator`

Example of using `std::fill_n` and `std::generate_n`.

```
1 std::vector<int> data;
2 // data == {}
3
4 std::fill_n(std::back_inserter(data), 5, 11);
5 // data == {11, 11, 11, 11, 11}
6
7 std::ranges::generate_n(std::back_inserter(data), 5,
8     []() { return 7; });
9 // data == {11, 11, 11, 11, 11, 7, 7, 7, 7, 7}
```

Open in Compiler Explorer

2.12.3 `std::iota`

iota	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++23

The `std::iota` generates elements by consecutively assigning the result of applying the prefix `operator++`, starting with the initial value.

constraints	
domain	`forward_range`

Example of using `std::iota`.

```
1 std::vector<int> data(9, 0);
2 // data == {0, 0, 0, 0, 0, 0, 0, 0, 0}
3
4 std::iota(data.begin(), data.end(), -4);
5 // data == {-4, -3, -2, -1, 0, 1, 2, 3, 4}
```

Open in Compiler Explorer

Notably, the `std::iota` algorithm is also an outlier in the support added with the C++20 standard. The `std::iota` algorithm did not receive a range version. However, we do have access to an iota view.

Example of using both finite and infinite `std::views::iota`.

```
1 std::vector<int> data;
2
3 std::ranges::transform(std::views::iota(1, 10), std::views::iota(5),
4     std::back_inserter(data), std::plus<>{});
5 // data == { 6, 8, 10, 12, 14, 16, 18, 20, 22 }
```

Open in Compiler Explorer

Here we take advantage of the finite view constructor `std::views::iota(1,10)` to establish the output size (line 3), which allows us to use the infinite view `std::views::iota(5)` for the second parameter. Functionally, we could swap even the second

view for a finite one. However, this would impose an additional (and unnecessary) boundary check.

2.13 Copy and move

The standard offers a wide range of copy algorithms in roughly three categories: simple copies and moves, selective copies and copies with reordering.

2.13.1 `std::copy`, `std::move`

The `std::copy` and `std::move` algorithms provide a forward copy and move. The direction is important for overlapping ranges, so we do not overwrite the yet-to-be copied elements.

For the forward direction, the output iterator is not permitted to be within `[first, last)` (of the input range). Consequently, only the tail of the output range can overlap with the input range.

copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

move	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

constraints	
domain	`input_range -> output_iterator`
parallel domain	`forward_range -> forward_iterator`

Example of a non-overlapping and permitted overlapping case of `std::copy`.

```cpp
std::vector<std::string> data{ "a", "b", "c", "d", "e", "f"};

std::copy(data.begin(), data.begin()+3, data.begin()+3);
// data = { "a", "b", "c", "a", "b", "c" }

// Overlapping case:
std::copy(std::next(data.begin()), data.end(), data.begin());
// data = { "b", "c", "a", "b", "c", "c" }
```

Open in Compiler Explorer

Move operates identically, except it casts each element to an rvalue before the assignment, turning copies into moves.

Example of using `std::move`.

```cpp
std::vector<std::string> data{ "a", "b", "c", "d", "e", "f"};

std::move(data.begin(), data.begin()+3, data.begin()+3);
// data = { ?, ?, ?, "a", "b", "c" }
// Note: most implementations will set
//       a moved-out-of std::string to empty.
```

Open in Compiler Explorer

Significantly, whether std::move will move depends on the underlying element type. If the underlying type is copy-only, std::move will behave identically to std::copy.

Example of using `std::move` with a copy-only type.

```cpp
struct CopyOnly {
    CopyOnly() = default;
    CopyOnly(const CopyOnly&) = default;
    CopyOnly& operator=(const CopyOnly&) {
        std::cout << "Copy assignment.\n";
        return *this;
    };
};

std::vector<CopyOnly> test(6);

std::move(test.begin(), test.begin()+3, test.begin()+3);
// 3x Copy assignment
```

Open in Compiler Explorer

2.13.2 `std::copy_backward`, `std::move_backward`

copy_backward	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

move_backward	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

The `std::copy_backward` and `std::move_backward` are variants that copy in the opposite direction, starting at the back of the range. Because of this, the head of the output range can now overlap with the input range.

constraints	
domain	`bidirectional_range -> bidirectional_iterator`

The output iterator cannot be within `(first, last]` and will be treated as the end iterator for the destination range, meaning that the algorithm will write the first value to `std::prev(end)`.

Example of a non-overlapping and permitted overlapping case of `std::copy_backward`.

```cpp
std::vector<std::string> data{ "a", "b", "c", "d", "e", "f"};
std::vector<std::string> out(9, "");
// out == {"", "", "", "", "", "", "", "", ""}

std::copy_backward(data.begin(), data.end(), out.end());
// out == {"", "", "", "a", "b", "c", "d", "e", "f"}

std::copy_backward(data.begin(), std::prev(data.end()), data.end());
// data == { "a", "a", "b", "c", "d", "e" }
```

Open in Compiler Explorer

2.13.3 `std::copy_n`

copy_n	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::copy_n` algorithm is the counted variant of `std::copy` that accepts an input range specified using an iterator and the number of elements.

constraints	
domain	`input_iterator -> output_iterator`
parallel domain	`forward_iterator -> forward_iterator`

The algorithm cannot check whether the requested count is valid and does not go out of bounds, so this burden is on the caller.

Example of using `std::copy_n`.

```
1 std::list<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
2 std::vector<int> out;
3
4 std::copy_n(data.begin(), 5, std::back_inserter(out));
5 // out == { 1, 2, 3, 4, 5 }
```

Open in Compiler Explorer

2.13.4 `std::copy_if`, `std::remove_copy`, `std::remove_copy_if`

copy_if	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

remove_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

remove_copy_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::copy_if`, `std::remove_copy` and `std::remove_copy_if` are selective copy algorithms.

constraints		
domain	`input_range -> output_range`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	N/A	`unary_predicate`

The `std::remove_copy` algorithm will copy elements that do not match the provided value. The `std::copy_if` and `std::remove_copy_if` algorithms will copy elements based on a predicate, with `std::copy_if` copying elements for which the predicate returns true and `std::remove_copy_if` copying elements for which the predicate returns false.

Example demonstrating differences between `std::copy_if`, `std::remove_copy` and `std::remove_copy_if`.

```
1 std::vector<int> data{ 1, 2, 3, 4, 5, 6, 7, 8, 9};
2 std::vector<int> even, odd, no_five;
3
4 auto is_even = [](int v) { return v % 2 == 0; };
5
6 std::ranges::copy_if(data, std::back_inserter(even), is_even);
7 // even == { 2, 4, 6, 8 }
```

```
 8
 9  std::ranges::remove_copy_if(data, std::back_inserter(odd), is_even);
10  // odd == { 1, 3, 5, 7, 9 }
11
12  std::ranges::remove_copy(data, std::back_inserter(no_five), 5);
13  // no_five == { 1, 2, 3, 4, 6, 7, 8, 9 }
```

Open in Compiler Explorer

2.13.5 `std::sample`

sample	
introduced	C++17
constexpr	N/A
parallel	N/A
rangified	C++20

The `std::sample` algorithm is a random selective copy algorithm. The algorithm will copy a random selection of N elements from the source range to the destination range utilising the provided random number generator.

constraints	
domain	`forward_range -> output_iterator` `input_range -> random_access_iterator`

The two domains of this algorithm are due to the stable nature of the sampling, maintaining the order of elements from the source range. This feature requires either the input range to be at least a forward range or the destination range needs to be a random-access range.

Example of using `std::sample`.

```
1  std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
2  std::vector<int> out;
3
4  std::sample(data.begin(), data.end(), std::back_inserter(out),
5      5, std::mt19937{std::random_device{}()});
6  // e.g. out == {2, 3, 4, 5, 9}
7  // guaranteed ascending, because source range is ascending
```

Open in Compiler Explorer

2.13.6 `std::replace_copy`, `std::replace_copy_if`

replace_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

replace_copy_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::replace_copy` and `std::replace_copy_if` algorithms operate like `std::copy`; however, they will copy a provided value instead of specific elements.

