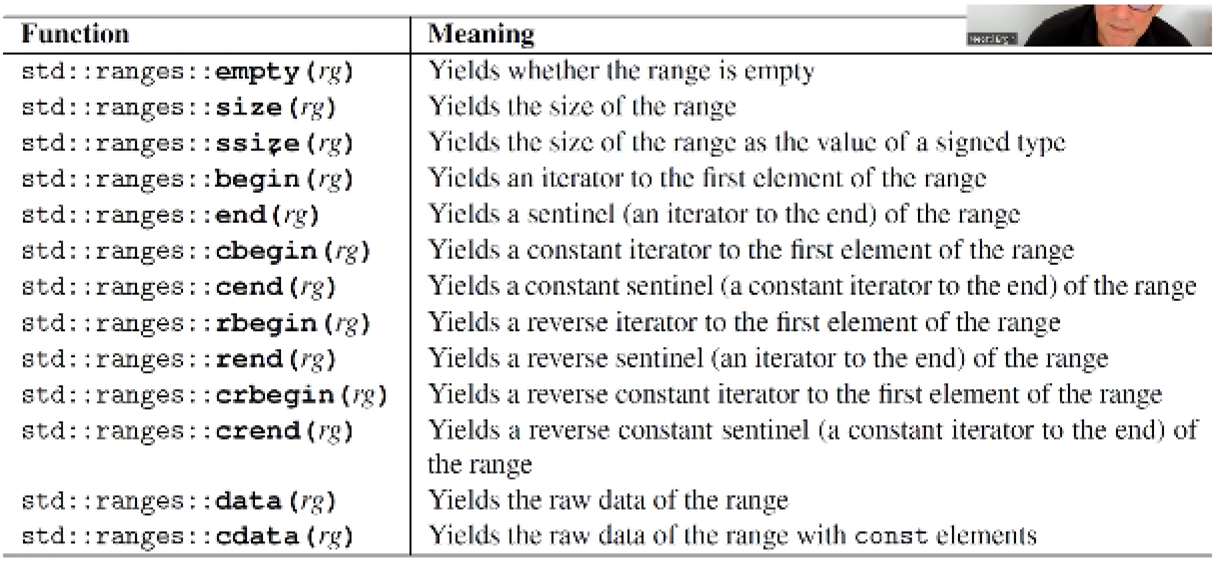
13.Hafta

> 22_16_09_2023

İçindekiler

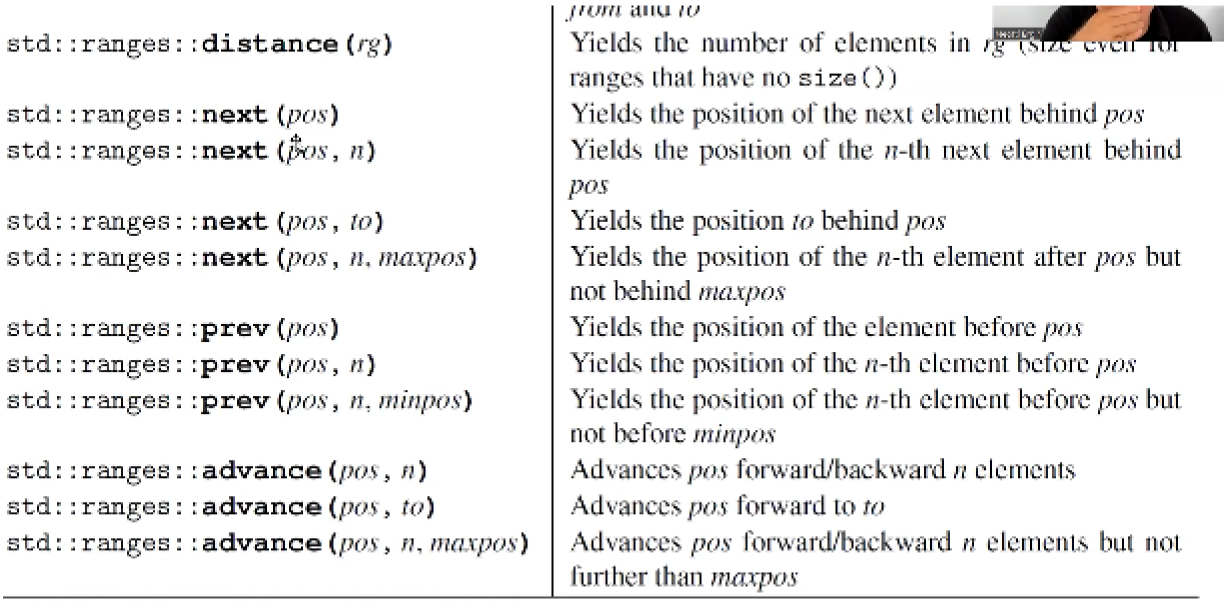
Ranges Kütüphanesi

Generic fonksiyonlar

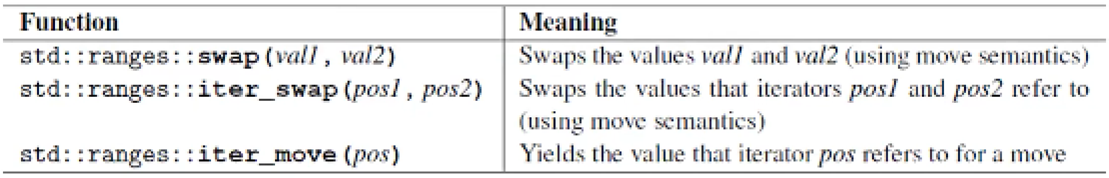


Function	Meaning
`std::ranges::empty(rg)`	Yields whether the range is empty
`std::ranges::size(rg)`	Yields the size of the range
`std::ranges::ssize(rg)`	Yields the size of the range as the value of a signed type
`std::ranges::begin(rg)`	Yields an iterator to the first element of the range
`std::ranges::end(rg)`	Yields a sentinel (an iterator to the end) of the range
`std::ranges::cbegin(rg)`	Yields a constant iterator to the first element of the range
`std::ranges::cend(rg)`	Yields a constant sentinel (a constant iterator to the end) of the range
`std::ranges::rbegin(rg)`	Yields a reverse iterator to the first element of the range
`std::ranges::rend(rg)`	Yields a reverse sentinel (an iterator to the end) of the range
`std::ranges::crbegin(rg)`	Yields a reverse constant iterator to the first element of the range
`std::ranges::crend(rg)`	Yields a reverse constant sentinel (a constant iterator to the end) of the range
`std::ranges::data(rg)`	Yields the raw data of the range
