6
Ranges and Views

This chapter will pick up right where we left off in the previous chapter about algorithms and their limitations. Views from the Ranges library are a powerful complement to the Algorithm library, which allows us to compose multiple transformations into a lazy evaluated view over a sequence of elements. After reading this chapter, you will understand what range views are and how to use them in combination with containers, iterators, and algorithms from the standard library.

Specifically, we'll cover the following major topics:

- The composability of algorithms
- Range adaptors
- Materializing views into containers
- Generating, transforming, and sampling elements in a range

Before we get into the Ranges library itself, let's discuss why it's been added to C++20, and why we'd want to use it.

The motivation for the Ranges library

With the introduction of the Ranges library to C++20 came some major improvements to how we benefit from the standard library when implementing algorithms. The following list shows the new features:

- Concepts that define requirements on iterators and ranges can now be better checked by the compiler and provide more help during development
- New overloads of all functions in the `<algorithm>` header are constrained with the concepts just mentioned and accept ranges as arguments rather than iterator pairs
- Constrained iterators in the iterator header
- Range views, which make it possible to compose algorithms

This chapter will focus on the last item: the concept of views, which allow us to compose algorithms to avoid the unnecessary copying of data to owning containers. To fully understand the importance of this, let's begin by demonstrating the lack of composability within the algorithm library.

Limitations of the Algorithm library

standard library algorithms are lacking in one fundamental aspect: composability. Let's examine what is meant by that by looking at the last example from *Chapter 5*, *Algorithms*, where we discussed this briefly. If you remember, we had a class to represent a `Student` in a particular year and with a particular exam score:

```
struct Student {
  int year_{};
  int score_{};
  std::string name_{};
```

```
  // ...
};
```

If we wanted to find the highest score from a big collection of students in their second year, we would probably use `max_element()` on `score_`, but as we only want to take the students in a specific year into account, it gets tricky. By using the new algorithms that accept both ranges and projections (refer to *Chapter 5*, *Algorithms*), we might end up with something like this:

```
auto get_max_score(const std::vector<Student>& students, int year) {
  auto by_year = [=](const auto& s) { return s.year_ == year; };
  // The student list needs to be copied in
  // order to filter on the year
  auto v = std::vector<Student>{};
  std::ranges::copy_if(students, std::back_inserter(v), by_year);
  auto it = std::ranges::max_element(v, std::less{}, &Student::score_);
  return it != v.end() ? it->score_ : 0;
}
```

Here is an example of how it can be used:

```
auto students = std::vector<Student>{
  {3, 120, "Niki"},
  {2, 140, "Karo"},
  {3, 190, "Sirius"},
  {2, 110, "Rani"},
   // ...
};
```

```
auto score = get_max_score(students, 2);
std::cout << score << '\n';
// Prints 140
```

This implementation of `get_max_score()` is easy to understand, but it creates unnecessary copies of `Student` objects when using `copy_if()` and `std::back_inserter()`.

You may now be thinking that `get_max_score()` could be written as a simple `for-` loop, which relieves us of extra allocation(s) due to `copy_if()`:

```
auto get_max_score(const std::vector<Student>& students, int year) {
  auto max_score = 0;
  for (const auto& student : students) {
    if (student.year_ == year) {
      max_score = std::max(max_score, student.score_);
    }
  }
  return max_score;
}
```

Although this is easily achievable in this small example, we would like to be able to implement this algorithm by composing small algorithmic building blocks, rather than implementing it from scratch using a single `for` -loop.

What we would like is a syntax that is as readable as using algorithms, but with the ability to avoid constructing new containers for every step in the algorithm. This is where the views from the Ranges library come into play. Although the Ranges library contains a lot more than views, the major difference from

the Algorithm library is the ability to compose what is essentially a different kind of iterator into a lazy evaluated range.

