5
Algorithms

The use of containers from the standard library is widely employed among C++ programmers. It's rare to find C++ code bases without references to `std::vector` or `std::string`, for example. However, in my experience, standard library algorithms are much less frequently used, even though they offer the same kind of benefits as containers:

- They can be used as building blocks when solving complex problems
- They are well documented (including references, books, and videos)
- Many C++ programmers are already familiar with them
- Their space and runtime costs are known (complexity guarantees)
- Their implementations are well crafted and efficient

If this wasn't enough, C++ features such as lambdas, execution policies, concepts, and ranges have all made the standard algorithms more powerful and, at the same time, friendlier to use.

In this chapter, we will take a look at how we can write efficient algorithms in C++ using the **Algorithm library**. You will learn the benefits of using the standard library algorithms as building blocks in your application, both performance-wise and readability-wise.

In this chapter, you will learn about:

- The algorithms in the C++ standard library
- Iterators and ranges – the glue between the containers and the algorithms
- How to implement a generic algorithm that can operate on standard containers
- Best practices when using C++ standard algorithms

Let's begin by taking a look at the standard library algorithms and how they came to be what they are today.

Introducing the standard library algorithms

Integrating the standard library algorithms into your C++ vocabulary is important. In this introduction, I will present a set of common problems that can be solved effectively by using the standard library algorithms.

C++20 comes with a dramatic change to the Algorithm library by the introduction of the **Ranges library** and the language feature of *C++ concepts*. So, before we start, we need a brief background of the history of the C++ standard library.

Evolution of the standard library algorithms

You have probably heard about STL algorithms or STL containers. And hopefully, you have heard about the new **Ranges library** introduced with C++20. There have been a lot of additions to the standard library in C++20. And before going further, I need to clear up some terminology. We'll start with the STL.

The **STL**, or the **Standard Template Library**, was initially the name of a library added to the C++ standard library in the 1990s. It contained algorithms, containers, iterators, and function objects. The name has

been sticky, and we have become accustomed to hearing and talking about the STL algorithms and containers. The C++ standard does not mention the STL however; instead, it talks about the *standard library* and their individual components such as the **Iterator library** and the **Algorithm library**. I will try to avoid using the name STL in this book and instead talk about the standard library or individual libraries when needed.

Now on to the Ranges library and what I will call the **constrained algorithms**. The Ranges library is a library added to the standard library in C++20 that introduced a completely new header called `<ranges>`, which we will talk more about in the next chapter. But the addition of the Ranges library also had a big impact on the `<algorithm>` header by introducing overloaded versions of all previously existing algorithms. I will refer to these algorithms as the *constrained algorithms* because they are constrained using C++ concepts. So, the `<algorithm>` header now includes the old iterator-based algorithms and the new algorithms constrained with C++ concepts that can operate on ranges. This means that the algorithms we will discuss in this chapter come in two flavors, as demonstrated in the following example:

```
#include <algorithm>
#include <vector>
auto values = std::vector{9, 2, 5, 3, 4};
// Sort using the std algorithms
std::sort(values.begin(), values.end());
// Sort using the constrained algorithms under std::ranges
std::ranges::sort(values);
std::ranges::sort(values.begin(), values.end());
```

Note that both versions of `sort()` live in the `<algorithm>` header but they are distinguished by different namespaces and signatures. This chapter will use both flavors, but in general, I recommend using the new

constrained algorithms whenever possible. The benefits will hopefully become apparent after reading this chapter.

Now you are ready to start learning about how you can use ready-made algorithms to solve common problems.

Solving everyday problems

I will list here some common scenarios and useful algorithms just to give you a taste of the algorithms that are available in the standard library. The are many algorithms in the library, and I will only present a few in this section. For a quick but complete overview of the standard library algorithms, I recommend the talk from *CppCon 2018, 105 STL Algorithms in Less Than an Hour*, by Jonathan Boccara, available at **https://sched.co/FnJh**.

Iterating over a sequence

It's useful to have a short helper function that can print the elements of a sequence. The following generic function works with any container that holds elements that can be printed to an output stream using `operator<<()`:

```
void print(auto&& r) {
  std::ranges::for_each(r, [](auto&& i) { std::cout << i << ' '; });
}
```

The `print()` function is using `for_each()`, which is an algorithm imported from the `<algorithm>` header. `for_each()` calls the function that we provide once for each element in the range. The return value of the

function we provide is ignored and has no effect on the sequence we pass to `for_each()`. We can use `for_each()` for side effects such as printing to `stdout` (which we do in this example).

A similar, very general algorithm is `transform()`. It also calls a function for each element in a sequence, but instead of ignoring the return value, it stores the return value of the function in an output sequence, like this:

```
auto in = std::vector{1, 2, 3, 4};
auto out = std::vector<int>(in.size());
auto lambda = [](auto&& i) { return i * i; };
std::ranges::transform(in, out.begin(), lambda);
print(out);
// Prints: "1 4 9 16"
```

This code snippet also demonstrates how we can use our `print()` function defined earlier. The `transform()` algorithm will call our lambda once for each element in the input range. To specify where the output will be stored, we provide `transform()` with an output iterator, `out.begin()`. We will talk a lot more about iterators later on in this chapter.

With our `print()` function in place and a demonstration of some of the most general algorithms, we will move on to look at some algorithms for generating elements.

Generating elements

Sometimes we need to assign a sequence of elements with some initial values or reset an entire sequence. The following example fills a vector with the value -1:

```
auto v = std::vector<int>(4);
std::ranges::fill(v, -1);
print(v);
// Prints "-1 -1 -1 -1 "
```

The next algorithm, `generate()`, calls a function for each element and stores the return value at the current element:

```
auto v = std::vector<int>(4);
std::ranges::generate(v, std::rand);
print(v);
// Possible output: "1804289383 846930886 1681692777 1714636915 "
```

In the preceding example, the `std::rand()` function is called once for each element.

The last generating algorithm I will mention is `std::iota()` from the `<numeric>` header. It generates values in increasing order. The start value must be specified as a second argument. Here is a short example that generates values between 0 and 5:

```
auto v = std::vector<int>(6);
std::iota(v.begin(), v.end(), 0);
print(v); // Prints: "0 1 2 3 4 5 "
```

This sequence is already sorted, but it more commonly happens that you have an unordered collection of elements that needs sorting, which we will look at next.

Sorting elements

Sorting elements is a very common operation. There are sorting-algorithm alternatives that are good to know about, but in this introduction, I will only show the most conventional version, simply named `sort()`:

```
auto v = std::vector{4, 3, 2, 3, 6};
std::ranges::sort(v);
print(v);      // Prints: "2 3 3 4 6 "
```

As mentioned, this is not the only way to sort, and sometimes we can use a partial sorting algorithm to gain performance. We will talk more abort sorting later in this chapter.

Finding elements

Another very common task is to find out whether a specific value is in a collection or not. Maybe we want to know how many instances of some specific value there are in a collection. These algorithms that search for values can be implemented more efficiently if we know that the collection is already sorted. You saw this in *Chapter 3*, *Analyzing and Measuring Performance*, where we compared linear search with binary search.

Here we begin with the `find()` algorithm, which doesn't require a sorted collection:

```
auto col = std::list{2, 4, 3, 2, 3, 1};
auto it = std::ranges::find(col, 2);
if (it != col.end()) {
```

```
    std::cout << *it << '\n';
  }
```

If the element we are looking for could not be found, `find()` returns the `end()` iterator of the collection. In the worst case, `find()` needs to inspect all elements in the sequence, therefore it runs in *O(n)* time.