For `std::replace_copy`, the algorithm will replace elements matching a value for `std::replace_copy_if` the algorithm will replace elements for which the predicate evaluates to true.

constraints		
domain	`input_range -> output_iterator`	
parallel domain	`forward_range -> forward_iterator`	
invocable	default	custom
	N/A	`unary_predicate`

Example of using `std::replace_copy` and `std::replace_copy_if`.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> out, odd;

std::ranges::replace_copy(data, std::back_inserter(out), 5, 10);
// out == { 1, 2, 3, 4, 10, 6, 7, 8, 9 }

auto is_even = [](int v) { return v % 2 == 0; };
std::ranges::replace_copy_if(data, std::back_inserter(odd),
                             is_even, -1);
// odd == { 1, -1, 3, -1, 5, -1, 7, -1, 9 }
```

Open in Compiler Explorer

2.13.7 `std::reverse_copy`

reverse_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::reverse_copy` algorithm copies elements in reverse order.

constraints	
domain	`bidirectional_range -> output_iterator`
parallel domain	`bidirectional_range -> forward_iterator`

Not to be confused with the `std::copy_backwards`, which copies elements in the original order. The `std::reverse_copy` does not permit the source, and the destination ranges to overlap.

Example of using `std::reverse_copy`.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<int> out;

std::ranges::reverse_copy(data, std::back_inserter(out));
// out == { 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

Open in Compiler Explorer

2.13.8 `std::rotate_copy`

rotate_copy	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::rotate_copy` algorithm will copy elements `[middle, last)`, followed by `[first, middle)`, which mirrors the behaviour of the `std::rotate` algorithm.

constraints	
domain	`(forward_range, forward_iterator) -> output_iterator`
parallel domain	`(forward_range, forward_iterator) -> forward_iterator`

The input and output ranges cannot overlap.

Example of using `std::rotate_copy`.

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

std::ranges::rotate_copy(data, data.begin() + 4,
                         std::back_inserter(out));
// out == { 5, 6, 7, 8, 9, 1, 2, 3, 4 }
```

Open in Compiler Explorer

2.14 Uninitialized memory algorithms

The uninitialized memory algorithms are a group of relatively low-level algorithms designed to help when implementing manual memory management. These algorithms provide the functionality to transactionally construct, copy, move and destroy sequences of elements on top of raw memory.

The counted variants of the algorithms are not listed in this section[1]. However, note that all algorithms in this section that operate on ranges have a counted variant, where the range is specified using an iterator and number of elements. These variants are used in some of the examples to demonstrate.

2.14.1 `std::construct_at`, `std::destroy_at`

construct_at	
introduced	C++20
constexpr	C++20
parallel	N/A
rangified	C++20

destroy_at	
introduced	C++17
constexpr	C++20
parallel	N/A
rangified	C++20

The `std::construct_at` and `std::destroy_at` algorithms will construct/destroy a single element at a given address. If additional arguments are specified, `std::construct_at` will forward these to the objects' constructor.

Example of using `std::create_at` to create a `std::string` object using the arguments eight and 'X', which results in a string filled with eight copies of the X character.

```cpp
alignas(alignof(std::string)) char mem[sizeof(std::string)];
auto *ptr = reinterpret_cast<std::string*>(mem);

std::construct_at(ptr, 8, 'X');
// *ptr == "XXXXXXXX", ptr->length() == 8
std::destroy_at(ptr);
```

Open in Compiler Explorer

[1] The names of these algorithms are particularly long and obnoxious.

2.14.2 `std::uninitialized_default_construct`, `std::uninitialized_value_construct`, `std::uninitialized_fill`, `std::destroy`

The three uninitialized algorithms cover the default initialization, value initialization and copy initialization of elements. The `std::destroy` algorithm provides the destruction of elements without deallocating the underlying memory.

constraints	
domain	`forward_range` `forward_iterator (counted)`

un…default_construct	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

un…value_construct	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

uninitialized_fill	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

destroy	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

Example demonstrating the use of the counted variants of the uninitialized construction algorithm and the `std::destroy_n` algorithms. Note that for `std::string`, there is no difference between default and value construction.

```
1  alignas(alignof(std::string)) char buffer[sizeof(std::string)*10];
2  auto *begin = reinterpret_cast<std::string*>(buffer);
3  auto *it = begin;
4
5  it = std::uninitialized_default_construct_n(it, 3);
6  it = std::uninitialized_fill_n(it, 2, "Hello World!");
7  it = std::uninitialized_value_construct_n(it, 3);
8  it = std::uninitialized_fill_n(it, 2, "Bye World!");
9
10 // {"", "", "", "Hello World!", "Hello World!", "", "", "",
11 //  "Bye World!", "Bye World!"}
12
13 std::destroy_n(begin, 10);
```

Open in Compiler Explorer

2.14.3 `std::uninitialized_copy`, `std::uninitalized_move`

The `std::unitialized_copy` and `std::unitialized_move` algorithms follow the behaviour of `std::copy` and `std::move` algorithms with the distinction that the destination range is uninitialized memory.

constraints	
domain	`input_range -> forward_iterator` `input_iterator -> forward_iterator (counted)`
parallel domain	`forward_range -> forward_iterator` `forward_iterator -> forward_iterator (counted)`

uninitialized_copy	
introduced	C++98
constexpr	N/A
parallel	C++17
rangified	C++20

uninitialized_move	
introduced	C++17
constexpr	N/A
parallel	C++17
rangified	C++20

Example of using `std::uninitialized_copy` and `std::uninitialized_move`.

```cpp
alignas(alignof(std::string)) char buff1[sizeof(std::string)*5];
alignas(alignof(std::string)) char buff2[sizeof(std::string)*5];
std::vector<std::string> data = {
    "hello", "world", "and", "everyone", "else"};

auto *bg1 = reinterpret_cast<std::string*>(buff1);
std::uninitialized_copy(data.begin(), data.end(), bg1);
// buff1 == { "hello", "world", "and", "everyone", "else"}
// data == { "hello", "world", "and", "everyone", "else"}
std::destroy_n(bg1, 5);

auto *bg2 = reinterpret_cast<std::string*>(buff2);
std::uninitialized_move(data.begin(), data.end(), bg2);
// buff2 == { "hello", "world", "and", "everyone", "else"}
// data == { ?, ?, ?, ?, ?}
// In most implementations a moved-out-of string will be empty.
std::destroy_n(bg2, 5);
```

Open in Compiler Explorer

Transactional behaviour

The main benefit of using the uninitialized memory algorithms is that they correctly handle transactional behaviour. Transactionality is important in cases where the constructor of an object can throw. If one of the objects fails to construct, the algorithms will correctly roll back by destructing already constructed objects.

Example demonstrating the roll-back behaviour of uninitialized algorithms when the third invocation of the constructor throws. Note that the exception is re-thrown after the partial work is rolled back.

```cpp
struct Custom {
    static int cnt;
    Custom() {
        if (++cnt >= 3)
            throw std::runtime_error("Deliberate failure.");
        std::cout << "Custom()\n";
    }
    ~Custom() {
        std::cout << "~Custom()\n";
    }
};

int Custom::cnt = 0;

int main() {
    alignas(alignof(Custom)) char buffer[sizeof(Custom)*10];
```

```
17      auto *begin = reinterpret_cast<Custom*>(buffer);
18
19      try {
20          std::uninitialized_default_construct_n(begin, 10);
21          std::destroy_n(begin, 10); // not reached
22      } catch (std::exception& e) {
23          std::cout << e.what() << "\n";
24      }
25  }
26  /* OUTPUT:
27      Custom()
28      Custom()
29      ~Custom()
30      ~Custom()
31      Deliberate failure.
32  */
```

Open in Compiler Explorer

2.15 Heap data structure

The standard offers a convenient wrapper for a max-heap data structure through `std::priority_queue`. However, when using `std::priority_queue`, we lose access to the underlying data, which might be inconvenient.

2.15.1 `std::make_heap`, `std::push_heap`, `std::pop_heap`

A Heap data structure is a binary tree where each element satisfies the heap property: the value at the parent is greater or equal to the value of its children.

When discussing the heap algorithms, we will be referring to an array representation of the heap, where the children of the element at index i are elements at indexes $2i + 1$ and $2i + 2$.

The valuable property of a heap is that it can be constructed in linear time and then provide logarithmic complexity for extracting the maximum value and inserting new values into the heap.

constraints		
domain	`random_access_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

`make_heap`	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

`push_heap`, `pop_heap`	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

The `std::make_heap` algorithm reorders elements in the given range such that the elements maintain the max-heap property, that is, the element at index i compares greater or equal to the elements at indexes $2i + 1$ and $2i + 2$.

Example of using `std::make_heap` to construct a max-heap and a min-heap (using a custom comparator).

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

std::make_heap(data.begin(), data.end());
// data == {9, 8, 7, 4, 5, 6, 3, 2, 1} - different ordering possible
// 9 >= 8, 9 >= 7
// 8 >= 4, 8 >= 5
// 7 >= 6, 7 >= 3
// ...

std::make_heap(data.begin(), data.end(), std::greater<>{});
// data == {1, 2, 3, 4, 5, 6, 7, 8, 9} - different ordering possible
// 1 <= 2, 1 <= 3
// 2 <= 4, 2 <= 5
// 3 <= 6, 3 <= 7
// ...
```

Open in Compiler Explorer

The `std::push_heap` and `std::pop_heap` algorithms simulate push and pop operations for the max-heap data structure. However, because they operate on top of a range, they cannot manipulate the underlying data structure. Therefore, they use the last element of the range as the input/output.

The `std::push_heap` algorithm will insert the last element of the range into the heap, and `std::pop_heap` will extract the maximum element to the last position of the range. As previously mentioned, both operations have logarithmic complexity.

Example of using `std::push_heap` and `std::pop_heap`.

