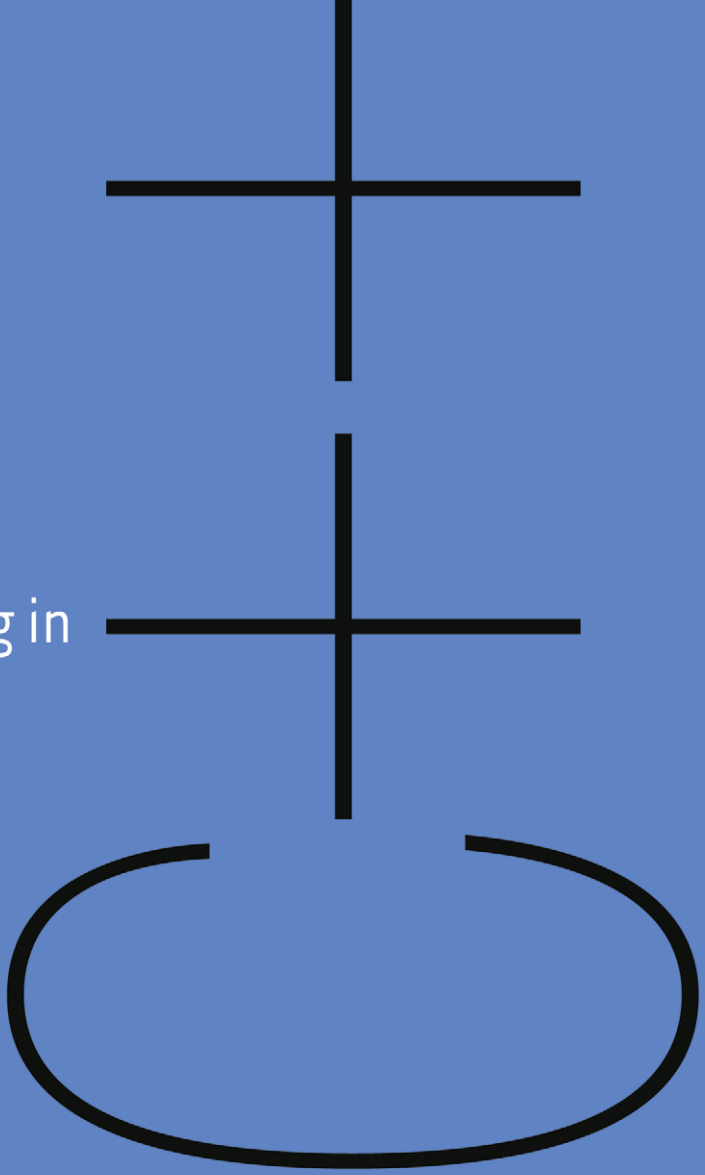

Functional Programming in C++

How to improve your C++ programs using functional techniques

Sample Chapter

Ivan Čukić

MANNING

Functional Programming in C++

by Ivan Cukic

Chapter 7

brief contents

7 Ranges

This chapter covers

- The problems of passing iterator pairs to algorithms
- What ranges are and how to use them
- Creating chained range transformations using the pipe syntax
- Understanding range views and actions
- Writing succulent code without `for` loops

In chapter 2, you saw why you should avoid writing raw `for` loops and that you should instead rely on using generic algorithms provided to you by the STL. Although this approach has significant benefits, you've also seen its downsides. The algorithms in the standard library were not designed to be easily composed with each other. Instead, they're mostly focused on providing a way to allow implementation of a more advanced version of an algorithm by applying one algorithm multiple times.

A perfect example is `std::partition`, which moves all items in a collection that satisfy a predicate to the beginning of the collection, and returns an iterator to the first element in the resulting collection that doesn't satisfy the predicate. This allows you to create a function that does multigroup partitioning—not limited to predicates that return `true` or `false`—by invoking `std::partition` multiple times.

As an example, you're going to implement a function that groups people in a collection based on the team they belong to. It receives the collection of persons, a function that gets the team name for a person, and a list of teams. You can perform `std::partition` multiple times—once for each team name—and you'll get a list of people grouped by the team they belong to.

Listing 7.1 Grouping people by the team they belong to

```
template <typename Persons, typename F>
void group_by_team(Persons& persons,
                   F team_for_person,
                   const std::vector<std::string>& teams)
{
    auto begin = std::begin(persons);
    const auto end = std::end(persons);

    for (const auto& team : teams) {
        begin = std::partition(begin, end,
                [&](const auto& person) {
                    return team == team_for_person(person);
                });
    }
}
```

Although this way to compose algorithms is useful, a more common use case is to have a resulting collection of one operation passed to another. Recall the example from chapter 2: you had a collection of people, and you wanted to extract the names of female employees only. Writing a `for` loop that does this is trivial:

```
std::vector<std::string> names;

for (const auto& person : people) {
    if (is_female(person)) {
        names.push_back(name(person));
    }
}
```

If you wanted to solve the same problem by using the STL algorithms, you'd need to create an intermediary collection to copy the persons that satisfy the predicate (`is_female`) and then use `std::transform` to extract the names for all persons in that collection. This would be suboptimal in terms of both performance and memory.