`std::ranges::cdata(rg)`	Yields the raw data of the range with `const` elements

Table 7.3. Generic functions for dealing with the elements of ranges

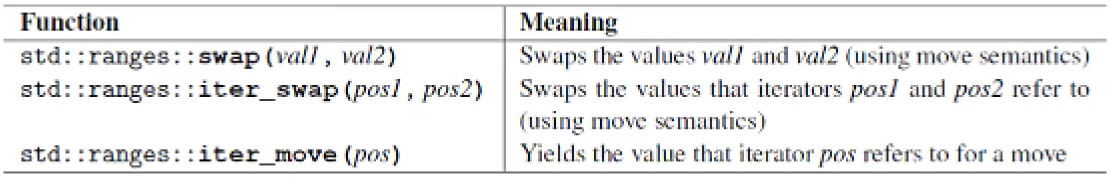


	from and *to*
`std::ranges::distance(rg)`	Yields the number of elements in *rg* (size even for ranges that have no `size()`)
`std::ranges::next(pos)`	Yields the position of the next element behind *pos*
`std::ranges::next(pos, n)`	Yields the position of the *n*-th next element behind *pos*
`std::ranges::next(pos, to)`	Yields the position *to* behind *pos*
`std::ranges::next(pos, n, maxpos)`	Yields the position of the *n*-th element after *pos* but not behind *maxpos*
`std::ranges::prev(pos)`	Yields the position of the element before *pos*
`std::ranges::prev(pos, n)`	Yields the position of the *n*-th element before *pos*
`std::ranges::prev(pos, n, minpos)`	Yields the position of the *n*-th element before *pos* but not before *minpos*
`std::ranges::advance(pos, n)`	Advances *pos* forward/backward *n* elements
`std::ranges::advance(pos, to)`	Advances *pos* forward to *to*
`std::ranges::advance(pos, n, maxpos)`	Advances *pos* forward/backward *n* elements but not further than *maxpos*