```cpp
std::vector<int> data = { 1, 1, 2, 2 };
std::make_heap(data.begin(), data.end());
// data == { 2, 2, 1, 1} - different ordering possible

// Push 9 to the heap
data.push_back(9);
// data == { [heap_part], 9 }
std::push_heap(data.begin(), data.end());
// data == { 9, 2, 1, 1, 2 } - different ordering possible

// Push 7 to the heap
data.push_back(7);
// data == { [heap_part], 7 }
std::push_heap(data.begin(), data.end());
// data == { 9, 2, 7, 1, 2, 1 } - different ordering possible

std::pop_heap(data.begin(), data.end());
```

```
// data == { [heap_part], 9 }
std::pop_heap(data.begin(), std::prev(data.end()));
// data == { [heap_part], 7, 9 }
```

Open in Compiler Explorer

2.15.2 `std::sort_heap`

The `std::sort_heap` will reorder the elements in a heap into a sorted order. Note that this is the same as repeatedly calling `std::pop_heap`; hence the algorithm has $O(n \log n)$ complexity.

sort_heap	
introduced	C++98
constexpr	C++20
parallel	N/A
rangified	C++20

constraints		
domain	random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

Example of using `std::sort_heap`.

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

std::make_heap(data.begin(), data.end());
// data == {9, 8, 7, 4, 5, 6, 3, 2, 1} - different ordering possible

std::sort_heap(data.begin(), data.end());
// data == {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Open in Compiler Explorer

2.15.3 `std::is_heap`, `std::is_heap_until`

The `std::is_heap` and `std::is_heap_until` algorithms check the heap invariant.

is_heap	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

is_heap_until	
introduced	C++11
constexpr	C++20
parallel	N/A
rangified	C++20

constraints		
domain	random_access_range	
invocable	default	custom
	operator<	strict_weak_ordering

The two algorithms follow the same logic as `std::is_sorted` and `std::is_sorted_until`, returning a boolean and an iterator to the first out-of-order element respectively.

Example of using `std::is_heap` and `std::is_heap_until`.

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

bool test1 = std::is_heap(data.begin(), data.end());
```

```
// test1 == false
auto it1 = std::is_heap_until(data.begin(), data.end());
// *it1 == 2

std::make_heap(data.begin(), data.end());

bool test2 = std::is_heap(data.begin(), data.end());
// test2 == true
auto it2 = std::is_heap_until(data.begin(), data.end());
// it2 == data.end()
```

Open in Compiler Explorer

2.15.4 Comparison with `std::priority_queue`

As mentioned at the beginning of this section, the heap algorithms provide effectively the same functionality as `std::priority_queue`. To demonstrate the differences, let's look at two versions of a topk algorithm: an algorithm that takes a range and returns the top k elements in sorted order as a `std::vector`.

Example of implementing a topk algorithm using a `std::priority_queue`.

```
auto topk_queue(
    std::input_iterator auto begin,
    std::sentinel_for<decltype(begin)> auto end,
    size_t k) {

    using vtype = std::iter_value_t<decltype(begin)>;
    using arrtype = std::vector<vtype>;

    std::priority_queue<vtype, arrtype, std::greater<vtype>> pq;

    while (begin != end) {
        pq.push(*begin);
        if (pq.size() > k)
            pq.pop();
        ++begin;
    }

    arrtype result(k);
    for (auto &el: result | std::views::reverse) {
        el = std::move(pq.top());
        pq.pop();
    }

    return result;
}

```

Open in Compiler Explorer

The example relies on C++20 concepts to provide a generic interface and should therefore work with any range on input, returning a `std::vector` of the same element type.

When using the priority queue, we can utilize the simple `push()` and `pop()` interface provided (lines 12 and 14). However, extracting all data from the queue is only possible by repeatedly applying `pop()` until the queue is empty (line 20).

Example of implementing a topk algorithm using heap algorithms.

```
 1 auto topk_heap(
 2     std::input_iterator auto begin,
 3     std::sentinel_for<decltype(begin)> auto end,
 4     size_t k) {
 5
 6     std::vector<std::iter_value_t<decltype(begin)>> result;
 7
 8     while (begin != end) {
 9         result.push_back(*begin);
10         std::ranges::push_heap(result, std::greater<>{});
11
12         if (result.size() > k) {
13             std::ranges::pop_heap(result, std::greater<>{});
14             result.pop_back();
15         }
16
17         ++begin;
18     }
19
20     std::ranges::sort_heap(result, std::greater<>{});
21     return result;
22 }
```

Open in Compiler Explorer

When using the heap algorithms, we need to manually manage the underlying data structure (lines 9-10 and 13–14). However, we do not need to extract the data, and on top of that, we could omit the final `std::sort_heap` (line 20) if we do not need the top k elements in sorted order.

2.16 Search and compare algorithms

The search and compare category provides straightforward linear (when compared against a single value) and quadratic (when compared against a range) complexity algorithms.

find, find_if	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

find_if_not	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

2.16.1 `std::find`, `std::find_if`, `std::find_if_not`

The std::find algorithm provides a basic linear search. The standard provides three variants, one searching by value and two variants using a predicate.

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==` (`find`)	`unary_predicate`

Example of utilizing `std::find` to find delimiters in a string.

```
1  std::string data = "John;Doe;April;1;1900;";
2  auto it = data.begin(), token = data.begin();
3  std::vector<std::string> out;
4
5  while ((token = find(it, data.end(), ';')) != data.end()) {
6      out.push_back("");
7      std::copy(it, token, std::back_inserter(out.back()));
8      it = std::next(token);
9  }
10 // out == { "John", "Doe", "April", "1", "1900" }
```

Open in Compiler Explorer

If we want to search for categories of elements, we can use `std::find_if` and `std::find_if_not` since these two variants search using a predicate.

Example of utilizing `std::find_if_not` to find leading and trailing whitespace.

```
1  std::string data = "   hello world!  ";
2
3  auto begin = std::find_if_not(data.begin(), data.end(),
4      [](char c) { return isspace(c); });
5  if (begin == data.end()) // only spaces
6      return 0;
7
8  std::string out;
9  std::copy(begin,
10     std::find_if_not(data.rbegin(), data.rend(),
11         [](char c) { return isspace(c); }
12     ).base(),
13     std::back_inserter(out));
14 // out == "hello world!"
```

Open in Compiler Explorer

2.16.2 `std::adjacent_find`

adjacent_find	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::adjacent_find` is a binary find algorithm that searches for pairs of adjacent elements in a single range.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`binary_predicate`

If the algorithm finds a pair of elements, it will return an iterator to the first of the two elements (end iterator otherwise).

Example of using `std::adjacent_find` to find the first pair of equal elements and the first pair of elements that sum up to more than ten.

```
1 std::vector<int> data = { 1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 9 };
2 auto it1 = std::adjacent_find(data.begin(), data.end());
3 // *it1 == 4, i.e. {4, 4}
4
5 auto it2 = std::adjacent_find(data.begin(), data.end(),
6     [](int l, int r) { return l + r > 10; });
7 // *it2 == 5, i.e. {5, 6}
```

Open in Compiler Explorer

2.16.3 `std::search_n`

search_n	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::search_n` algorithm searches for n instances of the given value.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`binary_predicate`

The interface to `std::search_n` can be a bit confusing. The algorithm accepts the number of instances and the value to search for as two consecutive arguments, followed by an optional custom comparator function.

Example of using `std::search_n` to find two consecutive elements equal to 3, three elements equal to 3 (in modulo 5 arithmetic) and finally, two elements equal to 0.

```
1 std::vector<int> data = { 1, 0, 5, 8, 3, 3, 2 };
2
3 auto it1 = std::search_n(data.begin(), data.end(), 2, 3);
4 // *it1 == 3, i.e. {3, 3}
5
```

```cpp
auto it2 = std::search_n(data.begin(), data.end(), 3, 3,
    [](int l, int r) { return l % 5 == r % 5; });
// *it2 == 8, i.e. {8, 3, 3}

auto it3 = std::search_n(data.begin(), data.end(), 2, 0);
// it3 == data.end(), i.e. not found
```

Open in Compiler Explorer

Note that `std::search_n` is one exception to the _n naming scheme.

