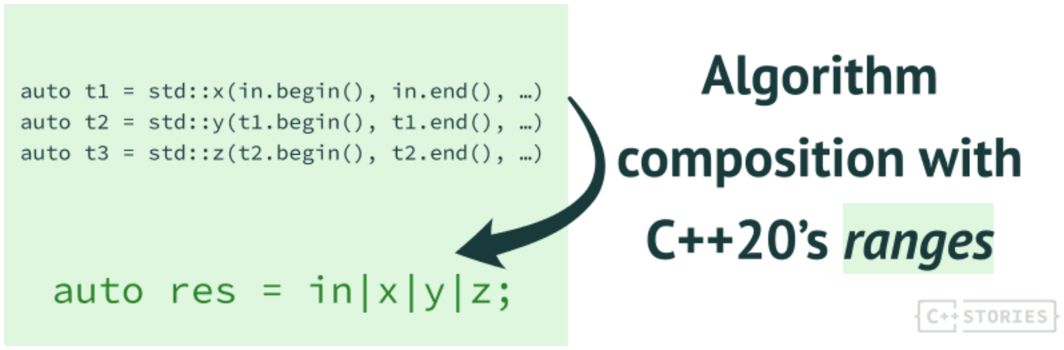
C++20 Ranges: The Key Advantage - Algorithm Composition



```
auto t1 = std::x(in.begin(), in.end(), …)
auto t2 = std::y(t1.begin(), t1.end(), …)
auto t3 = std::z(t2.begin(), t2.end(), …)

auto res = in|x|y|z;
```

Conceptually a Range is a simple concept: it's just a pair of two iterators - to the beginning and to the end of a sequence (or a sentinel in some cases). Yet, such an abstraction can radically change the way you write algorithms. In this blog post, I'll show you a **key change** that you get with C++20 Ranges.

By having this one layer of abstraction on iterators, we can express more ideas and have different computation models.

Let's look at a simple example in "regular" STL C++.

It starts from a list of numbers, selects even numbers, skips the first one and then prints them in the reverse order:

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

int main() {
    const std::vector numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto even = [](int i) { return 0 == i % 2; };

    std::vector<int> temp;
    std::copy_if(begin(numbers), end(numbers), std::back_inserter(temp), even);
    std::vector<int> temp2(begin(temp)+1, end(temp));

    for (auto iter = rbegin(temp2); iter!=rend(temp2); ++iter)
        std::cout << *iter << ' ';
}
```

Play @Compiler Explorer.

The code does the following steps:

- It creates `temp` with all even numbers from `numbers`,
- Then, it skips one element and copies everything into `temp2`,
- And finally, it prints all the elements from `temp2` in the reverse order.

(*): Instead of `temp2` we could just stop the reverse iteration before the last element, but that would require to find that last element first, so let's stick to the simpler version with a temporary container…

(*): The early version of this article contained a different example where it skipped the first two elements, but it was not the best one and I changed it (thanks to various comments).

I specifically used names `temp` and `temp2` to indicate that the code must perform additional copies of the input sequence.

And now let's rewrite it with Ranges:
```
#include <algorithm>
#include <vector>
#include <iostream>
#include <ranges>   // new header!

int main() {
    const std::vector numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto even = [](int i) { return 0 == i % 2; };

    std::ranges::reverse_view rv{
        std::ranges::drop_view {
            std::ranges::filter_view{ numbers, even }, 1
        }
    };
    for (auto& i : rv)
        std::cout << i << ' ';;
}
```

Play @Compiler Explorer.

Wow! That's nice!

This time, we have a completely different model of computation: Rather than creating temporary objects and doing the algorithm step by step, we wrap the logic into a composed view.

Before we discuss code, I should bring two essential topics and loosely define them to get the basic intuition:

> ***Range*** *- Ranges are an abstraction that allows a C++ program to operate on elements of data structures uniformly. On minimum a range contains defines begin() and end() to elements. There are several different types of ranges: containers, views, sized ranges,*
>
> ***Container*** *- It's a range that owns the elements.*
>
> ***View*** *- This is a range that doesn't own the elements that the begin/end points to. A view is cheap to create, copy and move.*

Our code does the following (inside out)

- We start from `filter_view` that additionally takes a predicate `even`,
- Then, we add `drop_view` (drop one element from the previous step),
- And the last view is to apply a `reverse_view` view on top of that,
- The last step is to take that view and iterate through it in a loop.

Can you see the difference?

The view `rv` doesn't do any job when creating it. We only compose the final `receipt`. The execution happens *lazy* only when we iterate through it.

Let's have a look at one more example with string trimming:

Here's the standard version:
```
const std::string text { "    Hello World" };
std::cout << std::quoted(text) << '\n';

auto firstNonSpace = std::find_if_not(text.begin(), text.end(), ::isspace);
std::string temp(firstNonSpace, text.end());
std::transform(temp.begin(), temp.end(), temp.begin(), ::toupper);

