容器，ranges与算法
Container, Ranges and
Algorithm

现代C++基础
Modern C++ Basics

Jiaming Liang, undergraduate from Peking University

- **Part 2**
- **Ranges**

- **Generator**

- **Function**

- **Algorithms**

Before that…

- We need to talk about `std::unique_ptr` first.
  - You need to learn *Move Semantics* and *Memory Management* to understand it thoroughly, but now we'll give it a simple introduction.
  - It's one of "smart pointers", i.e. it utilizes RAII to manage the resource.
    - Before, if you want to allocate something on heap, you'll use `new`.
    - However, it's easy to forget to `delete` it, especially when there are lots of return points in the function…
    - You can think `std::unique_ptr` with a ctor to `new` something and a dtor to `delete` something, so that exiting the function will trigger it automatically.

- As its name, it has **unique ownership** to the resource.
  - That is, you cannot copy it to another `std::unique_ptr`; otherwise destruction will `delete` the pointer twice, which is UB.
  - You need to **move it**, i.e. one gives up the ownership of the resource, and grants it to another.
  - `std::unique_ptr<T> ptr{ std::make_unique<T>(params) }, ptr2;`
    - `ptr2 = std::move(ptr);`

Containers, ranges and algorithms

Ranges

Ranges

- You may find it annoying when:
  - You cannot specify any other operations in range-based for, i.e. you have to iterate over all elements, iterate over more than one container.
  - You have to specify `xx.begin(), xx.end()` in e.g. `std::sort()` over and over again…

- Ranges will solve these problems!
  - Using ranges is very like functional programming.
  - It still has many problems now:
    - The optimization is not perfect, i.e. it may be slightly slower than normal loop.
    - The supported ranges-related infrastructures are limited.
    - User customization is hard.
  - It's useful, but I think it may be not good enough to use widely before C++26. But you may try it in your own project currently!
  - Notice that what we teach is based on C++23; Ranges in C++20 is too limited and has many bugs.

Ranges

- There are three important components in ranges:
  - Range: A type that provides a begin iterator and end sentinel, so that it can be iterated over.
    - `std::begin/end()`, and semantically the complexity is amortized $O(1)$.
    - A container is a range obviously.
  - View: A range that can be moved in $O(1)$, copied in $O(1)$ (or cannot be copied) and destructed in $O(1)$ (do not consider the destruction of elements).
    - As its name "view", it's usually a type inspecting elements (like span that we've taught).
    - It's also a kind of range, but usually it's non-owning compared with containers.
  - Range adaptor:
    - A functor that can transform a range/ranges to a view. Since a view is also a range, it can transform a view to a view, and use a new adaptor to continue to transform.
    - You can connect a range with a range adaptor with `operator|`, which consists of a *pipeline*.
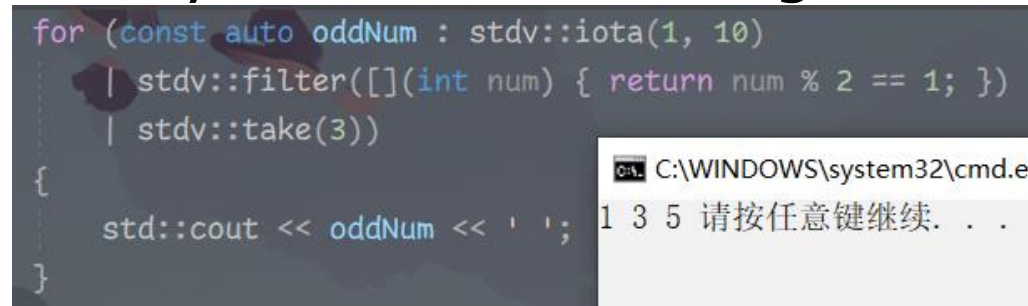
Ranges

- Note1: According to iterators, range also has concepts like `input/output/forward/bidirectional/random_access/contiguous_range`.

- Note2: A significant feature of the view generated by range adaptor is that calculation happens **lazily**.
  - That is, the calculation will happen iff. you want to get the next value of new view.
  - For example, you want to get elements that are less than 5 in a vector.
    - If you test all elements first, and write them to a new vector, and you use the new vector to do something else, then it happens **eagerly**.
    - But range only provides a way to iterate, so when the user wants the first element that is less than 5, you can go through the vector and pause at the first element that satisfies it; when the user wants the next, you can continue to go through.
      - This just makes an illusion to the user that you have all satisfied elements.

Ranges

- Notice that range and view are all *concept*; that is, it only regulates what a type should be, but not a type itself.
  - There is nothing like `range a`; we can only say e.g. vector is a range.
- They're all defined in `<range>`; all views are defined as `std::ranges::xx_view`, and all range adaptors are defined as `std::views::xx`.
  - Personally, I'd like to use alias as `namespace stdr = std::ranges` and `namespace stdv = std::views`.
  - There are also some range factories, e.g.
    - `stdr::iota_view{lower, upper = INF}/stdv::iota(lower, upper=INF)`, a sequence that starts from `lower` and `++` until `upper`(i.e. [lower, upper)).
      - If you've used Python, it's very similar to `range(a, b)`, except that in C++ `stdv::iota(a)` means [a, INF) rather than [0, a).
    - We'll cover more of them later…

Ranges

- So, let's see how you can utilize ranges first!

```
for (const auto oddNum : stdv::iota(1, 10)
    | stdv::filter([](int num) { return num % 2 == 1; })
    | stdv::take(3))
{

    std::cout << oddNum << ' ';
}
```

C:\WINDOWS\system32\cmd.e
1 3 5 请按任意键继续. . .

- This produce a view that:
    - Choose a number, increasing from 1 to 10 (excluding).
    - Then it's filtered so that the number is odd.
        - If it's even, then it won't go to the next range adaptor.
    - Finally, only the first 3 results are taken.
- Notice that any range is valid; e.g. you can change `stdv::iota` to `std::vector<int>`.

Ranges

- Some views are read-only, i.e. you cannot change the element referenced by the view.
  - `std::iota_view` is like this, because it just stores an integer, there is no real "integer sequence".
  - Thus, you cannot use `auto&` to get the element.

- Some views are writable.
  - E.g. if you change it to `v | xx` where `v` is `std::vector<int>`, since the integer sequence really exists, view to it is writable.
  - You can use `auto& num` and changing `num` is also changing original elements in vector.
  - But `const std::vector` is still read-only.

- A single read-only view will make the following views and the user get merely a read-only element.

- So, let's introduce some range adaptors!
- Writable:
  - `stdv::filter(Pred)`: drop the element if the predicate function `Pred` returns `false`.
  - `stdv::take(x)`: take first `x` elements (but not exceed `end`).
  - `stdv::take_while(Pred)`: take elements until `Pred` returns `false` (or `end`).
  - `stdv::drop(x)`: drop first `x` elements.
  - `stdv::drop_while(Pred)`: drop elements until `Pred` returns `false`.
  - `stdv::reverse`: reverse the elements; requires the range to be `bidirectional_range`.
  - `stdv::keys`: get the first element from a tuple-like thing.
    - Particularly, when you iterate over the map, since the key is `const`, the view is still read-only.
  - `stdv::values`: get the second element from a tuple-like thing.
  - `stdv::elements<i>`: get the `i`th element from a tuple-like thing.
  - `stdv::stride(k)`: use `k` as stride to iterate.
    - e.g. `std::iota(1, 10) | std::stride(2)` gets `{1, 3, 5, 7, 9}`.
    - This is a generalization of Python `range(a, b, k)`, which is only for integer rather than any range.

- There are also some adaptors that combine some ranges and/or return a tuple (still writable).
  - `stdv::join`: The element of range `R` is also range; this requires that their elements are of the same type, and **flattens** the ranges to a single range.
    - E.g. `std::vector<std::vector<int>> v{ {1, 2}, {3, 4, 5}, {6}, {7, 8, 9}};` `v | stdv::join` gets {1, 2, 3, 4, 5, 6, 7, 8, 9}.
  - `stdv::join_with(xx)`: Similar to join, but fill the interval with `xx`.
    - E.g. `std::vector<std::vector<int>> v{ {1, 2}, {3, 4, 5}, {6}, {7, 8, 9}};` `v | stdv::join_with(10)` gets {1, 2, **10**, 3, 4, 5, **10**, 6, **10**, 7, 8, 9}.
    - This is more useful in concatenating strings, e.g. `std::vector<std::string> v{"Are", "you", "Okay?"}`, `v | stdv::join_with(' ')` gets "Are you Okay?".
  - `stdv::zip(r1, r2, …)`: zip values from ranges to a tuple, e.g. `auto [a, b, c]: stdv::zip(v1, v2, v3)`.
    - Different from join, it accepts ranges as parameters instead of by `operator|` with a range of ranges.
    - If zipped ranges have different lengths, it will terminate when the shortest reaches its end.
  - `stdv::cartesian_product(r1, r2, …)`: return a tuple of elements from the cartesian product of these ranges.
    - E.g. $\{1, 2\} \times \{3, 4\} = \{\{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}\}$

- So, you can iterate over multiple containers like:

```cpp
std::vector<int> v1{ 1,2,3 }, v2{ 4,5,6 }, v3{ 7,8,9 };
for (auto [a, b, c] : stdv::zip(v1, v2, v3))
{
    std::cout << std::format("{} {} {}\n", a, b, c);
}
```

- Notice that though we use `auto` here, `a, b, c` **are all references** to the elements in vectors since it returns `std::tuple<T&, U&, …>`.
  - You cannot use `auto&`, since it returns a temporary tuple (though this tuple contains references…)!
  - You cannot make them read-only by e.g. `const auto`, `const auto&`.
    - You are using `const std::tuple<T&, U&, …>`!
      - But since the reference itself is never changed (i.e. the referenced object is same), the const tuple doesn't prohibit it, and you can still write them.
      - You can make an analogy with pointers; `const T*` will make the content read-only, `T* const` will make the pointer itself unchangeable (and this is reference!).
    - What you need is `std::tuple<const T&, const U&, …>`.
  - You should add a `| stdv::as_const` instead.
  - So in fact, so-called "read-only" is just return a `const T&` or `T` or tuple of `const T&/T`; "writable" is a `T&` or tuple of `T&`.
    - By remembering this, you will know what every element represents in binding.

Ranges

- `stdv::enumerate`: return `std::tuple<Integer, T&>`; Integer is index whose type is diff type of iterator.
  - For example, `std::list l{7, 3};  l | stdv::enumerate` will get [0, 7], [1, 3].
  - Thus, `auto[index, ele]` will still bind on `T&`.
- There are also some range adaptors that **produce many views** instead of a single view (elements in each view are still writable):
  - You need `for(auto v : …)` to get a view from their result, and `for(auto& ele:v)` to iterate over the view.
  - `stdv::slide(width)`: slide the range in a sliding window of `width`.
    - E.g. `std::vector v{1, 2, 3, 4, 5}; v | stdv::slide(3)` gets `{{1, 2, 3}, {2, 3, 4}, {3, 4, 5}}`.
  - `stdv::adjacent<width>`: same as `stdv::slide(width);`
    - The difference is that it uses a template parameter so it's slightly more efficient; and it should be able to be determined in compile time.
    - E.g. `int a = xx; stdv::adjacent<a>` is wrong while `stdv::slide(a)` is right.
  - `stdv::pairwise`: i.e. `stdv::adjacent<2>`.

Ranges

- `stdv::chunk(width)`: partition the range by `width`.
  - E.g. `std::vector v{1, 2, 3, 4, 5, 6}; v | stdv::chunk(2)` gets `{{1, 2}, {3, 4}, {5, 6}}`.
  - If `size % chunk_size != 0`, the last chunk size is the remainder.
- `stdv::chunk_by(pred2)`: partition the range by `pred2`, i.e. a view will stop when `pred2(*it, *next(it))` return `false`.
  - E.g. `std::vector v{1, 2, 5, 4, 3, 4}; v | stdv::chunk_by(std::less<int>{})` gets `{{1, 2, 5}, {4}, {3, 4}}`.
- `stdv::split(xx)`: split the range by delimiter `xx`, inverse operation of join.
  - E.g. `std::string str{ "Are you Okay?" }; str | stdv::split(' ')` gets `{ "Are", "You", "Okay?"}`.
- `stdv::lazy_split(xx)`: similar to split, but split only accepts `random_access_range` and will preserve the range category while lazy split will always result in `forward_range`.
  - So for view generated by `lazy_split`, `.size()` cannot be called, and it cannot be converted to random-access range (e.g. to use it in `sort`).

Caching behavior of views

- Some views may cache the begin hint once you call `begin()`.
  - Obviously, they're for those who cannot grab `begin` directly, i.e. `filter` `/drop_while/split/chunk_by`, to fulfill requirement of amortized $O(1)$.
    - Particularly, `drop` only caches for non-random-access or non-size-determined (i.e. no `.size()`) range, since it needs to `++` for several times to get it.
    - `slide(n)` is similar, but it only needs to maintain [begin, end) to be `n`, so it will optimize for some bidirectional range so that end is cached instead of begin.
      - We'll tell you what does "some" mean sooner.
  - This means that views with these caching adaptors behave differently before/after calling `begin`, which is counter-intuitive sometimes!

Notice that `lazy_split` doesn't cache `*begin` (i.e. get the whole range) like `split`; it delays comparison to the iteration of `*begin` (that's why it's only `forward_range`!).

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
auto f = [](int e) { std::cout << e << ' '; return e % 2 == 0; };
auto cachedView = v | stdv::filter(f);

int cnt = 0;
for (auto& ele : cachedView) cnt++;
std::cout << '\n' << cnt << '\n';
cnt = 0;
for (auto& ele : cachedView) cnt++;
std::cout << '\n' << cnt << '\n';
```

```
C:\WINDOWS\system32\cmd.exe
1 2 3 4 5 6
3
3 4 5 6
3
请按任意键继续. . .
```

Caching behavior of views

- Essentially, they cache the iterator (for at least forward range), but this is not forced, so many implementations may cache index for random-access range.
  - So, it's sometimes recommended to not modify the container after calling begin.
    - For writing, if we add `v[1] = 1` after the first loop, then though `1 % 2 != 0`, it won't be filtered since begin is seen as satisfied before, and it's already cached.
    - For inserting, if index is cached, you may also get wrong result if you insert before cached position. But other ranges may be correct as long as cached iterator is not invalidated after insertion.
    - For removing, similar to insertion; more severely, removal may cause size < offset or the iterator removed, so invalid memory access happens!
    - You may also pay attention to possible iterator invalidation, e.g. unordered containers are forward range so iterator is cached, but rehashing may invalidate all of iterators.

Caching behavior of views

- Besides, caching makes it vulnerable to parallelism, since two parallel `.begin()` (e.g. two loops) may operate caching result simultaneously, which causes data race.
- More importantly, `const auto&` that is widely used to refer to a read-only container may not fit for views.
  - Because `.begin()` may write its members, so `const` view may be not iterable.
  - Besides, this `const` is shallow, since it's only a view; if you want to make it only reads the underlying range, use `stdv::as_const` (like `std::span<const T>` instead of `const std::span<T>`).

- Finally, cache may or may not be copied, so copied view may have different behaviors from the original view.
  - E.g. if you change `std::list` to `std::vector`, then cache is preserved in MS…

```
std::list<int> v{ 1, 2, 3, 4, 5, 6 };
auto f = [](int e) { std::cout << e << ' '; return e % 2 == 0; };
auto cachedView = v | stdv::filter(f);

int cnt = 0;
for (auto& ele : cachedView) cnt++;
std::cout << '\n' << cnt << '\n';
cnt = 0;
auto view2 = cachedView;
....
for (auto& ele : view2) cnt++;
std::cout << '\n' << cnt << '\n';
```

```
C:\WINDOWS\system32\cmd.exe
1 2 3 4 5 6
3
1 2 3 4 5 6
3
请按任意键继续. . . ▪
```

Caching behavior of views

- All in all, views only guarantee that you will get the same elements for a non-modified range, when the predicate function doesn't change any outer states (like print).
  - It's usually encouraged to use views **ad hoc**, i.e. use it once and immediately like in a loop.
    - Then it's safe to write the element, as writable view possibly needs.
  - Or, at least you only read the views and preserving the underlying container unchanged, and no outer states are changed.
  - Otherwise, you need to ensure `.begin()` is not called, or pay much much attention to caching behavior.

Ranges

- Read-only:
  - Either make the view const, i.e. `std::as_const`; this will return `const T&` or `tuple<const T&, …>`.
  - Or return value, i.e. transform-related, which will return `T` or `tuple<T, …>`.
    - `stdv::zip_transform(TransformN, r1, r2, …)`: return a view of `TransformN(ele1, ele2, …)`.
      - It can only be the starting point of `operator|`.
    - `stdv::adjacent_transform<N>(TransformN)`: return a view of `TransformN(…)`, where … is the elements of the sliding window (i.e. unpacked view).
    - `stdv::transform(Transform)`: transform element to another element.
    - Here `Transform` means `Ret(T) or Ret(const T&)`; you should transform by return value e.g. `return a + 1`, not by e.g. `T& a → a++`!
      - Writable ranges that need cache shouldn't change value by `&` too.
  - For transform-related case, you need to pay special attention!
    - The result is uncached, so when its value are needed, it may be called for multiple times…

Caveat on transform

- For example:

```cpp
std::vector<int> v{ 1,2,3,4,5 };
int cnt = 0;
auto r = v
    | stdv::transform([&cnt](const int ele) {
        std::cout << std::format("cnt={},ele={}\n", cnt, ele);
        cnt++;
        return cnt + ele;
    })
    | stdv::take_while([](const int ele) { return ele < 10; });
for (const auto& i : r);
```

- You may expect that the result is "cnt=0,ele=1", etc., until "cnt=4,ele=5", since 4+5+1<10 is false, so the first four elements are taken by take_while.

- But unfortunately, it's like:
  - Only the first three elements are taken...