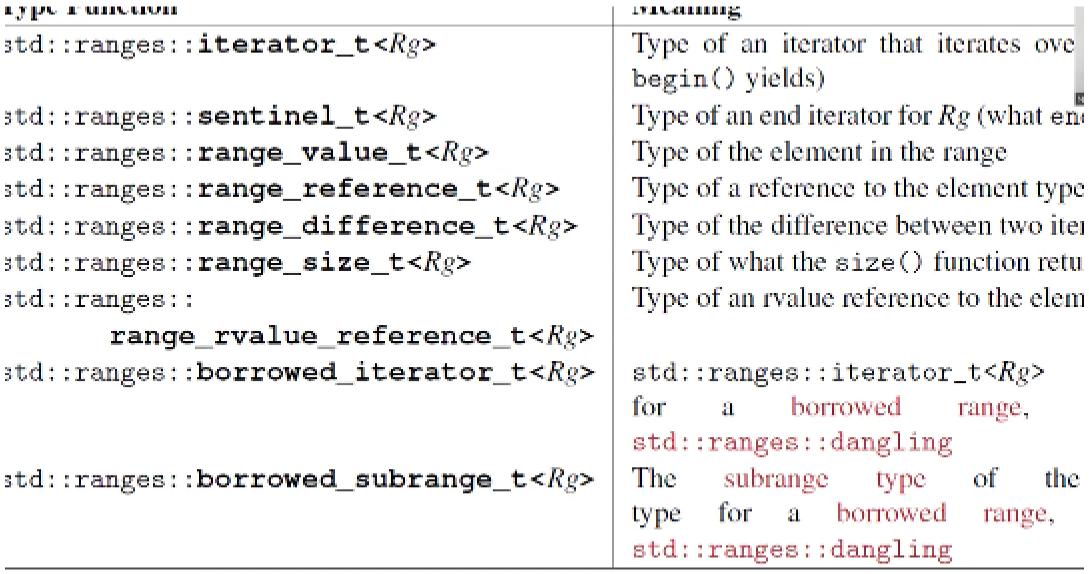
Function	Meaning
`std::ranges::swap(val1, val2)`	Swaps the values *val1* and *val2* (using move semantics)
`std::ranges::iter_swap(pos1, pos2)`	Swaps the values that iterators *pos1* and *pos2* refer to (using move semantics)
`std::ranges::iter_move(pos)`	Yields the value that iterator *pos* refers to for a move

Table 7.5. Generic functions for swapping and moving elements/values



Function	Meaning
std::ranges::swap(*val1*, *val2*)	Swaps the values *val1* and *val2* (using move semantics)
std::ranges::iter_swap(*pos1*, *pos2*)	Swaps the values that iterators *pos1* and *pos2* refer to (using move semantics)
std::ranges::iter_move(*pos*)	Yields the value that iterator *pos* refers to for a move

Table 7.5. Generic functions for swapping and moving elements/values



Type Function	Meaning
std::ranges::iterator_t<*Rg*>	Type of an iterator that iterates ove begin() yields)
std::ranges::sentinel_t<*Rg*>	Type of an end iterator for *Rg* (what en
std::ranges::range_value_t<*Rg*>	Type of the element in the range
std::ranges::range_reference_t<*Rg*>	Type of a reference to the element type
std::ranges::range_difference_t<*Rg*>	Type of the difference between two ite
std::ranges::range_size_t<*Rg*>	Type of what the size() function retu
std::ranges::range_rvalue_reference_t<*Rg*>	Type of an rvalue reference to the elem
std::ranges::borrowed_iterator_t<*Rg*>	std::ranges::iterator_t<*Rg*> for a borrowed range, std::ranges::dangling
std::ranges::borrowed_subrange_t<*Rg*>	The subrange type of the type for a borrowed range, std::ranges::dangling

Table 7.7. Generic functions that yield the types involved when using ranges

```
void print(std::string_view sv, auto beg, auto end)
{
    std::cout << msg;
    for(auto pos = beg; pos != end; ++pos)
        std::cout << ' ' << *pos;
    std::cout << '\n';
}

int main()
{
    std::vector inCall{1,2 ,3, 4, 5, 6, 7, 8, 9, 10};
    std::vector outCall{1,2 ,3, 4, 5, 6, 7, 8, 9, 10};
    auto result= std::ranges::transform(inCall, outCall.begin(), [](int x)
{return x * x;});

    print("inCall : ", inCall.begin(), inCall.end());
    print("outCall : ", outCall.begin(), outCall.end());
}
```

format kütüphanesi ile de kullanabiliyoruz, bu özellik C++23 ile geld. Her view bir range fakat her range bir view değil. Belirli concept özellikleri sağlayan rangeler view olabilir. Begin ve end bilgilerinin aynı türden olmayabilir. Bir çok range'i iki kere dolaşmak gerekmiyor. Var olan fonksiyonlar, begin ve end'in aynı türden olduğu varsayılarak tasarlanmış

