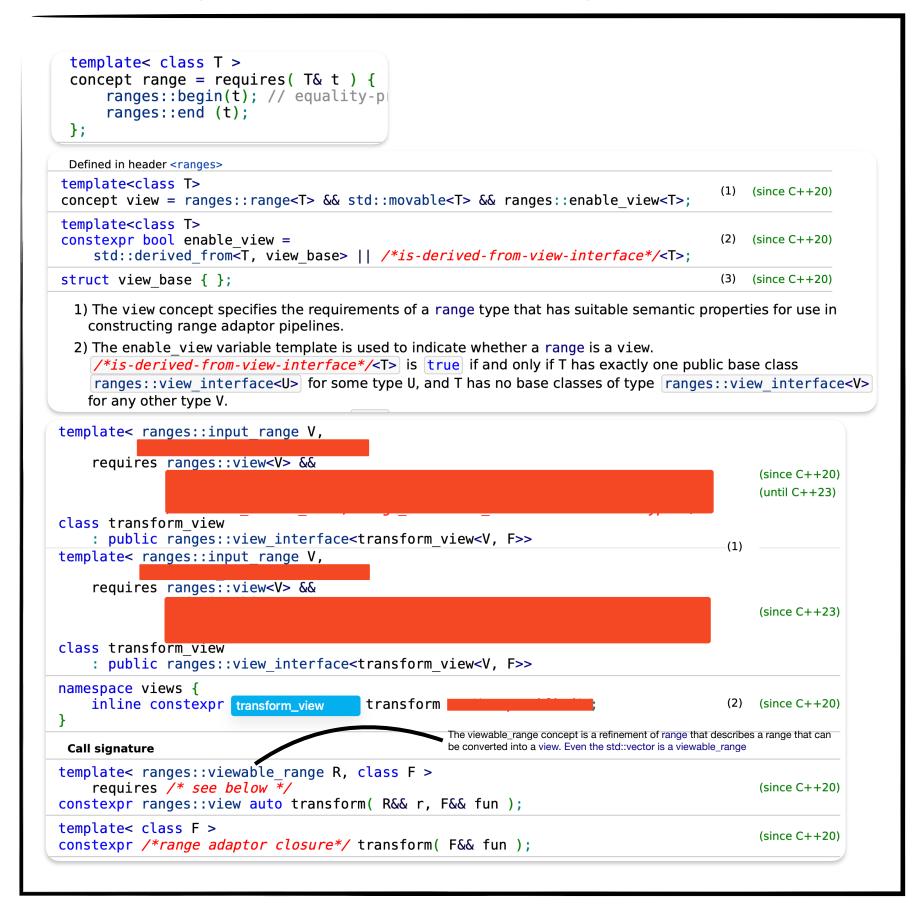
Range & View concepts & Range adaptors



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

```
void step1() { cout << "Step 1 ... \n"; }</pre>
void step2() { cout << "Step 2 ... \n"; }</pre>
  vector<int> v {1,4,7,10,5};
   auto lTransformed = views::filter(v, [](int value) {stepl(); return value % 2 == 0; }) |
                views::transform([](int n)
                                     {step2(); return n*n; })
                views::transform([](int n)
                                      {step3(); return n/2; });
  auto it {|Transformed.begin()};
                                     Step 1 ...
                                     Step 1 ...
                                     cout << *it << "\n";
                                     Step 2 ...
                                     Step 3 ...
                                      Step 1 ...
                                     Step 1 ...
                                      _____ 4 _____
  cout << *it << "\n";
                                     Step 2 ...
                                     Step 3 ...
  it++;
                                                  ==== 5 =========
                                     Step 1 ...
  cout << " ----- 6 ----- \n"; ======== 6 ======= 6 ===========
  cout << *it << "\n";
                                     Step 2 ...
                                     Step 3 ...
```

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.

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}
}
```