```
cnt=0, ele=1
cnt=1, ele=1
cnt=2, ele=2
cnt=3, ele=2
cnt=4, ele=3
cnt=5, ele=3
cnt=6, ele=4
```

Caveat on transform

- Astonishingly, the lazy evaluation is not that lazy...
  - Every transform is called twice; why?
  - Try:
    ```
    for (auto it = r.begin(); it != r.end(); it++) {
        //std::cout << *it << ' ';
    }
    ```
    ```
    cnt=0, ele=1
    cnt=1, ele=2
    cnt=2, ele=3
    cnt=3, ele=4
    cnt=4, ele=5
    ```
  - So, `++` will trigger transform since `take_while` needs the transformed value to advance the iterator (i.e. by `Pred(*it)`).
  - `const auto& ele = *it` will trigger it again!
    - If you just use `take(2)`, since `take` doesn't need the transformed value, transform happens exactly once.
  - That's because the result doesn't reference some existing elements, but generate from temporary; every time you need it, lazy evaluation generate it again.
    - You can think that other range adaptors can cache the original iterator, so that evaluation happens exactly once.
  - You need to pay special attention to transform-related adaptors, and their performance may be unsatisfying since function body may be called twice!

Ranges

- Sometimes, you may want to convert a range to a e.g. container, which needs to eagerly evaluate all.
    - Then you can use `stdr::to`, e.g. `stdr::to<std::vector>()`.
        - Notice that a template is provided instead of e.g. `std::vector<int>`.
    - This is implemented by supporting a ctor accepting `(std::from_range_t, Range)` or `(Range)` in containers since C++23.
- There are also some naïve range factories, just list here.
    - `stdv::single(obj)`: make a view that copies/moves an object, i.e. the view owns only a single element.
    - `stdv::empty<T>`: create an empty view.
    - `stdv::repeat(r, k = INF)`: repeat a range `r` for `k` times, i.e. like a range `[r, r, r, …]`
    - `stdv::istream<xx>(stream)`: similar to `istream_iterator`; it caches value.
- Several other things will be covered in the future.

Ranges

- You can also get a subrange from a range: `stdr::subrange(first, last)`.
  - If the iterator is not random-access, you can also add a `(size)` as the last parameter, so that if you use `size()` in the future, it's $O(1)$ instead of $O(n)$.
    - It's you who guarantee `std::distance(first, last) == size`!
    - Ranges that can get `size()` in $O(1)$ have the concept as `sized_range`.
- To be specific, we've said that you can use e.g. `std::size(), std::begin()` to get the size/iterator of container.
  - In a broad sense, `stdr::size/begin()` is more powerful.
    - It's safer because you cannot get an iterator from a temporary range (whose iterator will be dangling!).
    - Also you definitely get an iterator; E.g. `std::begin(c)` only calls `c.begin()`, but if `c.begin()` returns an integer instead of an iterator, it's still right…
      - `stdr::begin` will make compilation error in this case.
    - Besides, some ranges may only be able to use `stdr::` version.

Ranges

ranges::begin (C++20)

ranges::end (C++20)

ranges::cbegin (C++20)

ranges::cend (C++20)

ranges::rbegin (C++20)

ranges::rend (C++20)

ranges::crbegin (C++20)

ranges::crend (C++20)

ranges::size (C++20)

ranges::ssize (C++20)

ranges::empty (C++20)

ranges::data (C++20)

ranges::cdata (C++20)

- Note1: some ranges have no .size(), if they cannot determine their size in $O(1)$.
  - i.e. they're not sized_range.
  - For example, iota may be unbounded.
  - Or e.g. filter/take_while/lazy_split/split/join/ join_with/…
  - Or some ranges that don't have size itself, e.g. std::forward_list.
- Note2: data() is only callable for contiguous_range; otherwise compile error.
  - This will differentiate split_view and lazy_split_view; the latter cannot call data() on the new view since it'll generate forward_range.

Ranges

- The views generated by range factories and adaptors also have methods as:

base (C++20)	returns a copy of the underlying (adapted) view (public member function)
begin (C++20)	returns an iterator to the beginning (public member function)
end (C++20)	returns an iterator or a sentinel to the end (public member function)
size (C++20)	returns the number of elements. Provided only if the underlying (adapted) range satisfies sized_range. (public member function)
empty (C++20)	returns whether the derived view is empty. Provided if it satisfies sized_range or forward_range. (public member function of std::ranges::view_interface<D>)
cbegin (C++23)	returns a constant iterator to the beginning of the range. (public member function of std::ranges::view_interface<D>)
cend (C++23)	returns a sentinel for the constant iterator of the range. (public member function of std::ranges::view_interface<D>)
operator bool (C++20)	returns whether the derived view is not empty. Provided if ranges::empty is applicable to it. (public member function of std::ranges::view_interface<D>)
front (C++20)	returns the first element in the derived view. Provided if it satisfies forward_range. (public member function of std::ranges::view_interface<D>)
back (C++20)	returns the last element in the derived view. Provided if it satisfies bidirectional_range and common_range. (public member function of std::ranges::view_interface<D>)
operator[] (C++20)	returns the nth element in the derived view. Provided if it satisfies random_access_range. (public member function of std::ranges::view_interface<D>)

Ranges*

CPO is in fact more complex since it's related to ADL (**not covered in our course!**); here we only give a very general idea. See 如何理解 C++ 中的 定制点对象 这一概念？为什么要这样设计？ - Mick235711的回答 - 知乎 for more details.

The reason why we don't cover ADL is that it may evolve completely in C++26, and ADL before is a little bit obscure to novices. If you're interested, please refer to Chapter 13 of *C++ Templates 2nd.ed*.

- Now let's learn some basic concepts in ranges (**optional**).

- Customization Point Object(CPO): a const functor object that can be default-initialized and copyable.
  - All CPOs from the same class are equivalent, i.e. same parameters on `operator()` will get the same result, and any copy will not change the functionality of the CPO.

- Range Adaptor Object(RAO): A CPO that accepts ranges as first parameters and transform it/them to a view.

- Range Adaptor Closure Object(RACO): A RAO that supports pipe operator, i.e. `RACO(Range)` is equivalent to `Range | RACO`.
  - E.g. zip is not RACO, only an RAO.

Ranges*

- There are some important things about RAO:
  - It stores the **decayed** arguments, and "forward" them to call.
  - We'll cover decay in the next lecture (*Type Safety*), but you've known that C-array will be decayed to pointers.
    - References will be decayed to value.
  - So first, RAO stores arguments by value; this will make some objects "rvalue" when calling.
    - Some ranges are only viewable when not rvalue, e.g. `std::initializer_list`.
  - Besides, since pointers are not range but arrays are, decay will cause some astonishing things…
- All in all, if you don't understand currently, that's fine; but its outcomes should be remembered:

- 1. Array decay problem:

```cpp
std::vector<std::string> v{ "This", "is", "it" };
std::string result = v | std::views::join_with(' ') | std::ranges::to<std::string>();
// result is "This is it".
```

  - However, we cannot use join_with(" ") (i.e. double spaces); but we can use join_with(std::string{" "})!
    - This can be coded as " "s or " "sv when using namespace std::literals; we'll cover it in the future.
  - " " is char[3], which is a range because it has begin and end; but when it decays to char*, end has disappeared!
    - But string will not decay, so it's Okay.

- 2. Rvalue problem:
  - Similarly, if you join_with a vector<vector<int>> v, you cannot use v | join_with(k) where k is an initializer_list<int>!
  - But you can use join_with(v, k), since it doesn't call a function, but directly construct the new view, which will not "forward" it as rvalue.
    - i.e. join_with(v, k) is a view; join_with(k) is an RACO (which then needs to "forward"), and join_with(k)(v) should produce join_with(v, k).
  - You can also make k a container, e.g. array, vector etc. are all OK.

Sentinels

- Finally, we introduce sentinels.
  - We say that range is an iterator pair; but it's in fact beyond that.
  - An iterator begin and a sentinel end is enough!
  - For example, you can define your own sentinel for `const char*`:

```cpp
struct NullTerm {
  bool operator== (auto pos) const {
    return *pos == '\0';   // end is where iterator points to '\0'
  }
};
```

    - You can use it for `stdr::subrange` to construct a range.
      - It's still allowed to iterate over by range-based loop!
    - You can also use it in range-version algorithms, but not in common algorithms.
      - Common algorithms need `begin` and `end` to have the same type (i.e. concept as `common_range`); You may use range adaptor `stdv::common` to transform an iterator-sentinel pair to a common range.
    - So, for bidirectional range that is not common, `reverse/slide` will still cache the begin, because you cannot use `end.base()` to get the iterator.

Final word

- Some minor things are not covered:
  - 1. non-propagating cache, i.e. `chunk/lazy_split/join/join_with` will preserve something for input range, which is like cached state we've said before (and will cause the same problem!).
    - `join/join_with` for range of reference of range will do so too.
  - 2. `std::unreachable_sentinel`: a sentinel always return `false` for ==; using it as sentinel is same as `while(true){ …; it++; }`.
  - 3. `stdv::counted(it, n)`: Ranges can also be defined as a begin iterator with a count.
    - It's similar with `std::counted_iterator` with an end.
  - 4. RACO customization since C++23; we don't cover them because it needs advanced knowledge in the future.
    - You may learn it yourself after learning the whole course.
  - 5. `stdv::as_rvalue/std::move_sentinel`; you'll know what it is in *Move Semantics*.
  - 6. `.cbegin()/.cend()`: similar to `stdv::as_const`.

Containers, ranges and algorithms

Generator

Generator

- If you code Python before, you'll find that ranges are very similar to Python, e.g. filter, enumerate, map, etc.
  - Lazy evaluation is also very familiar…
  - Generator in Python is like this!
    - For generator expression, it's just like a lazy-evaluated pipeline.
    - But what about generator functions?

- For example:

```python
def func(end):
    begin = 0;
    while begin != end:
        yield begin
        begin += 1


generator = func(10)
for num in generator:
    print(num)
```

When the function reaches `yield`, it will pause and "return" the number; when iterator moves forward, the function will resume and continue to execute until the next `yield` or real `return`.

Generator

- Since C++23, generator is also supported by ***coroutine***.
  - Coroutines **cooperates** with each other and yield their execution flow themselves.
    - By contrast, for threads, they usually compete with each other and are interrupted by OS to give other threads chances to execute.
  - Generator is just the same; when you need the next value, you just **yield** your execution flow to the generator function; when the generator function completes its task, it will give back the right of execution.
    - Obviously, contexts of a coroutine are saved, so that you can pause and resume.
    - In Python before 3.6, coroutine is implemented by generator…
  - Generator is also an `input_range` and view; it provides `begin()` and `end()` to iterate, and `++` the iterator will resume the function.
    - NOTICE: Call .begin() will start the coroutine!

Generator

- For example:

```cpp
std::generator<int> func(int end)
{
    std::cout << "Ahh..";
    for (int begin = 0; begin < end; begin++)
        co_yield begin;
    co_return;
}
```

```cpp
auto generator = func(3);
```

- When you call `generator.begin()`, the generator function will pause at `co_yield begin`.
  - When you dereference the iterator, you will get the `begin` (i.e. 0) now.

```cpp
for (auto it = generator.begin(); it != generator.end(); it++)
    std::cout << *it;
// or
for (auto& num : generator)
    std::cout << num;
```

Generator

- Some notes:
  - Note1: one generator can only be used once.
    - i.e. You cannot call begin() twice.
    - Every generator represents an execution context, e.g. when begin is 3, you cannot make the execution "flows backwards" to make begin become 2.
    - You need a new generator if you want to iterate again.
  - Note2: generator has `operator=`, but it actually swaps two generators, i.e. swap their execution contexts.
    - The iterators are not invalidated after swapping; they still refer to the original context.
  - Note3: `co_return` can be omitted.
  - Note4: saving contexts also needs memory, so an allocator can also be provided as the last template parameter; we omit that now.
  - Note5: generator is also a view, which is only `input_range`.
- We'll cover coroutines deeply in the future!

Containers, ranges and algorithms

Function

Function

- Function
  - Pointer to member functions (in fact not that important…)
  - Callable parameter
  - Reinforcement on lambda

Pointer to member functions

- We've talked about pointer to functions; but member functions are special and different from normal functions.
    - So how can we get their pointers?
- First, member functions can be static or non-static.
    - Static ones are just normal functions with some `Class::`; their pointers are same as normal ones.
    - However, non-static ones are always bound to some specific objects, i.e. there is a `this` pointer as a parameter. This is what we hope to talk about.
- So pointer to member functions just means "an incomplete function", and when you want to use it, you need to bind an object to make it complete.

Pointer to member functions

- Wait…You may think "That bothers! Why don't I just use `obj.xx`?"
  - We'll talk about it later; let's see the syntax first.

- For example:

```cpp
class Person
{
public:
    unsigned int age;
    void AddAge(unsigned int n) {
        age += n;
    }
};
```

```cpp
using FuncPtr = void (Person::*)(unsigned int);

int main()
{
    Person p{ 0 }, *pointer = &p;
    FuncPtr ptr = &Person::AddAge;
    (p.*ptr)(5);
    (pointer->*ptr)(5);
    return 0;
}
```

- Define a pointer to member function is `Ret (Class::*)(params)`.
  - Unlike normal functions, `&` is necessary in `&Person::AddAge` (i.e. no decay).
  - Bind an object by `.*`, or an object pointer by `->*`.
  - Paratheses are added in `(p.*ptr)` since the precedence of `operator()` is higher, so no paratheses will cause compile error.

Pointer to member functions

- Similarly, you can also define pointer to data members. The usage is same. `using VarPtr = int Person::*;`

- It's quite troublesome to remember the type itself, so you can either use `auto ptr = &Person::AddAge` or `using FuncPtr = decltype(&Person::AddAge).`

- Also, `.*` and `->*` are quite weird, so you may use `std::invoke` defined in `<functional>` since C++17:

```
Person p{ 0 }, *pointer = &p;
auto ptr = &Person::AddAge;
std::invoke(ptr, p, 5);
std::invoke(ptr, pointer, 5);
```

  - This function will call the callable with corresponding operator.
    - For pointer to member functions + object, `.*`.
    - For pointer to member functions + object pointer, `->*`.
    - Otherwise, try ().

Pointer to member functions

- Since C++23, `std::invoke_r<Result>` is provided, which will convert the invoke result to type `Result`.

- Finally, why is it useful sometimes?
    - The pointer only conveys type instead of a specific member!
    - Before, I've written a simple SIMD program for blending two pictures.

```cpp
struct RGB
{
    std::uint8_t r;
    std::uint8_t g;
    std::uint8_t b;
};
```

```cpp
reg1 = _mm_set_epi16(RGBdata1[i + 7].r, RGBdata1[i + 6].r, RGBdata1[i + 5].r,
          RGBdata1[i + 4].r, RGBdata1[i + 3].r, RGBdata1[i + 2].r,
          RGBdata1[i + 1].r, RGBdata1[i].r);

reg2 = _mm_set_epi16(RGBdata2[i + 7].r, RGBdata2[i + 6].r, RGBdata2[i + 5].r,
          RGBdata2[i + 4].r, RGBdata2[i + 3].r, RGBdata2[i + 2].r,
          RGBdata2[i + 1].r, RGBdata2[i].r);

reg1 = _mm_mullo_epi16(reg1, alphaReg1);
reg2 = _mm_mullo_epi16(reg2, alphaReg2);
resultReg1 = _mm_add_epi16(reg1, reg2);
// for simplicity, code below omitted; there are also manys operation to get the result.
// ...
// store back
_mm_store_si128((__m128i*)buffer, resultReg1);
for(int k = 0; k < 16; k++)
{
    outRGBdata[i + k].r = buffer[k];
}
```

RGBdata is `vector<RGB>`, and this is blending r component.

Pointer to members

- So if I hope to blend to g and b, I have to paste the code twice.
  - What if I find a bug? Oops, paste again!
- By pointer to member, you can code like:
- Then the whole process is like:

```
SIMDBlendColor(RGBdata1.data() + i, RGBdata2.data() + i,
               outRGBdata.data() + i, &RGB::r, alphaReg1, alphaReg2);
SIMDBlendColor(RGBdata1.data() + i, RGBdata2.data() + i,
               outRGBdata.data() + i, &RGB::g, alphaReg1, alphaReg2);
SIMDBlendColor(RGBdata1.data() + i, RGBdata2.data() + i,
               outRGBdata.data() + i, &RGB::b, alphaReg1, alphaReg2);
```

- This is clean and beautiful…
- However, pointer to members is still rarely used actually…
  - Final word: sizeof them is implementation defined.