The main issue is that STL algorithms take iterators to the beginning and end of a collection as separate arguments instead of taking the collection itself. This has a few implications:

- The algorithms can't return a collection as a result.
- Even if you had a function that returned a collection, you wouldn't be able to pass it directly to the algorithm: you'd need to create a temporary variable so you could call `begin` and `end` on it.
- For the previous reasons, most algorithms mutate their arguments instead of leaving them immutable and just returning the modified collection as the result.

These factors make it difficult to implement program logic without having at least local mutable variables.

7.1 Introducing ranges

There have been a few attempts to fix these problems, but the concept that proved to be most versatile was ranges. For the time being, let's think of ranges as a simple structure that contains two iterators—one pointing to the first element in a collection, and one pointing to the element after the last one.

> **NOTE** Ranges haven't become part of the standard library yet, but an ongoing effort exists for their inclusion into the standard, currently planned for C++20. The proposal to add ranges into C++ standard is based on the range-v3 library by Eric Niebler, which we'll use for the code examples in this chapter. An older but more battle-tested library is Boost.Range. It's not as full of features as range-v3, but it's still useful, and it supports older compilers. The syntax is mostly the same, and the concepts we'll cover apply to it as well.

What are the benefits of keeping two iterators in the same structure instead of having them as two separate values? The main benefit is that you can return a complete range as a result of a function and pass it directly to another function without creating local variables to hold the intermediary results.

Passing pairs of iterators is also error prone. It's possible to pass iterators belonging to two separate collections to an algorithm that operates on a single collection, or to pass the iterators in in incorrect order—to have the first iterator point to an element that comes after the second iterator. In both cases, the algorithm would try to iterate through all elements from the starting iterator until it reached the end iterator, which would produce undefined behavior.

By using ranges, the previous example becomes a simple composition of `filter` and `transform` functions:

```
std::vector<std::string> names =
    transform(
        filter(people, is_female),
        name
    );
```

The `filter` function will return a range containing elements from the `people` collection that satisfy the `is_female` predicate. The `transform` function will then take this result and return the range of names of everybody in the filtered range.

You can nest as many range transformations such as `filter` and `transform` as you want. The problem is that the syntax becomes cumbersome to reason about when you have more than a few composed transformations.

For this reason, the range libraries usually provide a special *pipe* syntax that overloads the | operator, inspired by the UNIX shell pipe operator. So, instead of nesting the function calls, you can pipe the original collection through a series of transformations like this:

```
std::vector<std::string> names = people | filter(is_female)
                                        | transform(name);
```

As in the previous example, you're filtering the collection of persons on the `is_female` predicate and then extracting the names from the result. The main difference here is, after you get accustomed to seeing the operator | as meaning *pass through a transformation* instead of *bitwise or*, this becomes easier to write and reason about than the original example.

7.2 Creating read-only views over data

A question that comes to mind when seeing code like that in the previous section is how efficient it is, compared to writing a `for` loop that does the same thing. You saw in chapter 2 that using STL algorithms incurs performance penalties because you need to create a new vector of persons to hold the copies of all females from the `people` collection in order to be able to call `std::transform` on it. From reading the solution that uses ranges, you may get the impression that nothing has changed but the syntax. This section explains why that's not the case.

7.2.1 Filter function for ranges

The `filter` transformation still needs to return a collection of people so you can call `transform` on it. This is where the magic of ranges comes into play. A range is an abstraction that represents a collection of items, but nobody said it's a collection—it just needs to behave like one. It needs to have a start, to know its end, and to allow you to get to each of its elements.

Instead of having `filter` return a collection like `std::vector`, it'll return a range structure whose `begin` iterator will be a smart proxy iterator that points to the first element in the source collection that satisfies the given predicate. And the `end` iterator will be a proxy for the original collection's `end` iterator. The only thing the proxy iterator needs to do differently than the iterator from the original collection is to point only at the elements that satisfy the filtering predicate (see figure 7.1).

In a nutshell, every time the proxy iterator is incremented, it needs to find the next element in the original collection that satisfies the predicate.

Listing 7.2 Increment operator for the filtering proxy iterator

```
auto& operator++()
{
    ++m_current_position;
    m_current_position =
        std::find_if(m_current_position,
                     m_end,
                     m_predicate);

    return *this;
}
```

Iterator to the collection you're filtering. When the proxy iterator is to be incremented, find the first element after the current one that satisfies the predicate.

Starts the search from the next element

If no more elements satisfy the predicate, returns an iterator pointing to the end of the source collection, which is also the end of the filtered range

With a proxy iterator for filtering, you don't need to create a temporary collection containing copies of the values in the source collection that satisfy the predicate. You've created a new *view* of existing data.