This is what the previous example would look if it was written using views from the Ranges library:

```
auto max_value(auto&& range) {
  const auto it = std::ranges::max_element(range);
  return it != range.end() ? *it : 0;
}
auto get_max_score(const std::vector<Student>& students, int year) {
  const auto by_year = [=](auto&& s) { return s.year_ == year; };
  return max_value(students
    | std::views::filter(by_year)
    | std::views::transform(&Student::score_));
}
```

Now we are back to using algorithms and can, therefore, avoid mutable variables, `for`-loops, and `if`-statements. The extra vector that held students in a specific year in our initial example has now been eliminated. Instead, we have composed a range view, which represents all of the students filtered by the `by_year` predicate, and then transformed to only expose the score. The view is then passed to a small utility function `max_value()`, which uses the `max_element()` algorithm to compare the scores of the selected students in order to find the maximum value.

This way of composing algorithms by chaining them together and, at the same time, avoiding unnecessary copying is what motivates us to start using views from the Ranges library.

Understanding views from the Ranges library

Views in the Ranges library are lazy evaluated iterations over a range. Technically, they are only iterators with built-in logic, but syntactically, they provide a very pleasant syntax for many common operations.

The following is an example of how to use a view to square each number in a vector (via iteration):

```
auto numbers = std::vector{1, 2, 3, 4};
auto square = [](auto v) {  return v * v; };
auto squared_view = std::views::transform(numbers, square);
for (auto s : squared_view) {  // The square lambda is invoked here
  std::cout << s << " ";
}
// Output: 1 4 9 16
```

The variable `squared_view` is not a copy of the `numbers` vector with the values squared; it is a proxy object for numbers with one slight difference—every time you access an element, the `std::transform()` function is invoked. This is why we say that a view is lazy evaluated.

From the outside, you can still iterate over `squared_view` in the same way as any regular container and, therefore, you can perform regular algorithms such as `find()` or `count()`, but, internally, you haven't created another container.

If you want to store the range, the view can be materialized to a container using `std::ranges::copy()`. (This will be demonstrated later on in this chapter.) Once the view has been copied back to a container, there is no longer any dependency between the original and the transformed container.

With ranges, it is also possible to create a filtered view where only a part of the range is visible. In this case, only the elements that satisfy the condition are visible when iterating the view:

```
auto v = std::vector{4, 5, 6, 7, 6, 5, 4};
auto odd_view =
  std::views::filter(v, [](auto i){ return (i % 2) == 1; });
for (auto odd_number : odd_view) {
  std::cout << odd_number << " ";
}
// Output: 5 7 5
```

Another example of the versatility of the Ranges library is the possibility it offers to create a view that can iterate over several containers as if they were a single list:

```
auto list_of_lists = std::vector<std::vector<int>> {
  {1, 2},
  {3, 4, 5},
  {5},
  {4, 3, 2, 1}
};
auto flattened_view = std::views::join(list_of_lists);
for (auto v : flattened_view)
  std::cout << v << " ";
// Output: 1 2 3 4 5 5 4 3 2 1

auto max_value = *std::ranges::max_element(flattened_view);
// max_value is 5
```

Now that we have looked briefly at some examples using views, let's examine the requirements and properties that are common for all views

Views are composable

The full power of views comes from the ability to combine them. As they don't copy the actual data, you can express multiple operations on a dataset while, internally, only iterating over it once. To understand how views are composed, let's look at our initial example, but without using the pipe operator for composing the views; instead, let's construct the actual view classes directly. Here is how this looks:

```
auto get_max_score(const std::vector<Student>& s, int year) {
  auto by_year = [=](const auto& s) { return s.year_ == year; };

  auto v1 = std::ranges::ref_view{s}; // Wrap container in a view
  auto v2 = std::ranges::filter_view{v1, by_year};
  auto v3 = std::ranges::transform_view{v2, &Student::score_};
  auto it = std::ranges::max_element(v3);
  return it != v3.end() ? *it : 0;
}
```

We begin by creating a `std::ranges::ref_view`, which is a thin wrapper around a container. In our case, it turns the vector `s` into a view that is cheap to copy. We need this because our next view, `std::ranges::filter_view`, requires a view as its first parameter. As you can see, we compose our next view by referring to the previous view in the chain.