```cpp
using RGBColorPtr = std::uint8_t RGB::*;
void SIMDBlendColor(RGB* begin1, RGB* begin2,
                    RGB* outBegin, RGBColorPtr color,
                    __m128i alphaReg1, __m128i alphaReg2)
{
    __m128i resultReg1, reg1, reg2;
    alignas(16) std::uint8_t buffer[16];
    reg1 = _mm_set_epi16((begin1 + 7)->*color, (begin1 + 6)->*color,
                         (begin1 + 5)->*color, (begin1 + 4)->*color,
                         (begin1 + 3)->*color, (begin1 + 2)->*color,
                         (begin1 + 1)->*color, begin1->*color);

    reg2 = _mm_set_epi16((begin2 + 7)->*color, (begin2 + 6)->*color,
                         (begin2 + 5)->*color, (begin2 + 4)->*color,
                         (begin2 + 3)->*color, (begin2 + 2)->*color,
                         (begin2 + 1)->*color, begin2->*color);

    reg1 = _mm_mullo_epi16(reg1, alphaReg1);
    reg2 = _mm_mullo_epi16(reg2, alphaReg2);
    resultReg1 = _mm_add_epi16(reg1, reg2);
    // ...
    _mm_store_si128((__m128i*)buffer, resultReg1);
    for(int k = 0; k < 16; k++)
    {
        (outBegin + k)->*color = buffer[k];
    }
    return;
}
```

Function

- Function
  - Pointer to member functions
    - Callable parameter
  - Reinforcement on lambda

Function as parameter

- Sometimes, we need to pass a function as parameter.
  - Of course, function pointer will come into your mind immediately.
    - This is widely used in C.
  - But there are two problems:
    - Sometimes the parameter type is not strictly regulated, e.g. since `int` can be converted to `double`, `func(double)` is also acceptable.
    - In C++, usually what you need is just "something callable", i.e. a functor is allowed.
  - Both of them cannot be solved by function pointer!
- There are two general ways in C++ for it:
  - Use a template parameter; `<algorithm>` adopts this way to accept callable, e.g. predicate function for comparison in `sort`.
  - Use a `std::function` defined in `<functional>`.

std::function

- `std::function<RetType(Args…)>` can adopt almost all callable that have the return type **convertible** to `RetType` and accept `Args`.

- For example, `std::function<int(B, std::string, float)> f` can be constructed from:
  - A normal function/static member function, e.g. `int func(B, std::string, float){…}`.
  - A functor, e.g. `struct A{ int operator()(B, std::string, float) { … }}`.
  - A closure (no matter if it has captures or not), e.g.`[&](B, std::string, float)->int { … }`.
  - A non-static member function, e.g. `int B::MFunc(std::string, float)`.
    - You can think member function as a function with the first param `this` to understand it…
      - Similarly, you can use `(B*, …)`, `(B&, …)`, etc. to accept a member function.
      - The original `B` object won't be change if the type is specified as `B`, i.e. copy by value.
  - `nullptr`, as if a null function pointer (this is also the default-ctor).

std::function

- The member function even preserves polymorphism, i.e. if it accept `Parent&` but you pass a child class object and the member function is `virtual`, then it'll call the children version!

- After getting the `std::function`, you can just use `operator()` to call it.
  - Calling null function will `throw std::bad_function_call`.
  - You can judge whether it's null by e.g. `if(f)` or `if(f != nullptr)` as if a normal function pointer because it supports `operator bool` and `operator ==/!=` with `nullptr`.

- Even more powerful, you can bind some parameters to get new functors.
  - E.g. you can use `std::bind(any_callable, params)` to get a `std::function`.
  - For example:

```cpp
void print_num(int i)
{
    std::cout << i << '\n';
}
```

```cpp
std::function<void()> f_display_31337 = std::bind(print_num, 31337);
f_display_31337();
```
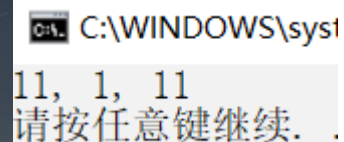
std::function

- Notice: If you want to bind a reference, you need `std::ref()` or `std::cref()`.
  - Otherwise, the bound parameter is copied, and reference parameter of the function will change this copied parameter rather than the bound one.
  - For example:

```cpp
class Person
{
public:
    unsigned int age = 1;
};

void AddAge(Person& p) { p.age += 10; }

int main()
{
    Person p, p1, p2;
    std::function<void(Person&)> f{ AddAge };
    std::function<void()> f1 = std::bind(AddAge, p1),
        f2 = std::bind(AddAge, std::ref(p2));
    f(p), f1(), f2();
    std::cout << std::format("{}, {}, {}\n", p.age, p1.age, p2.age);
    return 0;
}
```

```
C:\WINDOWS\sys1
11, 1, 11
请按任意键继续.
```

Reference wrapper

- `std::(c)ref()` in fact create `std::reference_wrapper<(const) T>`, which can be seen as an instantiated reference.
  - For example, you cannot store reference types in containers.
  - Then, you can use e.g. `std::vector<std::reference_wrapper<T>>`!
  - It's in fact a wrapper of pointer, but it cannot be null, just like reference.
  - You can assign an object to it, use `.get()` or convert it to `T&` to get the reference of it.
    - If it stores a function, you can use `(params)` to call it directly (`i.e. operator()`).
  - Different from reference, it can be bound to another object by `operator=`, just like pointer.
- It's also used to denote "it should be a reference" explicitly in some methods in standard library, e.g. `std::bind_xx`.
  - Usually you then need to pay attention to the lifetime problem, i.e. whether the referenced variable is destructed (dangling) when it's used.

std::function

- The first time I know `std::function`, I'm really astonished.
  - It's so powerful that it seems only a `std::function` is enough.
- Obviously, freedom comes at a price.
- There are two defects:
  - Performance: It roughly causes 10%-20% performance loss compared with direct function call.
    - It may need to `new/delete` a customized functor in the ctor/dtor.
      - In performance-critical scenarios, `new/delete` is costly, since it's hundreds or thousands times slower than arithmetic operations.
        - Not to mention overhead beyond a function call…
    - Thus, library uses SOO (small object optimization), i.e. it will prepare a small buffer that isn't allocated dynamically.
      - When the functor is small enough, `new/delete` won't be triggered.
      - It's also regulated in the standard that constructing from a normal function pointer will never allocate dynamically.

std::function

- Notice that you should guarantee captures by reference of the lambda expressions are valid when you call the `std::function`.
  - For example, in asynchronous callback, if you capture a local variable by reference and exit the function, so the local variables are destructed; then the callback will see dangling reference…
- Finally, since lambda expressions are just anonymous struct, it's possible that it's not "small object" when captures are too large.
  - Then you can use `auto lambda = xx;` and pass `std::ref(lambda)` to ctor of `std::function`.
  - Also, you need to guarantee it's called before the current scope exits, i.e. the lambda is not destructed.
    - Otherwise, the reference to lambda is dangling!
    - That's also why you cannot pass ref to an immediate lambda (`std::ref([&xx](){…})`), because the lambda will be invalid immediately after the construction.
    - If you don't understand it, that's OK now; we'll talk more about **lifetime** in the future…

C++26 adds a `std::function_ref`, which is basically same as `std::function{std::ref{xx}}` here. It's recommended to be used as parameter, just like `std::string_view` (we'll also cover string view in the future…).

std::function

- The second defect of `std::function` is that it cannot really accept all functors…
  - When the functor is not copiable (e.g. move-only, like `std::unique_ptr`)!
  - This is because `std::function` supports `operator=`, which needs to copy the functor.
- Thus, since C++23, `std::move_only_function` is introduced.
  - The usage is similar to `std::function`, except that:
    - It cannot be copied.
    - When it's null, it won't throw exception; it's UB to call null `move_only_function`.
  - For example:

```cpp
void test(std::unique_ptr<int> a, int b){
    std::cout << *a << ' ' << b << '\n';
}

struct Test
{
    Test(std::unique_ptr<int> a): a0(std::move(a)){}
    std::unique_ptr<int> a0;
    void operator()(int b){  std::cout << *a0 << ' ' << b << '\n'; }
};
```

Move only function

- These are all OK:

```cpp
std::function<void(std::unique_ptr<int>, int)> a{test};
a(std::make_unique<int>(111), 222);
std::function<void(std::unique_ptr<int>)> b{
    std::bind(test, std::placeholders::_1, 333)};
    // or std::bind_back since C++23.
b(std::make_unique<int>(444));
```

- But when you want to bind a `unique_ptr` or pass a move-only functor, it's not that good…

```cpp
// Wrong: std::function<void(int)> c{ std::bind(test, std::make_unique<int>(555)) };
// Right, but uncallable: auto c{ std::bind(test, std::make_unique<int>(555)) };
// Wrong: std::function<void(int)> c{ Test{std::make_unique<int>(777)} };
```

- Since C++23:

```cpp
std::move_only_function<void(int)> d{ Test{std::make_unique<int>(777)} };
d(888);
```

- But it still cannot cooperate with `std::bind_xxx`.

- Even further, if the functor is only constructible, not movable, you can still use it:

```cpp
std::move_only_function<void(int)> c(
    std::in_place_type<Test>, std::make_unique<int>(555));
c(666);
decltype(c) d;
d = std::move(c);
```

  - It's like `emplace`, we only pass params.
  - The `move_only_function` is still movable…

Move only Function

- Additionally, `std::move_only_function` will respect cv-qualifier, ref-qualifier of member functions, and `noexcept` specifier.
  - You've already know cv-qualifier; we'll mention ref-qualifier in the future.
  - For example:

```cpp
struct A
{
    void operator()() {/* ... */} // not const!
};
const std::function<void()> func(A{});
func(); // okay!
```

  - In a word (other qualifiers are similar to `const`):
    - `F<Ret(Args…) const>` will not accept a functor that has no `operator() const`.
      - `std::function` doesn't allow adding qualifiers here.
    - `F<Ret(Args…)>` will accept any functor, but calling `operator()` of `const F` acquires a functor to have `operator() const`.
      - `const std::function` will still tolerate calling non-const `operator()`.
  - Since C++26, `std::copyable_function` are added to substitute `std::function` to also respect these qualifiers.
    - Yet another dirty example of C++…

functional

- There are also some predefined template functors in `<functional>`.
  - As you've seen in *Containers*, `std::less<T>` is one of them.
- Arithmetic functors: `plus, minus, multiplies, divides, modulus, negate`(i.e. -x)
- Comparison functors: `equal_to, not_equal_to, greater, less, greater_equal, less_equal, compare_three_way` (since C++20).
- Logical functors: `logical_and, logical_or, logical_not`.
  - `not_fn` in C++17 accepts an functor `F`; its `operator(…)` is `!F(…)`.
- Bitwise functors: `bit_and, bit_or, bit_xor, bit_not`(i.e. ~x).
- `identity`, since C++20, just return itself.

functional

- These `Functor<T>` all have `auto operator()(const T&, const T&) const`.
- Then, let's observe what will `std::set<std::string> s` do!
  - i.e. `std::set<std::string, std::less<std::string>>` .
  - For `s.find("test")`, it will construct a temporary `std::string` from "test".
    - Then, it will compare this key with lots of inner nodes, i.e. `std::less<std::string>{}(TempStr, key)`.
  - However, this construction seems unnecessary…
    - All in all, comparison will only read the string, and why not just using the original C-string?
    - Even if `s.find` accepts any type as a template, `std::less<T>` will still convert it.
      - And even more times since any comparison needs a construction!
    - So, we also need a template `operator()`, which can accept any type and use their comparison operator directly.

functional

- Just like this:

```
struct less
{
    template<typename T, typename U>
    bool operator()(const T& a, const U& b) const{
        return a < b;
    }
};
```

- Considering that `void` is not a valid type for object, so `std::less<void>` is meaningless before.
  - Thus, since C++14, it's specialized as above (all functors in `<functional>` are specialized in this way).
  - You can also abbreviate it as `std::less<>`.

- It's called *transparent operator* because it defines a type as `is_transparent` (any type is fine, e.g. `using … = void`), so that the container will recognize it and use the template-version `find`.

Transparent operator

- The methods that support transparent operators are called *heterogeneous methods*.

- So, we just need `std::set<std::string, std::less<>>` to enable this feature to gain performance!
  - You can pass a C-string, `std::string`, `std::string_view` covered in the future, or your type with a comparison operator with `std::string`!

- Similarly, unordered containers can also be improved.
  - E.g. use `std::equal_to<>`.
  - So what about the hash function? Can it be transparent?

Transparent operator

- But different types may have different hash way…
  - E.g. `std::hash<double>(1.0)` is usually different from `std::hash<int>(1)`.
  - This property usually doesn't exist in comparison; `1.0 == 1.0` and `1.0 == 1` are same, so `std::equal_to<double>` can be changed to `std::equal_to<>`.
- The whole hash procedures are calculating hash value, mapping to bucket and using `equal_to` to find the exact value.
  - This assumes that equal values will hash to the same hash value!
  - Thus, it's usually dangerous to use transparent hash, so there is no `std::hash<>` specialization.
  - However, there exist some cases that we can convert possible types to a low-cost type (e.g. view type); for example, C-string, `std::string`, `std::string_view` can all be converted to `std::string_view` with low cost, and it's fine to use hash function of `std::string_view`.

Transparent operator

```
class Hasher
{
public:
    using is_transparent = void;
    size_t operator()(std::string_view sv) const {
        return std::hash<std::string_view>{}(sv);
    }
};
```

- You may define your own hash functor:
  - Notice: transparent hash should usually be non-template so that you can guarantee keys all use the same hash way.
    - It always accept a deterministic type!

- Then, you can use `std::unordered_set<std::string, Hasher, std::equal_to<>>` to gain performance.

- Notice that only hash and equal are **both** transparent will the e.g. `find()` uses this feature.

- Final word: a few member functions haven't supported transparent operators though there is a proposal in C++23 to improve them. But there are too many things to discuss for C++ committee in C++23, so they have to leave this proposal to C++26 to save time for more important features.
  - All in all, it will be fully supported in C++26.

Function

- Function
  - Pointer to member functions
  - Callable parameter
  - Reinforcement on lambda

Lambda expression

- There are several things for lambda to be covered.
- First, lambda expression can be returned from function like **auto** `func(…)` since C++14.
- We call lambda expression without capture *stateless lambda*; otherwise *stateful lambda*.
- We said that lambda expression is an anonymous struct, so `decltype(…)` will deduce a unique type.
  - Theoretically, you can use the type to construct a new functor.
  - E.g. `auto l = [](int a, int b) { return a + b; };`
    `using T = decltype(l); T l2{}.`
  - However, this is in fact legal only since C++20…
    - The anonymous struct also has a copy ctor & assignment operator.
    - Stateful lambda is not constructible, copiable and assignable.

Lambda expression

- This is useful for e.g. stateless compare functor of containers.
  - Before, you **can't** use `std::set<xx, LambdaType>` since `LambdaType` cannot be used to construct a new functor, i.e. `LambdaType{}(…)` is impossible.
  - You can only use `std::set<xx> s{ …, Compare}` to pass a functor.
  - After C++20, this limit is eliminated!
- Besides, C++20 supports *unevaluated lambda*, i.e. you can use `using T = decltype([](int a, int b) { return a + b; });` directly instead of declaring a real lambda and `decltype` it.
  - `decltype, sizeof`, … doesn't really need to evaluate the expression, e.g. `sizeof(a += b)` will not really perform `a += b`.
- Final word: You can cooperate lambda expression with initialization of variables by calling it immediately:

```
void test()
{
    static int onlyInitOnce = []() { return 1; }();
}
```

Containers, ranges and algorithms

Algorithms

Algorithm

- Algorithms are consisted of:
  - Iterator pairs, to denote the range of applying algorithm.
    - Also [begin, end).
  - Predicate function / transform function, get several parameters from the iterating sequence and judge / operate on them. **Usually** they won't transform the element itself, so params should be `const&` or copy by value.
    - The function accepts the underlying value rather than the iterator.
    - Also the `operator()` (`operator<`, `operator>`, etc., if needed) usually should be const. But it's basically Okay if not.
    - If the type can somehow be converted to a functor type (e.g. by `operator S()`, a user-defined conversion function), then it's also Okay;
      - Sometimes we call it surrogate (代理) call function.
  - Mostly return an iterator that satisfies the requirement of the algorithm. If no iterator satisfies, return end.

Algorithm

- They will **never** change the size of sequence, except:
  - You're using e.g. `back_inserter`, the iterator itself will change the size.
  - `std::erase/std::erase_if`; this is in fact not in `<algorithm>`, but in the container's header.
  - So, you need to guarantee the destination range is big enough, e.g. by resizing vector.
- Callables of algorithm are of value type instead of reference type.
  - If it's just a normal function, then it's a copy of function pointer.
  - Or if it's a temporary (like an immediate lambda), it's also not a burden.
    - We'll tell you why at *Move Semantics*…
  - But if it's a normal functor, like `auto func = […](){…}; std::some_algorithm(…, func)`.
    - Then a copy will happen, i.e. members / captures will be copied.
  - So, if you really want to pass into a reference to functor, you can use `std::ref(func)`.
    - You can assign an object to it, use `.get()` or convert it to `T&` to get the reference of it.

Remember?
    - If it stores a function, you can use `(params)` to call it directly (`i.e. operator()`).

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax
    - Permutation
    - There are also algorithms related to random number, which will be covered in the future.
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Search

- There are two kinds of search algorithms:
  - Linear search, the complexity is $O(\Pi range\_length)$.
    - Find a single value: (begin, end, …)
      - std::find: (value), return the first iterator that equals to value.
      - std::find_if(_not_): (Pred), return the first iterator that makes Pred(ele) return true(_false_).
    - Find one of values in a range: (begin1, end1, begin2, end2[, Pred2])
      - std::find_first_of.
        - **Find first** element in [begin1, end1) that is **one of** [begin2, end2).
        - For each ele1 in [begin1, end1)
          - std::find(begin2, end2, ele1).
          - If not exist (i.e. get end iterator), continue loop; else return the iterator of **ele1**.
        - The complexity is $O(NM)$.

Search

- Find a sub-sequence in a sequence (**Pattern matching**): `(begin1, end1, begin2, end2[, Pred2])`.
  - Optional `Pred2` acts as `operator==`.
  - `std::search`: find the first occurrence of `[begin2, end2)` in `[begin1, end1)`. Return the begin iterator of this occurrence in `[begin1, end1)`.
  - `std::find_end`: find the last occurrence of `[begin2, end2)` in `[begin1, end1)`. Return the begin iterator of this occurrence in `[begin1, end1)`.
  - The complexity of these two methods is $O(NM)$!
    - However, we've learnt that KMP in pattern matching is $O(N + M)$.
    - We'll tell you how to do so later…
- Others:
  - `std::adjacent_find(begin, end[, Pred2])`, return the iterator that `*it == *std::next(it)`.
  - `std::search_n(begin, end, count, value, [, Pred2])`, return the iterator that `[it, std::next(it, N))` is all `value`. $O(N)$.
    - `std::next` doesn't mean requiring random-access; just denote a range.

Search

- Examples:

```cpp
auto test(std::vector<int>& v) -> generator<decltype(v.begin())>
{
    std::vector sub{ 2, 0, 4 }, sub2{ 4, 4, 2 };
    co_yield std::find(v.begin(), v.end(), 3);
    co_yield std::find_if(v.begin(), v.end(),
            [](const int& elem) { return elem % 2 == 1; });
    co_yield std::find_first_of(v.begin(), v.end(), sub.begin(), sub.end());
    co_yield std::search(v.begin(), v.end(), sub2.begin(), sub2.end());
    co_yield std::find_end(v.begin(), v.end(), sub2.begin(), sub2.end());
    co_yield std::search_n(v.begin(), v.end(), 3, 2);
    co_yield std::adjacent_find(v.begin(), v.end());
}

int main()
{
    std::vector v{ 1,3,4,4,2,4,4,2 };
    for (auto it : test(v))
        std::cout << (it - v.begin()) << '\n';
    return 0;
}
```

Search


```cpp
std::string total = "This is it.";
std::string pattern = "is";
std::boyer_moore_searcher searcher{ pattern.begin(), pattern.end() };
for (auto it = std::search(total.begin(), total.end(), searcher); it != total.end();
    it = std::search(std::next(it), total.end(), searcher))
{
    std::cout << total.substr(it - total.begin()) << '\n';
}
```

```
C:\WINDOWS\system32\cmd.e
is is it.
is it.
请按任意键继续. . . _
```

- Since C++17, pattern matching can be linear.
  - However, the algorithm is not KMP since KMP is usually not optimal in real applications.
    - In real cases e.g. search engine, the possibility of matching is usually low, the brute-force algorithm may be even faster than KMP because its low constant.
      - That's possibly why it's not supported before…
  - The linear algorithm is BM (Boyer-Moore) algorithm.
  - The corresponding overload is std::search(begin, end, Searcher), which needs random access iterator (common case!).
    - The searcher should be constructed from the pattern [begin2, end2) and has an operator()(begin, end) that returns iterator pair [it1_begin, it1_end) in [begin, end) to denote the range of first occurrence (but std::search only returns it1_begin, so you can use searcher directly if you want).
    - The standard library provides three searchers in <functional>, i.e. std::default_searcher (brute-force), std::boyer_moore_searcher and std::boyer_moore_horspool_searcher.

BM algorithm is rather hard, so I recommend you to watch THU DSA course (数据结构（下）-第十三章（串）-DE两节课) if you're interested. BMH just uses BC table while BM uses BC+GS table, so the latter is linear while the former has lower preparation and space overhead but not linear.

Search

- The second kind is binary search, which is applied on **sorted** sequence.
  - Since associative containers have their own methods, they don't use it.
  - Only random-access containers supports $O(\log N)$ complexity; otherwise it's $O(N)$ since go to the medium point needs to iterate one by one.
    - But the comparison complexity is always $O(\log N)$.
  - All of them accept `(begin, end, value[, Pred2])`, and `Pred2` acts as `operator<`.
    - If `a < b && b < a`, then it's regarded as equal.
    - Notice that many algorithms require `operator<` to be strictly ordered, i.e. `a < b`, `b < a` cannot both be `true`.
      - Otherwise they may be less efficient or even go wrong totally!
  - `std::binary_search`: return bool, denoting whether `value` exists in `[begin, end)`.
  - `std::lower_bound`: return `it` so that `value` is the lower bound of `[it, end)`.
    - `[begin, it)` is all less than `value`. This is same as `std::map`.

Search

- `std::upper_bound`: return `it` so that `value` is the upper bound of `[begin, it)`.
  - `[it, end)` is all greater than `value`.
- `std::equal_range`: return an iterator pair `(it1, it2)` so that `value` is equal to `[it1, it2)`.
  - i.e. `[std::lower_bound, std::upper_bound)`.
- These three functions all have the same meanings as `std::map`.
- Notice that predicate function can accept two different types if the `value` is of a different type.
  - For example, the object is very huge but it just use id (an integer) as the key.
  - So, we can just use the key instead of constructing a huge empty object.
  - For `std::lower_bound`, it only uses `Pred2(Element, KeyType)`.
  - For `std::upper_bound`, it only uses `Pred2(KeyType, Element)`.
  - For the other two, you need to define both of two overloads.

Search

- Examples:

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

auto [it1, it2] = std::equal_range(v.begin(), v.end(), 3);
std::cout << std::format("{}, {}, {}, {}, {}\n",
    std::binary_search(v.begin(), v.end(), 5),
    it1 - v.begin(), it2 - v.begin(),
    std::lower_bound(v.begin(), v.end(), 2) - v.begin(),
    std::upper_bound(v.begin(), v.end(), 3) - v.begin()
);
```

C:\WINDOWS\system32\cmd.exe

```
false, 2, 4, 1, 4
```

- Particularly, you can use different types:

```cpp
class Person
{
public:
    Person(int init_id, const std::string& init_name) : id_{ init_id },
        name_{ init_name }, auxHugeData_(10000) {};
    auto GetID() const { return id_; }
private:
    int id_;
    std::string name_;
    std::vector<int> auxHugeData_;
};
```

```cpp
bool PersonEqualToID(const Person& p, const int id) {
    return p.GetID() == id;
}

bool PersonLessThanID(const Person& p, const int id) {
    return p.GetID() < id;
}

bool IDLessThanPerson(const int id, const Person& p) {
    return p.GetID() > id;
}
```

Search

```
std::vector<Person> persons{
    { 100, "Liang" }, { 200, "Li" },
    { 300, "Liu" }, { 400, "LiuLiu" }
};

std::cout << std::format("{}, {}\n",
    std::lower_bound(persons.begin(), persons.end(), 199, PersonLessThanID) - persons.begin(),
    std::upper_bound(persons.begin(), persons.end(), 299, IDLessThanPerson) - persons.begin()
);
```

C:\WINDOWS\system32\cmd.exe

```
1, 2
```

- However, for `std::equal_range` and `std::binary_search`, since they need two-way comparison (i.e. to know `==`, `ele1 < ele2` and `ele2 < ele1` should both compile), predicate function can only accept convertible types.
  - Usually you can define `operator<`.

```
bool operator<(const Person& p, const int id) {
    return p.GetID() < id;
}

bool operator<(const int id, const Person& p) {
    return p.GetID() > id;
}
```

Search


```
auto [it1, it2] = std::equal_range(persons.begin(), persons.end(), 200);
std::cout << std::format("{}, {}, {}\n",
    it1 - persons.begin(), it2 - persons.begin(),
    std::binary_search(persons.begin(), persons.end(), 200)
);
retu 1, 2, true
```

C:\WINDOWS\system32\cmd.exe

1, 2, true

- Final word: you can also define `auto operator<=>(int)` and `auto operator==(int)` as member function to deduce these operators!

Algorithms

- Algorithms
  - Search
  - **Comparison**
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax
    - Permutation
  - Numeric algorithms
  - Parallel algorithms
  - Range-version algorithms

Comparison

- `std::equal(begin1, end1, …[, Pred2])`: return a bool.
  - `(begin2)`: assuming `std::distance(begin1, end1) == n`, then it compares `[begin1, end1)` with `[begin2, begin2 + n)`.
  - `(begin2, end2)`: It compares `[begin1, end1)` with `[begin2, end2)`.
    - Since C++14; it may be dangerous to use the first version if `[begin2, begin2 + n)` is not all valid (e.g. compare two vectors, with the first has larger size)…
  - Similarly, `Pred2(Type1, Type2)` is enough; Not forced to be same.
- `std::lexicographical_compare(begin1, end1, begin2, end2[, Pred2])`: return a bool; `Pred2` acts as `operator<`.
  - Compare until `ele1 < ele2 || ele2 < ele1`, return that bool.
    - `false` if loop is over.
- `std::lexicographical_compare_three_way(begin1, end1, begin2, end2[, Pred2])`: return an ordering; `Pred2` acts as `operator<=>`.

Comparison

- `std::mismatch(begin1, end1, …[, Pred2])`
  - `(begin2), (begin2, end2)` since C++14.
  - Similar to `std::equal`, but will return an iterator pair `(it1, it2)` denoting the first occurrence of mismatching.
- These algorithms are all $O(N)$.


```
std::vector ids{ 100, 200, 300, 400 }, ids2{ 100, 300, 200};
auto [it1, it2] = std::mismatch(persons.begin(), persons.end(), ids2.begin(), ids2.end(), PersonEqualToID);
std::cout << std::boolalpha << it1 - persons.begin() << ' ' << it2 - ids2.begin() << ' '
    << std::equal(persons.begin(), persons.end(), ids.begin(), ids.end(), PersonEqualToID);
```

C:\WINDOWS\system32\cmd.exe

1 1 true请按任意键继续. . .

```
std::vector ids{ 100, 200, 100, 300 }, ids2{ 100, 200, 400, 300 };
std::cout << std::lexicographical_compare(persons.begin(), persons.end(), ids.begin(), ids.end())
    << ' ' << std::lexicographical_compare(persons.begin(), persons.end(), ids2.begin(), ids2.end());
```

C:\WINDOWS\system32\cmd.exe

0 1请按任意键继续. . .

Similar to `equal_range`, this needs two-way comparison.

Algorithms

- Algorithms
  - Search
  - Comparison
- Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax
    - Permutation
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Counting

- `std::all_of/any_of/none_of(begin, end, Pred)`: return whether all/any/none of elements in `[begin, end)` make `Pred` return `true`.

- `std::count(begin, end, value)`: return how many elements in `[begin, end)` are equal to `value`.

- `std::count_if(begin, end, Pred)`: return how many elements in `[begin, end)` make `Pred` return `true`.

- These algorithms are all $O(N)$.

- Examples:

```
std::vector<int> v{ 1,1,3,5 };
std::cout << std::format("{},{},{},{},{}\n",
    std::all_of(v.begin(), v.end(), [](const int ele) { return ele % 2 == 1; }),
    std::any_of(v.begin(), v.end(), [](const int ele) { return ele == 2; }),
    std::none_of(v.begin(), v.end(), [](const int ele) { return ele / 2 == 1; }),
    std::count(v.begin(), v.end(), 1),
    std::count_if(v.begin(), v.end(), [](const int ele) { return ele >= 3; }));
```

```
C:\WINDOWS\system32\cmd.exe
true, false, false, 2, 2
```

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - **Generating and Transforming**
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax
  - Permutation
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Generating and Transforming

- `std::fill(begin, end, value)`: Fill all elements in `[begin, end)` with `value`.
- `std::fill_n(begin, count, value)`: Fill all elements in `[begin, next(begin, count))` with `value`.
- `std::generate(begin, end, Gen)`: For each element in `[begin, end)`, `ele = Gen()`.
  - Call n times, so different with fill.
  - `std::generate_n(begin, count, Gen)`: Similarly.
- `std::for_each(begin, end, Transform)`: For each element in `[begin, end)`, call `Transform(ele)`
  - `std::for_each_n(begin, count, Transform)`: Similarly, since C++17.
  - Usually a for-loop is enough, so they're not frequently used (except for their parallel version, covered later).   All `_n` algorithms return iterator `begin + n`.

`for_each` is the only algorithm that uses the passed functor as return value.

Generating and Transforming

- `std::transform`: there are unary/binary transforms.
  - `(begin, end, dstBegin, Transform)`: unary, similar to `for_each`, but return value is the transformed result and will be written to `dstBegin` sequence and the original sequence is unchanged.
    - You need to make sure destination range is large enough, e.g. the vector should be resized.
    - `dstBegin` may not be in `(begin, end)`, since transformed value will overwrite so elements later are wrong.
      - Particularly, `dstBegin == begin` means the value itself is covered.
  - `(begin, end, begin2, dstBegin, Transform)`: binary, the transformation is `Transform(it1, it2)`; The transformed result will be written to `dstBegin`.
    - Still, pay attention to valid range of `begin2` and `dstBegin`…
- These algorithms are all $O(N)$.

Generating and Transforming

- Examples:

```cpp
std::vector<int> v(1000), dst(1000);
std::fill(v.begin(), v.end(), 1);
std::fill_n(v.begin(), v.size() / 2, 2);
std::generate(v.begin(), v.end(), []() { return rand(); });
std::for_each(v.begin(), v.end(), [](int& ele) { ele *= 3; });
std::for_each_n(v.begin(), v.size() / 2, [](int& ele) { ele /= 2; });
std::transform(v.begin(), v.end(), dst.begin(), [](const int& ele) { return ele / 2; });
std::transform(v.begin(), v.end(), dst.begin(), dst.begin(), // read from dst and write back
        [](const int eleFromV, const int eleFromDst) { return eleFromV + eleFromDst; });
```

- Notice that `std::for_each` needs to transform by accepting `T&` while `std::transform` does it by return value.

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
- Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax

- Permutation
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Modifying

- These algorithms are the most important part, because it's not easy for you to write correctly or efficiently e.g. by a for-loop.
    - `std::remove(begin, end, value)`/`std::remove_if(begin, end, Pred)`: return the iterator so that `[begin, it)` has no element that is equal to `value` or make `Pred` return `true`.
        - Notice that algorithms don't erase any iterator!
        - So, `[it, end)` has invalid elements, and if you want to erase them, you need call the container's methods `erase`.
            - Notice that values of invalid elements are unspecified. You'll know why in the future.
        - This is called ***Erase-remove idiom***.
            - Since C++20, they're integrated as methods `std::erase/std::erase_if`.
    - `std::unique(begin, end[, Pred2])`: return the iterator so that `[begin, it)` has no **adjacent** equal element; `Pred2` acts as `operator==`.
        - This is same as `list.unique()` as we've said, but `list.unique` will erase the iterator while `std::unique` will not (similar to `remove`).
        - So it's usually only used in random-access containers.

Modifying

- These two algorithms are both $O(n)$, by the technique of *dual pointers*.
  - We introduce unique, and you can think about remove yourself.
  - 1. Prepare a "before" iterator and an "after" iterator, initialized as `begin`.
    - `before` always means the last checked value; since begin is definitely in the result, this is satisfied initially.
  - 2. Advance `after`.
  - 3. If `after` is not equal to `before`, advance `before` and copy/move `*after` to `*before`.
  - 4. Loop until `after` reaches `end`.
  - Corner case that `begin == end` should be considered, too.

Modifying

- We advance after so that it's at position 1 first.

before

| 0 | 1 | 1 | 4 | 5 | 3 | 3 | 3 |

after

*after == *before?

no, advance before, assign *after to *before

Advance after.

Modifying

before

after

*after == *before?

Yes, only advance after.

Modifying

before

0	1	1	4	5	3	3	3

after

*after == *before?

no, advance before, assign *after to *before.