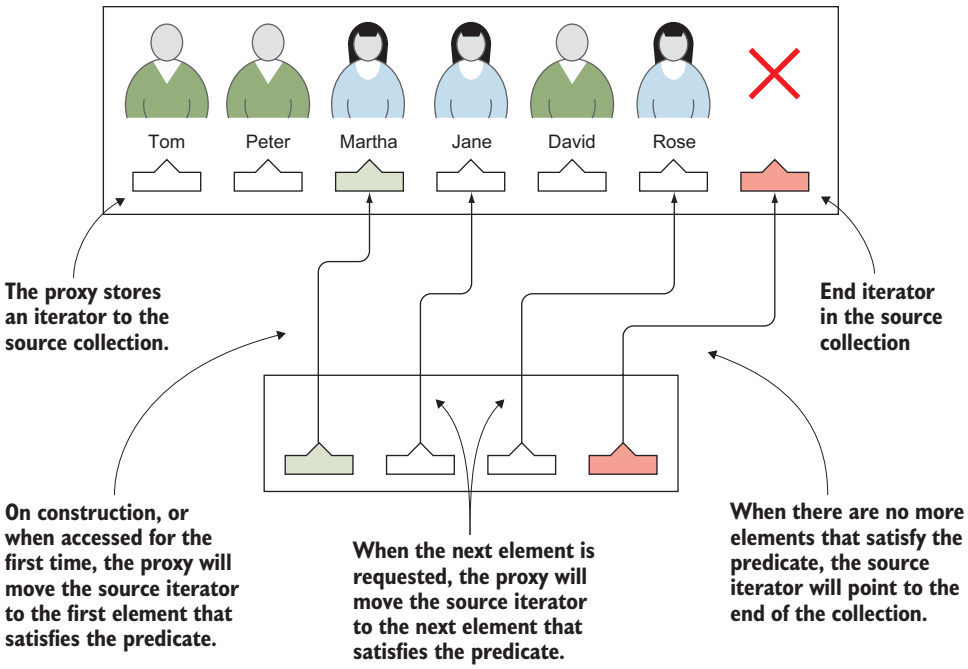

Figure 7.1 The view created by `filter` stores an iterator to the source collection. The iterator points to only the elements that satisfy the filtering predicate. The user of this view will be able to use it as if it were a normal collection of people with only three elements in it: Martha, Jane, and Rose.

You can pass this view to the `transform` algorithm, and it'll work just as well as it would on a real collection. Every time it requires a new value, it requests the proxy iterator to be moved one place to the right, and it moves to the next element that satisfies the predicate in the source collection. The `transform` algorithm goes through the original collection of people but can't *see* any person who isn't female.

7.2.2 Transform function for ranges

In a manner similar to `filter`, the `transform` function doesn't need to return a new collection. It also can return a view over the existing data. Unlike `filter` (which returns a new view that contains the same items as the original collection, just not all of them), `transform` needs to return the same number of elements found in the source collection, but it doesn't give access to the elements directly. It returns each element from the source collection, but transformed.

The increment operator doesn't need to be special; it just needs to increment the iterator to the source collection. This time, the operator to dereference the iterator will be different. Instead of returning the value in the source collection, you first apply the transformation function to it (see figure 7.2).

Listing 7.3 Dereference operator for the transformation proxy iterator

```
auto operator*() const
{
    return m_function(
            *m_current_position
        );
}
```

Gets the value from the original collection, applies the transformation function to it, and returns it as the value the proxy iterator points to

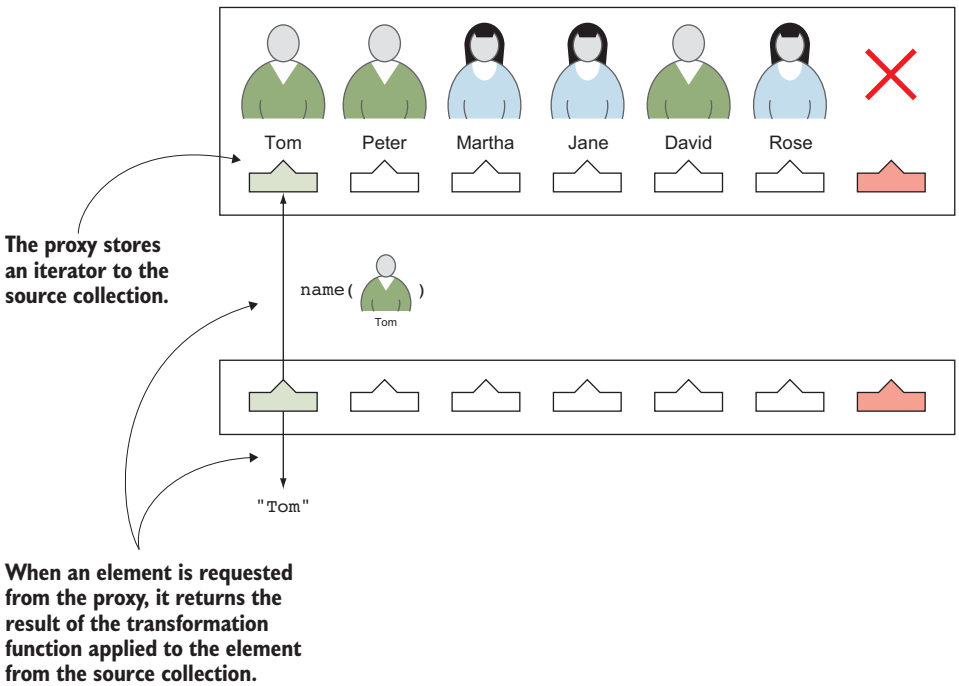

Figure 7.2 The view created by `transform` stores an iterator to the source collection. The iterator accesses all the elements in the source collection, but the view doesn't return them directly; it first applies the transformation function to the element and returns its result.

This way, just as with `filter`, you avoid creating a new collection that holds the transformed elements. You're creating a view that instead of showing original elements as they are, shows them transformed. Now you can pass the resulting range to another transformation, or you can assign it to a proper collection as in the example.

7.2.3 Lazy evaluation of range values

Even if you have two range transformations in the example—one `filter` and one `transform`—the calculation of the resulting collection takes only a single pass through the source collection, just as in the case of a handwritten `for` loop. Range views are evaluated lazily: when you call `filter` or `transform` on a collection, it defines a view; it doesn't evaluate a single element in that range.