This chain of composable views can, of course, be made arbitrarily long. The algorithm `max_element()` doesn't need to know anything about the complete chain; it only needs to iterate the range `v3`, as it was

an ordinary container.

The following diagram is a simplified view of the relationships between the `max_element()` algorithm, the views, and the input container:

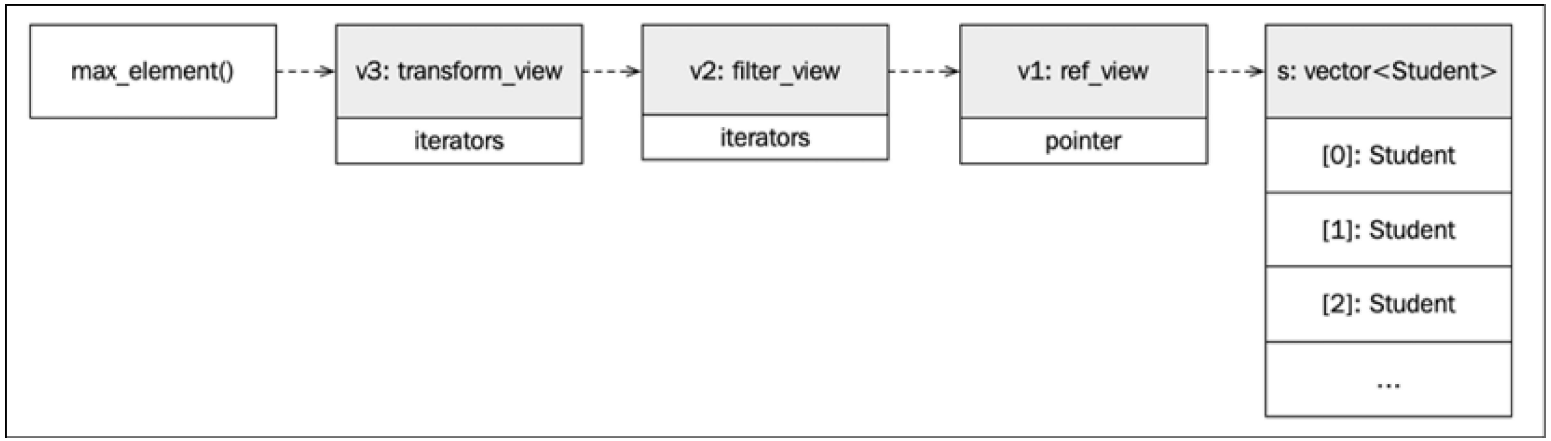

Figure 6.1: The top level algorithm, std::ranges::max_element(), pulls values from the views which lazyily process elements from the underlying container (std::vector)

Now, this style of composing views is a bit verbose, and if we were to try to remove the intermediate variables `v1` and `v2`, we would end up with something like this:

```
using namespace std::ranges; // _view classes live in std::ranges
auto scores =
  transform_view{filter_view{ref_view{s}, by_year},
    &Student::score_};
```

Now, this might not look syntactically elegant. By getting rid of the intermediate variables, we have something that is hard to read even to a trained eye. We are also forced to read the code from the inside out to understand the dependencies. Fortunately, the Ranges library provides us with range adaptors, which is the preferred way of composing views.

Range views come with range adaptors

As you have seen earlier, the Ranges library also allows us to compose views using range adaptors and pipe operators for a much more elegant syntax (you will learn more about using the pipe operator in your own code in *Chapter 10*, *Proxy Objects and Lazy Evaluation*). The previous code example could be rewritten by using range adaptor objects, and we would have something like this:

```
using namespace std::views; // range adaptors live in std::views
auto scores = s | filter(by_year) | transform(&Student::score_);
```

The ability to read a statement from left to right, rather than inside out, makes the code much easier to read. If you have used a Unix shell, you are probably familiar with this notation for chaining commands.

