Range & View concepts & Range adaptors

```
template< class T >
concept range = requires( T& t ) {
    ranges::begin(t); // equality-p
    ranges::end (t);
};
```

Defined in header <ranges>

```
template<class T>
concept view = ranges::range<T> && std::movable<T> && ranges::enable_view<T>;
```
(1) (since C++20)

```
template<class T>
constexpr bool enable_view =
    std::derived_from<T, view_base> || /*is-derived-from-view-interface*/<T>;
```
(2) (since C++20)

```
struct view_base { };
```
(3) (since C++20)

1) The view concept specifies the requirements of a range type that has suitable semantic properties for use in constructing range adaptor pipelines.
2) The enable_view variable template is used to indicate whether a range is a view. */*is-derived-from-view-interface*/*<T> is true if and only if T has exactly one public base class ranges::view_interface<U> for some type U, and T has no base classes of type ranges::view_interface<V> for any other type V.

```
template< ranges::input_range V,
    requires ranges::view<V> &&
class transform_view
    : public ranges::view_interface<transform_view<V, F>>
```
(1) (since C++20) (until C++23)

```
template< ranges::input_range V,
    requires ranges::view<V> &&
class transform_view
    : public ranges::view_interface<transform_view<V, F>>
```
(since C++23)

```
namespace views {
    inline constexpr transform_view transform
}
```
(2) (since C++20)

Call signature

The viewable_range concept is a refinement of range that describes a range that can be converted into a view. Even the std::vector is a viewable_range

```
template< ranges::viewable_range R, class F >
    requires /* see below */
constexpr ranges::view auto transform( R&& r, F&& fun );
```
(since C++20)

```
template< class F >
constexpr /*range adaptor closure*/ transform( F&& fun );
```
(since C++20)

Range adaptors takes a range (vector, list etc) and return a type that satisfies the "view" concept. For example transform_view.

When such type is created, it saves a reference to given range and the operation (callable) that needs to be applied over the range. It is **lazy evaluated**, doesn't nothing until the value is requested.

```
auto lTransformed = std::views::transform(v, [](int n){return n*n; }) |
```

Compiler generates

```
std::ranges::transform_view<
    std::ranges::ref_view<
        std::vector<int, std::allocator<int>>
    >,
    __lambda_52_50
> lTransformed = std::ranges::views::transform.operator()(v, __lambda_52_50);
```

reference to input

operation to execute on input

Functor

The **Subrange** class template combines together an iterator and a sentinel into a single view

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

// multiset::equal_range() returns a pair of iterators:
auto [left, right] = sorted.equal_range(5);

// We can use ranges::subrange to turn that into a range:
for (auto v : std::ranges::subrange(left, right)) {
    // Iterate over {5,5,5}
}
```