Let's modify the example to fetch the names of the first three females in the collection. You can use the `take(n)` range transformation, which creates a new view over the source range that shows only the first *n* items in that range (or fewer if the source range has fewer than `n` elements):

```
std::vector<std::string> names = people | filter(is_female)
                                        | transform(name)
                                        | take(3);
```

Let's analyze this snippet part by part:

1. When `people | filter(is_female)` is evaluated, nothing happens other than a new view being created. You haven't accessed a single person from the `people` collection, except potentially to initialize the iterator to the source collection to point to the first item that satisfies the `is_female` predicate.
2. You pass that view to `| transform(name)`. The only thing that happens is that a new view is created. You still haven't accessed a single person or called the `name` function on any of them.
3. You apply `| take(3)` to that result. Again, is creates a new view and nothing else.
4. You need to construct a vector of strings from the view you got as the result of the `| take(3)` transformation.

To create a vector, you must know the values you want to put in it. This step goes through the view and accesses each of its elements.

When you try to construct the vector of names from the range, all the values in the range have to be evaluated. For each element added to the vector, the following things happen (see figure 7.3):

1. You call the dereference operator on the proxy iterator that belongs to the range view returned by `take`.
2. The proxy iterator created by `take` passes the request to the proxy iterator created by `transform`. This iterator passes on the request.
3. You try to dereference the proxy iterator defined by the `filter` transformation. It goes through the source collection and finds and returns the first person that satisfies the `is_female` predicate. This is the first time you access any of the persons in the collection, and the first time the `is_female` function is called.
4. The person retrieved by dereferencing the `filter` proxy iterator is passed to the `name` function, and the result is returned to the `take` proxy iterator, which passes it on to be inserted into the `names` vector.

When an element is inserted, you go to the next one, and then the next one, until you reach the end. Now, because you've limited your view to three elements, you don't need to access a single person in the `people` collection after you find the third female.

This is lazy evaluation at work. Even though the code is shorter and more generic than the equivalent handwritten `for` loop, it does exactly the same thing and has no performance penalties.

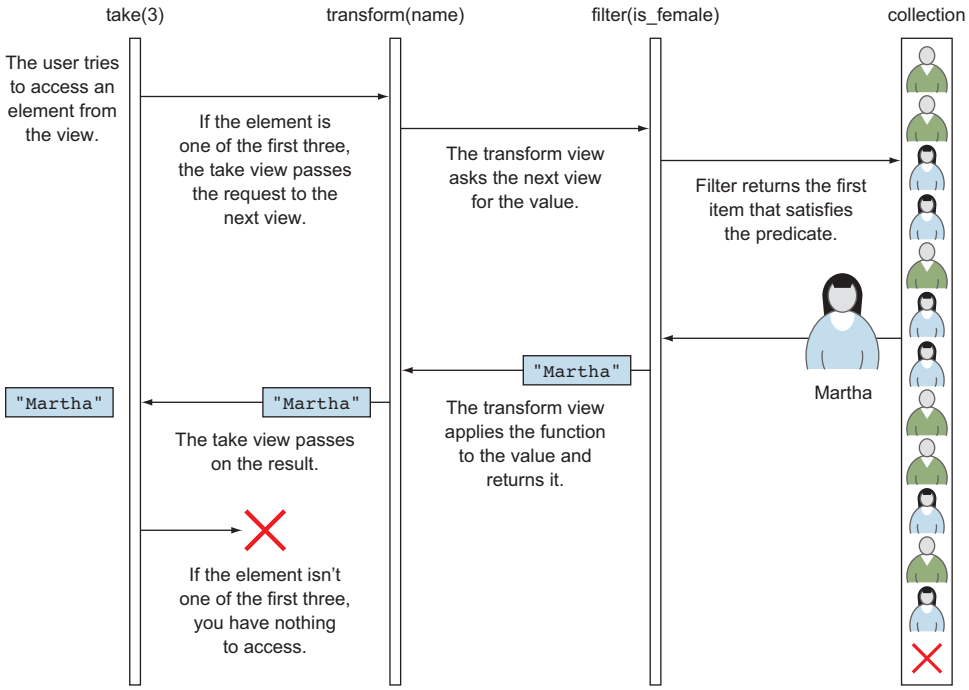

Figure 7.3 When accessing an element from the view, the view proxies the request to the next view in the composite transformation, or to the collection. Depending on the type of the view, it may transform the result, skip elements, traverse them in a different order, and so on.

7.3 Mutating values through ranges

Although many useful transformations can be implemented as simple views, some require changing the original collection. We'll call these transformations *actions* as opposed to *views*.

One common example for the action transformation is sorting. To be able to sort a collection, you need to access all of its elements and reorder them. You need to change the original collection, or create and keep a sorted copy of the whole collection. The latter is especially important when the original collection isn't randomly accessible (a linked list, for example) and can't be sorted efficiently; you need to copy its elements into a new collection that's randomly accessible and sort that one instead.

Views and actions in the range-v3 library

As mentioned earlier, because the range-v3 library is used as the base for the proposal for extending the STL with ranges, we'll use it in the code examples, and we'll use its nomenclature. The range transformations that create views such as `filter`, `transform`, and `take` live in the `ranges::v3::view` namespace, whereas the actions live in `ranges::v3::action`. It's important to differentiate between these two, so we'll specify the namespaces `view` and `action` from now on.