Each view in the Ranges library has a corresponding range adaptor object that can be used together with the pipe operator. When using the range adaptors, we can also skip the extra `std::ranges::ref_view` since the range adaptors work directly with `viewable_ranges`, namely, a range that can be safely converted into a `view`.

You can think of a range adaptor as a global stateless object that has two functions implemented: `operator()()` and `operator|()`. Both functions construct and return view objects. The pipe operator is what is

being used in the preceding example. But it is also possible to use the call operator to form a view using a nested syntax with parentheses, like this:

```
using namespace std::views;
auto scores = transform(filter(s, by_year), &Student::score_);
```

Again, when using range adaptors, there is no need to wrap the input container in a `ref_view`.

To summarize, each view in the Ranges library consists of:

- A class template (the actual view type) that operates on view objects, for example, `std::ranges::transform_view`. These view types can be found under the namespace `std::ranges`.
- A range adaptor object that creates instances of the view class from ranges, for example, `std::views::transform`. All range adaptors implement `operator()()` and `operator|()`, which makes it possible to compose transformations using the pipe operator or by nesting. The range adaptor objects live under the namespace `std::views`.

Views are non-owning ranges with complexity guarantees

In the previous chapter, the concept of a range was introduced. Any type that provides the functions `begin()` and `end()`, where `begin()` returns an iterator and `end()` returns a sentinel, qualifies as a range. We concluded that all standard containers are ranges. Containers own their elements, so we can, therefore, call them owning ranges.

A view is also a range, that is, it provides `begin()` and `end()` functions. However, unlike containers, a view does not own the elements in the range that the view spans over.

The construction of a view is required to be a constant-time operation, *O(1)*. It cannot perform any work that depends on the size of the underlying container. The same goes for assigning, copying, moving, and destructing a view. This makes it easy to reason about performance when using views to combine multiple algorithms. It also makes it impossible for views to own elements, since that would require linear time complexity upon construction and destruction.

Views don't mutate the underlying container

At first glance, a view might look like a mutated version of the input container. However, the container is not mutated at all: all the processing is performed in the iterators. A view is simply a proxy object that, when iterated, *looks* like a mutated container.

This also makes it possible for a view to expose elements of types that are different from the types of the input elements. The following snippet demonstrates how a view transforms the element type from `int` to `std::string`:

```
auto ints = std::list{2, 3, 4, 2, 1};
auto strings = ints
  | std::views::transform([](auto i) { return std::to_string(i); });
```

Perhaps we have a function that operates on a container that we want to transform using range algorithms, and then we want to return and store it back in a container. For example, in the example above, we might want to actually store the strings in a separate container. You will learn how to do that in the next section.

Views can be materialized into containers

Sometimes, we want to store the view in a container, that is, **materialize** the view. All views can be materialized into containers, but it is not as easy as you would have hoped. A function template called `std::ranges::to<T>()`, which could turn a view into an arbitrary container type `T`, was proposed for C++20 but didn't quite make it. Hopefully we will get something similar in a future version of C++. Until then, we need to do a little more work ourselves in order to materialize views.