Modifying
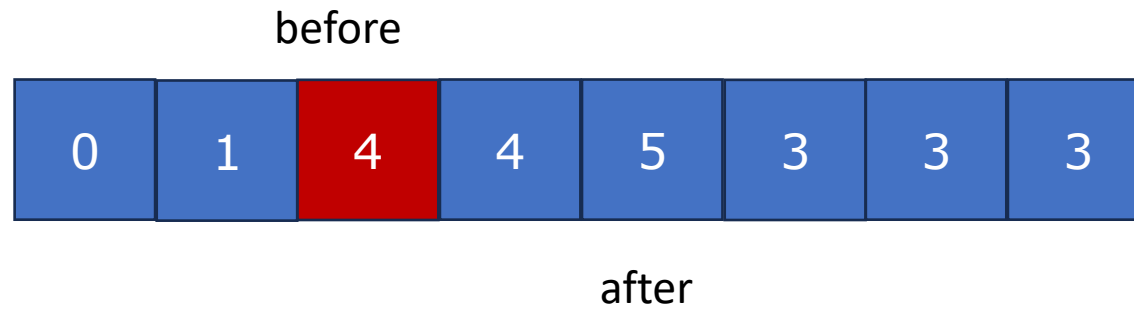
before

after

*after == *before?

no, advance before, assign *after to *before…

Modifying

- The final result is like:

before

after
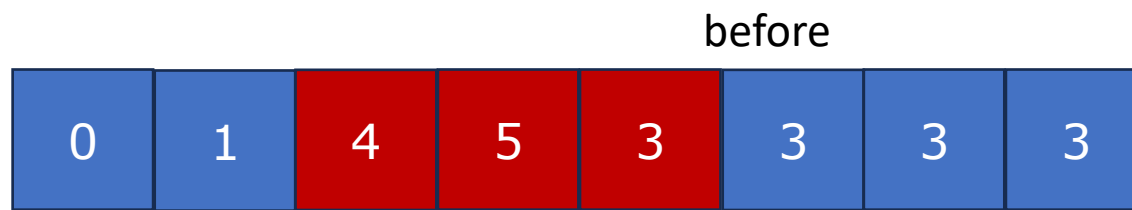
- Return before.
  - Removal is very similar, you may design the algorithm yourself!
  - It you really want to erase the removed elements, you need to call `vec.erase()` manually (i.e. Erase-remove idiom).
    - This is $O(n)$ instead of $O(n^2)$ as we said in vector.
    - Since C++20, you can also use `std::erase(_if)` for `remove + erase`.

```cpp
std::vector<int> v{ 0,1,1,4,5,3,3,3 };
auto it = std::unique(v.begin(), v.end());
v.erase(it, v.end());
std::println("{}", v);

auto it2 = std::remove_if(v.begin(), v.end(),
    [](const int ele) { return ele % 2 == 0; });
v.erase(it2, v.end());
std::println("{}", v);
```

C:\WINDOWS\system32\cmd.exe

```
[0, 1, 4, 5, 3]
[1, 5, 3]
```

Modifying

- You may notice that if the sequence is already unique, assignment will happen at every element…
  - In our example, there is a 1=1 if you've noticed.
  - This will happen only at the beginning of sequence that has already been unique, since checked value will be assigned to overwrite itself.
    - Once a non-unique position occurs, this will be broken since checked value stays at different position with current value.
  - So in MS-STL, there is a pre-loop to skip over the initial unique sequence.
  - Removal has similar optimization, i.e. skip over all non-removable elements.

Modifying

- `std::replace(begin, end, oldValue, newValue)`/`std::replace_if(begin, end, Pred, newValue)`: replace all values that equal to `oldValue`/ make `Pred` return `true` with `newValue`.
- `std::swap(x, y)`: swap two elements.
- `std::iter_swap(it1, it2)`: `std::swap(*it1, *it2)`.
- `std::swap_range(begin1, end1, begin2)`: swap two ranges one by one.
- `std::reverse(begin, end)`: reverse the range.
  - This is implemented by swap elements that one starts from head and the other starts from tail.
- `std::rotate(begin, mid, end)`: left rotate `[begin, mid)`.

| 2 | 3 | 4 | 4 | 5 |

  - If you want to right rotate, you can use reversed iterator.
  - A more intuitive way is "circular shift", which left shifts and fill dropped elements on the other side.
  - Here is `rotate(v.begin(), v.begin() + 3, v.end())`, which left shift `{2,3,4}` to another side.
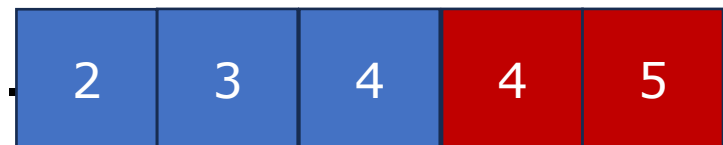
Modifying

- Rotate is also $O(n)$, and there are 3 typical implementations.
- We first think about brute-force method: swap each element to its position (just like bubble sort). This will be $O\big(k(n-k)\big)$, where k is `std::distance(begin, mid)`.
  - We can notice that to place the element in the right position, we move it one by one. So the simplest idea is just making it one-step!
  - Think about `arr[0]`; the next element that will occupy it is `arr[dis]`; to fill in `arr[dis]`, `arr[dis * 2]` is needed.
    - When `dis * n` exceeds size, modulus it (or for practical performance, `-= n` once exceeds it).
  - Once `(dis * n) % size` reaches 0, just use `arr[0]` to fill in it, and this circular group (循环群) is all in place.
    - The number of circular groups is congruence class of `mk % n`, which is `gcd(n,k)`.
    - Every element in every circular group is moved once, and gcd is $O(n)$, thus the total complexity is $O(n)$.

Notice that gcd has stricter complexity (i.e. $O(\log \frac{\min(n,k)}{\gcd(n,k)})$), but we don't cover them here.

Modifying

- The only problem of this algorithm is that it's cache-unfriendly, because we need to jump a lot.
- So in MS-STL, the algorithm is optimized by swapping all groups together.

Modifying

- When all groups are swapped to the end, the result is like:

Step3:Loop until Next==Last.Then keep Step1.

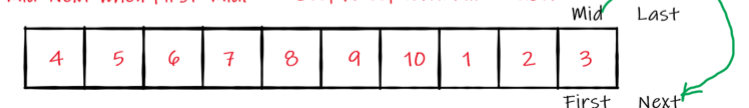|							Mid	Last
4	5	6	7	8	9	10	2	3	1

First⟶ Next ⟶
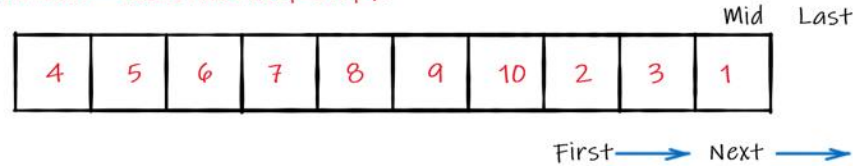
- The complexity is $\Theta(n - k)$.
- Then we need to adjust shifted elements to be in right place, i.e. circular shift the rest.
  - Now, we need `rotate(first, mid, last)`, so perform the loop before again.
    - i.e. $n' = k, k' = n - n \% k$; loop until $n \% k == 0$.
    - So the overall complexity is $\Theta(n - k) + \Theta(k - (n - n \% k)) + \cdots$, the next complexity will cancel the first $-k$, so the final complexity is $\Theta(n)$.
  - Practical solution (i.e. no recursion, loop by resetting the iterators):

Step4:Let Next=Mid when Next==Last.

								Mid	Last
4	5	6	7	8	9	10	1	3	2

First Next(New) Next(Old)
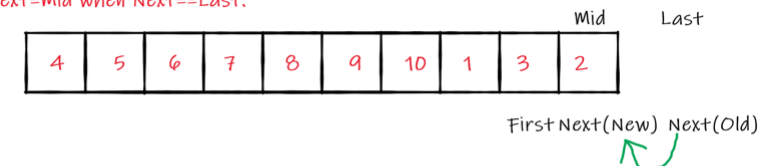
Step5:Let Mid=Next when First=Mid.   Step6:Loop until mid==Last.

								Mid	Last
4	5	6	7	8	9	10	1	2	3

First   Next

Modifying

- The third implementation is by reverse.
  - In fact, reverse is the basis of many algorithms.
  - [begin, mid-1] [mid, end-1].
  - Reverse two sub-sequences, get [mid-1, begin] [end-1, mid]
  - Reverse the total sequence, get [mid, end-1][begin, mid-1]
  - That's it; you rotate the sequence by reversing 3 times!
    - The complexity is $\Theta(k) + \Theta(n-k) + \Theta(n)$.
  - This is also more cache-friendly than the first method.
- In msvc, this is used in random-access iterator or bidirectional iterator; since this needs going back, forward iterator will not use it but the last implementation that we've mentioned.

Modifying

- `std::shift_left/right(begin, end, n)`: Since C++20; left/right shift `[begin, end)` by `n`;
  - Different from rotate since they're not "circular"; the dropped elements are permanently dropped (invalid).
  - `shift_left` returns the end of resulting range, `[it, end)` is invalid; `shift_right` returns the begin of resulting range , `[begin, it)` is invalid.

```cpp
std::vector<int> v{ 0,1,9,4,5,3,3,3 };
std::replace(v.begin(), v.end(), 3, 8);
std::println("{}", v);
std::replace_if(v.begin(), v.end(),
    [](const int ele) { return ele % 2 == 0; }, 7);
std::println("{}", v);
std::reverse(v.begin(), v.end());
std::println("{}", v);
std::rotate(v.begin(), v.begin() + 3, v.end());
std::println("{}", v);
```

```cpp
auto it = std::shift_left(v.begin(), v.end(), 2);
v.erase(it, v.end()); // erase invalid elements.
std::println("{}", v);
it = std::shift_right(v.begin(), v.end(), 3);
v.erase(v.begin(), it);
std::println("{}", v);
```

```
C:\WINDOWS\system32\cmd.exe

[0, 1, 9, 4, 5, 8, 8, 8]
[7, 1, 9, 7, 5, 7, 7, 7]
[7, 7, 7, 5, 7, 9, 1, 7]
[5, 7, 9, 1, 7, 7, 7, 7]
[9, 1, 7, 7, 7, 7]
[9, 1, 7]
```

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax
    - Permutation
  - Numeric algorithms
  - Parallel algorithms
  - Range-version algorithms

Copying

- You've known `std::copy` in stream iterator.
  - `std::copy(begin1, end1, dstBegin);`
    - Still pay attention to the length of destination.
  - `std::copy_if(begin1, end1, dstBegin[, Pred]);`
  - `std::copy_n(begin1, n, dstBegin);`
  - These three functions all copy forwards, i.e. from begin to end.
  - `std::copy_backward(begin1, end1, dstEnd);` copy things from `[begin1, end1)` to `[prev(dstEnd, distance), dstEnd)`, but **backwards**.
    - You may remember that insertion in vector can be implemented by `copy_backward`. Difference between forwards and backwards only happens when two ranges **overlap**.
  - Similarly, there are `std::move(_backward)`, which moves rather than copies elements. We'll mention them in *Move Semantics*.

Copying

- Finally, some modifying algorithms have a copy-version, i.e. the result range will be output to a new range instead of in-place.
  - The original range is unchanged.
  - They just add an output iterator after the end iterator...
  - std::remove_copy(begin1, end1, dstBegin, value).
  - std::remove_copy_if(begin1, end1, dstBegin, Pred).
  - std::unique_copy(begin1, end1, dstBegin[, Pred2]).
  - std::reverse_copy(begin1, end1, dstBegin).
  - std::rotate_copy(begin1, mid1, end1, dstBegin).
  - std::replace_copy(begin1, end1, dstBegin, oldValue, newValue).
  - std::replace_copy_if(begin1, end1, dstBegin, Pred, newValue).
  - We don't show how they're used; it's just adding an output destination.

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
- Partition and Sort
  - Heap
  - Set operations
  - MinMax

- Permutation
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Partition and Sort

- Partition is part of quick sort, and the provided algorithms are similar, so we put them together.
- For partition:
  - Partition denotes that a range is divided into two parts; assuming predicate function `Pred`, then there exists an iterator `it` (i.e. partition point) so that all elements in `[begin, it)` make `Pred` return `true` while `[it, end)` make `Pred` return `false`.
  - `std::is_partitioned(begin, end, Pred)`: return whether the range is a partition.
  - `std::partition(begin, end, Pred)`: rearrange a range to be a partition; return the partition point `it`.
    - Similarly, there is an `std::partition_copy`, which needs a true output iterator and false output iterator.
  - These two algorithms are all $O(N)$.

Partition and Sort
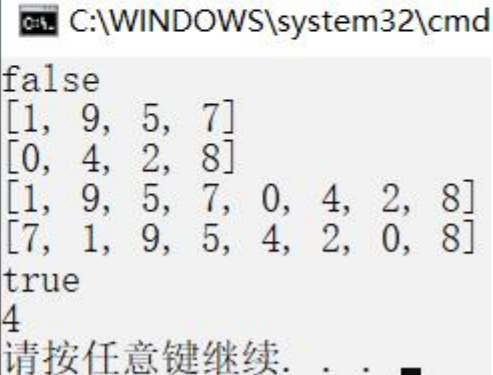
- `std::stable_partition(begin, end, Pred)`: similar to `std::partition`, but each sub-partition preserves the original order.
  - E.g. D C A B where A, C v.s. B, D, then the stable partition will be C A D B, so that C is still before A, and D is still before B, as the original order denotes.
  - There is no `std::stable_partition_copy`.
  - $O(N)$ space complexity, $O(N)$ time complexity.
    - If space is insufficient, the time complexity is $O(N \log N)$.
- `std::partition_point(begin, end, Pred)`: **assume the range is a partition**, find the partition point iterator (i.e. the first element of false range).
  - Notice that since it's already a partition, it's binary searched; $O(\log N)$ therefore.

Partition and Sort

- Notice that output of `partition` and `partition_copy` can be different!

```cpp
std::vector<int> v{ 0,1,9,4,5,2,7,8, }, v2(v.size()), v3(v.size()), v4(v);
auto isOdd = [](const int ele) { return ele % 2; };
std::cout << std::boolalpha;

std::cout << std::is_partitioned(v.begin(), v.end(), isOdd) << '\n';
auto [it1, it2] = std::partition_copy(v.begin(), v.end(), v2.begin(), v3.begin(), isOdd);
v2.erase(it1, v2.end()); v3.erase(it2, v3.end());
std::println("{}", v2); std::println("{}", v3);
std::stable_partition(v.begin(), v.end(), isOdd);
std::println("{}", v);
std::partition(v4.begin(), v4.end(), isOdd);
std::println("{}", v4);
std::cout << std::is_partitioned(v.begin(), v.end(), isOdd) << '\n'
<< std::partition_point(v.begin(), v.end(), isOdd) - v.begin() << '\n';
```

```
C:\WINDOWS\system32\cmd
false
[1, 9, 5, 7]
[0, 4, 2, 8]
[1, 9, 5, 7, 0, 4, 2, 8]
[7, 1, 9, 5, 4, 2, 0, 8]
true
4
请按任意键继续. . . _
```

Partition and Sort

- The implementation of stable partition is like:
  - When the memory is enough, prepare a buffer; move the `false` range to the buffer and move the `true` range to be consecutive (just like perform `std::remove_if()`, with removed range saved in buffer);
    - Then move the buffer elements back.
  - When the memory is not enough, divide the sequence into two halves and stable partition each half.
    - This will form `[true, false]`, `[true, false]` sequence.
    - Rotate the middle `[false, true]` so that the final result is totally partitioned.
    - $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$, so it's overall $O(n \log n)$.
      - You'll learn why in *Algorithm Design and Analysis*.

Partition and Sort

- For sort: there is always an optional predicate function `Pred2`, acts as `operator<`. The sorted result is ascending.
  - `std::is_sorted(begin, end[, Pred2])`: check whether a range is sorted.
  - `std::is_sorted_until(begin, end[, Pred2])`: return the iterator `it` so that `[begin, it)` is sorted.
  - These two are just checking whether it's ascending, so $O(N)$.
  - `std::sort(begin, end[, Pred2])`: sort the range.
    - $O(N \log N)$
  - `std::stable_sort(begin, end[, Pred2])`: sort the range; equal elements will preserve the original order.
    - $O(N)$ space complexity, $O(N \log N)$ time complexity; if space is insufficient, $O(N (\log N)^2)$ time complexity.

Partition and Sort

- Since `std::sort` requires the complexity exactly $O(N \log N)$, quick sort is not enough.
  - It's $O(N \log N)$ on average but $O(N^2)$ in the worst case!
- The widely-used algorithm in C++ standard library is *Introspective Sort* (abbr. *IntroSort*).
  - It integrates insertion sort, heap sort and quick sort.
  - When the element number is **low enough**, insertion sort is used.
    - We've learnt that its constant is really small so even if it's $O(N^2)$, it's faster than other algorithms.
  - When the recursion is too deep, heap sort is used.
    - This is to prevent the worst case.
    - Though heap sort is stably $O(N \log N)$, the constant is large due to random access when jumping from children to parent, so it's cache-unfriendly.
  - Otherwise quick sort.