Finding using binary search

If we know that the collection is already sorted, we can use one of the binary search algorithms: `binary_search()`, `equal_range()`, `upper_bound()`, or `lower_bound()`. If we are using these functions with containers that provide random access to their elements, they are all guaranteed to run in *O(log n)* time. You will gain a better understanding of how algorithms can provide complexity guarantees, even though they are operating on different containers, when we talk about iterators and ranges later in this chapter (there's a section coming up named, funnily enough, *Iterators and Ranges*).

In the following examples, we will use a sorted `std::vector` with the following elements:

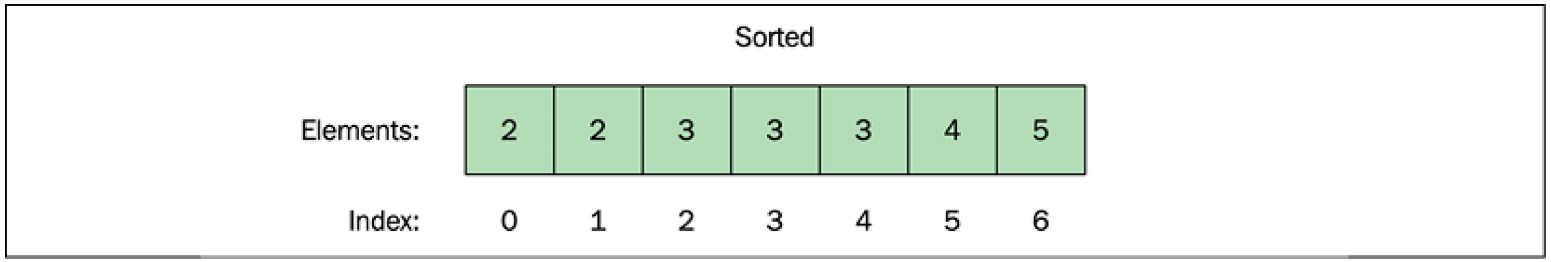


Figure 5.1: An sorted std::vector with seven elements

The `binary_search()` function returns `true` or `false` depending on whether the value we searched for could be found:

```
auto v = std::vector{2, 2, 3, 3, 3, 4, 5};    // Sorted!
bool found = std::ranges::binary_search(v, 3);
std::cout << std::boolalpha << found << '\n'; //   Output: true
```

Before calling `binary_search()`, you should be absolutely sure that the collection is sorted. We can easily assert this in our code with the use of `is_sorted()` as follows:

```
assert(std::ranges::is_sorted(v));
```

This check will run in *O(n)*, but will only be called when asserts are activated and hence will not affect the performance of your final program.

The sorted collection we are working with contains multiple 3s. What if we want to know the position of the first 3 or last 3 in the collection? In that case, we can use `lower_bound()` for finding the first 3, or `upper_bound()` for finding the element past the last 3:

```
auto v = std::vector{2, 2, 3, 3, 3, 4, 5};
auto it = std::ranges::lower_bound(v, 3);
if (it != v.end()) {
  auto index = std::distance(v.begin(), it);
  std::cout << index << '\n'; // Output: 2
}
```

This code will output `2` since that is the index of the first 3. To get the index of an element from an iterator, we use `std::distance()` from the `<iterator>` header.

In the same manner, we can use `upper_bound()` to get an iterator to the element *past* the last 3:

```
const auto v = std::vector{2, 2, 3, 3, 3, 4, 5};
auto it = std::ranges::upper_bound(v, 3);
if (it != v.end()) {
  auto index = std::distance(v.begin(), it);
  std::cout << index << '\n'; // Output: 5
}
```

If you want both the upper bound and lower bound, you can instead use `equal_range()`, which returns the subrange of the collection containing the 3s:

```
const auto v = std::vector{2, 2, 3, 3, 3, 4, 5};
auto subrange = std::ranges::equal_range(v, 3);
if (subrange.begin() != subrange.end()) {
  auto pos1 = std::distance(v.begin(), subrange.begin());
  auto pos2 = std::distance(v.begin(), subrange.end());
  std::cout << pos1 << " " << pos2 << '\n';
} // Output: "2 5"
```

Now let's explore some other useful algorithms for inspecting a collection.

Testing for certain conditions

There are three very handy algorithms called `all_of()`, `any_of()`, and `none_of()`. They all take a range, a unary predicate (a function that takes one argument and returns `true` or `false`), and an optional projection function.

Let's say we have a list of numbers and a small lambda that determines whether a number is negative or not:

```
const auto v = std::vector{3, 2, 2, 1, 0, 2, 1};
const auto is_negative = [](int i) { return i < 0; };
```

We can check if none of the numbers are negative by using `none_of()`:

```
if (std::ranges::none_of(v, is_negative)) {
  std::cout << "Contains only natural numbers\n";
}
```

Further, we can ask if all elements in the list are negative by using `all_of()`:

```
if (std::ranges::all_of(v, is_negative)) {
  std::cout << "Contains only negative numbers\n";
}
```

Lastly, we can see whether the list contains at least one negative number using `any_of()`:

```
if (std::ranges::any_of(v, is_negative)) {
  std::cout << "Contains at least one negative number\n";
}
```

It's easy to forget about these small, handy building blocks that reside in the standard library. But once you get into the habit of using them, you will never look back and start writing these by hand again.

Counting elements

The most obvious way to count the number of elements that equals some value is to call `count()`:

```
const auto numbers = std::list{3, 3, 2, 1, 3, 1, 3};
int n = std::ranges::count(numbers, 3);
std::cout << n;              // Prints: 4
```

The `count()` algorithm runs in linear time. However, if we know that the sequence is sorted and we are using a vector or some other random-access data structure, we could instead use `equal_range()`, which will run in *O(log n)* time. The following is an example:

```
const auto v = std::vector{0, 2, 2, 3, 3, 4, 5};
assert(std::ranges::is_sorted(v)); // O(n), but not called in release
auto r = std::ranges::equal_range(v, 3);
int n = std::ranges::size(r);
std::cout << n;              // Prints: 2
```

The `equal_range()` function finds the subrange that contains all elements with the value we want to count. Once the subrange is found, we can retrieve the length of the subrange using `size()` from the `<ranges>` header.

Minimum, maximum, and clamping

I want to mention a set of small but extremely useful algorithms that are essential knowledge for a seasoned C++ programmer. The functions `std::min()`, `std::max()`, and `std::clamp()` are sometimes forgotten and instead we too often find ourselves writing code like this:

```
const auto y_max = 100;
auto y = some_func();
if (y > y_max) {
  y = y_max;
}
```

The code ensures that the value of `y` is within a certain limit. This code works, but we can avoid the mutable variable and the `if` statement by using `std::min()` as follows:

```
const auto y = std::min(some_func(), y_max);
```

Both the mutable variable and the `if` statement that clutter our code have been eliminated by instead using `std::min()`. We can use `std::max()` for similar scenarios. If we want to limit a value to within both a minimum and a maximum value, we might do it like this:

```
const auto y = std::max(std::min(some_func(), y_max), y_min);
```

But, since C++17, we now have `std::clamp()` that does this for us in one function. So instead, we could just use `clamp()` as follows:

```
const auto y = std::clamp(some_func(), y_min, y_max);
```

Sometimes we need to find the extreme values in an unsorted collection of elements. For this purpose, we can use `minmax()`, which (unsurprisingly) returns the minimum and maximum values of a sequence. Combined with structured binding, we can print the extreme values as follows:

```
const auto v = std::vector{4, 2, 1, 7, 3, 1, 5};
const auto [min, max] = std::ranges::minmax(v);
std::cout << min << " " << max;     // Prints: "1 7"
```

We can also find the position of the minimum or maximum element by using `min_element()` or `max_element()`. Instead of returning the value, it returns an iterator pointing at the element we are looking for. In the following example, we are finding the minimum element:

```
const auto v = std::vector{4, 2, 7, 1, 1, 3};
const auto it = std::ranges::min_element(v);
std::cout << std::distance(v.begin(), it); // Output: 3
```

This snippet of code prints `3`, which is the index of the first minimum value that was found.