Imagine you have a function `read_text` that returns text represented as a vector of words, and you want to collect all the words in it. The easiest way to do this is to sort the words and then remove consecutive duplicates. (We'll consider all words to be lowercase in this example, for the sake of simplicity.)

You can get the list of all words that appear in given text by piping the result of the `read_text` function through `sort` and `unique` actions, as illustrated in figure 7.4 and shown here:

```
std::vector<std::string> words =
        read_text() | action::sort
                    | action::unique;
```

Because you're passing a temporary to the `sort` action, it doesn't need to create a copy to work on; it can reuse the vector returned by the `read_text` function and do the sorting in place. The same goes for `unique`—it can operate directly on the result of the `sort` action. If you wanted to keep the intermediary result, you would use `view::unique` instead, which doesn't operate on a real collection, but creates a view that skips all repeated consecutive occurrences of a value.

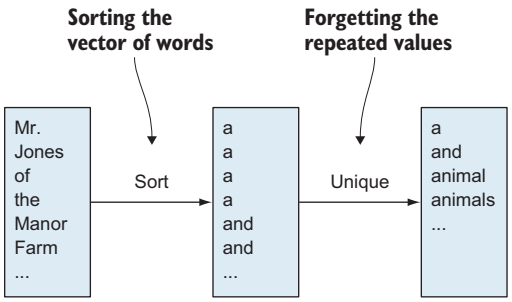

Figure 7.4 To get a list of words that appear in text, it's sufficient to sort them and then remove the consecutive repeated values.

This is an important distinction between views and actions. A view transformation creates a lazy view over the original data, whereas an action works on an existing collection and performs its transformation eagerly.

Actions don't have to be performed on temporaries. You can also act on lvalues by using the operator `|=`, like so:

```
std::vector<std::string> words = read_text();
words |= action::sort | action::unique;
```

This combination of views and actions gives you the power to choose when you want something to be done lazily and when you want it to be done eagerly. The benefits

of having this choice are that you can be lazy when you don't expect all items in the source collection to need processing, and when items don't need to be processed more than once; and you can be eager to calculate all elements of the resulting collection if you know they'll be accessed often.

7.4 Using delimited and infinite ranges

We started this chapter with the premise that a range is a structure that holds one iterator to the beginning and one to the end—exactly what STL algorithms take, but in a single structure. The `end` iterator is a strange thing. You can never dereference it, because it points to an element after the last element in the collection. You usually don't even move it. It's mainly used to test whether you've reached the end of a collection:

```
auto i = std::begin(collection);
const auto end = std::end(collection);
for (; i != end; i++) {
    // ...
}
```

It doesn't really need to be an iterator—it just needs to be something you can use to test whether you're at the end. This special value is called a *sentinel*, and it gives you more freedom when implementing a test for whether you've reached the end of a range. Although this functionality doesn't add much when you're working with ordinary collections, it allows you to create delimited and infinite ranges.

7.4.1 Using delimited ranges to optimize handling input ranges

A *delimited range* is one whose end you don't know in advance—but you have a predicate function that can tell you when you've reached the end. Examples are null-terminated strings: you need to traverse the string until you reach the `'\0'` character, or traverse the input streams and read one token at a time until the stream becomes invalid—until you fail to extract a new token. In both cases, you know the beginning of the range, but in order to know where the end is, you must traverse the range item by item until the end test returns `true`.

Let's consider the input streams and analyze the code that calculates the sum of the numbers it reads from the standard input:

```
std::accumulate(std::istream_iterator<double>(std::cin),
                std::istream_iterator<double>(),
                0);
```

You're creating two iterators in this snippet: one proper iterator, which represents the start of the collection of doubles read from `std::cin`, and one special iterator that doesn't belong to any input stream. This iterator is a special value that the `std::accumulate` algorithm will use to test whether you've reached the end of the collection; it's an iterator that behaves like a sentinel.

The `std::accumulate` algorithm will read values until its traversal iterator becomes equal to the end iterator. You need to implement `operator==` and `operator!=` for

`std::istream_iterator`. The equality operator must work with both the proper iterators and special sentinel values. The implementation has a form like this:

```
template <typename T>
bool operator==(const std::istream_iterator<T>& left,
                const std::istream_iterator<T>& right)
{
    if (left.is_sentinel() && right.is_sentinel()) {
          return true;

     } else if (left.is_sentinel()) {
          // Test whether sentinel predicate is
          // true for the right iterator

     } else if (right.is_sentinel()) {
          // Test whether sentinel predicate is
          // true for the left iterator

    } else {
          // Both iterators are normal iterators,
          // test whether they are pointing to the
          // same location in the collection
    }
}
```

You need to cover all the cases—whether the left iterator is a sentinel, and the same for the right iterator. These are checked in each step of an algorithm.

This approach is inefficient. It would be much easier if the compiler knew something was a sentinel at compile time. This is possible if you lift the requirement that the end of a collection has to be an iterator—if you allow it to be anything that can be equally compared to a proper iterator. This way, the four cases in the previous code become separate functions, and the compiler will know which one to call based on the involved types. If it gets two iterators, it'll call `operator==` for two iterators; if it gets an iterator and a sentinel, it'll call `operator==` for an iterator and a sentinel; and so on.