2.16.4 `std::find_first_of`

find_first_of	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

Using `std::find_if`, we can easily search for a category of elements. However, sometimes it is more convenient to list the elements we are looking for exhaustively.

constraints		
domain	`(input_range, forward_range)`	
parallel domain	`(forward_range, forward_range)`	
invocable	default	custom
	`operator==`	`binary_predicate`

Note that we are shifting from linear search to $O(m * n)$ time complexity since, for each element of the first range, we need to compare it to all elements in the second range (worst case).

Example of using `std::find_first_of`.

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

auto it = std::find_first_of(haystack.begin(), haystack.end(),
    needles.begin(), needles.end());
// *it == 3, i.e. haystack[2]
```

Open in Compiler Explorer

2.16.5 `std::search`, `std::find_end`

search, find_end	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

Both `std::search` and `std::find_end` algorithms search for a sub-sequence in a sequence. The `std::search` algorithm will return the first instance, and `std::find_end` will return the last.

constraints		
domain	`(forward_range, forward_range)`	
parallel domain	`(forward_range, forward_range)`	
invocable	default	custom
	`operator==`	`binary_predicate`

Example of using `std::search` and `std::find_end`.

```
std::string haystack = "abbabba";
std::string needle = "bba";

auto it1 = std::search(haystack.begin(), haystack.end(),
    needle.begin(), needle.end());
// it1..end == "bbabba"

auto it2 = std::find_end(haystack.begin(), haystack.end(),
    needle.begin(), needle.end());
// it2..end == "bba"
```

Open in Compiler Explorer

Searchers

Since C++17, we also can specify custom searchers for the search algorithm. Apart from the basic one, the standard implements Boyer-Moore and Boyer-Moore-Horspool string searchers that offer different best-case, worst-case and average complexity.

Example of using `std::search` with custom searchers.

```
std::string haystack = "abbabba";
std::string needle = "bba";

auto it1 = std::search(haystack.begin(), haystack.end(),
    std::default_searcher(needle.begin(), needle.end()));

auto it2 = std::search(haystack.begin(), haystack.end(),
    std::boyer_moore_searcher(needle.begin(), needle.end()));

auto it3 = std::search(haystack.begin(), haystack.end(),
    std::boyer_moore_horspool_searcher(needle.begin(), needle.end()));
// it1 == it2 == it3
```

Open in Compiler Explorer

2.16.6 `std::count`, `std::count_if`

`count, count_if`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::count` and `std::count_if` algorithms count the number of matching elements.

constraints		
domain	`input_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator==`	`unary_predicate`

The element searched for can be specified using a value (`std::count`) or a predicate (`std::count_if`).

Example of using `std::count` and `std::count_if`.

```
1  std::vector<int> data = { 1, 2, 3, 2, 1, 2, 3, 2, 1 };
2
3  auto one_cnt = std::count(data.begin(), data.end(), 1);
4  // one_cnt == 3
5
6  auto even_cnt = std::count_if(data.begin(), data.end(),
7      [](int v) { return v % 2 == 0; });
8  // even_cnt == 4
```

Open in Compiler Explorer

equal	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

2.16.7 `std::equal`, `std::mismatch`

The `std::equal` algorithm provides an equality comparison for ranges.

constraints		
domain	(input_range, input_iterator)	
parallel domain	(forward_range, forward_iterator)	
invocable	default	custom
	operator==	binary_predicate

Example of using `std::equal`.

```
1  std::vector<int> first = { 1, 2, 3, 4, 5 };
2  std::vector<int> second = { -1, -2, -3, -4, -5 };
3
4  bool test1 = std::equal(first.begin(), first.end(), second.begin());
5  // test1 == false
6
7  bool test2 = std::equal(first.begin(), first.end(), second.begin(),
8      [](int l, int r) { return std::abs(l) == std::abs(r); });
9  // test2 == true
```

Open in Compiler Explorer

mismatch	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

The `std::mismatch` algorithm behaves exactly like `std::equal`; however, instead of returning a simple boolean, it returns a pair of iterators denoting the mismatched elements.

constraints		
domain	(input_range, input_iterator)	
parallel domain	(forward_range, forward_iterator)	
invocable	default	custom
	operator==	binary_predicate

Example of using `std::mismatch`.

```
1  std::vector<int> first = { 1, 2, 3, 4, 5 };
2  std::vector<int> second = { 1, 2, 2, 4, 5 };
3
4  auto it_pair = std::mismatch(first.begin(), first.end(),
5                               second.begin());
6  // *it_pair.first == 3, *it_pair.second == 2
```

Open in Compiler Explorer

On top of the basic variants, both `std::equal` and `std::mismatch` offer a version where both ranges are fully specified (since C++14). These versions can detect mismatches beyond the first range's scope.

constraints		
domain	`(input_range, input_range)` since C++14	
parallel domain	(forward_range, forward_range)	
invocable	default	custom
	`operator==`	`binary_predicate`

Example of detecting a mismatch beyond the scope of the first range.

```
1  std::vector<int> first = { 1, 2, 3, 4, 5 };
2  std::vector<int> second = { 1, 2, 3, 4, 5, 6 };
3
4  bool test1 = std::equal(first.begin(), first.end(), second.begin());
5  // test1 == true, cannot detect mismatch in number of elements
6
7  bool test2 = std::equal(first.begin(), first.end(),
8      second.begin(), second.end());
9  // test2 == false, different number of elements -> not equal.
10
11 auto pair_it = std::mismatch(first.begin(), first.end(),
12     second.begin(), second.end());
13 // pair_it.first == first.end()
14 // *pair_it.second == 6
```

Open in Compiler Explorer

2.17 Min-Max algorithms

This group contains algorithms that operate with minimums and maximums. However, two theoretical topics are relevant here, and we need to talk about them first: `std::initializer_list` and `const_cast`.

For functions that accept an `std::initializer_list`, it is worth keeping in mind that `std::initializer_list` is constructed by copy; its internal array of

elements is copy constructed from the listed elements. Therefore we need to be careful when using `std::initializer_list` outside of compile-time contexts.

Example demonstrating case when utilizing `std::initializer_list` leads to excessive copies.

```cpp
struct X {
    static int copy_cnt;
    X(int v) : value(v) {}
    X(const X& other) : value(other.value) {
        ++copy_cnt;
    }
    int value;
    friend auto operator <=>(const X&, const X&) = default;
};

int X::copy_cnt = 0;

void example() {
    X a{1}, b{2}, c{3}, d{4}, e{5};
    auto max = std::max({a, b, c, d, e});
    // max.value == 5
    // X::copy_cnt == 6
}
```

Open in Compiler Explorer

In this example, using `std::initializer_list` leads to six copies (which we count using the static data member `X::copy_cnt`). Five copies result from passing in the variables a to e into the `std::initializer_list`, and one is the result of the return from `std::max`.

In rare cases, we can force constness on a mutable entity. If the constness is undesirable, using `const_cast` to cast away the const is an option.

Example demonstrating the valid and invalid uses for `const_cast`.

```cpp
int x = 10, y = 20;

auto& v = const_cast<int&>(std::min(x, y));
v = 5;
// x == 5, y == 20

// !IMPORTANT! the following compiles, but is undefined behaviour
// i.e. the program is ill-formed
const int z = 3;
auto& w = const_cast<int&>(std::min(x, z));
w = 10;
```

Open in Compiler Explorer

Please remember that when using casts like `const_cast`, you effectively override the compiler's judgment. Therefore it is entirely up to you to ensure that the given cast is valid.

2.17.1 `std::min`, `std::max`, `std::minmax`

The basic versions of `std::min`, `std::max` and `std::minmax` operate on two elements, accepting their arguments by const-reference and returning by const-reference. Unfortunately, as mentioned earlier, this creates a constness problem, and we also must be careful to capture the result by value when passing in temporary objects.

min, max	
introduced	C++98
constexpr	C++14
parallel	N/A
rangified	C++20

minmax	
introduced	C++11
constexpr	C++14
parallel	N/A
rangified	C++20

Example demonstrating use of `std::min` and `std::max`.

```
int x = 10, y = 20;
int min = std::min(x, y);
// min == 10

std::string hello = "hello", world = "world";
std::string& universe =
    const_cast<std::string&>(std::max(hello, world));
universe = "universe";

std::string greeting = hello + " " + world;
// greeting == "hello universe"

int j = 20;
auto& k = std::max(5, j);
// IMPORTANT! only works because 5 < j
// would produce dangling reference otherwise
// k == 20
```

Open in Compiler Explorer

Capturing the result by value gets a bit more complicated with `std::minmax`, which returns a `std::pair` of const references. To avoid dangling references to expired prvalues we must explicitly name the result type. Unfortunately, there is no way to work around this problem when using `auto` or structured binding.

Example demonstrating use of `std::minmax`.