This was a brief introduction to some of the most common algorithms from the standard library. The run-time cost of algorithms is specified in the C++ standard and all library implementations need to adhere to these, even though the exact implementation can vary between different platforms. To understand how the complexity guarantees can be withheld for generic algorithms working with many different types of containers, we need to take a closer look at iterators and ranges.

Iterators and ranges

As seen in the previous examples, the standard library algorithms operate on iterators and ranges rather than container types. This section will focus on iterators and the new concept of ranges introduced in C++20. Using containers and algorithms correctly becomes easy once you have grasped iterators and ranges.

Introducing iterators

Iterators form the basis of the standard library algorithms and ranges. Iterators are the glue between data structures and algorithms. As you have already seen, C++ containers store their elements in very different ways. Iterators provide a generic way to navigate through the elements in a sequence. By having algorithms operate on iterators rather than container types, the algorithms become more generic and flexible since they do not depend on the type of container and the way the containers arrange their elements in memory.

At its core, an iterator is an object that represents a position in a sequence. It has two main responsibilities:

- Navigating in the sequence
- Reading and writing the value at its current position

The iterator abstraction is not at all a C++ exclusive concept, rather it exists in most programming languages. What differentiates the C++ implementation of the iterator concept from other programming languages is that C++ mimics the syntax of raw memory pointers.

Basically, an iterator could be considered an object with the same properties as a raw pointer; it can be stepped to the next element and dereferenced (if pointing to a valid address). The algorithms only use a few of the operations that a pointer allows, although the iterator may internally be a heavy object traversing a tree-like `std::map`.

Most of the algorithms found directly under the `std` namespace operate only on iterators, not containers (that is, `std::vector`, `std::map`, and so on). Many algorithms return iterators rather than values.

To be able to navigate in a sequence without going out of bounds, we need a generic way to tell when the iterator has reached the end of a sequence. That is what we have sentinel values for.

Sentinel values and past-the-end iterators

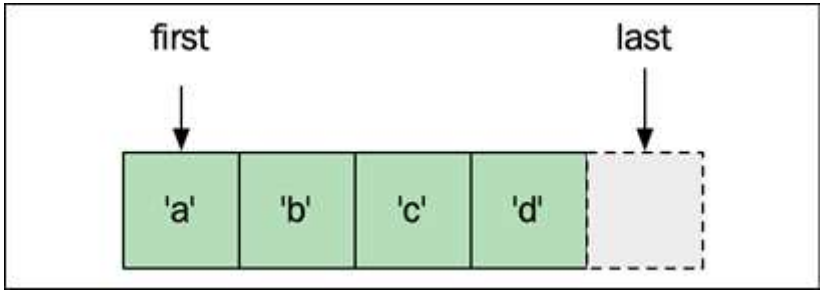
A **sentinel value** (or simply a sentinel) is a special value that indicates the end of a sequence. Sentinel values make it possible to iterate a sequence of values without knowing the size of the sequence in advance. An example usage of sentinel values are C-style strings that are null-terminated (in this case, the sentinel is the `'\0'` character). Instead of keeping track of the length of null-terminated strings, the pointer to the beginning of the string and the sentinel at the end is enough to define a sequence of characters.

The constrained algorithms use an iterator to define the first element in a sequence and a sentinel to indicate the end of the sequence. The only requirement of the sentinel is that it can be compared against the iterator, which in practice means that `operator==()` and `operator!=()` should be defined to accept combinations of a sentinel and an iterator:

```
bool operator=!(sentinel s, iterator i) {
 // ...
}
```

Now that you know what a sentinel is, how would we create a sentinel to indicate the end of a sequence? The trick here is to use something called a **past-the-end iterator** as a sentinel. It is simply an iterator that points to an element *after* (or past) the last element in the sequence we define. Take a look at the following code snippet and diagram:

```
auto vec = std::vector {
  'a','b','c','d'
};
auto first = vec.begin();
auto last = vec.end();
```



As seen in the preceding diagram, the `last` iterator now points to an imagined element after `'d'`. This makes it possible to iterate over all the elements in the sequence by using a loop:

```
for (; first != last; ++first) {
  char value = *first; // Dereference iterator
  // ...
```

We can use the past-the-end sentinel to compare it against our iterator, `it`, but we cannot dereference the sentinel since it doesn't point to an element of the range. This concept of past-the-end iterators has a long history and even works for built-in C arrays:

```
char arr[] = {'a', 'b', 'c', 'd'};
char* end = arr + sizeof(arr);
for (char* it = arr; it != end; ++it) { // Stop at end
   std::cout << *it << ' ';}
// Output: a b c d
```

Again, note that `end` actually points out of bounds, so we are not allowed to dereference it, but we are allowed to read the pointer value and compare it with our `it` variable.

Ranges

A range is a replacement for the iterator-sentinel pairs that we have used when referring to a sequence of elements. The `<range>` header contains multiple concepts that define requirements for different kinds

of ranges, for example, `input_range`, `random_access_range`, and so forth. These are all refinements of the most basic concept called `range`, which is defined like this:

```
template<class T>
concept range = requires(T& t) {
  ranges::begin(t);
  ranges::end(t);
};
```

This means that any type that exposes `begin()` and `end()` functions is considered a range (given that these functions return iterators).

For C++ standard containers, the `begin()` and `end()` functions will return iterators of the same type, whereas for C++20 ranges, this is not true in general. A range with identical iterator and sentinel types fulfills the concept of `std::ranges::common_range`. The new C++20 views (covered in the next chapter) return iterator-sentinel pairs that can be of different types. However, they can be transformed to a view that has the same type for the iterator and sentinel using `std::views::common`.

The constrained algorithms found in the `std::ranges` namespace can operate on ranges instead of iterator pairs. And since all standard containers (`vector`, `map`, `list`, and so on) fulfill the range concept, we can pass ranges directly to the constrained algorithms as follows:

```
auto vec = std::vector{1, 1, 0, 1, 1, 0, 0, 1};
std::cout << std::ranges::count(vec, 0); // Prints 3
```

Ranges are an abstraction of something iterable (something that can be looped over), and to some extent, they hide the immediate use of C++ iterators. However, iterators are still a major part of the C++ standard library and are also used extensively in the Ranges library.

The next thing you need to understand is the different kinds of iterators that exist.

Iterator categories

Now that you have a better understanding of how a range is defined and how we can know when we have reached the end of a sequence, it's time to look more closely at the operations that iterators can support in order to navigate, read, and write values.

Iterator navigation in a sequence can be done with the following operations:

- Step forward: `std::next(it)` or `++it`
- Step backward: `std::prev(it)` or `--it`
- Jump to an arbitrary position: `std::advance(it, n)` or `it += n`

Reading and writing a value at the position that the iterator represents is done by *dereferencing* the iterator. Here is how it looks:

- Read: `auto value = *it`
- Write: `*it = value`

These are the most common operations for iterators that are exposed by containers. But in addition, iterators might operate on data sources where a write or read implies a step forward. Examples of such data

sources could be user input, a network connection, or a file. These data sources require the following operations:

- Read only *and* step forward: `auto value = *it; ++it;`
- Write only *and* step forward: `*it = value; ++it;`

These operations are only possible to express with two succeeding expressions. The post-condition of the first expression is that the second expression must be valid. This also means that we can only read or write a value to a position once. If we want to read or write a new value, we must first advance the iterator to the next position.

Not all iterators support all of the operations in the preceding list. For example, some iterators can only *read* values and *step forward*, whereas others can both *read*, *write*, and *jump* to arbitrary positions.

