The Ranges Library

The hello world of ranges.

```
std::vector<int> numbers = {1, 2, 3, 4, 5, 6};
auto results = numbers | std::views::filter([](int n){ return n % 2 == 0;})
                       | std::views::transform([](int n){ return n * 2;});


for (auto v: results) std::cout << v << " ";
```

rangesFilterTransform.cpp

The Ranges Library

The Ranges Library extend the classical STL with three new features:

- Algorithms can
  1. operate direct on the container.
  2. operate on infinite data streams.
  3. be composed with the | operator.

➡ The ranges library enables programming in a functional style.

The Ranges Library

- Operate directly on the container

```
std::array<int, 6> rang{3, 1, 4, 1, 5, 9};
std::ranges::reverse_view revRang{rang};
for (int i : revRange) std::cout << i << ' ';

std::map<std::string, int> freqWord{ {"witch", 25},
      {"wizard", 33}, {"tale", 45}, {"dog", 4}, {"cat", 3} };
auto names = std::views::keys(freqWord);
for (const auto& n : names){ std::cout << n << " "; };
```

rangesEntireContainer.cpp

The Ranges Library

- Lazy evaluation

```
std::vector<int> vec;
for (int i: std::views::iota(0) | std::views::take(5)) {
    vec.push_back(i);
}
```

- Function composition

```
std::map<std::string, int> pa{ {"witch", 25}, {"wizard", 33},
    {"tale", 45}, {"dog", 4}, {"cat", 34}, {"fish", 23} };
 for (const auto& n : std::views::keys(pa) | std::views::reverse) {
    std::cout << n << " ";
 };
```

rangesLazy.cpp
rangesComposition.cpp

The Ranges Library

- More Details: ranges overload of `std::ranges::sort`

```
template <std::random_access_iterator I, std::sentinel_for<I> S,
          class Comp = ranges::less, class Proj = std::identity>
requires std::sortable<I, Comp, Proj>
constexpr I sort(I first, S last, Comp comp = {}, Proj proj = {});
```

  - Comparators: `Comp`
  - Projections: `Proj`
  - Sentinel: `std::sentinel_for<I>`
  - Concepts: `std::random_access_iterator`, `std::sortable<I, Comp, Proj>`, `std::sentinel_for<I>`

The Ranges Library

- A projection is a mapping of a set into a subset. By default, the projection is the identitiy.

```
struct PhoneBookEntry{
    std::string name;
    int number;
};

std::vector<PhoneBookEntry> phoneBook{ {"Brown", 111}, {"Smith", 444},
    {"Grimm", 666}, {"Butcher", 222}, {"Taylor", 555}, {"Wilson", 333} };

std::ranges::sort(phoneBook, {}, &PhoneBookEntry::name);
std::ranges::sort(phoneBook, std::ranges::greater(), &PhoneBookEntry::number);
```

rangeProjection.cpp

The Ranges Library

`std::range`

- is a set of elements that can be iterated over.
- has a begin iterator and a sentinel element.

- The container of the Standard Template Library (STL) are ranges.

Concept (`std::ranges::` missing)	Additional Interface	Container (`std::` missing)

`input_range`	`++It, It++, *It`
	`It == It2, It != It2`
		`unordered_set, unordered_map`
		`unordered_multiset, unordered_multimap`
		`forward_list`

`bidirectional_range`	`--It, It--`	`set, map, multiset, multimap` `list`

`random_access_range`	`It[i]`
	`It += n, It -= n`
	`It + n, It -n`
	`n + It`
	`It - It2`
	`It < It2, It <= It2,`
	`It < It2, It >= It2`
		`deque`

`contigious_range`		`array, vector, string`

The Ranges Library

A sentinel specifies the end of a range. End iterators are sentinels.

```
struct Space {
    bool operator== (auto pos) const { return *pos == ' '; }
};
const char* rainerGrimm = "Rainer Grimm";
std::ranges::for_each(rainerGrimm, Space{}, [] (char c) { std::cout << c; });


struct NegativeNumber {
    bool operator== (auto num) const { return *num < 0; }
};
std::vector<int> myVec{5, 10, 33, -5, 10};
std::ranges::transform(std::begin(myVec), NegativeNumber{},
                       std::begin(myVec), [](auto num) { return num * num; });
```

sentinel.cpp

The Ranges Library

A view

- is a lightweight range.
- has no data.
- has constant time copy, move, or assign operations.
- can be composed and is lazy.

```
std::vector<int> vec;
for (int i: std::views::iota(0) | std::views::take(5)) {
    vec.push_back(i);
}
```

The Ranges Library

View	Description
`std::all_view, std::views::all`	Takes all elements
`std::ref_view`	Takes all elements from another view
`std::filter_view, std::views::filter`	Takes all elements that satisfy the predicate
`std::transform_view, std::views::transform`	Transforms all elements
`std::take_view, std::views::take`	Takes the first N elements of another view
`std::take_while_view, std::views::take_while`	Takes the elements of another view as long as these elements satisfy the predicate
`std::drop_view, std::views::drop`	Ignores the first N elements of another view
`std::drop_while_view, std::views::drop_while`	Ignores the elements of another view as long as these elements do not satisfy the predicate

The Ranges Library

View	Description
`std::join_view, std::views::join`	Joins a view of ranges
`std::split_view, std::views::split`	Splits a view using a separator
`std::common_view, std::views::common`	Converts a view into a `std::common_range`
`std::reverse_view, std::views::reverse`	Iterates in reverse order
`std::basic_istream_view, std::views::istream_view`	Applies >> to the view
`std::elements_view, std::views::elements`	Creates a view on the n-th element of tuples
`std::keys_view, std::views::keys`	Creates a view on the first element of pairs
`std::values_view, std::views::values`	Creates a view on the second element of pairs

The Ranges Library

Criteria	STL Algorithms	Ranges Algorithms
STL libraries `algorithm`, `memory`, and `numeric`	All supported	`numeric` missing
Concepts support	No	Yes
Unified lookup rules	No	Yes
Safety	No	Yes
Execution policy	Yes	No

sortList.cpp
begin.cpp
rangeAccess.cpp

C++20

- [Modernes C++ Blog]
- [C++20: Get the Details]