- view'ların kopyalanması ve taşınması çoğunlukla constant time.
-

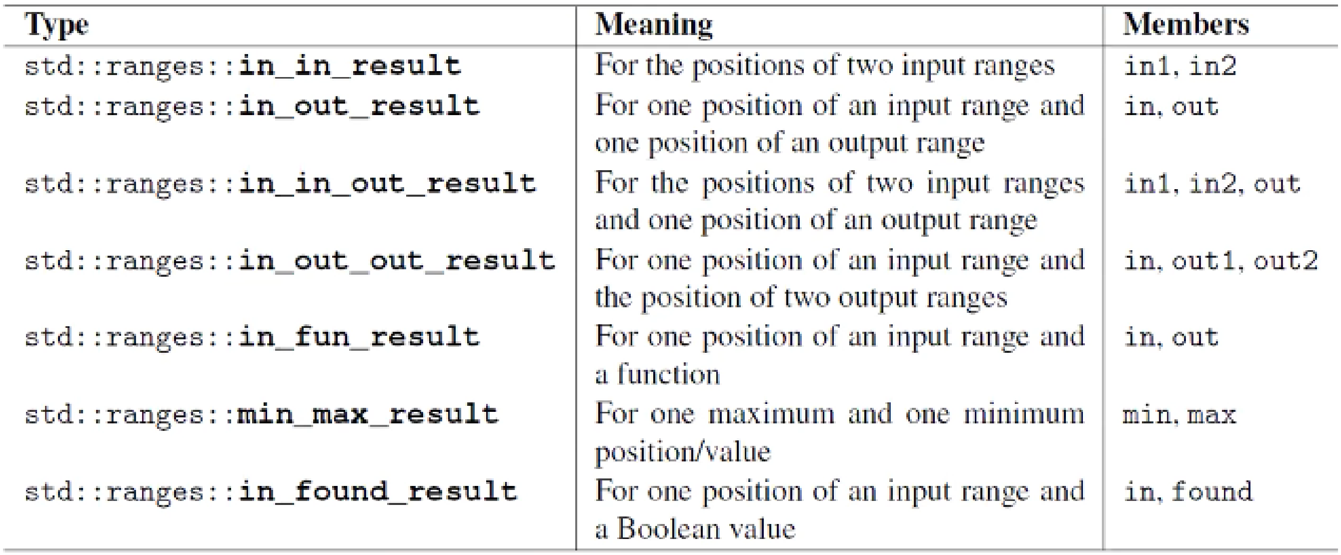
```
int main()
{
    using namespace std;
    vector<int> ivec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto vw = vec | views::filter([](int x){return x% 2 == 0;});

    cout << format("{}", vw);
}
```