```
struct X {
    int rank;
    auto operator <=> (const X&) const = default;
};

X a{1}, b{2};
auto [first, second] = std::minmax(b, a);
// first.rank == 1, second.rank == 2
```

```cpp

// Operating on prvalues requires capture by value
std::pair<int,int> result = std::minmax(5, 10);
// result.first = 5, result.second = 10
```

Open in Compiler Explorer

The C++14 and C++20 standards introduced additional variants of the min-max algorithms that return by value. The change to return by value resolves the dangling reference issue while simultaneously introducing the potential for excessive copies.

constraints		
domain	C++14: `initializer_list` C++20 range version: `input_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

Example of `std::initializer_list` and range variants of `std::min`, `std::max` and `std::minmax`.

```cpp
auto min = std::min({5, 3, -2, 0});
// min == -2

auto minmax = std::minmax({5, 3, -2, 0});
// minmax.first == -2, minmax.second == 5

std::list<int> data{5, 3, -2, 0};
auto max = std::ranges::max(data);
// max == 5
```

Open in Compiler Explorer

2.17.2 `std::clamp`

clamp	
introduced	C++17
constexpr	C++17
parallel	N/A
rangified	C++20

The `std::clamp` algorithm takes three arguments, the value, the minimum and the maximum bound and will clamp the value between the provided minimum and maximum:

- if $value < minimum$, `std::clamp` returns the minimum
- if $maximum < value$, `std::clamp` returns the maximum
- otherwise, `std::clamp` returns the value

Because the algorithm accepts its arguments by const reference and returns a const reference, it shares the same issues as the min-max algorithms regarding const correctness and dangling references.

Examples of using `std::clamp` with prvalues and with lvalues and `const_cast` to get mutable access to the original variable.

```cpp
int a = std::ranges::clamp(10, 0, 20);
// a == 10 (0 < 10 && 10 < 20)

int b = std::clamp(-20, 0, 20);
// b == 0 (-20 < 0)

int c = std::clamp(30, 0, 20);
// c == 20 ( 30 > 20 )

int x = 10, y = 20, z = 30;
int &w = const_cast<int&>(std::clamp(z, x, y));
w = 99;
// x == 10, y == 99, z == 30
```

Open in Compiler Explorer

2.17.3 `std::min_element`, `std::max_element`, `std::minmax_element`

The element versions of min-max algorithms operate on ranges and, instead of returning by const-reference or value, return an iterator to the minimum or maximum elements.

constraints		
domain	`forward_range`	
parallel domain	`forward_range`	
invocable	default	custom
	`operator<`	`strict_weak_ordering`

`min_element`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`max_element`	
introduced	C++98
constexpr	C++20
parallel	C++17
rangified	C++20

`minmax_element`	
introduced	C++11
constexpr	C++20
parallel	C++17
rangified	C++20

Example of using element versions of min-max algorithms.

```cpp
std::vector<int> data = { 5, 3, -2 , 0};
auto i = std::min_element(data.begin(), data.end());
// *i == -2 (i.e. data[2])
auto j = std::max_element(data.begin(), data.end());
// *j == 5 (i.e. data[0])

auto k = std::minmax_element(data.begin(), data.end());
// *k.first == -2, *k.second == 5
```

Open in Compiler Explorer

You might be wondering whether we can cause the same dangling reference (iterator) issue with the element versions of min-max algorithms. Fortunately, one very nice feature of C++20 ranges is the protection against precisely this problem.

Example demonstrating protection from dangling iterators.

```cpp
auto i = std::ranges::min_element(std::vector<int>{5, 3, -2, 0});
// decltype(i) == std::ranges::dangling

std::vector<int> data = { 5, 3, -2 , 0};
auto j = std::ranges::min_element(std::span(data.begin(), 2));
// *j == 3, std::span is a borrowed_range
```

Open in Compiler Explorer

All ranged versions of algorithms that return iterators will return the `std::ranges::dangling` type when invoked on a temporary range. This would preclude the use case of using `std::span` to sub-reference a range, which is why the range algorithms have an additional concept of a `borrowed_range`. Such ranges can be passed in as temporaries since they do not own their elements.

Chapter 3

Introduction to Ranges

The C++20 standard introduced the Ranges library (a.k.a. STLv2), which effectively replaces existing algorithms and facilities. In this chapter, we will go over the main changes.

Note that Ranges are one of the features that landed in C++20 in a partial state. Despite C++23 introducing plenty of additional features, we already know that many features will still not make it into C++23.

3.1 Reliance on concepts

The standard has always described what is required of each argument passed to an algorithm. However, this description was purely informative, and the language did not offer first-class tools to enforce these requirements. Because of this, the error messages were often confusing.

Concepts are a new C++20 language feature that permits library implementors to constrain the arguments of generic code. Concepts are beyond the scope of this book; however, there are two consequences worth noting.

First, the definitions of all concepts are now part of the standard library, e.g. you can look up what exactly it means for a type to be a `random_access_range` and also use these concepts to constrain your code.

Example of using standard concepts in user code. The function accepts any random access range as the first argument and an output iterator with the same underlying type as the second argument.

```
template <std::ranges::random_access_range T>
auto my_function(T&& rng,
    std::output_iterator<std::ranges::range_value_t<T>> auto it) {
    if (rng.size() >= 5)
        *it++ = rng[4];
    if (rng.size() >= 7)
        *it++ = rng[6];
}
```

Open in Compiler Explorer

Second, error messages now reference unsatisfied constraints instead of reporting an error deep in the library implementation.

```
note: candidate: 'template<class _Iter, class _Sent, class _Comp, class _Proj>
    requires (random_access_iterator<_Iter>)
```

3.2 Notion of a Range

Non-range algorithms have always operated strictly using iterators. Conceptually, the range of elements has been defined by two iterators `[first, last)` or when working with all elements from a standard data structure `[begin, end)`.

Range algorithms formally define a range as an iterator and a sentinel pair. Moving the sentinel to a different type unlocks the potential to simplify code where an end iterator doesn't naturally map to an element. The standard offers two default sentinel types, `std::default_sentinel` and `std::unreachable_sentinel`.

Apart from simplifying certain use cases, sentinels also allow for infinite ranges and potential performance improvements.

Example of an infinite range when the data guarantees termination. Using `std::unreachable_sentinel` causes the boundary check to be optimized-out, removing one comparison from the loop.

```
std::vector<int> dt = { 1, 2, 3, 4, 5, 6, 7, 8, 9};
std::ranges::shuffle(dt, std::mt19937(std::random_device()()));

auto pos = std::ranges::find(
    dt.begin(),
    std::unreachable_sentinel,
    7);
```

Open in Compiler Explorer

We can only use this approach when we have a contextual guarantee that the

algorithm will terminate without going out of bounds. However, this removes one of the few instances when algorithms couldn't perform as well as handwritten code.

And finally, with the introduction of the range, algorithms now provide range overloads, leading to more concise and easier-to-read code.

Example of using the range overload of `std::sort`.

```
1 std::vector<int> dt = {1, 4, 2, 3};
2 std::ranges::sort(dt);
```

Open in Compiler Explorer

3.3 Projections

Each range algorithm comes with an additional argument, the projection. The projection is effectively a baked-in transform operation applied before an element is passed into the main functor.

The projection can be any invocable, which includes member pointers.

Example of using a projection to sort elements of a range based on a computed value from an element method.

```
1 struct Account{
2     double value();
3 };
4
5 std::vector<Account> data = get_data();
6
7 std::ranges::sort(data, std::greater<>{}, &Account::value);
```

Open in Compiler Explorer

Because the projection result is only used for the main functor, it does not modify the output of the algorithm except for `std::ranges::transform`.

Example of the difference between `std::ranges::copy_if`, which uses the projection result for the predicate and `std::ranges::transform`, which uses the projection result as the input for the transformation function (therefore making it affect the output type).

```
1 struct A{};
2 struct B{};
3
4 std::vector<A> data(5);
5
6 std::vector<A> out1;
7 // std::vector<B> out1; would not compile
8 std::ranges::copy_if(data, std::back_inserter(out1),
```

```
 9      [](B) { return true; }, // predicate accepts B
10      [](A) { return B{}; }); // projection projects A->B
11
12  std::vector<B> out2;
13  std::ranges::transform(data, std::back_inserter(out2),
14      [](auto x) { return x; },  // no-op transformation functor
15      [](A) { return B{}; }); // projection projects A->B
```

Open in Compiler Explorer

3.4 Dangling iterator protection

Because range versions of algorithms provide overloads that accept entire ranges, they open the potential for dangling iterators when users pass in a temporary range.

However, for some ranges, passing in a temporary is not an issue. To distinguish these two cases and to prevent dangling iterators, the range library has the concepts of a borrowed range and a special type `std::ranges::dangling`.

Borrowed ranges are ranges that "borrow" their elements from another range. Primary examples are `std::string_view` and `std::span`. For borrowed ranges, the lifetime of the elements is not tied to the lifetime of the range itself.

Example demonstrating the different handling for a borrowed range `std::string_view` and a `std::string`.