Now if we think about a few basic algorithms, it becomes obvious that the requirements on the iterators vary between different algorithms:

- If an algorithm counts the number of occurrences of a value, it requires the *read* and *step forward* operations
- If an algorithm fills a container with a value, it requires the *write* and *step forward* operations
- A binary search algorithm on a sorted collection requires the *read* and *jump* operations

Some algorithms can be implemented more efficiently depending on what operations the iterators support. Just like containers, all algorithms in the standard library have complexity guarantees (using big O notation). For an algorithm to fulfill a certain complexity guarantee, it puts *requirements* on the iterators it operates on. These requirements are categorized into six basic iterator categories that relate to each other as shown in the following diagram:

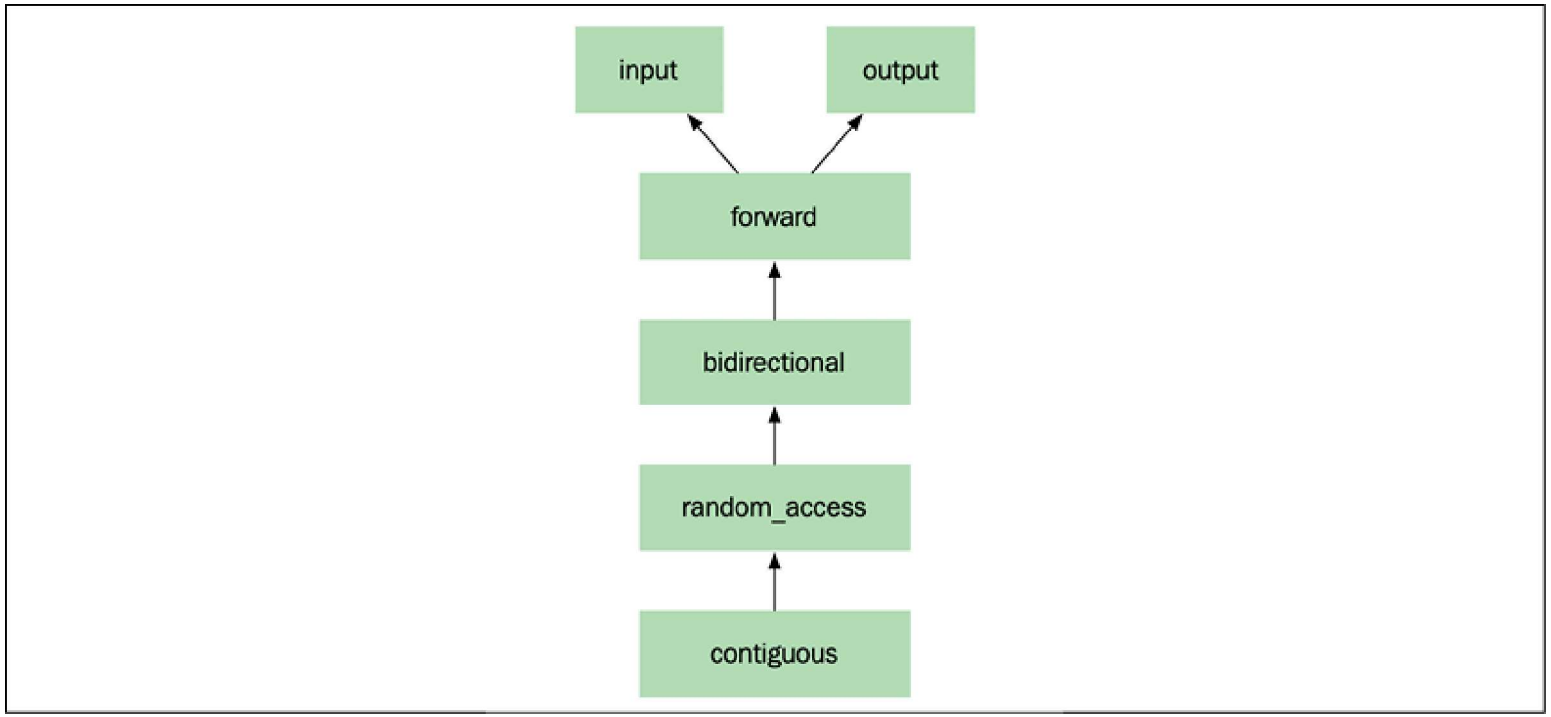


Figure 5.2: The six iterator categories and their relation to each other

The arrows indicate that an iterator category also has all the capabilities of the category it points at. For example, if an algorithm requires a forward iterator, we can just as well pass it a bidirectional iterator, since a bidirectional iterator has all the capabilities of a forward iterator.

The six requirements are formally specified by the following concepts:

- `std::input_iterator` : Supports *read only and step forward* (once). One-pass algorithms such as `std::count()` can use input iterators. `std::istream_iterator` is an example of an input iterator.
- `std::output_iterator` : Supports *write only and step forward* (once). Note that an output iterator can only write, not read. `std::ostream_iterator` is an example of an output iterator.

- `std::forward_iterator` : Supports *read* and *write* and *step forward*. The value at the current position can be read or written multiple times. Singly linked lists such as `std::forward_list` expose forward iterators.
- `std::bidirectional_iterator` : Supports *read*, *write*, *step forward*, and *step backward*. The doubly linked `std::list` exposes bidirectional iterators.
- `std::random_access_iterator` : Supports *read*, *write*, *step forward*, *step backward*, and *jump* to an arbitrary position in constant time. The elements inside `std::deque` can be accessed with random access iterators.
- `std::contiguous_iterator` : The same as random access iterators, but also guarantees that the underlying data is a contiguous block of memory, such as `std::string` , `std::vector` , `std::array` , `std::span` , and the (rarely used) `std::valarray` .

The iterator categories are very important for understanding the time-complexity requirements of the algorithms. Having a good understanding of the underlying data structures makes it fairly easy to know what iterators typically belong to which containers.

We are now ready to dig a little deeper into the common patterns used by most of the standard library algorithms.

Features of the standard algorithms

To get a better understanding of the standard algorithms, it's good to know a bit about the features and common patterns used by all algorithms in the `<algorithm>` header. As already stated, the algorithms under the `std` and `std::ranges` namespaces have a lot in common. We will start here with the general principles that are true for both the `std` algorithms and the constrained algorithms under `std::range` .

Then, in the next section, we will move on to discuss the features that are specific to the constrained algorithms found under `std::ranges`.

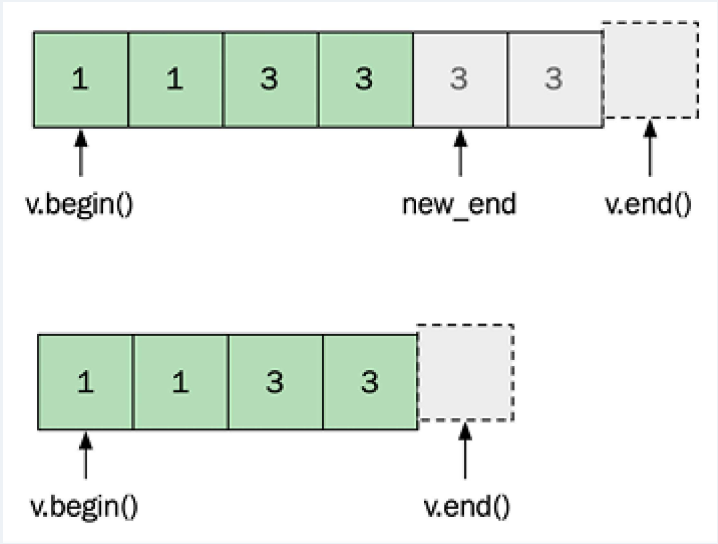
Algorithms do not change the size of the container

Functions from `<algorithm>` can only modify the elements in a specified range; elements are never added or deleted from the underlying container. Therefore, these functions never alter the size of the container that they operate on.