> **Range-based for loops and sentinels**
>
> The range-based `for` loop, as defined in C++11 and C++14, requires both the `begin` and `end` to have the same type; they need to be iterators. The sentinel-based ranges can't be used with the range-based `for` loop in C++11 and C++14. This requirement was removed in C++17. You can now have different types for the `begin` and `end`, which effectively means the `end` can be a sentinel.

7.4.2 Creating infinite ranges with sentinels

The sentinel approach gives you optimizations for delimited ranges. But there's more: you're now able to easily create infinite ranges as well. Infinite ranges don't have an end, like the range of all positive integers. You have a start—the number 0—but no end.

Although it's not obvious why you'd need infinite data structures, they come in handy from time to time. One of the most common examples for using a range of integers is

enumerating items in another range. Imagine you have a range of movies sorted by their scores, and you want to write out the first 10 to the standard output along with their positions as shown in listing 7.4 (see example:top-movies).

To do this, you can use the `view::zip` function. It takes two ranges[1] and pairs the items from those ranges. The first element in the resulting range will be a pair of items: the first item from the first range and the first item from the second range. The second element will be a pair containing the second item from the first range and the second item from the second range, and so on. The resulting range will end as soon as any of the source ranges ends (see figure 7.5).

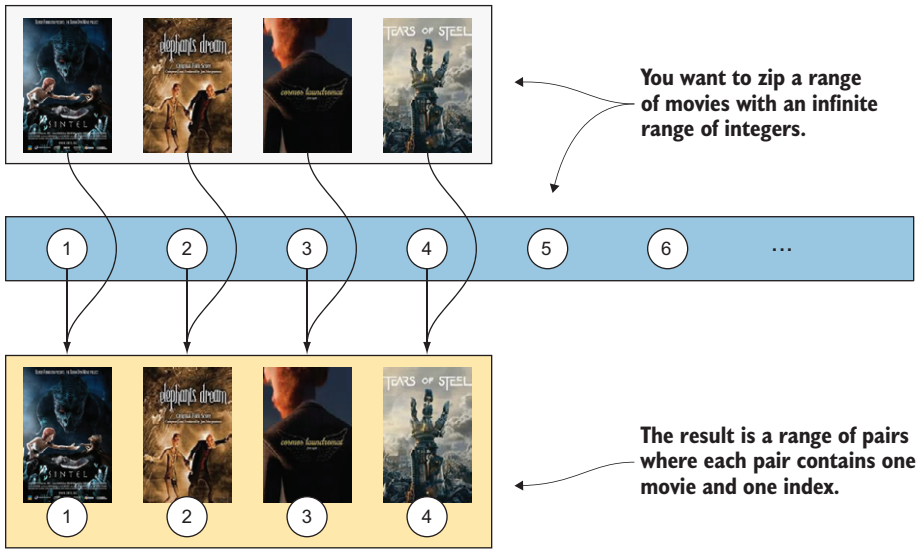

Figure 7.5 The range doesn't have a notion of an item index. If you want to have the indices for the elements in a range, you can zip the range with the range of integers. You'll get a range of pairs, and each pair will contain an item from the original range along with its index.

Listing 7.4 Writing out the top 10 movies along with their positions

```
template <typename Range>
void write_top_10(const Range& xs)
{
    auto items =
        view::zip(xs, view::ints(1))
            | view::transform([](const auto& pair) {
                  return std::to_string(pair.second) +
                         " " + pair.first;
              })
            | view::take(10);
```

Zips the range of movies with the range of integers, starting with 1. This gives a range of pairs: a movie name and the index.

The transform function takes a pair and generates a string containing the rank of the movie and the movie name.

You're interested in the first 10 movies.

[1] `view::zip` can also zip more than two ranges. The result will be a range of n-tuples instead of a range of pairs.

```
    for (const auto& item : items) {
        std::cout << item << std::endl;
    }
}
```

Instead of the infinite range of integers, you could use integers from 1 to `xs.length()` to enumerate the items in `xs`. But that wouldn't work as well. As you've seen, you could have a range and not know its end, you probably couldn't tell its size without traversing it. You'd need to traverse it twice: once to get its size, and once for `view::zip` and `view::transform` to do their magic. This is not only inefficient, but also impossible to do with some range types. Ranges such as the input stream range can't be traversed more than once; after you read a value, you can't read it again.

Another benefit of infinite ranges isn't in using them, but in designing your code to be able to work on them. This makes your code more generic. If you write an algorithm that works on infinite ranges, it'll work on a range of any size, including a range whose size you don't know.

7.5 Using ranges to calculate word frequencies

Let's move on to a more complicated example to see how programs can be more elegant if you use ranges instead of writing the code in the old style. You'll reimplement the example from chapter 4: calculating the frequencies of the words in a text file. To recap, you're given a text, and you want to write out the *n* most frequently occurring words in it. We'll break the problem into a composition of smaller transformations as before, but we're going to change a few things to better demonstrate how range views and actions interact with each other.

The first thing you need to do is get a list of lowercase words without any special characters in them. The data source is the input stream. You'll use `std::cin` in this example.

The range-v3 library provides a class template called `istream_range` that creates a stream of tokens from the input stream you pass to it:

```
std::vector<std::string> words =
        istream_range<std::string>(std::cin);
```

In this case, because the tokens you want are of `std::string` type, the range will read word by word from the standard input stream. This isn't enough, because you want all the words to be lowercase and you don't want punctuation characters included. You need to transform each word to lowercase and remove any nonalphanumeric characters (see figure 7.6).