```
 1  const char* c_str = "1234567890";
 2
 3  // find on a temporary string_view
 4  auto sep1 = std::ranges::find(std::string_view(c_str), '0');
 5  // OK, string_view is a borrowed range, *sep1 == '0',
 6
 7  int bad = 1234567890;
 8
 9  // find on a temporary string
10  auto sep2 = std::ranges::find(std::to_string(bad), '0');
11  // decltype(sep2) == std::ranges::dangling, *sep2 would not compile
```

Open in Compiler Explorer

User types can declare themselves as borrowed ranges by specializing the `enable_borrowed_range` constant.

Example demonstrating declaring MyView as a borrowed range.

```
1  template<typename T> inline constexpr bool
2      std::ranges::enable_borrowed_range<MyView<T>> = true;
```

Open in Compiler Explorer

3.5 Views

One of the core problems with algorithms is that they are not easily composable. As a result, the code using algorithms is often quite verbose and requires additional copies when working with immutable data.

Views aim to address this problem by providing cheap to copy and move facilities composed at compile-time.

Example of composing several views.

```cpp
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8, 9};
for (auto v : data | std::views::reverse |
         std::views::take(3) | std::views::reverse) {
    // iterate over 7, 8, 9 (in order)
}
```

Open in Compiler Explorer

It is worth noting that views are stateful and should be treated as stateful lambdas (e.g. for iterator validity and thread safety).

Example of `std::views::drop` caching behaviour.

```cpp
std::list<int> data{1, 2, 3, 4, 5, 6, 7, 8};
auto view = data | std::views::drop(3);

for (auto v : view) {
    // iterate over: 4, 5, 6, 7, 8
}

// Note, if we used std::vector, push could invalidate
// the cached iterator inside of std::views::drop.
data.push_front(99);
for (auto v : view) {
    // iterate over: 4, 5, 6, 7, 8
}

// Fresh view
for (auto v : data | std::views::drop(3)) {
    // iterate over: 3, 4, 5, 6, 7, 8
}

const auto view2 = data | std::views::drop(3);
// for (auto v : view2) {}
// Wouldn't compile, std::views::drop requires mutability.
```

Open in Compiler Explorer

Chapter 4

The views

In this chapter, we will go over all the views the standard library offers.

4.1 `std::views::keys`, `std::views::values`

The `std::views::keys` and `std::views::values` operate on ranges of pair-like elements. The `std::views::keys` will produce a range of the first elements from each pair. The `std::views::values` will produce a range of the second elements from each pair.

Example of decomposing a `std::unordered_map` into a view over the keys and a view over the values.

```cpp
std::unordered_map<int,double> map{
    {0, 1.0}, {1, 1.5}, {2, 2.0}, {3, 2.5}
};

std::vector<int> keys;
std::ranges::copy(std::views::keys(map), std::back_inserter(keys));
// keys == {0, 1, 2, 3} in unspecified order (std::unordered_map)

std::vector<double> values;
std::ranges::copy(std::views::values(map),
        std::back_inserter(values));
// values == {1.0, 1.5, 2.0, 2.5}
// in unspecified order matching order of keys
```

Open in Compiler Explorer

4.2 `std::views::elements`

The `std::views::elements` will produce a range of the Nth elements from a range of tuple-like elements.

Example of creating element views for each tuple's second and third elements in the source range.

```
std::vector<std::tuple<int,int,std::string>> data{
    {1, 100, "Cat"}, {2, 99, "Dog"}, {3, 17, "Car"},
};

std::vector<int> second;
std::ranges::copy(data | std::views::elements<1>,
    std::back_inserter(second));
// second == {100, 99, 17}

std::vector<std::string> third;
std::ranges::copy(data | std::views::elements<2>,
    std::back_inserter(third));
// third == {"Cat", "Dog", "Car"}
```

Open in Compiler Explorer

4.3 `std::views::transform`

The transform view applies a transformation functor to every element of the range.

Example of using `std::views::transform` that also changes the base type of the range.

```
std::vector<double> data{1.2, 2.3, 3.1, 4.5, 7.1, 8.2};

std::vector<int> out;
std::ranges::copy(data |
    std::views::transform([](auto v) -> int {
        return v*v;
    }), std::back_inserter(out));
// out == {1, 5, 9, 20, 50, 67}
```

Open in Compiler Explorer

4.4 `std::views::take`, `std::views::take_while`

Both views are formed from the leading elements of the source range. In the case of `std::views::take`, the view consists of the first N elements. In the case of `std::views::take_while`, the view consists of the sequence of elements for which the predicate evaluates true.

Example of a view of the first three elements and a view of the leading sequence of odd elements.

```
std::vector<int> data{1, 3, 5, 7, 2, 4, 6, 8};

std::vector<int> out1;
std::ranges::copy(data | std::views::take(3),
    std::back_inserter(out1));
// out1 == {1, 3, 5}

std::vector<int> out2;
std::ranges::copy(data |
    std::views::take_while([](int v) { return v % 2 != 0; }),
    std::back_inserter(out2));
// out2 == {1, 3, 5, 7}
```

Open in Compiler Explorer

4.5 `std::views::drop`, `std::views::drop_while`

The drop views are the inverse of take views. The `std::views::drop` consists of all but the first N elements. The `std::views::drop_while` consists of all but the leading sequence of elements for which the predicate evaluates true.

Example of a view of all but the first three elements and a view skipping over the leading sequence of odd elements.

```
std::vector<int> data{1, 3, 5, 7, 2, 4, 6, 8};

std::vector<int> out1;
std::ranges::copy(data | std::views::drop(3),
    std::back_inserter(out1));
// out1 == {7, 2, 4, 6, 8}

std::vector<int> out2;
std::ranges::copy(data |
    std::views::drop_while([](int v) { return v % 2 != 0; }),
    std::back_inserter(out2));
// out2 == {2, 4, 6, 8}
```

Open in Compiler Explorer

4.6 `std::views::filter`

The filter view consists of all elements that satisfy the provided predicate.

Example of a view of even elements.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8};

std::vector<int> even;
std::ranges::copy(data |
    std::views::filter([](int v) { return v % 2 == 0; }),
    std::back_inserter(even));
// even == {2, 4, 6, 8}
```

Open in Compiler Explorer

4.7 `std::views::reverse`

The reverse view is the reverse iteration view for bidirectional ranges.

Example of a reverse view.

```
std::vector<int> data{1, 2, 3, 4};

std::vector<int> out;
std::ranges::copy(data | std::views::reverse,
    std::back_inserter(out));
// out == {4, 3, 2, 1}
```

Open in Compiler Explorer

4.8 `std::views::counted`

The counted view adapts an iterator and number of elements into a view.

Example of using a counted view to iterate over a subrange.

```
std::vector<int> data{1, 2, 3, 4, 5, 6, 7, 8};

std::vector<int> out;
std::ranges::copy(std::views::counted(std::next(data.begin()), 3),
    std::back_inserter(out));
// out == {2, 3, 4}
```

Open in Compiler Explorer

4.9 `std::views::common`

The common view adapts a view into a common range, a range with a begin and end iterator of matching types. Non-range versions of algorithms require a common range.

Example of using adapting a view for a non-range algorithm.

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

std::vector<int> out;
auto view = data |
    std::views::filter([](int v) { return v % 2 == 0; }) |
    std::views::common;

std::copy(view.begin(), view.end(), std::back_inserter(out));
// out == {2, 4, 6, 8}
```

Open in Compiler Explorer

4.10 `std::views::all`

The all view is a view of all the elements from a range.

Example of creating a view over all elements. Note that view over all elements is the default.

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

std::vector<int> out;
std::ranges::copy(std::views::all(data), std::back_inserter(out));
// out == {1, 2, 3, 4}
```

Open in Compiler Explorer

4.11 `std::views::split`, `std::views::lazy_split`, `std::views::join_view`

The two split views split a single range into a view over sub-ranges. However, they differ in their implementation. The `std::view::split` maintains the bidirectional, random access or contiguous properties of the underlying range, and the `std::view::lazy_split` does not, but it does support input ranges.