In the previous example, we converted `ints` into `std::strings`, as follows:

```
auto ints = std::list{2, 3, 4, 2, 1};
auto r = ints
  | std::views::transform([](auto i) { return std::to_string(i); });
```

Now, if we want to materialize the range `r` to a vector, we could use `std::ranges::copy()` like this:

```
auto vec = std::vector<std::string>{};
std::ranges::copy(r, std::back_inserter(vec));
```

Materializing views is a common operation, so it would be handy if we had a generic utility for this case. Say that we want to materialize some arbitrary view into a `std::vector`; we could use some generic programming to come up with the following convenient utility function:

```
auto to_vector(auto&& r) {
  std::vector<std::ranges::range_value_t<decltype(r)>> v;
  if constexpr(std::ranges::sized_range<decltype(r)>) {
    v.reserve(std::ranges::size(r));
  }
```

```
  std::ranges::copy(r, std::back_inserter(v));
  return v;
}
```

This snippet is taken from Timur Doumler's blog post, **https://timur.audio/how-to-make-a-container-from-a-c20-range**, which is well worth a read.

We haven't talked much about generic programming yet in this book, but the next few chapters will explain the use of `auto` argument types and `if constexpr`.

We are using `reserve()` to optimize the performance of this function. It will preallocate enough room for all of the elements in the range to avoid further allocations. However, we can only call `reserve()` if we know the size of the range, and therefore we have to use the `if constexpr` statement to check whether the range is a `size_range` at compile time.

With this utility in place, we can transform a container of some type into a vector holding elements of another arbitrary type. Let's see how to convert a list of integers to a vector of `std::strings` using `to_vector()`. Here is an example:

```
auto ints = std::list{2, 3, 4, 2, 1};
auto r = ints
  | std::views::transform([](auto i) { return std::to_string(i); });
auto strings = to_vector(r);
// strings is now a std::vector<std::string>
```

Remember that once the view has been copied back to a container, there is no longer any dependency between the original and the transformed container. This also means that the materialization is an eager operation, whereas all view operations are lazy.

Views are lazy evaluated

All of the work that is performed by a view happens lazily. This is the opposite of the functions found in the `<algorithm>` header, which perform their work immediately on all elements when they are called.

You have seen that the `std::views::filter` view can replace the algorithm `std::copy_if()`, and that the `std::views::transform` view can replace the `std::transform()` algorithm. When we use the views as building blocks and chain them together, we benefit from lazy evaluation by avoiding unnecessary copies of the container elements required by the eager algorithms.

But what about `std::sort()`? Is there a corresponding sorting view? The answer is no because it would require the view to first collect all the elements eagerly in order to find the first element to return. Instead, we have to do that ourselves by explicitly calling sort on our view. In most cases, we also need to materialize the view before sorting. We can clarify this with an example. Assume that we have a vector of numbers that we have filtered by some predicate, like this:

```
auto vec = std::vector{4, 2, 7, 1, 2, 6, 1, 5};
auto is_odd = [](auto i) { return i % 2 == 1; };
auto odd_numbers = vec | std::views::filter(is_odd);
```

If we try to sort our view `odd_numbers` using `std::ranges::sort()` or `std::sort()`, we will get a compilation error:

```
std::ranges::sort(odd_numbers); // Doesn't compile
```

The compiler complains about the types of iterators provided by the `odd_numbers` range. The sorting algorithm requires random access iterators, but that's not the type of iterators that our view provides, even though the underlying input container is a `std::vector`. What we need to do is to materialize the view before sorting:

```
auto v = to_vector(odd_numbers);
std::ranges::sort(v);
// v is now 1, 1, 5, 7
```

But why is this necessary? The answer is that this is a consequence of lazy evaluation. The filter view (and many other views) cannot preserve the iterator types of the underlying range (in this case, the `std::vector`) when evaluation needs to be lazy by reading one element at a time.

So, are there any views that can be sorted? Yes, an example would be `std::views::take`, which returns the first *n* elements in a range. The following example compiles and runs fine without the need for materializing the view before sorting:

```
auto vec = std::vector{4, 2, 7, 1, 2, 6, 1, 5};
auto first_half = vec | std::views::take(vec.size() / 2);
std::ranges::sort(first_half);
// vec is now 1, 2, 4, 7, 2, 6, 1, 5
```

The quality of the iterators has been preserved and it's therefore possible to sort the `first_half` view. The end result is that the first half of the elements in the underlying vector `vec` have been sorted.