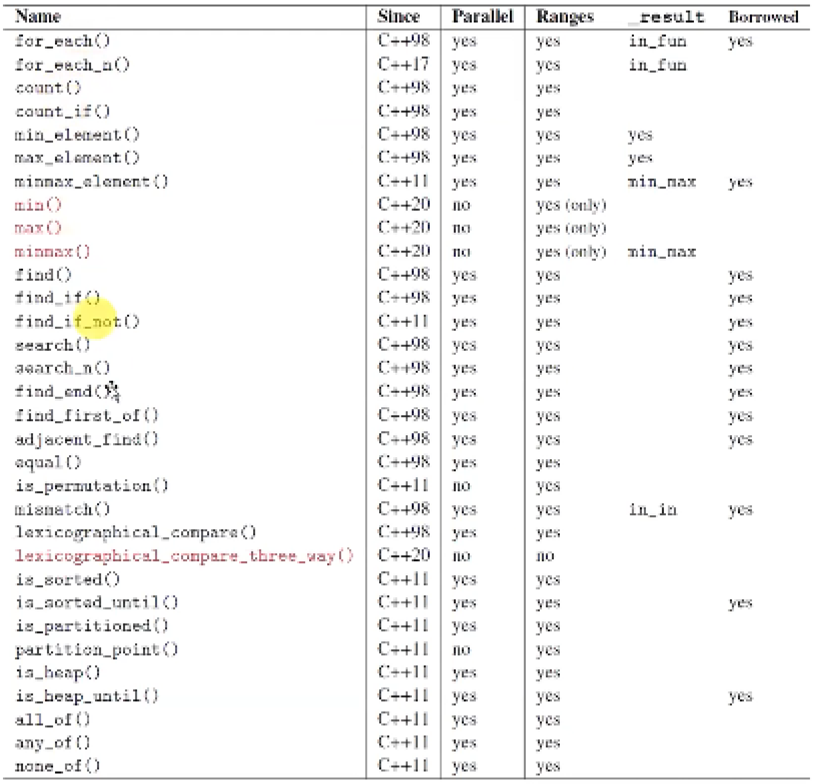
- yazma algoritmaları her zaman yazdığı konumdan bir sonraki konuma döndürüyor.

```
int main()
{
    vector<string> svec;
    rfill(svec, 10, rname);
    ranges::copy(svec, ostream_iterator<string>{cout, " "});
    vector<size_t> destvec(100);
    transform(svec.begin(), svec.end(), destvec.begin(), [](const string
&s){return s.size();});
    transform(svec.begin(), svec.end(), svec.begin(), [](const string &s)
{return s+"can";});//şeklinde svec'i de değiştirebiliriz.
    cout << "distance = "<< distance(destvec.begin)
}
```



Type	Meaning	Members
std::ranges::**in_in_result**	For the positions of two input ranges	in1, in2
std::ranges::**in_out_result**	For one position of an input range and one position of an output range	in, out
std::ranges::**in_in_out_result**	For the positions of two input ranges and one position of an output range	in1, in2, out
std::ranges::**in_out_out_result**	For one position of an input range and the position of two output ranges	in, out1, out2
std::ranges::**in_fun_result**	For one position of an input range and a function	in, out
std::ranges::**min_max_result**	For one maximum and one minimum position/value	min, max
std::ranges::**in_found_result**	For one position of an input range and a Boolean value	in, found

burada `in_fun_result` geri dönüş değeri in ve fun döndürüyor.



Name	Since	Parallel	Ranges	_result	Borrowed
for_each()	C++98	yes	yes	in_fun	yes
for_each_n()	C++17	yes	yes	in_fun	
count()	C++98	yes	yes		
count_if()	C++98	yes	yes		
min_element()	C++98	yes	yes	yes	
max_element()	C++98	yes	yes	yes	
minmax_element()	C++11	yes	yes	min_max	yes
min()	C++20	no	yes (only)		
max()	C++20	no	yes (only)		
minmax()	C++20	no	yes (only)	min_max	
find()	C++98	yes	yes		yes
find_if()	C++98	yes	yes		yes
find_if_not()	C++11	yes	yes		yes
search()	C++98	yes	yes		yes
search_n()	C++98	yes	yes		yes
find_end()	C++98	yes	yes		yes
find_first_of()	C++98	yes	yes		yes
adjacent_find()	C++98	yes	yes		yes
equal()	C++98	yes	yes		
is_permutation()	C++11	no	yes		
mismatch()	C++98	yes	yes	in_in	yes
lexicographical_compare()	C++98	yes	yes		
lexicographical_compare_three_way()	C++20	no	no		
is_sorted()	C++11	yes	yes		
is_sorted_until()	C++11	yes	yes		yes
is_partitioned()	C++11	yes	yes		
partition_point()	C++11	no	yes		
is_heap()	C++11	yes	yes		
is_heap_until()	C++11	yes	yes		yes
all_of()	C++11	yes	yes		
any_of()	C++11	yes	yes		
none_of()	C++11	yes	yes		

Table 7.12. Non-modifying algorithms

Views

- Bir range'i view haline getirebiliriz bunu adaptör ve factory ile yapabiliriz. Bir range argüman verilyor ve geri dönüş değeri olarak bir view döndürüyor.