Example of using split view to parse a version number.

```cpp
std::string version = "1.23.13.42";
std::vector<int> parsed;

std::ranges::copy(version |
    std::views::split('.') |
    std::views::transform([](auto v) {
        int result = 0;
        // from_chars requires contiguous range
```

```
 9          std::from_chars(v.data(), v.data()+v.size(), result);
10          return result;
11      }),
12      std::back_inserter(parsed));
13  // parsed == {1, 23, 13, 42}
```

Open in Compiler Explorer

The join view flattens a view of ranges.

Example of using `std::views::lazy_split` to split a string into tokens and then join them using the `std::views::join`.

```
 1  std::string_view data = "Car Dog Window";
 2  std::vector<std::string> words;
 3  std::ranges::for_each(data | std::views::lazy_split(' '),
 4      [&words](auto const& view) {
 5          // string constructor needs common range.
 6          auto common = view | std::views::common;
 7          words.emplace_back(common.begin(), common.end());
 8      });
 9  // words == {"Car", "Dog", "Window"}
10
11  auto joined = data | std::views::lazy_split(' ') | std::views::join |
    ↪  std::views::common;
12  std::string out(joined.begin(), joined.end());
13  // out == "CarDogWindow"
```

Open in Compiler Explorer

4.12 `std::views::empty`, `std::views::single`

The empty view is an empty view (containing no elements), and the single view is a view that contains a single element. Note that the view owns the single element, which will be copied/moved alongside the view.

Example of using `std::views::empty` and `std::views::single`.

```
1  std::vector<int> out;
2  std::ranges::copy(std::views::empty<int>, std::back_inserter(out));
3  // out == {}
4
5  std::ranges::copy(std::views::single(42), std::back_inserter(out));
6  // out == {42}
```

Open in Compiler Explorer

4.13 `std::views::iota`

The iota view represents a generated sequence formed by repeatedly incrementing an initial value.

Example of using the finite and infinite iota view.

```cpp
std::vector<int> out1;
std::ranges::copy(std::views::iota(2,5), std::back_inserter(out1));
// finite view [2, 5), out == {2, 3, 4}

std::vector<int> out2;
std::ranges::copy(std::views::iota(42) | std::views::take(5),
std::back_inserter(out2));
// infinite view starting with 42, take(5) takes the first five
↪  elements from this view
// out2 == {42, 43, 44, 45, 46}
```

Open in Compiler Explorer

4.14 `std::views::istream`

The istream view provides similar functionality to istream iterator, except in the form of a view. It represents a view obtained by successively applying the istream input operator.

Example of using istream view.

```cpp
std::ranges::for_each(std::views::istream<int>(std::cin), [](int v) {
    // iterate over integers on standard input
});
```

Open in Compiler Explorer

Chapter 5

Bits of C++ theory

This chapter will dive deep into the various topics referenced throughout the book. While this chapter serves as a reference, the topics are still presented in a heavily simplified, example-heavy format. For a proper reference, please refer to the C++ standard.

5.1 Argument-dependent lookup (ADL)

When calling a method without qualification (i.e. not specifying the namespace), the compiler needs to determine the set of candidate functions. As a first step, the compiler will do an unqualified name lookup, which starts at the local scope and examines the parent scopes until it finds the first instance of the name (at which point it stops).

Example of unqualified lookup. Both calls to `some_call` will resolve to `::A::B::some_call` since this is the first instance discovered by the compiler.

```
1  namespace A {
2  void some_call(int) {}
3  void some_call(const char*) {}
4  namespace B {
5  void some_call(double) {}
6
7  void calling_site() {
8      some_call(1);   // A::B::some_call
9      some_call(2.0); // A::B::some_call
10     // some_call("hello world"); will not compile
11     // no conversion from const char* -> double
12 }
13 }
14 }
```

Open in Compiler Explorer

Due to the simplicity of unqualified lookup, we need an additional mechanism to discover overloads. Notably, it is a requirement for operator overloading since operator calls are unqualified. This is where argument-dependent lookup comes in.

Without ADL, any call to a custom operator overload would have to be fully qualified, requiring the function call syntax.

```
namespace Custom {
struct X {};

std::ostream& operator << (std::ostream& s, const X&) {
    s << "Output of X\n";
    return s;
}
}

void calling_site() {
    Custom::X x;
    Custom::operator << (std::cout, x); // OK, explicit call
    std::cout << x; // Requires ADL
}
```

Open in Compiler Explorer

While the full rules for ADL are quite complex, the heavily simplified version is that the compiler will also consider the innermost namespace of all the arguments when determining the viable function overloads.

```
namespace A {
    struct X {};
    void some_func(const X&) {}
}

namespace B {
    struct Y {};
    void some_func(const Y&) {}
}

void some_func(const auto&) {}

void calling_site() {
    A::X x; B::Y y;
    some_func(x); // Calls A::some_func
    some_func(y); // Calls B::some_func
}
```

Open in Compiler Explorer

Arguably the true power of ADL lies in the interactions with other language

features, so let's look at how ADL interacts with friend functions and function objects.

5.1.1 Friend functions vs ADL

Friend functions (when defined inline) do not participate in the normal lookup (they are part of the surrounding namespace but are not visible). However, they are still visible to ADL, which permits a common implementation pattern for a default implementation with a customization point through ADL.

Example demonstrating a default implementation with a customization point through ADL. The default implementation needs to be discovered during the unqualified lookup; therefore, if any custom implementation is visible in the surrounding namespaces, it will block this discovery.

```
 1 namespace Base {
 2     template <typename T>
 3     std::string serialize(const T&) {
 4         return std::string{"{Unknown type}"};
 5     }
 6 }
 7
 8 namespace Custom {
 9 struct X {
10     friend std::string serialize(const X&) {
11         return std::string{"{X}"};
12     }
13 };
14
15 struct Y {};
16 }
17
18 void calling_site() {
19     Custom::X x;
20     Custom::Y y;
21
22     auto serialized_x = serialize(x);
23     // serialized_x == "{X}"
24
25     // auto serialized_y = serialize(y); // Would not compile.
26
27     using Base::serialize;  // Pull in default version.
28     auto serialized_y = serialize(y);
29     // serialized_y == "{Unknown type}"
30 }
31
```

Open in Compiler Explorer

5.1.2 Function objects vs ADL

The second notable interaction occurs with non-function symbols. In the context of this section, the important ones are function objects (and lambdas).

Argument-dependent lookup will not consider non-function symbols. This means that a function object or a lambda must be either visible to an unqualified lookup or need to be fully qualified.

Example demonstrating a lambda stored in an inline variable that can only be invoked through a qualified call.

```
1  namespace Custom {
2      struct X {};
3
4      constexpr inline auto some_func = [](const X&) {};
5  }
6
7  void calling_site() {
8      Custom::X x;
9      // some_func(x); // Will not compile, not visible to ADL.
10     Custom::some_func(x); // OK
11 }
```

Open in Compiler Explorer

Finally, discovering a non-function symbol during the unqualified lookup phase will prevent ADL completely.

Example demonstrating a non-function object preventing ADL, making a friend function impossible to invoke.

```
1  namespace Custom {
2      struct X {
3          friend void some_func(const X&) {}
4      };
5  }
6
7  constexpr inline auto some_func = [](const auto&) {};
8
9  void calling_site() {
10     Custom::X x;
11     some_func(x); // calls ::some_func
12     // Because ADL is skipped, Custom::some_func cannot be called.
13 }
```

Open in Compiler Explorer

5.1.3 C++20 ADL customization point

With the introduction of concepts in C++20, we now have a cleaner way to introduce a customization point using ADL.

The concept on line 4 will be satisfied if an ADL call is valid, i.e. there is a custom implementation. This is then used on lines 8 and 13 to differentiate between the two overloads. One calls the custom implementation, and the other contains the default implementation. The inline variable on line 18 is in an inline namespace to prevent collision with friend functions in the same namespace. However, because friend functions are only visible to ADL, a fully qualified call will always invoke this function object.

```
 1 namespace dflt {
 2 namespace impl {
 3     template <typename T>
 4     concept HasCustomImpl = requires(T a) { do_something(a); };
 5
 6     struct DoSomethingFn {
 7         template <typename T> void operator()(T&& arg) const
 8         requires HasCustomImpl<T> {
 9             do_something(std::forward<T>(arg));
10         }
11
12         template <typename T> void operator()(T&&) const
13         requires (!HasCustomImpl<T>) { /* default implementation */ }
14     };
15 }
16
17 inline namespace var {
18     constexpr inline auto do_something = impl::DoSomethingFn{};
19 }
20 }
21
22 namespace custom {
23     struct X { friend void do_something(const X&){}; };
24     struct Y {};
25 }
26
27 void calling_site() {
28     custom::X x;
29     custom::Y y;
30     dflt::do_something(x); // calls custom::do_something(const X&)
31     dflt::do_something(y); // calls default implementation
32 }
```

Open in Compiler Explorer

This approach has several benefits, in particular on the call site. We no longer need to remember to pull in the namespace of the default implementation. Furthermore, because the call is now fully qualified, we avoid the problem of symbol collisions that can potentially completely prevent ADL.