Partition and Sort

- Briefly:

```
void IntroSort(It begin, It end, size_t depth)
{
    if(std::distance(begin, end) <=ISORT_MAX)
    {
        InsertionSort(begin, end);
        return;
    }
    if(depth == 0)
    {
        HeapSort(begin, end);
        return;
    }
    It pivot = ChoosePivot(begin, end);
    Partition(begin, pivot, end);
    IntroSort(begin, pivot, depth - 1);
    IntroSort(std::next(pivot), end, depth - 1);
}
```

- In MS-STL, ISORT_MAX = 32.

- However, to reduce the possibility of heap sort, it's still important to choose a good pivot.
  - It cannot be random since it will be hard for the user to debug.
    - E.g. when there is some memory leak in sorting, the user need deterministic behavior to watch the leak reason.
  - In GCC/Clang, it's chosen from {begin, mid, end - 1}.
    - Use the middle value as pivot.
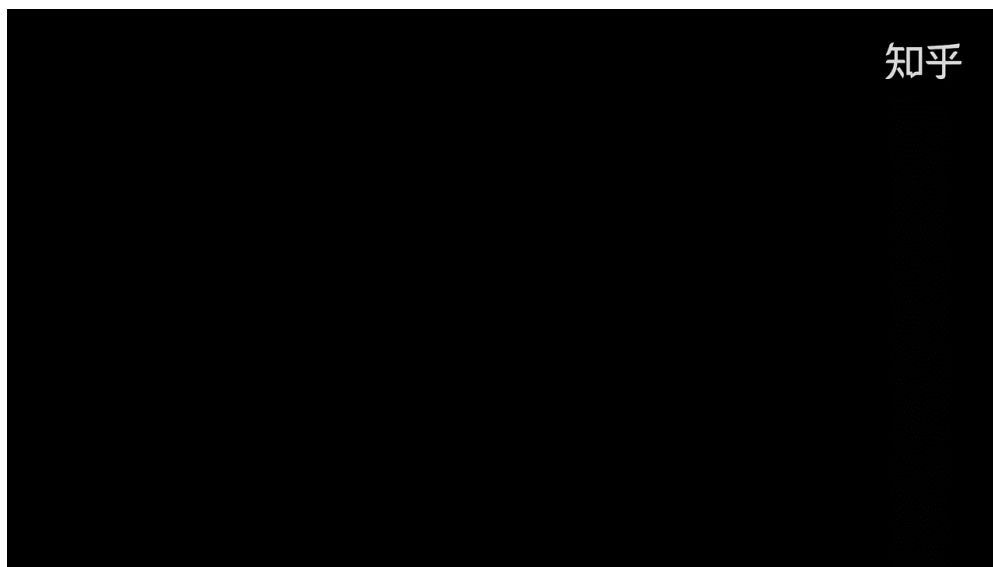
Partition and Sort

- We know that if we simply choose begin, then ascending sequence will reach $O(N^2)$.
  - i.e. every time partition will only drop one element.
- By choosing one in three, this case is eliminated.
  - However, another case called *pipe organ*(管风琴) will still fall in worst case!
  - Every time the pivot falls at either left or right, so that the partition is bad enough.
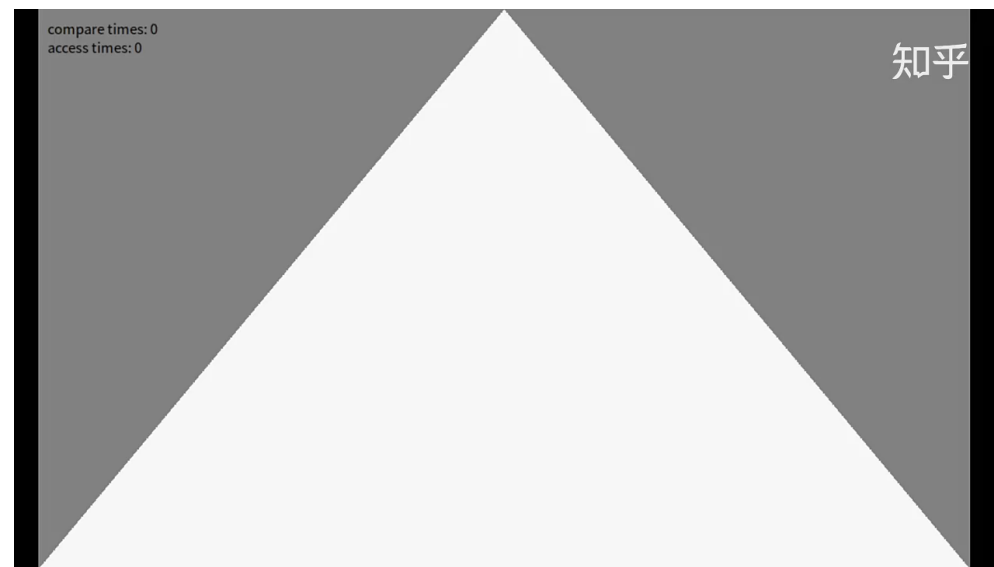 - Let's see an animation…
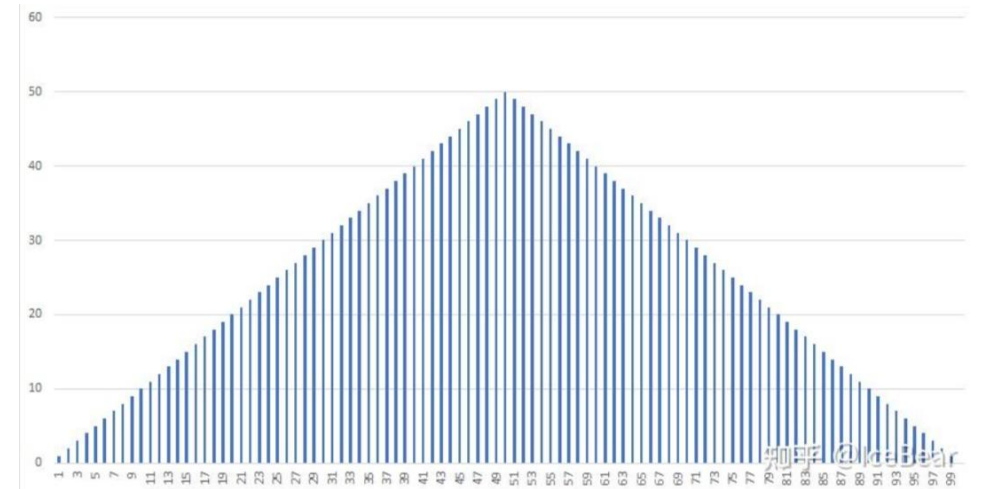
Credit: IceBear@zhihu.

Partition and Sort

Random sequence

Pipe organ sequence

compare times: 0
access times: 0

知乎

知乎

When the depth is too high, every local region will be heap sort.
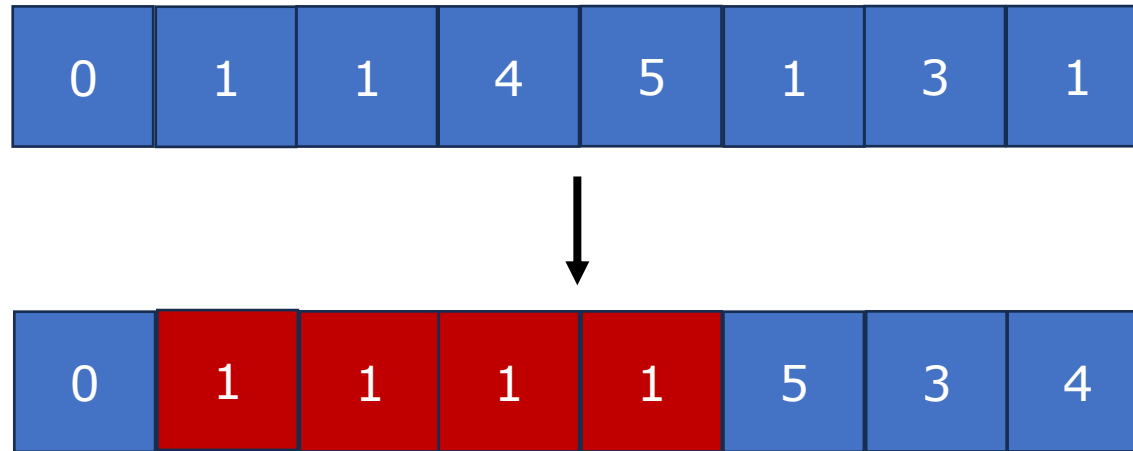
Partition and Sort

- In MS-STL, this is solved by:
  - 9 feasible pivots: when the element number is high enough, msvc will choose 1 in 9 instead of 1 in 3. It's even harder to fall in worst case.

```
// sort median element to middle
using _Diff        = _Iter_diff_t<_RanIt>;
const _Diff _Count = _Last - _First;
if (40 < _Count) { // Tukey's ninther
    const _Diff _Step     = (_Count + 1) >> 3; // +1 can't overflow because range was made inclusive in caller
    const _Diff _Two_step = _Step << 1; // note: intentionally discards low-order bit
    _Med3_unchecked(_First, _First + _Step, _First + _Two_step, _Pred);
    _Med3_unchecked(_Mid - _Step, _Mid, _Mid + _Step, _Pred);
    _Med3_unchecked(_Last - _Two_step, _Last - _Step, _Last, _Pred);
    _Med3_unchecked(_First + _Step, _Mid, _Last - _Step, _Pred);
} else {
    _Med3_unchecked(_First, _Mid, _Last, _Pred);
}
```

Partition and Sort

- MS-STL also optimizes partition.
  - When many elements are equal, $O(N)$ is basically enough.
  - So MS-STL will group elements that are equal to the pivot, and then sort the left/right respectively.
  - When all elements are equal, exactly $O(N)$.

0	1	1	4	5	1	3	1

0	1	1	1	1	5	3	4

Partition and Sort

- Procedures of partition are like:
  - Let `it0=pivot`, decrease & increase it so that `[PFirst, PLast)` is equal to pivot (P means partition).
  - Let `GFirst = PLast, GLast = PFirst`.
  - Increase `GFirst` when `*GFirst` is greater than `*PFirst`, which satisfies partition.
    - When they are equal, this means it's an equal element to pivot, and we need to swap `*GFirst` with `*PLast`, and increase `PLast`.
    - When less than, it should appear at left, so we break the loop.
  - Decrease `GLast` when `*GLast` is less than `*PFirst`, which satisfies partition; the sub-cases are similar as `GFirst`.
  - Finally swap `GFirst` and `GLast` since they're a pair of unsatisfying elements; swapping them will satisfy!
    - Corner case: if `GFirst` or `GLast` reaches `end` or `begin`, the equal range will be "shifted" so that the other side has space for partition.

Partition and Sort

- Corner case process:

```
if (_Glast == _First && _Gfirst == _Last) {
    return pair<_RanIt, _RanIt>(_Pfirst, _Plast);
}

if (_Glast == _First) { // no room at bottom, rotate pivot upward
    if (_Plast != _Gfirst) {
        _STD iter_swap(_Pfirst, _Plast);
    }

    ++_Plast;
    _STD iter_swap(_Pfirst, _Gfirst);
    ++_Pfirst;
    ++_Gfirst;
} else if (_Gfirst == _Last) { // no room at top, rotate pivot downward
    if (--_Glast != --_Pfirst) {
        _STD iter_swap(_Glast, _Pfirst);
    }

    _STD iter_swap(_Pfirst, --_Plast);
} else {
```

- Recursion limit is estimated by $1.5 \log N$ (until `_Ideal==0`).

```
_Ideal = (_Ideal >> 1) + (_Ideal >> 2); // allow 1.5 log2(N) divisions
```

- Finally, the quick sort is like:

```
for (;;) {
    if (_Mid.first - _First < _Last - _Mid.second) { // loop on second half
        _Sort_unchecked(_First, _Mid.first, _Ideal, _Pred);
        _First = _Mid.second;
    } else { // loop on first half
        _Sort_unchecked(_Mid.second, _Last, _Ideal, _Pred);
        _Last = _Mid.first;
    }
}
```

This seems like only sorting single side, but in fact it's double sides; you can think it yourself why.

Partition and Sort

- Particularly, in Go and Rust, unstable sort called Pattern-Defeating Quick sort (PDQsort) is implemented.
  - As its name, it tries to break patterns that cause worst cases in quick sort.
  - When the size of one side is too low, it's thought as "unbalanced", when the sequence will be "shuffled".
    - The shuffle is not random, just swaps elements at some distances.
    - This will cancel the worst pattern so that the worst case complexity is $O(N \log N)$.
  - A C++ implementation is here; benchmarked in my machine, it's generally equal to or faster than msvc's `std::sort`.
    - However, this implementation is also optimized by considering e.g. cache line size, which is unfair; but basically it should be faster even if they're disabled.

- For stable sort, it's usually merge sort when enough memory is available; otherwise divide and conquer until the space is enough.
  - When the elements are few, also insertion sort.

Partition and Sort

- `std::partial_sort(begin, mid, end)`: sort $[begin, end)$ so that $[begin,$ $mid)$ is sorted (i.e. same as the $[begin, mid)$ in the full sorted range).
  - $O(M \log N)$, usually heap sort.
  - Similarly, there is an `std::partial_sort_copy`.
- `std::nth_element(begin, mid, end)`: rearrange $[begin, end)$ so that `*mid` is sorted (i.e. same as the `*mid` in the full sorted range) and the whole range is partitioned by it.
  - The procedures are similar to quick sort.
    - Select a pivot.
    - Partition the range by the pivot.
    - If pivot is after mid, search in $[begin, pivot)$; else search in $[pivot, end)$.
    - When the length of sequence is low enough, just insertion sort.
  - $O(N)$; this is proved by Knuth in *Mathematical Analysis of Algorithms*.

```cpp
std::vector<int> v{ 0,1,9,4,5,2,7,8 }, v2(v);
std::partial_sort(v.begin(), v.begin() + 6, v.end());
std::println("{}", v);
std::nth_element(v2.begin(), v2.begin() + 2, v2.end());
std::cout << v2[2];

return 0;
```

C:\WINDOWS\system32\cmd.exe
```
[0, 1, 2, 4, 5, 7, 9, 8]
2请按任意键继续. . .
```

Partition and Sort

- `std::merge(begin1, end1, begin2, end2, dstBegin[, Pred2])`: merge two (sorted) ranges and output to `dstBegin`; just like merge in merge sort.
  - It's same as `std::list::merge`, except that it needs a destination.
  - $O(M + N)$; basically only meaningful for two sorted ranges though not force it.
- `std::inplace_merge(begin, mid, end)`: merge sorted `[begin, mid)` and `[mid, end)` to a sorted `[begin, end)`.
  - It's same as merge in merge sort, whose space complexity is $O(N)$ and time complexity is $O(N)$.
  - If the memory is not enough, $O(N \log N)$.
  - The implementation is like stable partition, don't cover here.

```cpp
std::vector<int> v{ 0,1,5,6,10,13 }, v2{ 2,3,4,9,12 }, v3(20);
auto it = std::merge(v.begin(), v.end(), v2.begin(), v2.end(), v3.begin());
v3.erase(it, v3.end()); // or you can v3(v1.size() + v2.size()) when resizing.
std::println("{}", v3);
auto total = stdv::concat(v, v2) | stdr::to<std::vector>();
std::inplace_merge(total.begin(), total.begin() + v.size(), total.end());
std::println("{}", total);

return 0;
```

```
[0, 1, 2, 3, 4, 5, 6, 9, 10, 12, 13]
[0, 1, 2, 3, 4, 5, 6, 9, 10, 12, 13]
```

Notice that `stdv` doesn't have `concat`; we use range-v3 in fact.

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
- Heap
  - Set operations
  - MinMax
  - Permutation
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Heap

- We've talked about heap in `priority_queue`.
  - Needs random-access iterator.
  - `a[i] <= a[(i – 1) / 2]`.
  - Percolate down/up.
- Methods are part of similar to sort; an optional `Pred2` acts as `operator<`.
  - `std::is_heap(_until)(begin, end[, Pred2])`.
  - `std::make_heap(begin, end[, Pred2])`: Floyd algorithm.
  - These two methods are $O(N)$.
  - `std::push_heap(begin, end[, Pred2])`: assuming that `[begin, prev(end))` is a heap; then see the last element(i.e. `prev(end)`) as the newly added element, percolate it up.
    - `[begin, end)` will be a new heap.