Adaptörler

Bir source range'i alıp, bize view özelliğinde bir range döndürüyor.

```
int main()
{
    uısing namespace std;
    vector<int> ivec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto vw = views::take(vec,4); //take'in dönüşünün de bir range var ve bunu doğrudan kullanabiliriz. Fakat
    //
}
```

- Templeta oldukları için argümana bağlı olarak farklı tür döndürebilirler.
- Her adaptörün döndürdüğü range aynı özelliklere sahip olmayabilir.
- view olup olmadığını aşağıdaki gibi kontrol edebiliriz.

```
int main()
{
    using namespace std;
    vector<int> ivec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto vw = vec | views::filter([](int x){return x% 2 == 0;});
    cout <<"sizeof vw: "<<  sizeof(vw);
    static_assert(ranges::view<decltype(vw)>);
}
```

Lazy evaluation

Burada bir uyarlama yapılmasına rağmen, bu uyarlama biz bur öğe almak istediğimizde yapılıyor.

```
int main()
{
    using namespace std;
    vector<int> ivec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    auto filter = std::views::filter(ivec, [](int x){
        std::cout << "deneme"<< "\n";
        return x % 2 == 0;});
        //bu fonksiyon çağırılmadı

        auto iter = fiter.begin();
        //fonksiyon burada çağırılıyor.
}
```

implementasyonda, taban sınıf olarak kullanılan bir base-class var.

- CRTP kullanılarak yapılmış.

Bunlar composible, buradan elde edilen range başka bir range adaptörüne argüman olarak verilebilir. Bu şekilde kopyalama yapılmadan, bir range üzerinde bir çok işlem yapılabilir.

```
int main()
{
    using namespace std;
    vector<int> ivec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> dest;
    auto vw = vec | views::filter([](int x){return x% 2 == 0;}) |
views::transform([](int x){return x * x;});
    cout << format("{}", vw);
}
```

ya da,

```
int main()
{
    using namespace std;
    vector<int> ivec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    vector<int> dest;
    for(auto val : vw::transform(vw::filetr(vec,[](int x){return x % 2 == 0;}), [](int x){return x * x;}))
        cout << val << " ";
}
```

- Adaptör range'i alıp range veriyor, factory ise range almadan bir range döndürüyor.
- View'ların çoğu referans semantiği ile çalışıyor. View'in sarmaladığı range açısından sürekli referans semantiği kullanılıyor.
- Range'leri c++23 ile artık container'a dönüştürebiliyoruz.
- Bir range'in size() fonksiyonu varsa bu range ilgili concepti doğruluyor.
- Normalde sized-range olmayan bir containeri da böyle kullanabiliriz.

```
int main()
{
    using namespace std;
    namespace vw = std::views;
    namespace rng = std::ranges;

    for (  auto i : vw::iota(10) | vw::take(5) )
        cout << i << " ";
    vector<int> ivec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    std::ranges::subrange sbvec{next(ivec.begin()), prev(ivec.end())};
    constexpr bool b1 = std::ranges::sized_rangez<decltype(sbvec)>;
    std::cout << b1 << "\n";
    std::list<int> ilist{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    //std::ranges::subrange sblist{(ilist.begin()), (ilist.end(), ilist.size()}; sized rnage now
    std::ranges::subrange sblist{next(ilist.begin()), prev(ilist.end())}; //not a sized range
    costexpr bool b2 = std::ranges::sized_rangez<decltype(sblist)>;
    std::cout << b2 << "\n"; // false

}
```

```
template <std::ranges::input_range Range>
std::ranges::range_value_t<Range> get_min(Range &&rng)
{
    if(ranges::empty(rng))
        throw std::invalid_argument{"range is empty"};
    auto pos = std::ranges::begin(rng);
    auto min = *pos;
    while(++pos != std::ranges::end(rng))
    {
        if(*pos < min)
            min = *pos;
    }
    return min;
}
```

`common` fonksiyonu elimizde common range olmayan bir durum varsa bu durumda fonksiyon bize common range dödnürüyor.

`Subrange` bize range oluşturan bir adaptör

`auto vw = std::views::common(std::ranges::subrange(vec.begin(), vec.end()));` şeklinde kullanılabilir.

- Standart kullanılan Sentinal türü

```
template <auto ENDVAL>
struct Sentinel{
    bool operator==(auto pos)const
    {
        return pos == ENDVAL;
    }
};
```

```
template <std::ranges::random_access_range Range>
auto left_half(Range r)
{
    return std::ranges::subrange(std::begin(r), std::begin(r) +
std::rarnges::sizes(r)/2);
}

template <std::ranges::random_access_range Range>
auto right_half(Range r)
{
    return std::ranges::subrange(std::begin(r)+std::ranges::size(r),
std::end(r));
}
```

- 23_17_09_2023 tarihli ders tekrar izle