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

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 */                                                                 (since C++20)
constexpr ranges::view auto transform( R&& r, F&& fun );

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<          reference to input
        std::vector<int, std::allocator<int>>
    >,                                                    Functor
    __lambda_52_50  operation to execute on input
> lTransformed = std::ranges::views::transform.operator()(v, __lambda_52_50);
```

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}
}
```

Which is the evaluation order for piped views?
Short answer: Left ⟶ Right

```
void step1() { cout << "Step 1 ... \n"; }
void step2() { cout << "Step 2 ... \n"; }
void step3() { cout << "Step 3 ... \n"; }

int main()
{
    vector<int> v {1,4,7,10,5};

    auto lTransformed = views::filter(v, [](int value)  {step1(); return value % 2 == 0; }) |
                        views::transform([](int n)      {step2(); return n*n; }) |
                        views::transform([](int n)      {step3(); return n/2; });

    cout << " ================ 1 =============== \n";
    auto it {lTransformed.begin()};

    cout << " ================ 2 =============== \n";
    cout << *it << "\n";

    cout << " ================ 3 =============== \n";
    it++;

    cout << " ================ 4 =============== \n";
    cout << *it << "\n";

    cout << " ================ 5 =============== \n";
    it++;

    cout << " ================ 6 =============== \n";
    cout << *it << "\n";
}
```

```
 ================ 1 ===============
Step 1 ...
Step 1 ...
 ================ 2 ===============
Step 2 ...
Step 3 ...
8
 ================ 3 ===============
Step 1 ...
Step 1 ...
 ================ 4 ===============
Step 2 ...
Step 3 ...
50
 ================ 5 ===============
Step 1 ...
 ================ 6 ===============
Step 2 ...
Step 3 ...
0
```

Any movement of iterator result into finding the next starting point in the sequence

Requesting the value triggers the rest of the execution