5.2 Integral and floating-point types

Arguably one of the most error-prone parts of C++ is integral and floating-point expressions. As this part of the language is inherited from C, it relies heavily on fairly complex implicit conversion rules and sometimes interacts unintuitively with more static parts of C++ language.

5.2.1 Integral types

There are two phases of potential type changes when working with integral types. First, promotions are applied to types of lower rank than int, and if the resulting expression still contains different integral types, a conversion is applied to arrive at a common type.

The ranks of integral types are defined in the standard:

1. `bool`
2. `char`, `signed char`, `unsigned char`
3. `short int`, `unsigned short int`
4. `int`, `unsigned int`
5. `long int`, `unsigned long int`
6. `long long int`, `unsigned long long int`

Promotions

As mentioned, integral promotions are applied to types of lower rank than `int` (e.g. `bool`, `char`, `short`). Such operands will be promoted to int (if `int` can represent all the values of the type, `unsigned int` if not).

Promotions are generally harmless and invisible but can pop up when we mix them with static C++ features (more on that later).

Example of `uint16_t` (both a and b) being promoted to `int`.

```
1 uint16_t a = 1;
2 uint16_t b = 2;
3
4 auto v = a - b;
5 // v == -1, decltype(v) == int
```

Open in Compiler Explorer

Conversions

Conversions apply after promotions when the two operands are still of different integral types.

If the types are of the same signedness, the operand of the lower rank is converted to the type of the operand with the higher rank.

Example of integral conversion. Both operands have the same signedness.

```
int a = -100;
long int b = 500;

auto v = a + b;
// v == 400, decltype(v) == long int
```

Open in Compiler Explorer

When we mix integral types of different signedness, there are three possible outcomes.

When the unsigned operand is of the same or higher rank than the signed operand, the signed operand is converted to the type of the unsigned operand.

Example of integral conversion. The operands have different signedness but the same rank.

```
int a = -100;
unsigned b = 0;

auto v = a + b;
// v ~ -100 + (UINT_MAX + 1), decltype(v) == unsigned
```

Open in Compiler Explorer

When the type of the signed operand can represent all values of the unsigned operand, the unsigned operand is converted to the type of the signed operand.

Example of integral conversion. The operands have different signedness, and the type of the signed operand can represent all values of the unsigned operand.

```
unsigned a = 100;
long int b = -200;

auto v = a + b;
// v = -100, decltype(v) == long int
```

Open in Compiler Explorer

Otherwise, both operands are converted to the unsigned version of the signed operand type.

Example of integral conversion. The operands have different signedness, and the type of the signed operand cannot represent all values of the unsigned operand.

```
long long a = -100;
unsigned long b = 0; // assuming sizeof(long) == sizeof(long long)

auto v = a + b;
// v ~ -100 + (ULLONG_MAX + 1), decltype(v) == unsigned long long
```

Open in Compiler Explorer

Due to these rules, mixing integral types can sometimes lead to non-intuitive behaviour.

Demonstration of a condition that changes value based on the type of one of the operands.

```
int x = -1;
unsigned y = 1;
long z = -1;

auto t1 = x > y;
// x -> unsigned, t1 == true

auto t2 = z < y;
// y -> long, t2 == true
```

Open in Compiler Explorer

C++20 safe integral operations

The C++20 standard introduced several tools that can be used to mitigate the issues when working with different integral types.

Firstly, the standard introduced `std::ssize()`, which allows code that relies on signed integers to avoid mixing signed and unsigned integers when working with containers.

Demonstration of using `std::ssize` to avoid a signed-unsigned mismatch between the index variable and the container size.

```
std::vector<int> data{1,2,3,4,5,6,7,8,9};
// std::ssize returns ptrdiff_t, avoiding mixing
// a signed and unsigned integer in the comparison
for (ptrdiff_t i = 0; i < std::ssize(data); i++) {
    std::cout << data[i] << " ";
}
std::cout << "\n";
// prints: "1 2 3 4 5 6 7 8 9"
```

Open in Compiler Explorer

Second, a set of safe integral comparisons was introduced to correctly compare values of different integral types (without any value changes caused by conversions).

Demonstration of safe comparison functions.

```
int x = -1;
unsigned y = 1;
long z = -1;

auto t1 = x > y;
auto t2 = std::cmp_greater(x,y);
// t1 == true, t2 == false

auto t3 = z < y;
auto t4 = std::cmp_less(z,y);
// t3 == true, t4 == true
```

Open in Compiler Explorer

Finally, a small utility `std::in_range` will return whether the tested type can represent the supplied value.

Demonstration of `std::in_range`.

```
auto t1 = std::in_range<int>(UINT_MAX);
// t1 == false
auto t2 = std::in_range<int>(0);
// t2 == true
auto t3 = std::in_range<unsigned>(-1);
// t3 == false
```

Open in Compiler Explorer

5.2.2 Floating-point types

The rules for floating-point types are a lot simpler. The resulting type of an expression is the highest floating-point type of the two arguments, including situations when one of the arguments is an integral type (highest in order: `float`, `double`, `long double`).

Importantly, this logic is applied per operator, so ordering matters.

In this example, both expressions end up with the resulting type `long double`; however, in the first expression, we lose precision by first converting to `float`.

```
auto src = UINT64_MAX - UINT32_MAX;
auto m = (1.0f * src) * 1.0L;
auto n = 1.0f * (src * 1.0L);
// decltype(m) == decltype(n) == long double

```

```
6  std::cout << std::fixed << m << "\n" << n << "\n" << src << "\n";
7  // 18446744073709551616.000000
8  // 18446744069414584320.000000
9  // 18446744069414584320
```

Open in Compiler Explorer

Ordering is one of the main things to remember when working with floating-point numbers (this is a general rule, not specific to C++). Operations with floating-point numbers are not associative.

Demonstration of non-associativity of floating point expressions.

```
1  float v = 1.0f;
2  float next = std::nextafter(v, 2.0f);
3  // next is the next higher floating pointer number
4  float diff = (next-v)/2;
5  // diff is below the resolution of float
6  // importantly: v + diff == v
7
8  std::vector<float> data1(100, diff);
9  data1.front() = v; // data1 == { v, ... }
10 float r1 = std::accumulate(data1.begin(), data1.end(), 0.f);
11 // r1 == v
12 // we added diff 99 times, but each time, the value did not change
13
14 std::vector<float> data2(100, diff);
15 data2.back() = v; // data2 == { ..., v }
16 float r2 = std::accumulate(data2.begin(), data2.end(), 0.f);
17 // r2 != v
18 // we added diff 99 times, but we did that before adding to v
19 // the sum of 99 diffs is above the resolution threshold
```

Open in Compiler Explorer

Any operation with floating-point numbers of different magnitudes should be done with care.

5.2.3 Interactions with other C++ features

While integral types are implicitly inter-convertible, references to different integral types are not related types and will, therefore, not bind to each other. This has two consequences.

First, trying to bind an lvalue reference to a non-matching integral type will not succeed. Second, if the destination reference can bind to temporaries (rvalue, const lvalue), the value will go through an implicit conversion, and the reference will bind to the resulting temporary.

Demonstration of integral type behaviour with reference types.

```cpp
void function(const int& v) {}

long a = 0;
long long b = 0;
// Even when long and long long have the same size
static_assert(sizeof(a) == sizeof(b));
// The two types are unrelated in the context of references
// The following two statements wouldn't compile:
// long long& c = a;
// long& d = b;

// OK, but dangerous, implict conversion to int
// int temporary can bind to const int&
function(a);
function(b);
```

Open in Compiler Explorer

Finally, we need to talk about type deduction. Because type deduction is a static process, it does remove the opportunity for implicit conversions. However, this also brings potential issues.

Demonstration of potential problem of type deduction when using standard algorithms.

```cpp
std::vector<unsigned> data{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto v = std::accumulate(data.begin(), data.end(), 0);
// 0 is a literal of type int. Internally this means that
// the accumulator (and result) type of the algorithm will be
// int, despite iterating over a container of type unsigned.

// v == 45, decltype(v) == int
```

Open in Compiler Explorer

But at the same time, when mixed with concepts, we can mitigate implicit conversions while only accepting a specific integral type.

Demonstration of using concepts to prevent implicit conversions.

```cpp
template <typename T> concept IsInt = std::same_as<int, T>;
void function(const IsInt auto&) {}

function(0); // OK
// function(0u); // will fail to compile, deduced type unsigned
```

Open in Compiler Explorer

Index