Heap

- `std::pop_heap(begin, end[, Pred2])`: assuming that `[begin, end)` is a heap; use the last element(i.e. `prev(end)`) to overwrite heap top(i.e. `begin`), and percolate it down.
  - `[begin, prev(end))` will be a new heap.
- These two methods are all $O(\log N)$.
- `std::sort_heap(begin, end[, Pred2])`: assuming that `[begin, end)` is a heap; sort it.
  - Theoretically, "heap" is better than an arbitrary range, which may be a hint.
  - Practically, this method is implemented as heap sort, which is $O(N \log N)$.
  - However, heap sort is slower than `std::sort`…
    - You may use it if $O(\log N)$ stack growth is unacceptable for you, since heap sort is $O(1)$.
    - It's also reported that `std::sort_heap` has smaller code size.

https://stackoverflow.com/questions/12786518/reason-for-existence-of-sort-heap

Heap

- Notice that you cannot assume popping will swap top and last element (you may see the last one as invalid after popping).

```
std::vector<int> v{ 1,7,4,3,2,9 };
std::make_heap(v.begin(), v.end());
std::println("{}", v);
std::pop_heap(v.begin(), v.end());
std::println("{}", v);
v.back() = 100;
std::push_heap(v.begin(), v.end());
std::println("{}", v);
std::sort_heap(v.begin(), v.end());
std::println("{}", v);
```

```
C:\WINDOWS\system3:
[9, 7, 4, 3, 2, 1]
[7, 3, 4, 1, 2, 9]
[100, 3, 7, 1, 2, 4]
[1, 2, 3, 4, 7, 100]
请按任意键继续. . .
```

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
- Set operations
  - MinMax

- Permutation
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Set operations

- Set operations are used on **sorted** range, including `set`.
  - The optional `Pred2` acts as `operator<`.
  - `std::includes(begin1, end1, begin2, end2[, Pred2])`: check whether the second range is subset of the first range;
    - Assume `it1` in the first range and `it2` in the second range.
    - If `*it2 < *it1`, then since both ranges are sorted, `*it2` will never exist in the first range, i.e. return `false`.
    - Otherwise, if `!(*it1 < *it2)`, then proceed `it2` by one.
      - This means `*it1 == *it2`, so try to find next.
    - Proceed `it1` by one, continue to loop.

Set operations

it1

1	1	4	5	8

1	4	8

it2

!(*it1 < *it2) &&
!(*it2 < *it1),
it1++, it2++.

Set operations

it1

1	1	4	5	8

1	4	8
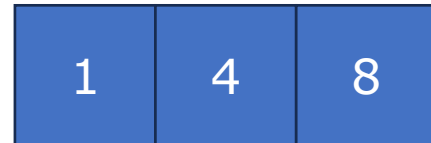
it2

*it1 < *it2, it1++

Set operations

it1

1	1	4	5	8

1	4	8

it2
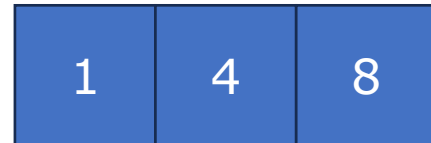
!(*it1 < *it2) &&
!(*it2 < *it1),
it1++, it2++.

Set operations

- We omit the following procedures; what if the second one is 2?

it1

1	1	4	5	8

1	2	8

*it2 < *it1, false!

it2

Set operations
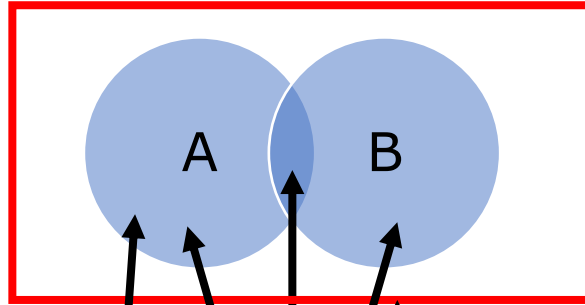
- std::set_intersection(begin1, end1, begin2, end2, dstBegin[, Pred2]): i.e. $A \cap B$.
- std::set_union(begin1, end1, begin2, end2, dstBegin[, Pred2]): i.e. $A \cup B$.
- std::set_symmetric_difference(begin1, end1, begin2, end2, dstBegin[, Pred2]): i.e. $A - B$.
- std::set_difference(begin1, end1, begin2, end2, dstBegin[, Pred2]): i.e. $A - (A \cap B)$.
- These algorithms are all $O(N)$, and the implementation is similar to std::includes; you may think their implementations yourself if you like.
  - So for multiset, intersection will preserve the less one, union will preserve the more one (this is still same as mathematical definition).

Set operations

- Here we use ranges-version algorithms since they're much easier here.
- You'll fully understand it in the sections later.

```cpp
template<typename Func>
void Test(std::vector<int>& v1, std::vector<int>& v2, const Func& set_op)
{
    std::vector<int> v3(v1.size() + v2.size());
    auto it3 = set_op(v1, v2, v3.begin()).out;
    v3.erase(it3, v3.end());
    std::println("{}", v3);
}

int main()
{
    std::vector<int> v{ 1,1,4,5,5,5,5,6 }, v2{ 2,3,5,5,6 };
    Test(v, v2, stdr::set_union);
    Test(v, v2, stdr::set_intersection);
    Test(v, v2, stdr::set_symmetric_difference);
    Test(v, v2, stdr::set_difference);
    Test(v2, v, stdr::set_difference);
    return 0;
}
```

```
[1, 1, 2, 3, 4, 5, 5, 5, 5, 6]
[5, 5, 6]
[1, 1, 2, 3, 4, 5, 5]
[1, 1, 4, 5, 5]
[2, 3]
```

Algorithms

- Algorithms
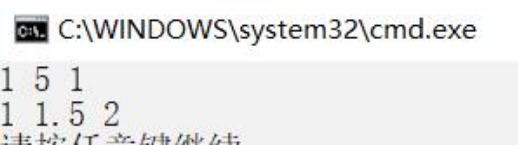 - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
    - Permutation
  - Numeric algorithms
  - Parallel algorithms
  - Range-version algorithms
- MinMax

MinMax

```cpp
std::vector<int> v{ 1, 5, 3, 2, 4 };
auto it1 = std::min_element(v.begin(), v.end()),
     it2 = std::max_element(v.begin(), v.end());
std::cout << *it1 << ' ' << *it2 << ' ' << std::min({ 1,5,3,2,4 }) << '\n'
    << std::clamp(0.5f, 1.0f, 2.0f) << ' '
    << std::clamp(1.5f, 1.0f, 2.0f) << ' '
    << std::clamp(2.5f, 1.0f, 2.0f) << '\n';
```

C:\WINDOWS\system32\cmd.exe

```
1 5 1
1 1.5 2
请按任意键继续
```

- Pred2 acts as operator<.
  - std::min/max/minmax(a, b[, Pred2]): return (pair of) reference of the smaller/bigger element.
    - You can also pass a initializer list, so that it will return the minimum/maximum element in the list. But it returns value instead of reference.
    - If they're equivalent, a / [a, b] will be returned.
    - Ranges-version can pass any range, which is much more convenient…
  - std::min_element/max_element/minmax_element(begin, end[, Pred2]): return the iterator of the minimum/maximum value in the range.
    - The comparison complexity of minmax_element is $\Theta(\frac{3}{2}n)$ rather than $\Theta(2n)$ as you may think. You'll learn it in *Algorithm Design and Analysis* in detail, so we don't tell you here (it's clever but rather easy once you know it!).
  - std::clamp(value, low, high): return value if it's within [low, high]. If value is less than low, return low; otherwise return high.
    - This is usually used in deep learning, e.g. to limit gradient.

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax

- Permutation
  - Numeric algorithms
  - Parallel algorithms
  - Range-version algorithms

Permutation

- Permutation means that two sequence are **unorderly equal**.
  - E.g. {1,2,3} and {2,1,3}.
  - `std::is_permutation(begin1, end1, begin2[, Pred2])`: Judge whether two ranges are premutation. `Pred2` acts as `operator==`.
    - Similar to `equal` and `mismatch`, `end2` can be provided since C++14.
    - $O(N^2)$; at best $O(N)$ if they are strictly equal.
  - `std::prev/next_permutation(begin, end[, Pred2])`: turn the sequence to the previous/next permutation.
    - For example, {1, 2, 3} have 6 permutations, and can be sorted by lexicographical comparison.
      - i.e. {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}.
      - The next premutation of {1, 3, 2} is {2, 1, 3}.
    - If it's the first/last permutation, prev/next will return `false`. So you can use `while(next_permutation)` to iterate over all permutations (but notice that the start point should be sorted one, so that it's the first in all permutations).
    - $O(N)$.

Permutation

- The implementation of `std::is_permutation` is simple; it will first check whether two sequences are at the same length.
  - It they're, then skip equal values at the beginning.
  - For the remaining values, loop over every element:
    - If it's not the first element in all equivalents, skip it. This will use `std::find` so it's $O(n)$.
    - Otherwise it's the first element, so we count the number of the equivalent in two ranges. This will use `std::count` so it's $O(n)$.
      - If their counts are different, return `false`.
      - So `std::find` is in fact to prevent redundant counting.
  - So it's totally $O(n^2)$.
  - Notice that this can be used for forward iterators; `std::next/prev_permutation` can only be applied on bidirectional iterators.

Permutation


```cpp
int main()
{
    std::vector<int> v{ 1, 2, 2, 3 }, v2{ 3,2,1,2 };
    std::cout <<std::is_permutation(v.begin(), v.end(),
        v2.begin(), v2.end()) << '\n';
    while(std::next_permutation(v.begin(), v.end()))
        std::println("{}", v);
    return 0;
```

```
C:\WINDOW
1
[1,  2,  3,  2]
[1,  3,  2,  2]
[2,  1,  2,  3]
[2,  1,  3,  2]
[2,  2,  1,  3]
[2,  2,  3,  1]
[2,  3,  1,  2]
[2,  3,  2,  1]
[3,  1,  2,  2]
[3,  2,  1,  2]
[3,  2,  2,  1]
```
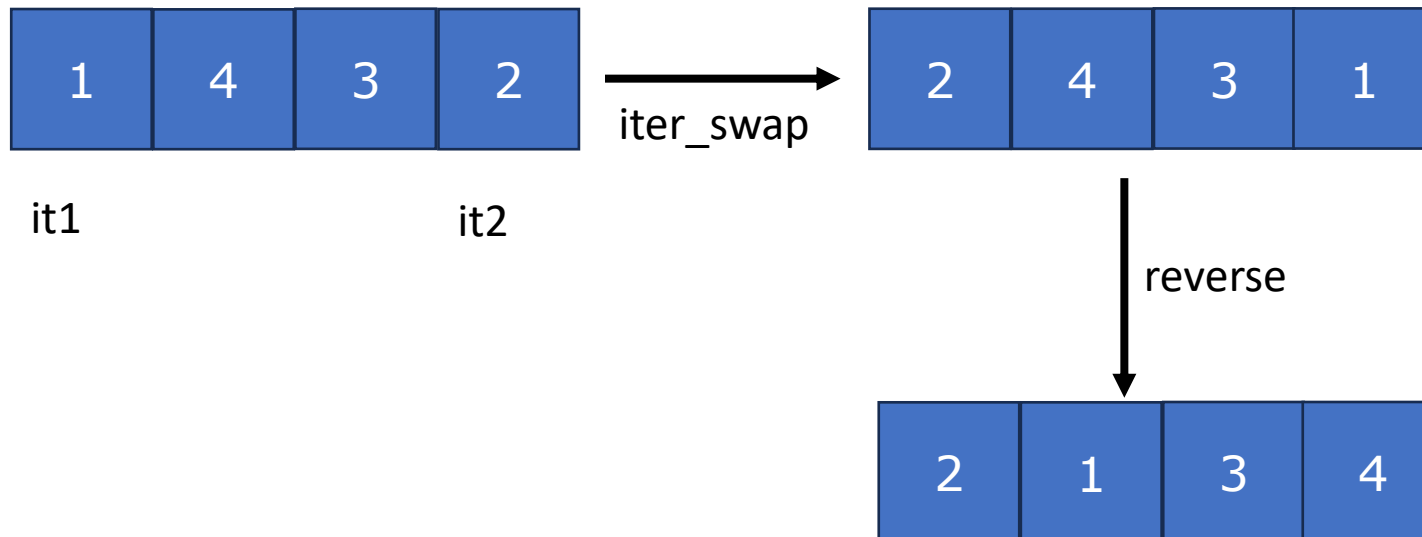
- For the implementation of `std::next/prev_permutation` (`next` as example):
  - It will find rightmost element smaller than its next element, say `it1`.
    - If `begin` doesn't match, pure descending, return `false`.
  - Then find the rightmost element that is not smaller than `it1`, say `it2`.
  - `std::iter_swap(it1, it2)`, and `std::reverse(std::next(it1), end)`.

1	4	3	2

it1   it2

→ iter_swap →

2	4	3	1

↓ reverse

2	1	3	4

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax

  - Permutation
- Numeric algorithms
- Parallel algorithms
- Range-version algorithms

Numeric algorithms

- They are all defined in `<numeric>`.
- As its name, these algorithms are frequently used in the field of numeric calculation, so they're solely split out.
  - They are all $O(N)$.
  - Usually you may get a value or output sequence.
- For the most basic ones:
  - `std::iota(begin, end, beginVal)`: fill in `[begin, end)` with `{beginVal, ++beginVal, …}`.
    - This can be used for any type that supports `operator++`.
  - `std::adjacent_difference(begin, end, dstBegin, Op = std::minus)`: as its name, output `{ val[0], val[1] – val[0], val[2] – val[1], …}`.
    - `–` can be substituted with `Op`.

Numeric algorithms

- `std::accumulate(begin, end, initVal, Op = std::plus)`: accumulate all values, return $initVal + \sum_{begin}^{end} val$.
  - `+` can be substituted with `Op`, e.g. `std::multiply` will get product.
- `std::partial_sum(begin, end, dstBegin, Op = std::plus)`: as its name, output `{ val[0], val[0] + val[1], val[0] + val[1] + val[2], …}`.
  - Return end iterator of output sequence.
  - Partial sum is really useful in range sum query, since `Sum[k2] – Sum[k1]` will get the original sum from `k1` to `k2`.
- `std::inner_product(begin1, end1, begin2, initVal, Op1 = std::plus, Op2 = std::multiplies)`: as its name, finally get `initVal + a[0] * b[0] + a[1] * b[1] + …`.

- These three algorithms are guaranteed to happen from first to last.
  - However, if we consider parallelism, many operations can be done independently and finally be collected.
  - So, there are versions for out-of-order operations since C++17.

Numeric algorithms

- `std::reduce(begin, end, initVal, Op = std::plus)`: same as accumulate.
- `std::inclusive_scan(begin, end, dstBegin, Op = std::plus[, initVal])`: same as partial sum, but can provide a `initVal` so that the sequence is `{initVal + val[0], initVal + val[0]+ val[1], …}`.
- `std::transform_reduce(begin1, end1, begin2, initVal, ReduceOp = std::plus, BiTransformOp = std::multiplies)`: same as inner product.

- Their orders are not guaranteed, so possible optimizations are allowed.
  - They all have parallel-version ones, which will be covered later.

- Beyond them, there are also some other unordered operations:
  - `std::transform_reduce(begin, end, initVal, ReduceOp, UnaryTransformOp)`: the previous one is $Init\ ROp\ BiOp(x_1, y_1)\ ROp\ BiOp(…) …$, this one is $Init\ ROp\ UnaryOp(x_1)\ ROp\ UnaryOp(…) …$.
    - No default operation parameter.

Numeric algorithms

- `std::exclusive_scan(begin, end, dstBegin, `**`initVal,`**` Op = std::plus)`: similar to partial sum, but exclude the element itself, i.e. the sequence is `{initVal, initVal + val[0], …, initVal + val[0] + … + val[n-2]}`.
  - Notice that since exclusive, the `initVal` is necessary, so it's not provided as the last optional parameter like `inclusive_scan`.
- `std::transform_inclusive_scan(begin, end, dstBegin, Op, UnaryTransformOp[, initVal])`
  - Equivalent to apply `UnaryTransformOp` to every element, and use `Op` to `inclusive_scan`.
- `std::transform_exclusive_scan(begin, end, dstBegin, initVal, Op, UnaryTransformOp)`
  - Equivalent to apply `UnaryTransformOp` to every element, and use `Op` to `exclusive_scan`.
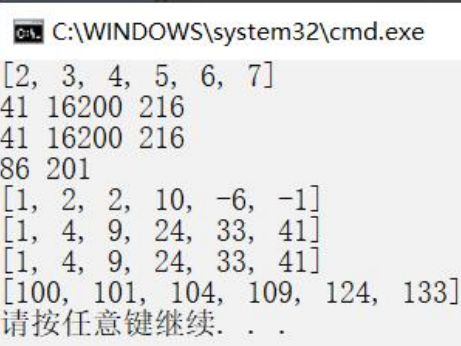 - `initVal` will not be transformed.