std::cout << std::quoted(temp) << '\n';
```

Play @Compiler Explorer.

And here's the ranges version:
```
const std::string text { "    Hello World" };
std::cout << std::quoted(text) << '\n';

auto conv = std::ranges::transform_view {
    std::ranges::drop_while_view{text, ::isspace},
    ::toupper
};

std::string temp(conv.begin(), conv.end());

std::cout << std::quoted(temp) << '\n';
```

Play @Compiler Explorer.

This time we compose `drop_while_view` with `transform_view`. Later once the view is ready, we can iterate and build the final `temp` string.

This article started as a preview for Patrons, sometimes even months before the publication. If you want to get extra content, previews, free ebooks and access to our Discord server, join **the C++ Stories Premium membership.**

The examples so far used views from the `std::ranges` namespace. But in C++20, we also have another namespace, `std::views`, which defines e a set of predefined Range adaptor objects. Those objects and the pipe operator allow us to have even shorter syntax.

We can rewrite the previous example into:
```
const std::vector numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

auto even = [](int i) { return 0 == i % 2; };

namespace sv = std::views;
for(auto& i : numbers | sv::filter(even) | sv::drop(1) | sv::reverse)
    std::cout << i << ' ';
```

Play @Compiler Explorer.

According to C++ Reference:

> *if C is a range adaptor object and R is a viewable_range, these two expressions are equivalent: C(R) and R | C.*

For our expression, we can read it from left to right:

- take `numbers` and apply `filter` view,
- then add `drop(1)`
- and the last step is to apply `reverse`.

And for the string trimming example we can write the following:

```
const std::string text { "    Hello World" };
std::cout << std::quoted(text) << '\n';

auto conv = text | std::views::drop_while(isspace) | std::views::transform(::toupper);
std::string temp(conv.begin(), conv.end());

std::cout << std::quoted(temp) << '\n';
```

Play @Compiler Explorer.

You might notice that I still need an additional step to build the final string out of a view. This is because Ranges are not complete in C++20, and we'll get more handy stuff in C++23.

In C++23, we'll be able to use `std::ranges::to<std::string>();` and thus the code will get even simpler:

```
auto temp = text | std::views::drop_while(isspace) | std::views::transform(::toupper) | std::ranges::to<
```

Now, `temp` is a `string` created from the view. The composition of algorithms and the creation of other containers will get even simpler.

Here's the list of predefined views that we get with C++20:

Name	Notes
`views::all`	returns a view that includes all elements of its range argument.
`filter_view/filter`	returns a view of the elements of an underlying sequence that satisfy a predicate.
`transform_view/transform`	returns a view of an underlying sequence after applying a transformation function to each element.
`take_view/take`	returns a view of the first `N` elements from another view, or all the elements if the adapted view contains fewer than `N`.
`take_while_view/take_while`	Given a unary predicate `pred` and a view `r`, it produces a view of the range `[begin(r), ranges::find_if_not(r, pred))`.
`drop_view/drop`	returns a view excluding the first `N` elements from another view, or an empty range if the adapted view contains fewer than `N` elements.
`drop_while_view/drop_while`	Given a unary predicate `pred` and a view `r`, it produces a view of the range `[ranges::find_if_not(r, pred), ranges::end(r))`.
`join_view/join`	It flattens a view of ranges into a view
`split_view/split`	It takes a view and a delimiter and splits the view into subranges on the delimiter. The delimiter can be a single element or a view of elements.
`counted`	A counted view presents a view of the elements of the counted range `([iterator.requirements.general]) i+[0, n)` for an iterator `i` and non-negative integer `n`.

Name	Notes
common_view/common	takes a view which has different types for its iterator and sentinel and turns it into a view of the same elements with an iterator and sentinel of the same type. It is useful for calling legacy algorithms that expect a range's iterator and sentinel types to be the same.
reverse_view/reverse	It takes a bidirectional view and produces another view that iterates the same elements in reverse order.
elements_view/elements	It takes a view of tuple-like values and a size_t, and produces a view with a value-type of the Nth element of the adapted view's value-type.
keys_view/keys	Takes a view of tuple-like values (e.g. std::tuple or std::pair), and produces a view with a value-type of the first element of the adapted view's value-type. It's an alias for elements_view<views::all_t<R>, 0>.
values_view/values	Takes a view of tuple-like values (e.g. std::tuple or std::pair), and produces a view with a value-type of the second element of the adapted view's value-type. It's an alias for elements_view<views::all_t<R>, 1>.

You can read their details in this section of the Standard: https://timsong-cpp.github.io/cppwp/n4861/range.factories

In this blog post, I gave only the taste of C++20 Ranges.

As you can see, the idea is simple: wrap iterators into a single object - a Range and provide an additional layer of abstraction. Still, as with abstractions in general, we now get lots of new powerful techniques. The computation model is changed for algorithm composition. Rather than executing code in steps and creating temporary containers, we can build a view and execute it once.

Have you started using ranges? What's your initial experience? Let us know in the comments below the article.