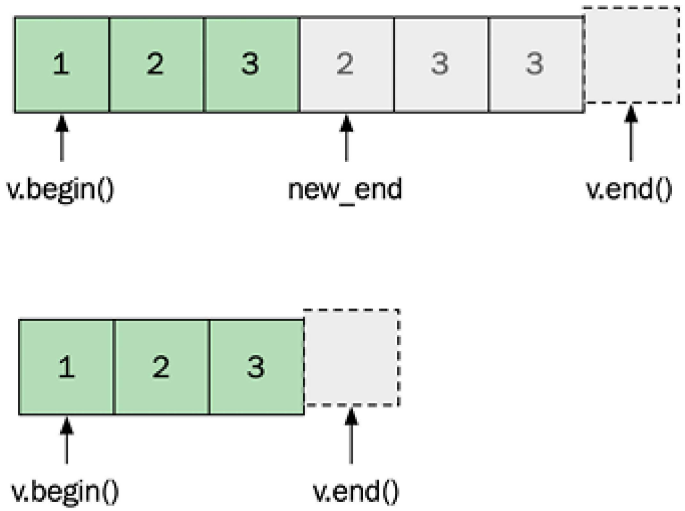
For example, `std::remove()` or `std::unique()` do not actually remove elements from a container (despite their names). Rather, it moves the elements that should be kept to the front of the container and then returns a sentinel that defines the new end of the valid range of elements:

Code example	Resulting vector

```
// Example with std::remove()
auto v = std::vector{1,1,2,2,3,3};
auto new_end = std::remove(
  v.begin(), v.end(), 2);
v.erase(new_end, v.end());
```



```
// Example with std::unique()
auto v = std::vector{1,1,2,2,3,3};
auto new_end = std::unique(
  v.begin(), v.end());
v.erase(new_end, v.end());
```



C++20 added new versions of the `std::erase()` and `std::erase_if()` functions to the `<vector>` header, which erases values immediately from the vector without the need to first call `remove()` followed by `erase()`.

The fact that standard library algorithms never change the size of a container means that we need to allocate data ourselves when calling algorithms that produce output.

Algorithms with output require allocated data

Algorithms that write data to an output iterator, such as `std::copy()` or `std::transform()`, require already allocated data reserved for the output. As the algorithms only use iterators as arguments, they cannot allocate data by themselves. To enlarge the container the algorithms operate on, they rely on the iterator being capable of enlarging the container it iterates.

If an iterator to an empty container is passed to the algorithms for output, the program will likely crash. The following example, where `squared` is empty, illustrates the problem:

```
const auto square_func = [](int x) { return x * x; };
const auto v = std::vector{1, 2, 3, 4};
auto squared = std::vector<int>{};
std::ranges::transform(v, squared.begin(), square_func);
```

Instead, you have to do either of the following:

- Preallocate the required size for the resulting container, or
- Use an insert iterator, which inserts elements into a container while iterating

The following snippet shows how to use preallocated space:

```
const auto square_func = [](int x) { return x * x; };
const auto v = std::vector{1, 2, 3, 4};
auto squared = std::vector<int>{};
squared.resize(v.size());
std::ranges::transform(v, squared.begin(), square_func);
```

The following snippet shows how to use `std::back_inserter()` and `std::inserter()` to insert values into a container that is not preallocated:

```
const auto square_func = [](int x) { return x * x; };
const auto v = std::vector{1, 2, 3, 4};
// Insert into back of vector using std::back_inserter
auto squared_vec = std::vector<int>{};
auto dst_vec = std::back_inserter(squared_vec);
std::ranges::transform(v, dst_vec, square_func);
// Insert into a std::set using std::inserter
auto squared_set = std::set<int>{};
auto dst_set = std::inserter(squared_set, squared_set.end());
std::ranges::transform(v, dst_set, square_func);
```

If you are operating on `std::vector` and know the expected size of the resulting container, you can use the `reserve()` member function before executing the algorithm in order to avoid unnecessary allocations. Otherwise, the vector may reallocate new chunks of memory several times during the algorithm.

Algorithms use operator==() and operator<() by default

For comparison, an algorithm relies on the fundamental `==` and `<` operators, as in the case of an integer. To be able to use your own classes with algorithms, `operator==()` and `operator<()` must either be provided by the class or as an argument to the algorithm.

By using the three-way comparison operator, `operator<=>()`, we can have the necessary operators generated by the compiler. The following example shows a simple `Flower` class, where `operator==()` is utilized by `std::find()`, and `operator<()` is utilized by `std::max_element()`:

```
struct Flower {
   auto operator<=>(const Flower& f) const = default;
   bool operator==(const Flower&) const = default;
   int height_{};
};
auto garden = std::vector<Flower>{{67}, {28}, {14}};
// std::max_element() uses operator<()
auto tallest = std::max_element(garden.begin(), garden.end());
// std::find() uses operator==()
auto perfect = *std::find(garden.begin(), garden.end(), Flower{28});
```

Apart from using the default comparison functions for the current type, it's also possible to use a custom comparator function, which we will explore next.

Custom comparator functions

Sometimes we need to compare objects without using the default comparison operators, for example, when sorting or finding a string by length. In those cases, a custom function can be provided as an additional argument. While the original algorithm uses a value (for example, `std::find()`), the version with a specific operator has the same name with `_if` attached at the end (`std::find_if()`, `std::count_if()`, and so on):

```
auto names = std::vector<std::string> {
  "Ralph", "Lisa", "Homer", "Maggie", "Apu", "Bart"
};
std::sort(names.begin(), names.end(),
          [](const std::string& a,const std::string& b) {
           return a.size() < b.size(); });
// names is now "Apu", "Lisa", "Bart", "Ralph", "Homer", "Maggie"
// Find names with length 3
auto x = std::find_if(names.begin(), names.end(),
  [](const auto& v) { return v.size() == 3; });
// x points to "Apu"
```

Constrained algorithms use projections

The constrained algorithms under `std::ranges` provide us with a handy feature called **projections**, which decreases the need for writing custom comparison functions. The preceding example in the previous section could be rewritten using the standard predicate `std::less` combined with a custom projection:

```
auto names = std::vector<std::string>{
  "Ralph", "Lisa", "Homer", "Maggie", "Apu", "Bart"
};
std::ranges::sort(names, std::less<>{}, &std::string::size);
// names is now "Apu", "Lisa", "Bart", "Ralph", "Homer", "Maggie"

// Find names with length 3
auto x = std::ranges::find(names, 3, &std::string::size);
// x points to "Apu"
```

It's also possible to pass a lambda as a projection parameter, which can be handy when you want to combine multiple properties in a projection:

```
struct Player {
  std::string name_{};
  int level_{};
  float health_{};
  // ...
};
auto players = std::vector<Player>{
  {"Aki", 1, 9.f},
  {"Nao", 2, 7.f},
  {"Rei", 2, 3.f}};
auto level_and_health = [](const Player& p) {
  return std::tie(p.level_, p.health_);
};
// Order players by level, then health
std::ranges::sort(players, std::greater<>{}, level_and_health);
```

The possibility to pass a projection object to the standard algorithms is a very welcome feature and really simplifies the use of custom comparisons.

Algorithms require move operators not to throw

All algorithms use `std::swap()` and `std::move()` when moving elements around, but only if the move constructor and move assignment are marked `noexcept`. Therefore, it is important to have these imple-

mented for heavy objects when using algorithms. If they are not available and exception free, the elements will be copied instead.

Note that if you implement a move constructor and a move assignment operator in your class, `std::swap()` will utilize them and, therefore, a specified `std::swap()` overload is not needed.

Algorithms have complexity guarantees

The complexity of each algorithm in the standard library is specified using big O notation. Algorithms are created with performance in mind. Therefore, they do not allocate memory nor do they have a time complexity higher than *O(n log n)*. Algorithms that do not fit these criteria are not included even if they are fairly common operations.