Numeric algorithms

```cpp
std::vector<int> v{ 1,3,5,15,9,8 }, v2(v.size()), dst(v.size());
std::iota(v2.begin(), v2.end(), 2);
std::println("{}", v2);
std::cout << std::accumulate(v.begin(), v.end(), 0) << ' ' <<
    std::accumulate(v.begin(), v.end(), 1, std::multiplies<>{}) << ' '
    << std::inner_product(v.begin(), v.end(), v2.begin(), 0) << '\n';

std::cout << std::reduce(v.begin(), v.end(), 0) << ' ' <<
    std::reduce(v.begin(), v.end(), 1, std::multiplies<>{}) << ' '
    << std::transform_reduce(v.begin(), v.end(), v2.begin(), 0) << '\n';

std::cout << std::transform_reduce(v.begin(), v.end(), v2.begin(), 0, // binary
    std::plus<>{}, [](const int ele1, const int ele2) { return ele1 + ele2 + 3; })
    << ' ' << std::transform_reduce(v.begin(), v.end(), 100, std::plus<>{},
        [](const int ele1) { return ele1 + 10; }) << '\n'; // unary

std::adjacent_difference(v.begin(), v.end(), dst.begin());
std::println("{}", dst);
std::partial_sum(v.begin(), v.end(), dst.begin());
std::println("{}", dst);
std::inclusive_scan(v.begin(), v.end(), dst.begin());
std::println("{}", dst);
std::exclusive_scan(v.begin(), v.end(), dst.begin(), 100);
std::println("{}", dst);
```

```
C:\WINDOWS\system32\cmd.exe
[2, 3, 4, 5, 6, 7]
41 16200 216
41 16200 216
86 201
[1, 2, 2, 10, -6, -1]
[1, 4, 9, 24, 33, 41]
[1, 4, 9, 24, 33, 41]
[100, 101, 104, 109, 124, 133]
请按任意键继续. . .
```

Numeric algorithms

- There are also three more algorithms defined in `<numeric>`:
  - `std::gcd(a, b)`: since C++17, return the *greatest common divisor* of `abs(a)` and `abs(b)`, where `a` and `b` should be non-bool integer.
    - Particularly, if `a == 0 && b == 0`, return 0; if only one of them is 0, return the other.
  - `std::lcm(a, b)`: since C++17, return the *least common multiple* of `abs(a)` and `abs(b)`, where `a` and `b` should be non-bool integer.
    - Particularly, if `a == 0 || b == 0`, return 0.
  - If `abs(result)` cannot be represented by the bigger type of a and b, UB.
    - E.g. `gcd(INT_MIN, 3)` is UB since `abs(INT_MIN)` is not representable by `int`.
    - `lcm` is more likely to exceed the limit, so pay attention.
  - `std::midpoint(a, b)`: since C++20, return `a + (b - a) / 2`.
    - Accept arithmetic types (except for bool) and pointer type.

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax

  - Permutation
- Numeric algorithms
- **Parallel algorithms**
- Range-version algorithms

Parallel algorithms

- As you can observe, many algorithms can utilize parallelism!

- There are two kinds of parallelism (in a single machine):
  - SIMD: *single instruction multiple data*, or ***vectorization***, e.g. AVX you've heard of in ICS.
    - For example, AVX can operate a 256-bit long register, which can load 4 double; it also provides instructions that operate these doubles simultaneously.
  - SIMT: *single instruction multiple threads*, or more exactly ***multithreading***[1], i.e. you use many threads to divide the problem and use every thread to solve a sub-problem.

- For example, `std::fill` a 1G array can obviously be parallelized.
  - For SIMD, you can e.g. fill in 32 bytes rather than only 8 bytes in one instruction, which theoretically reduce the problem to 256M.
  - For SIMT, you can e.g. use 4 threads to do together, so every thread just need to do 256M/4=64M work.
  - Anyway, you will get 16x times acceleration by parallelism!

1: SIMT is not same as multithreading, and is in fact stricter (which can also be deduced from the name). This is widely used in GPU and you may learn it yourself in the future.

Parallel algorithms

- Since C++17, parallel-version algorithms are added.
- To give parallel hint, you need constants defined in `<execution>`.
    - There are four execution policies defined in `std::execution`.
    - `seq`: sequenced policy, i.e. the algorithm cannot be parallelized.
    - `par`: parallel policy, i.e. the algorithm can utilize multithreading.
        - This requires that if multiple threads simultaneously execute the operation, the result is still correct.
    - `unseq`: unsequenced policy (since C++20), i.e. the algorithm can utilize SIMD.
        - This requires that if the operation is executed simultaneously in a single thread, the result is still correct.
    - `par_unseq`: parallel and unsequenced policy, i.e. the algorithm can utilize both multithreading and SIMD.

Parallel algorithms

- For example:

```cpp
int x = 0;
std::mutex m;
void f() {
    int a[] = {1,2};
    std::for_each(std::execution::par_unseq,
                  std::begin(a), std::end(a),
                  [&](int) {
                      m.lock();
                      ++x;
                      m.unlock();
                  });
}
```

- This is vectorization-unsafe but parallel-safe since one thread cannot lock the mutex for multiple times, but multiple threads can be prevented from data race by mutex.

- Or from another angle of view, we can discuss from data dependencies.
  - Of course, when data operation doesn't depends on each other (e.g. all elements `+= 1`), the operation can be `par_unseq`.

Parallel algorithms

- There are for kinds of data dependencies:
  - RAR: read after read, i.e. no data dependency. You can read a value for any times in any order. You can just use `par_unseq`.
    - E.g. `std::adjacent_find`, never try to write.
  - RAW: read after write, e.g. `a[j] = a[j - 1]+1`. You cannot parallelize it since e.g. `a[500]` needs `a[499]`, so chunk division is impossible; also, you cannot vectorize it.
  - WAR: write after read, e.g. `a[j-1] = a[j] + 1`. Also, you cannot parallelize it since `a[499]` needs `a[500]`, thus you have to wait until `a[499]` reads the value. However, vectorization is possible since you can read e.g. `a[1]…a[8]`, operate them(i.e. `+ 1`), and write back to `a[0]…a[7]`. Sometimes you can just use `unseq`, but some may still not e.g. add a `B[j] = A[j] *2`.
  - WAW: write after write, e.g. `a[j-1] = a[k]; a[j] = a[k] + 1;`. The final result is indeterministic if they interleave.

  Credit: https://d3f8ykwhia686p.cloudfront.net/1live/intel/CompilerAutovectorizationGuide.pdf

Parallel algorithms

- However, by adding locks, parallelism is still possible.
  - But you need to design dedicatedly since if the critical section is too large, it's just same as sequential since only one thread will work.
  - Another way is by *atomic variables*, which will be covered in the future.
- Anyway, personally the most frequently used one is `par_unseq` for me, and parallelism is usually designed by manually creating threads and designating works since parallel algorithms doesn't specify e.g. the thread number, which is too coarse-grained.
- Now let's talked about parallel algorithms.
  - You need to specify the execution policy as the first parameter.
  - We will only introduce the algorithms that change complexity; all algorithms can use `par_unseq` unless the predicate function/transformation function/… has dependencies (e.g. generate a Fibonacci sequence).

Parallel algorithms

- Parallelism itself doesn't change the big-O complexity (it just reduces the constant); however, some algorithms cannot be directly parallelized, so the algorithm itself is changed and then the complexity is also changed.
  - We'll not cover the parallel algorithms; you may learn it yourself if you're interested in e.g. HPC.
  - For `std::partition, std::nth_element`, the complexity is $O(N \log N)$ (but `==` will only use $O(N)$).
  - For `std::merge`, the sequential version is exactly $N - 1$ while parallel one is only guaranteed to be $O(N)$.
  - For `std::inplace_merge`, the sequential version is only $O(N \log N)$ when the memory is insufficient; parallel version is always $O(N \log N)$.
  - Since $O(\log N)$ is very small, the reduced constant will still accelerate the algorithm (profile anyway).

Parallel algorithms

- There are also some algorithms that cannot be parallelized.
  - Those who only operate on several values, e.g. `std::min/max/minmax/clamp`, `std::swap/iter_swap`.
  - Random algorithms, which aren't covered in this lecture.
  - `std::search` with a searcher; you can make your searcher parallel instead.
  - `std::push/pop/make/sort_heap`, obviously nothing to parallelize.
  - Permutation algorithms.
  - Binary search algorithm (including `std::partition_point`), and `copy/move_backward()` (use reversed iterators instead).
  - Ordered numeric algorithms, i.e. `std::iota/accumulate/inner_product/partial_sum`.

- Also, all parallel algorithms need at least forward iterators (even policy is `seq`!).

Parallel algorithms

- When the memory is not enough to prepare (e.g. create threads), the parallel algorithm will `throw std::bad_alloc`;
  - If the function body (e.g. the transform function) itself throws an exception, the program will be terminated (even policy is `seq`!).
- Note:16x acceleration that we've said is just theoretical result.
  - Due to the limitation of architecture, the acceleration rate is usually sub-linear, which will be less than 16 times.
  - For example, creating threads has overhead.
    - Thus, when the data amount is very small (e.g. size < 1024), usually parallel algorithms will just do things in one thread.
  - Besides, when the optimization level is high, the compiler will enable vectorization automatically.
    - So when transform function is cheap, it's also possibly not suitable.
  - Also, if the system doesn't support parallelism or vectorization, they will be silently turned to sequential one.

Anyway, profile it!

Parallel algorithms

- An example of parallel sort:
  - The first is debug mode, and the second is release mode.
  - We generate random numbers and time the procedure, which will be covered in the future courses.
    - If you want to test something before we learn them, you can just use framework similar to this piece.

Parallel algorithms

- Final note:
  - libc++ doesn't support parallel algorithms yet, though it's already 7 years later.
  - libstdc++ requires you to use TBB to enable parallel algorithms, otherwise still sequential.
    - That is, you need to install TBB and –I Your/TBB/Path –ltbb.
      - This will be easier if you use CMake/XMake…

Algorithms

- Algorithms
  - Search
  - Comparison
  - Counting
  - Generating and Transforming
  - Modifying
  - Copying
  - Partition and Sort
  - Heap
  - Set operations
  - MinMax
    - Permutation
- Numeric algorithms
- Parallel algorithms
- **Range-version algorithms**

Range-version algorithms

- We need to specify iterator pair in algorithms…
  - However, we usually just operate on the whole container, i.e. the iterator pair is `begin` and `end`.
  - Why not just specify a range?
- So, range-version algorithms (also called *constrained algorithms*) are introduced, with more flexible operations.
  - Defined in `std::ranges`, e.g. `stdr::sort(vec)` is OK!
  - You can additionally specify ***projection*** as the last parameter, i.e. transformation of elements before entering the real function.
    - This just changes criteria, the element itself is unchanged.
    - Multiple ranges may specify multiple projections.
  - The destination iterator still needs an iterator, since the original one is not an iterator pair too.

Range-version algorithms

- For example:

```
std::vector vec { -1, 2, -3, 4, -5, 6 };
auto print = [](int i) { std::cout << i << ", "; };
std::cout << "with abs() projection: \n";
std::ranges::sort(vec, {}, [](int i) { return std::abs(i); });
std::ranges::for_each(vec, print);

with abs() projection:
-1, 2, -3, 4, -5, 6,
```

- The second {} means using the default comparator, i.e. stdr::less{}.
  - Comparison functors also have stdr version.

Range-version algorithms

- Range-version algorithms also have these advantages:
  - It uses **_concept_** in C++20, which will produce more human-friendly error if your iterator doesn't satisfy the requirement of the algorithm.
  - The range doesn't need to be common (i.e. begin/end with the same type).
  - It's more secure, since if you pass a temporary container, `stdr::dangling` will be returned, and using it will cause compile error.
    - Views to underlying temporary container are forbidden too, e.g. `std::vector{1, 2, 3} | std::take(2)`.
  - Some algorithms are reinforced, e.g. `stdr::min/max` accept a total range and return `T`, instead of just two values or an `initializer_list`.
  - New methods are added.
  - You can use template argument to accept them directly.
  - It also has some ADL-related advantages, but we don't cover them since ADL is not part of our course.

Range-version algorithms

- However, there are also some defects.
  - They cannot utilize parallel algorithms.
  - Some algorithms are weakened, e.g. `std::search` cannot use searcher.
  - Their return values are not same as `std::`, so migration is not direct.
    - But they will provide all useful information, so it may not be defects.
    - `stdr::remove/unique` will return removed range (i.e. `[newEnd, end)`).
    - For those who return an iterator to **denote several ranges**, there are mainly four kinds of return type:
      - `stdr::in_in_result<I1,I2>`: have `.in1`, `.in2`, which means the result or end iterator of two ranges; used by `std::swap_ranges` and `std::mismatch`.
      - `stdr::in_out_result<I,O>`: have `.in`, `.out`, which means end iterators of two ranges. This is only used in algorithms with only one input range and an output range, which returns `{input_end, output_end}`.
        - `stdr::transform` that use unary transformation is also in this case.
        - Particularly, `stdr::set_difference` will only return the iterator of the first range though it accepts two ranges.

Range-version algorithms

- `stdr::in_in_out_result<I1,I2,O>`: have `.in1`, `.in2`, `.out`, which means end iterators of three ranges. This is widely used in algorithms with two input ranges and an output range which returns `{input1_end, input2_end, output_end}`.
  - This is used in `stdr::set_union/intersection/symmetric_difference`, `stdr::merge` and binary `stdr::transform`.
- `stdr::in_out_out_result<I1,I2,O>`: have `.in`, `.out1`, `.out2`, which means end iterators of three ranges. This is only used in `stdr::partition_copy`, which needs two output iterator for true/false ranges.

- There are also some special forms for return value.
  - `stdr::in_found_result<I>`: have `.in`, `.found`, with the second a `bool`; only used in `stdr::next/prev_permutation`.
  - `stdr::min_max_result<I>`: have `.min`, `.max`; only used in `stdr::minmax(_element)`, the former returns values, the latter iterators.

- Notice that `struct` can also use structure binding.

Set operations

- So now you can understand what we show in set operations!
  - 1. They can be passed by template parameter directly.
    - For `std::`, they are template functions, which is template rather than a normal function!
      - They need more complex form to be passed in…
    - You may think them as a functor with `operator()`.
  - 2. .out is used to grasp result.
  - 3. a range is passed directly instead of iterator pair.

```cpp
template<typename Func>
void Test(std::vector<int>& v1, std::vector<int>& v2, const Func& set_op)
{
    std::vector<int> v3(v1.size() + v2.size());
    auto it3 = set_op(v1, v2, v3.begin()).out;
    v3.erase(it3, v3.end());
    std::println("{}", v3);
}

int main()
{
    std::vector<int> v{ 1,1,4,5,5,5,5,6 }, v2{ 2,3,5,5,6 };
    Test(v, v2, stdr::set_union);
    Test(v, v2, stdr::set_intersection);
    Test(v, v2, stdr::set_symmetric_difference);
    Test(v, v2, stdr::set_difference);
    Test(v2, v, stdr::set_difference);
    return 0;
}
```

```
[1, 1, 2, 3, 4, 5, 5, 5, 5, 6]
[5, 5, 6]
[1, 1, 2, 3, 4, 5, 5]
[1, 1, 4, 5, 5]
[2, 3]
```

Range-version algorithms

- For numeric algorithms, only two are provided since C++23.
  - `std::iota` is `stdr::iota`, but the latter returns `stdr::out_value_result`.
  - `std::accumulate` is `stdr::fold_left(R, init, Op)` defined in `<algorithm>`.
    - You can also use `stdr::fold_left_first(R, Op)`, which will use the first element as `init`. It will return `std::optional<T>` since the first element may not exist.
      - We'll cover `std::optional` in *Error Handling*.
    - You can also proceed from right to left, i.e. `stdr::fold_right_last(R, Op)` or `stdr::fold_right(R, init, Op)`, but it requires bidirectional range.
    - There are also `stdr::fold_left_(first_)with_iter`, which will return `stdr::in_value_result`(i.e.{end, result}).
    - These methods don't exist in `std::`!
- There are also some other methods that only exist in `stdr::` since C++23:

Range-version algorithms

- `stdr::contains(R, val[, Proj])`: return whether `val` exists in the range `R`.
- `stdr::contains_subrange(R1, R2[, Proj1[, Proj2]])`: return whether `R2` exists in the range `R1`. This is $O(N)$, but cannot specify searcher.
- `stdr::starts_with/ends_with(R1, R2[, Proj1[, Proj2]])`: return whether `R1` starts with/end with `R2`. $O(N)$.
- `stdr::find_last((_if)_not)`: equals to `find_first` with reversed iterator.

- Iterator-related methods also have ranges-version:
  - You can optionally provide a bound as the last parameter for `advance/next/prev`, so that the iterator will never exceed it. This is safer if you are not sure how large is `n`.
    - E.g. `stdr::next(vec.begin(), 100, vec.end())` will return `vec.end()` if `vec.size() < 100`.
    - They also provide an overload `(it0, bound)`, which equals to `for(auto it = it0; it != bound; it++)`.
  - For `distance`, you can directly provide a range rather than two iterators too.

Summary

- Ranges
  - Range adaptors, the meaning of iterated value.
  - Some caveats on ranges.
- Generator
- Function
  - Pointer to members.
  - `std::function`.
    - `std::reference_wrapper`.
    - Prefer lambda to bind!
  - Transparent operators.
  - Supplement on lambda.

- Algorithms
- Numeric algorithms
- Parallel algorithms
- Ranges-version algorithms

Next lecture…

- We'll talk about lifetime and type security.
  - The former is the usual mistake for novices…
    - We've also mentioned in this course that you need to pay special attention to referenced variables, no matter captured by lambda or `std::ref` by bind.
  - The latter is also important, but may even be neglected by the professional and produce many UBs.