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>>
```
(since C++20) (until C++23)

(1)

```
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}
}
```

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";
}
```

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

Requesting the value triggers the rest of the execution

```
Type to enter text
================ 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
```

What is a sentinel?

When the end of a range is defined by an iterator, we need to know where that iterator should point to in advance.
For example, imagine we wanted our range of numbers to end at the first negative number.
A sentinel is something that can signal an algorithm to end.
Within the context of C++ ranges, sentinels are objects that can be compared to iterators, using the equality (==) operator. If the operator returns true, that's the signal for the algorithm to stop.

```
template <typename T>
struct Sentinel {
  bool operator==(T Iter) const {
    return Iter == ContainerEnd || *Iter < 0;
  }

  T ContainerEnd;
};
```

Sentinel that stop at first negative number

```
struct Sentinel {/*...*/};
```
Using our Sentinel you can define specific ranges, like the one below:

```
template <typename T>
class PositiveRange {
public:
  PositiveRange(
    std::initializer_list<T> Numbers)
    : Container{Numbers},
      Sentinel{Container.end()} {}

  auto begin() const{
    return Container.begin();
  }

  auto end() const{ return Sentinel; }

private:
  std::vector<T> Container;
  Sentinel<typename std::vector<
    T>::const_iterator> Sentinel;
};
```

Sentinel can be used with algorithms:

```
std::vector R{1, 4, 3, 8, -2, 5};

std::ranges::for_each(
  R.begin(), Sentinel{R.end()}, Log
);
```

Only with the one defined in std::ranges, why?

The type of begin iterator should match the type of end iterator

```
template< class InputIt, class T >
typename std::iterator_traits<InputIt>::difference_type
    count( InputIt first, InputIt last, const T& value );
```

std::count as it has been defined before ranges

Count algorithm as it is defined in std::ranges:
(it can accept either an input iterator and sentinel or a range

```
template< std::input_iterator I, std::sentinel_for<I> S,
    count( I first, S last, const T& value, Proj proj = {} );
```

```
template< ranges::input_range R, class T, class Proj = std::identity >
    count( R&& r, const T& value, Proj proj = {} );
```

What is a projection?

When an algorithm acts on intems in a collection, more often than not it is not interested in exact items in the collection but in their members. This is why most algorithms in std::ranges have a "Projection" template parameter.

```
template< ranges::input_range R, class T, class Proj = std::identity >
    count( R&& r, const T& value, Proj proj = {} );
```

```
#include <vector>
#include <iostream>
#include <algorithm>

UNCHANGED CODE | HIDE
struct Player {
  std::string Name;
  int Level;
};

int main() {
  UNCHANGED CODE | HIDE
  std::vector Party {
    Player {"Legolas", 49},
    Player {"Gimli", 47},
    Player {"Gandalf", 53}
  };

  std::ranges::sort(
    Party,
    [](int a, int b) { return a > b; },
    [](Player& P) { return P.Level; }
  );
```

Projection on class members

```
#include <vector>
#include <iostream>
#include <algorithm>

PLAYER CLASS | HIDE
class Player {
public:
  Player(std::string Name) : Name(Name) {};
  std::string GetName() const { return Name; }

private:
  std::string Name;
};

int main() {
  std::vector Party {
    Player{"Legolas"},
    Player{"Gimli"},
    Player{"Gandalf"}
  };

  std::ranges::sort(Party, {}, &Player::GetName);

  for (const auto& P : Party) {
    std::cout << P.GetName() << '\n';
  }
}
```

Projection on class members

```
#include <algorithm>
#include <iostream>
#include <vector>

int Project(int x) {
  return std::abs(x);
}

int main() {
  std::vector Nums{-3, 5, 0};
  std::ranges::sort(Nums, {}, Project);
  for (auto Num : Nums) {
    std::cout << Num << ", ";
  }
}
```

Projection on primitive types

Projection for primitive types:
(The "abs logic" can be embedded in predicate in this case)