You now have a good understanding of what views from the Ranges library are and how they work. In the next section, we will explore how to use the views that are included in the standard library.

Views in the standard library

So far in this chapter, we have been talking about views from the Ranges library. As was described earlier, these view types need to be constructed in constant time and also have constant-time copy, move, and assignment operators. However, in C++, we have talked about view classes before the Ranges library was added to C++20. These view classes are non-owning types, just like `std::ranges::view`, but without the complexity guarantees.

In this section, we will begin by exploring the views from the Ranges library that are associated with the `std::ranges::view` concept, and then move on to `std::string_view` and `std::span`, which are not associated with `std::ranges::view`.

Range views

There are already many views in the Ranges library, and I think we will see even more of them in future versions of C++. This section will provide a quick overview of some of the available views and also put them in different categories based on what they do.

Generating views

Generating views produce values. They can generate a finite or infinite range of values. The most obvious example in this category is `std::views::iota`, which produces values within a half-open range. The following snippet prints the values `-2`, `-1`, `0`, and `1`:

```
for (auto i : std::views::iota(-2, 2)) {
  std::cout << i << ' ';
}
// Prints -2 -1 0 1
```

By omitting the second argument, `std::views::iota` will produce an infinite number of values on request.

Transforming views

Transforming views are views that transform the elements of a range or the structure of the range itself. Some examples include:

- `std::views::transform`: Transforms the value and/or the type of each element
- `std::views::reverse`: Returns a reversed version of the input range
- `std::views::split`: Takes an element apart and splits each element into a subrange. The resulting range is a range of ranges
- `std::views::join`: The opposite of split; flattens out all subranges

The following example uses `split` and `join` to extract all digits from a string of comma-separated values:

```
auto csv = std::string{"10,11,12"};
auto digits = csv
```

```
  | std::views::split(',')     // [ [1, 0], [1, 1], [1, 2] ]
  | std::views::join;          // [ 1, 0, 1, 1, 1, 2 ]
for (auto i : digits) {   std::cout << i; }
// Prints 101112
```

Sampling views

Sampling views are views that select a subset of elements in a range, for example:

- `std::views::filter` : Returns only the elements that fulfill a provided predicate
- `std::views::take` : Returns the *n* first elements of a range
- `std::views::drop` : Returns all the remaining elements in a range after dropping the first *n* elements

You have seen plenty of examples using `std::views::filter` in this chapter; it's an extremely useful view. Both `std::views::take` and `std::views::drop` have a `_while` version, which accepts a predicate instead of a number. Here is an example using `take` and `drop_while` :

```
auto vec = std::vector{1, 2, 3, 4, 5, 4, 3, 2, 1};
auto v = vec
  | std::views::drop_while([](auto i) { return i < 5; })
  | std::views::take(3);
for (auto i : v) { std::cout << i << " "; }
// Prints 5 4 3
```

This example uses `drop_while` to discard values from the front that are less than 5. The remaining elements are passed to `take` , which returns the first three elements. Now to our last category of range

views.

Utility views

You have already seen some of the utility views in action in this chapter. They come in handy when you have something that you want to convert or treat as a view. Some examples in this category of views are `ref_view`, `all_view`, `subrange`, `counted`, and `istream_view`.

The following example shows you how to read a text file with floating-point numbers and then print them.

Assume that we have a text file called `numbers.txt` full of important floating-point numbers, like this:

```
1.4142 1.618 2.71828 3.14159 6.283 ...
```

We could then create a view of `floats` by using `std::ranges::istream_view`:

```
auto ifs = std::ifstream("numbers.txt");
for (auto f : std::ranges::istream_view<float>(ifs)) {
  std::cout << f << '\n';
}
ifs.close();
```

By creating a `std::ranges::istream_view` and passing it an `istream` object, we have a succinct way of processing data from files or any other input stream.