Listing 7.5 Getting a list of lowercase words that contain only letters or digits

```
std::vector<std::string> words =
        istream_range<std::string>(std::cin)
        | view::transform(string_to_lower)
        | view::transform(string_only_alnum)
        | view::remove_if(&std::string::empty);
```

Makes all words lowercase

Keeps only letters and digits

You may get empty strings as the result when a token doesn't contain a single letter or a digit, so you need to skip those.

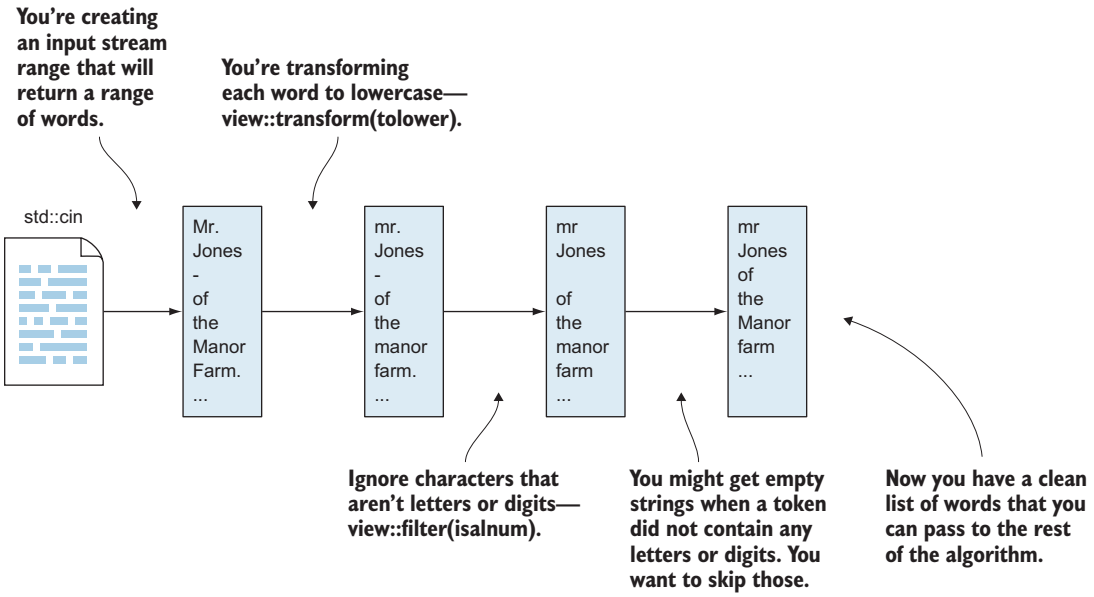

Figure 7.6 You have an input stream from which to read words. Before you can calculate the word frequencies, you need a list of words converted to lowercase with all punctuation removed.

For the sake of completeness, you also need to implement `string_to_lower` and `string_only_alnum` functions. The former is a transformation that converts each character in a string to lowercase, and the latter is a filter that skips characters that aren't alphanumeric. A `std::string` is a collection of characters, so you can manipulate it like any other range:

```
std::string string_to_lower(const std::string& s)
{
    return s | view::transform(tolower);
}

std::string string_only_alnum(const std::string& s)
{
    return s | view::filter(isalnum);
}
```

You have all the words to process, and you need to sort them (see figure 7.7). The `action::sort` transformation requires a randomly accessible collection, so it's lucky you declared `words` to be a `std::vector` of strings. You can request it to be sorted:

```
words |= action::sort;
```

Now that you have a sorted list, you can easily group the same words by using `view::group_by`. It'll create a range of word groups (the groups are, incidentally, also ranges). Each group will contain the same word multiple times—as many times as it appeared in the original text.

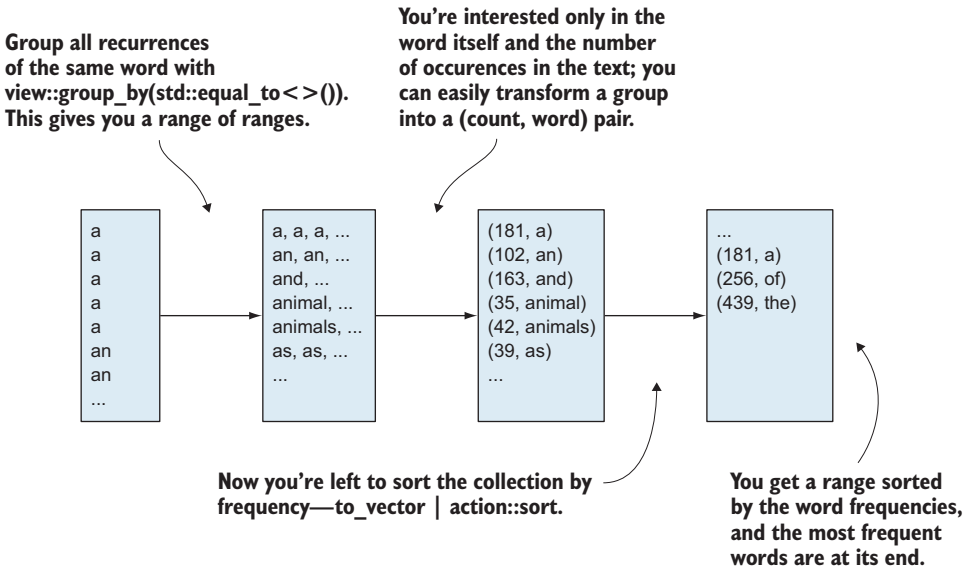

Figure 7.7 You get a range of sorted words. You need to group the same words, count how many words you have in each group, and then sort them all by the number of occurrences.