The views in the Ranges library have been carefully chosen and designed. There will most likely be more of them in upcoming versions of the standard. Being aware of the different categories of views helps us to keep them apart and make them easy to find when we need them.

Revisiting std::string_view and std::span

It's worth noting that the standard library provides us with other views outside of the Ranges library. Both `std::string_view` and `std::span` introduced in *Chapter 4*, *Data Structures* are non-owning ranges that are perfect to use in combination with the Ranges view.

There is no guarantee that these views can be constructed in constant time, as is the case with the views from the Ranges library. For example, constructing a `std::string_view` from a null-terminated C-style string could invoke a call to `strlen()`, which is an *O(n)* operation.

Suppose, for some reason, we have a function that resets the first `n` values in a range:

```
auto reset(std::span<int> values, int n) {
  for (auto& i : std::ranges::take_view{values, n}) {
    i = int{};
  }
}
```

There is no need to use a range adaptor with `values` in this case because `values` is already a view. By using `std::span`, we can pass both built-in arrays or a container such as `std::vector`:

```
int a[]{33, 44, 55, 66, 77};
reset(a, 3);
// a is now [0, 0, 0, 66, 77]
auto v = std::vector{33, 44, 55, 66, 77};
reset(v, 2);
// v is now [0, 0, 55, 66, 77]
```

In a similar way, we can use `std::string_view` together with the Ranges library. The following function splits the content of a `std::string_view` into a `std::vector` of `std::string` elements:

```
auto split(std::string_view s, char delim) {
 const auto to_string = [](auto&& r) -> std::string {
  const auto cv = std::ranges::common_view{r};
  return {cv.begin(), cv.end()};
 };
 return to_vector(std::ranges::split_view{s, delim}
  | std::views::transform(to_string));
}
```

The lambda `to_string` transforms a range of `char`s into a `std::string`. The `std::string` constructor requires identical iterator and sentinel types, therefore, the range is wrapped in a `std::ranges::common_view`. The utility `to_vector()` materializes the view and returns a `std::vector<std::string>`. `to_vector()` was defined earlier in this chapter.

Our `split()` function can now be used with both `const char*` strings and `std::string` objects, like this:

```
const char* c_str = "ABC,DEF,GHI";  // C style string
const auto v1 = split(c_str, ',');  // std::vector<std::string>
const auto s = std::string{"ABC,DEF,GHI"};
const auto v2 = split(s, ',');      // std::vector<std::string>
assert(v1 == v2);                   // true
```

We will now wrap this chapter up by talking a little bit about what we expect to see in the Ranges library in future versions of C++.

The future of the Ranges library

The Ranges library that got accepted in C++20 was based on a library authored by Eric Niebler, and is available at **https://github.com/ericniebler/range-v3**. Only a small subset of the components of this library have made their way into the standard at present, but more things are likely to be added soon.

In addition to many useful views that haven't been accepted yet, such as `group_by`, `zip`, `slice`, and `unique`, there is the concept of **actions** that can be piped in the same way that views can. However, instead of being lazy evaluated like views, actions perform eager mutations of ranges. Sorting is an example of a typical action.

If you cannot wait for these features to be added to the standard library, I recommend that you take a look at the range-v3 library.

Summary

This chapter presented a number of motivations behind using Range views to construct algorithms. By using views, we can compose algorithms efficiently, and with a succinct syntax, using the pipe operator. You also learned what it means for a class to be a view and how to use range adaptors that turn ranges into views.

A view does not own its elements. Constructing a range view is required to be a constant time operation and all views are evaluated lazily. You have seen examples of how we can convert a container into a view, and how to materialize a view back into an owning container.

Finally, we covered a brief overview of the views that come with the standard library, and the likely future of ranges in C++.

This chapter is the last in the series about containers, iterators, algorithms, and ranges. We will now move on to memory management in C++.