You can transform the range into pairs; the first item in the pair is the number of items in a group, and the second is the word. This will give you a range containing all the words in the original text along with the number of occurrences for each.

Because the frequency is the first item in a pair, you can pass this range through `action::sort`. You can do so as in the previous code snippet, by using `operator|=`, or you can do it inline by first converting the range to a vector, as shown next (see example:word-frequency). This will allow you to declare the `results` variable as `const`.

Listing 7.6 Getting a sorted list of frequency-word pairs from a sorted list

```
const auto results =
    words | view::group_by(std::equal_to<>())
          | view::transform([](const auto& group) {
                const auto begin = std::begin(group);
                const auto end   = std::end(group);
                const auto count = distance(begin, end);
                const auto word  = *begin;

                return std::make_pair(count, word);
            })
          | to_vector | action::sort;
```

Groups multiple occurrences of words from the words range

Gets the size of each group, and returns a pair consisting of the word frequency and the word

To sort the words by frequency, you first need to convert the range into a vector.

The last step is to write the n most frequent words to the standard output. Because the results have been sorted in ascending order, and you need the most frequent words,

not the least frequent ones, you must first reverse the range and then take the first *n* elements:

```
for (auto value: results | view::reverse
                         | view::take(n)) {
    std::cout << value.first << " " << value.second << std::endl;
}
```

That's it. In fewer than 30 lines, you've implemented the program that originally took a dozen pages. You've created a set of easily composable, highly reusable components; and, not counting the output, you haven't used any `for` loops.

Range-based for loop and ranges

As previously mentioned, the range-based `for` loop started supporting sentinels in C++17. The preceding code won't compile on older compilers. If you're using an older compiler, the range-v3 library provides a convenient `RANGES_FOR` macro that can be used as a replacement for the range-based `for`:

```
RANGES_FOR (auto value, results | view::reverse
                                | view::take(n)) {
   std::cout << value.first << " " << value.second << std::endl;
}
```

Additionally, if you sorted the range of words the same way you sorted the list of results (without `operator|=`), you'd have no mutable variables in your program.

TIP For more information and resources about the topics covered in this chapter, see https://forums.manning.com/posts/list/43776.page.

Summary

- One frequent source of errors when using STL algorithms is passing incorrect iterators to them—sometimes even iterators belonging to separate collections.
- Some collection-like structures don't know where they end. For those, it's customary to provide sentinel-like iterators; these work but have unnecessary performance overhead.
- The ranges concept is an abstraction over any type of iterable data. It can model normal collections, input and output streams, database query result sets, and more.
- The ranges proposal is planned for inclusion in C++20, but libraries provide the same functionality today.
- Range views don't own the data, and they can't change it. If you want to operate on and change existing data, use actions instead.
- Infinite ranges are a nice measure of algorithm generality. If something works for infinite ranges, it'll work for finite ones as well.
- By using ranges and thinking of program logic in terms of range transformations, you can decompose the program into highly reusable components.

PROGRAMMING LANGUAGES/C++

Functional Programming in C++

How to improve your C++ programs using functional techniques

Ivan Čukić

Well-written code is easier to test and reuse, simpler to parallelize, and less error prone. Mastering the functional style of programming can help you tackle the demands of modern apps and will lead to simpler expression of complex program logic, graceful error handling, and elegant concurrency. C++ supports FP with templates, lambdas, and other core language features, along with many parts of the STL.

Functional Programming in C++ helps you unleash the functional side of your brain, as you gain a powerful new perspective on C++ coding. You'll discover dozens of examples, diagrams, and illustrations that break down the functional concepts you can apply in C++, including lazy evaluation, function objects and invokables, algebraic data types, and more. As you read, you'll match FP techniques with practical scenarios where they offer the most benefit.

What's inside

- Writing safer code with no performance penalties
- Explicitly handling errors through the type system
- Extending C++ with new control structures
- Composing tasks with DSLs

Written for developers with two or more years of experience coding in C++.

Ivan Čukić is a core developer at KDE and has been coding in C++ since 1998. He teaches modern C++ and functional programming at the Faculty of Mathematics at the University of Belgrade.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/functional-programming-in-c-plus-plus

"Offers precise, easy-to-understand, and engaging explanations of functional concepts."
—Sumant Tambe, LinkedIn

"An excellent read. Comprehensive code examples illustrate the implementation of functional programming patterns using C++14/C++17 constructs."
—Keerthi Shetty
FactSet Research Systems

"Provides elegant, easy-to-grasp, ready-to-use examples that will improve the way you think about coding."
—Nikos Athanasiou, BETA CAE Systems

"Presents a new way of writing quality software and a new way of thinking."
—Gian Lorenzo Meocci, CommProve

"Particularly valuable for intermediate/advanced C++ developers who want to embrace reactive-style programming."
—Marco Massenzio, Apple

ISBN-13: 978-1-61729-381-8
ISBN-10: 1-61729-381-4
54999
9 781617 293818

MANNING US $49.99 / Can $65.99 [including eBook]