Note the exceptions of `stable_sort()`, `inplace_merge()`, and `stable_partition()`. Many implementations tend to temporarily allocate memory during these operations.

For example, let's consider an algorithm that tests whether a non-sorted range contains duplicates. One option is to implement it by iterating through the range and search the rest of the range for a duplicate. This will result in an algorithm with $O(n^2)$ complexity:

```
template <typename Iterator>
auto contains_duplicates(Iterator first, Iterator last) {
  for (auto it = first; it != last; ++it)
    if (std::find(std::next(it), last, *it) != last)
      return true;
```

```
  return false;
}
```

Another option is to make a copy of the full range, sort it, and look for adjacent equal elements. This will result in a time complexity of *O(n log n)*, the complexity of `std::sort()`. However, since it needs to make a copy of the full range, it still doesn't qualify as a building block algorithm. Allocating means that we cannot trust it not to throw:

```
template <typename Iterator>
auto contains_duplicates(Iterator first, Iterator last) {
  // As (*first) returns a reference, we have to get
  // the base type using std::decay_t
  using ValueType = std::decay_t<decltype(*first)>;
  auto c = std::vector<ValueType>(first, last);
  std::sort(c.begin(), c.end());
  return std::adjacent_find(c.begin(),c.end()) != c.end();
}
```

The complexity guarantees have been a part of the C++ standard library from the very beginning and are one of the major reasons behind its great success. Algorithms in the C++ standard library are designed and implemented with performance in mind.

Algorithms perform just as well as C library function equivalents

The standard C library comes with a number of low-level algorithms, including `memcpy()`, `memmove()`, `memcmp()`, and `memset()`. In my experience, sometimes people use these functions instead of their

equivalents in the standard Algorithm library. The reason is that people tend to believe that the C library functions are faster and, therefore, accept the trade-off in type safety.

This is not true for modern standard library implementation; the equivalent algorithms, `std::copy()`, `std::equal()`, and `std::fill()`, resort to these low-level C functions where plausible; hence, they provide both performance and type safety.

Sure, there might be exceptions where the C++ compiler is not able to detect that it is safe to resort to the low-level C-functions. For example, if a type is not trivially copyable, `std::copy()` cannot use `memcpy()`. But that's for good reason; hopefully, the author of a class that is not trivially copyable had good reasons for designing the class in such a way, and we (or the compiler) should not ignore that by not calling the appropriate constructors.

Sometimes, functions from the C++ Algorithm library even outperform their C library equivalents. The most prominent example is `std::sort()` versus `qsort()` from the C library. A big difference between `std::sort()` and `qsort()` is that `qsort()` is a *function* and `std::sort()` is a *function template*. When `qsort()` calls the comparison function, which is provided as a function pointer, it is generally a lot slower than calling an ordinary comparison function that may be inlined by the compiler when using `std::sort()`.

We will spend the remainder of this chapter going through some best practices when using the standard algorithms and implementing custom algorithms.

Writing and using generic algorithms

The Algorithm library contains generic algorithms. To keep things as concrete as possible here, I will show an example of how a generic algorithm can be implemented. This will provide you with some in-

sights into how to use the standard algorithms and at the same time demonstrate that implementing a generic algorithm is not that hard. I will intentionally avoid explaining all the details about the example code here, because we will spend a lot of time on generic programming later on in this book.

In the examples that follow, we will transform a simple non-generic algorithm into a full-fledged generic algorithm.

Non-generic algorithms

A generic algorithm is an algorithm that can be used with various ranges of elements, not only one specific type, such as `std::vector`. The following algorithm is an example of a non-generic algorithm that only works with `std::vector<int>`:

```
auto contains(const std::vector<int>& arr, int v) {
  for (int i = 0; i < arr.size(); ++i) {
    if (arr[i] == v) { return true; }
  }
  return false;
}
```

To find the element we are looking for, we rely on the interface of `std::vector` that provides us with the `size()` function and the subscript operator (`operator[]()`). However, not all containers provide us with these functions, and I don't recommend you write raw loops like this anyway. Instead, we need to create a function template that operates on iterators.

Generic algorithms

By replacing the `std::vector` with two iterators, and the `int` with a template parameter, we can transform our algorithm to a generic version. The following version of `contains()` can be used with any container:

```
template <typename Iterator, typename T>
auto contains(Iterator begin, Iterator end, const T& v) {
  for (auto it = begin; it != end; ++it) {
    if (*it == v) { return true; }
  }
  return false;
}
```

To use it with, for example, `std::vector`, you would have to pass the `begin()` and `end()` iterators:

```
auto v = std::vector{3, 4, 2, 4};
if (contains(v.begin(), v.end(), 3)) {
 // Found the value...
}
```

We could improve this algorithm by offering a version that accepts a range instead of two separate iterator parameters:

```
auto contains(const auto& r, const auto& x) {
  auto it = std::begin(r);
  auto sentinel = std::end(r);
```

```
  return contains(it, sentinel, x);
}
```

This algorithm does not force the client to provide the `begin()` and `end()` iterators because we have moved that inside the function. We are using the **abbreviated function template** syntax from C++20 to avoid spelling out explicitly that this is a function template. As a last step, we could add constraints to our parameter types:

```
auto contains(const std::ranges::range auto& r, const auto& x) {
  auto it = std::begin(r);
  auto sentinel = std::end(r);
  return contains(it, sentinel, x);
}
```

As you can see, there is really not that much code needed to create a robust generic algorithm. The only requirement on the data structure we pass to the algorithm is that it can expose `begin()` and `end()` iterators. You will learn more about constraints and concepts in *Chapter 8*, *Compile-Time Programming*.

Data structures that can be used by generic algorithms

This leads us to the insight that new custom data structures we create can be used by the standard generic algorithms as long as they expose the `begin()` and `end()` iterators or a range. As a simple example, we could implement a two-dimensional `Grid` structure where rows are exposed as a pair of iterators, like this:

```
struct Grid {
  Grid(std::size_t w, std::size_t h) : w_{w}, h_{h} {    data_.resize(w * h);
  }
  auto get_row(std::size_t y); // Returns iterators or a range

  std::vector<int> data_{};
  std::size_t w_{};
  std::size_t h_{};
};
```

The following diagram illustrates the layout of the `Grid` structure with the iterator pairs:

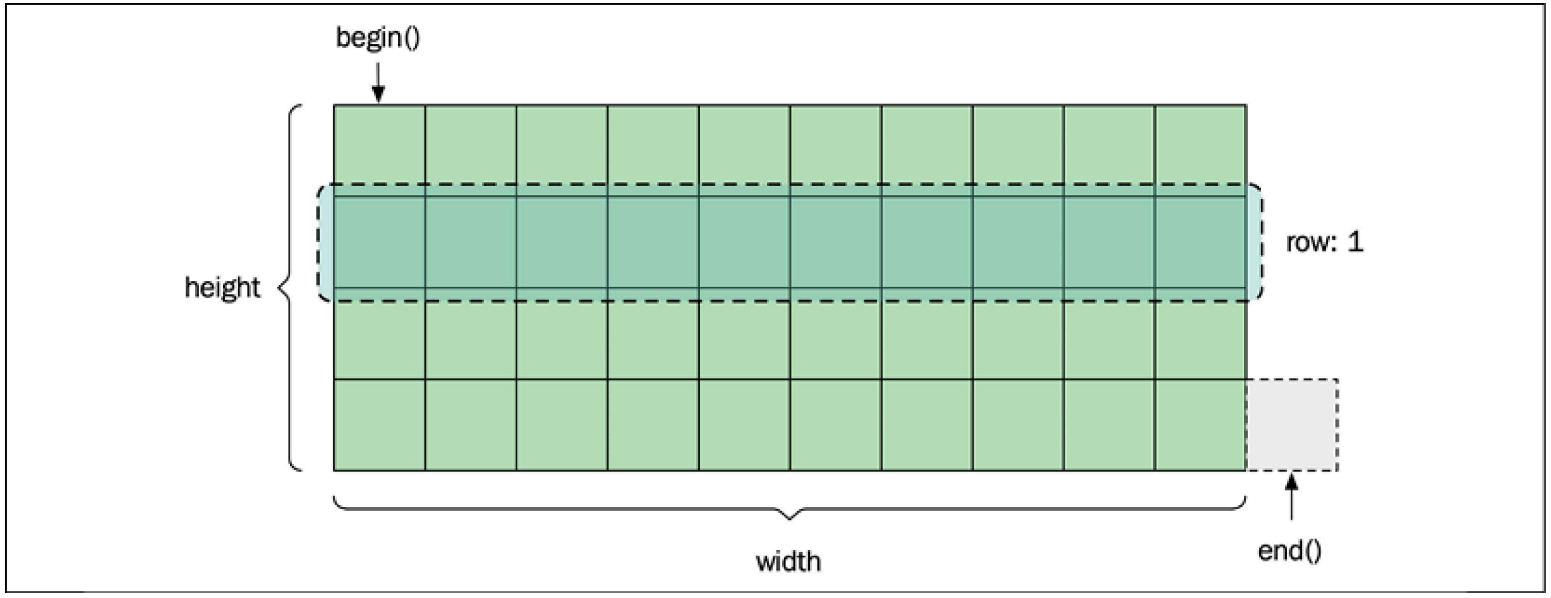


Figure 5.3: Two-dimensional grid built upon a one-dimensional vector

A possible implementation of `get_row()` would return a `std::pair` holding iterators that represent the beginning and the end of the row:

```
auto Grid::get_row(std::size_t y) {
  auto left = data_.begin() + w_ * y;
  auto right = left + w_;
  return std::make_pair(left, right);
}
```

The iterator pair representing a row can then be utilized by standard library algorithms. In the following example, we are using `std::generate()` and `std::count()`:

```
auto grid = Grid{10, 10};
auto y = 3;
auto row = grid.get_row(y);
std::generate(row.first, row.second, std::rand);
auto num_fives = std::count(row.first, row.second, 5);
```

While this works, it is a bit clumsy to work with `std::pair`, and it also requires the client to know how to handle the iterator pair. There is nothing that explicitly says that the `first` and `second` members actually denote a half-open range. Wouldn't it be nice if it could expose a strongly typed range instead? Fortunately, the Ranges library that we will explore in the next chapter provides us with a view type called `std::ranges::subrange`. Now, the `get_row()` function could be implemented like this:

```
auto Grid::get_row(std::size_t y) {
  auto first = data_.begin() + w_ * y;
```

```
  auto sentinel = first + w_;
  return std::ranges::subrange{first, sentinel};
}
```

We could be even lazier and use the handy view that is tailor-made for this scenario, called `std::views::counted()`

```
auto Grid::get_row(std::size_t y) {
  auto first = data_.begin() + w_ * y;
  return std::views::counted(first, w_);
}
```

A row returned from the `Grid` class could now be used with any of the constrained algorithms that accept ranges instead of iterator pairs:

```
auto row = grid.get_row(y);
std::ranges::generate(row, std::rand);
auto num_fives = std::ranges::count(row, 5);
```

That completes our example of writing and using a generic algorithm that supports both iterator pairs and a range. Hopefully, this has given you some insights about how to write data structures and algorithms in a generic way to avoid the combinatorial explosion that would occur if we had to write specialized algorithms for all types of data structures.

Best practices

Let's consider practices that will help you out when working with the algorithms we've been discussing. I will start by highlighting the importance of actually exploiting the standard algorithms.

Using the constrained algorithms

The constrained algorithms under `std::ranges` introduced with C++20 offer some benefits over the iterator-based algorithms under `std`. The constrained algorithms do the following:

- Support projections, which simplifies custom comparisons of elements.
- Support ranges instead of iterator pairs. There is no need to pass `begin()` and `end()` iterators as separate arguments.
- Are easy to use correctly and provide descriptive error messages during compilation as a result of being constrained by C++ concepts.

It's my recommendation to start using the constrained algorithms over the iterator-based algorithms.

You may have noticed that this book uses iterator-based algorithms in a lot of places. The reason for this is that not all standard library implementations support the constrained algorithms at the time of writing this book.

Sorting only for the data you need to retrieve

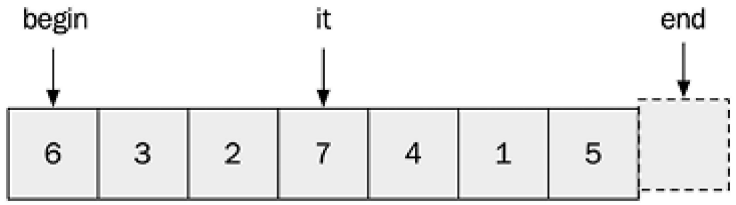
The Algorithm library contains three basic sorting algorithms: `sort()`, `partial_sort()`, and `nth_element()`. In addition, it also contains a few variants of those, including `stable_sort()`, but we will focus on these three as, in my experience, it is easy to forget that, in many cases, a complete sort can be avoided by using `nth_element()` or `partial_sort()` instead.

While `sort()` sorts the entire range, `partial_sort()` and `nth_element()` could be thought of as algorithms for inspecting parts of that sorted range. In many cases, you are only interested in a certain part of the sorted range, for example:

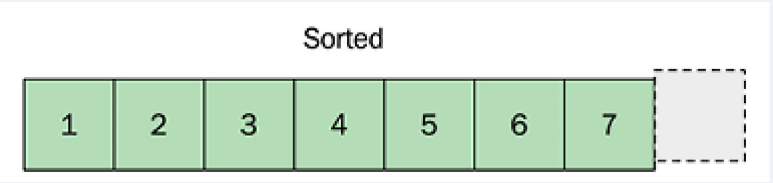
- If you want to calculate the median of a range, you require the value in the middle of the sorted range.
- If you want to create a body scanner that can be used by the mean 80% by height of a population, you require two values in the sorted range: the value located 10% from the tallest person, and the value located 10% from the shortest person.

The following diagram illustrates how `std::nth_element` and `std::partial_sort` process a range, compared to a fully sorted range:

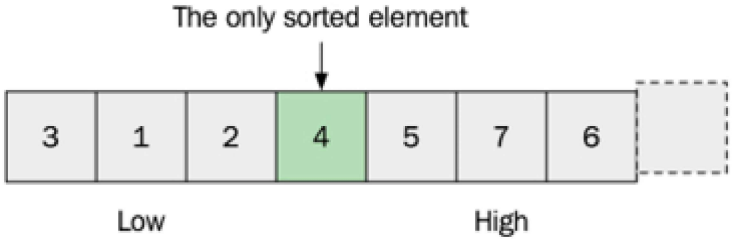
```
auto v = std::vector{6, 3, 2, 7,
                     4, 1, 5};
auto it = v.begin() + v.size()/2;
```



```
std::ranges::sort(v);
```




```
std::nth_element(v.begin(), it,
v.end());
```



```
std::partial_sort(v.begin(), it,
    v.end());
```

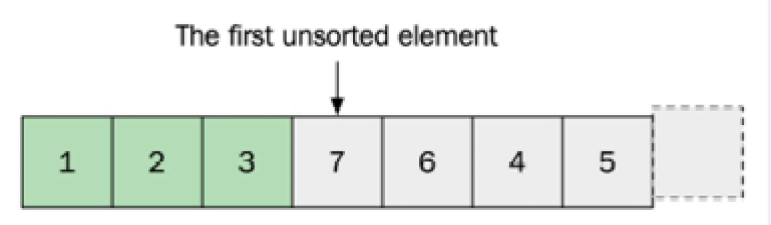


Figure 5.1: Sorted and non-sorted elements of a range using different algorithms

The following table shows their algorithmic complexity; note that *m* denotes the sub-range which is being fully sorted:

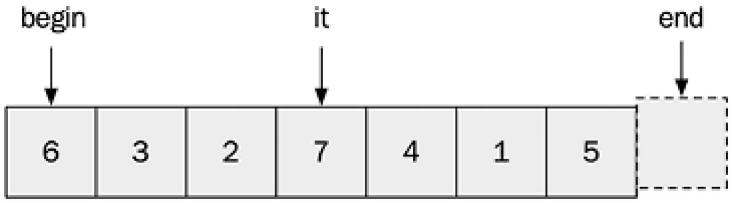
Algorithm	Complexity
`std::sort()`	*O(n log n)*
`std::partial_sort()`	*O(n log m)*
`std::nth_element()`	*O(n)*

Table 5.2: Algorithmic complexity

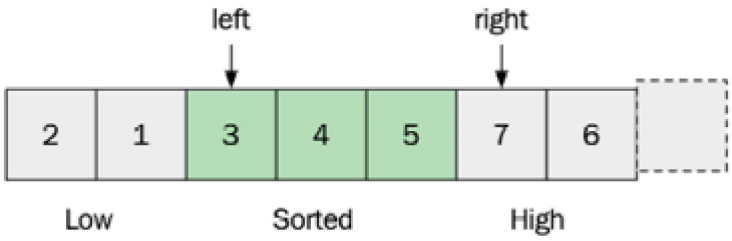
Use cases

Now that you have insights into `std:nth_element()` and `std::partial_sort()`, let's see how we can combine them to inspect parts of a range as if the entire range were sorted:

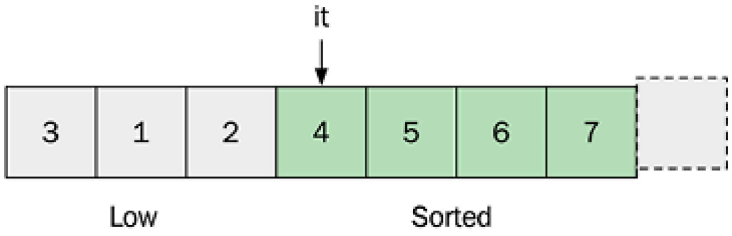
```
auto v = std::vector{6, 3, 2, 7,
                     4, 1, 5};
auto it = v.begin() + v.size()/2;
```




```
auto left = it - 1;
auto right = it + 2;
std::nth_element(v.begin(),
                 left, v.end());
std::partial_sort(left, right,
                  v.end());
```



```
std::nth_element(v.begin(), it,
                 v.end());
std::sort(it, v.end());
```




```
auto left = it - 1;
auto right = it + 2;
std::nth_element(v.begin(),
                 right, v.end());
std::partial_sort(v.begin(),
                  left, right);
std::sort(right, v.end());
```

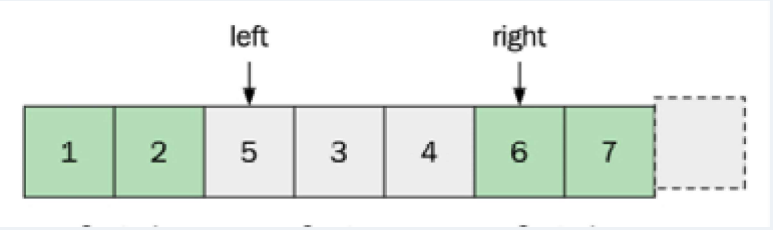


Figure 5.3: Combining algorithms and corresponding partially ordered results

As you can see, by using combinations of `std::sort()`, `std::nth_element()`, and `std::partial_sort()`, there are many ways to avoid sorting the entire range when not absolutely needed. This is an effective way to gain performance.

Performance evaluation

Let's see how `std::nth_element()` and `std::partial_sort()` measure up against `std::sort()`. We've measured this with a `std::vector` with 10,000,000 random `int` elements:

Operation	Code, where `r` is the range operated on	Time (Speedup)
Sort	`std::sort(r.begin(), r.end());`	760 ms (1.0x)
Find median	`auto it = r.begin() + r.size() / 2;` `std::nth_element(r.begin(), it, r.end());`	83 ms (9.2x)
Sort first tenth of range	`auto it = r.begin() + r.size() / 10;` `std::partial_sort(r.begin(), it, r.end());`	378 ms (2.0x)

Table 5.3: Benchmark results for partial sort algorithms

Use standard algorithms over raw for-loops

It's easy to forget that complex algorithms can be implemented by combining algorithms from the standard library. Maybe because of an old habit of trying to solve problems by hand and immediately starting to handcraft `for`-loops and working through the problem using an imperative approach. If this sounds familiar to you, my recommendation is to get to know the standard algorithms well enough so that you start considering them as the first choice.

I promote the use of standard library algorithms over raw `for`-loops, for a number of reasons:

- Standard algorithms deliver performance. Even though some of the algorithms in the standard library may seem trivial, they are often optimally designed in ways that are not obvious at first glance.
- Standard algorithms provide safety. Even simpler algorithms may have corner cases, which are easy to overlook.
- Standard algorithms are future-proof; a given algorithm can be replaced by a more suitable algorithm if you want to take advantage of SIMD extensions, parallelism, or even the GPU at a later stage (see *Chapter 14*, *Parallel Algorithms*).
- Standard algorithms are thoroughly documented.

In addition, by using algorithms instead of `for`-loops, the intention of each operation is clearly indicated by the name of the algorithm. The readers of your code do not need to inspect details inside raw `for`-loop to determine what your code does if you use standard algorithms as building blocks.

Once you get into the habit of thinking in terms of algorithms, you'll realize that many `for`-loops are most often a variation of a few simple algorithms such as `std::transform()`, `std::any_of()`, `std::copy_if()`, and `std::find()`.

Using algorithms will also make the code cleaner. You can often implement functions without nested code blocks and at the same time avoid mutable variables. This will be demonstrated in the following

example.

Example 1: Readability issues and mutable variables

Our first example is from a real-world code base, although variable names have been disguised. As it is only a cut-out, you don't have to understand the logic of the code. The example here is just to show you how the complexity is lowered when using algorithms compared with nested `for` -loops.

The original version looked like this:

```
// Original version using a for-loop
auto conflicting = false;
for (const auto& info : infos) {
 if (info.params() == output.params()) {
  if (varies(info.flags())) {
   conflicting = true;
   break;
  }
 }
 else {
  conflicting = true;
  break;
 }
}
```

In the `for` -loop version, it's hard to grasp when or why `conflicting` is set to `true` , whereas in the following versions of the algorithm, you can instinctively see that it happens if `info` fulfills a predicate.

Further, the standard algorithm version uses no mutable variables and can be written using a combination of a short lambda and `any_of()`. Here is how it looks:

```
// Version using standard algorithms
const auto in_conflict = [&](const auto& info) {
  return info.params() != output.params() || varies(info.flags());
};
const auto conflicting = std::ranges::any_of(infos, in_conflict);
```

Although it may overstate the point, imagine if we were to track a bug or parallelize it, the standard algorithm version using a lambda and `any_of()` would be far easier to understand and reason about.

Example 2: Unfortunate exceptions and performance problems

To further state the importance of using algorithms rather than `for`-loops, I'd like to show a few not-so-obvious problems that you may bump into when using handcrafted `for`-loops rather than standard algorithms.

Let's say we need a function that moves the first *n* elements from the front of a container to the back, like this:

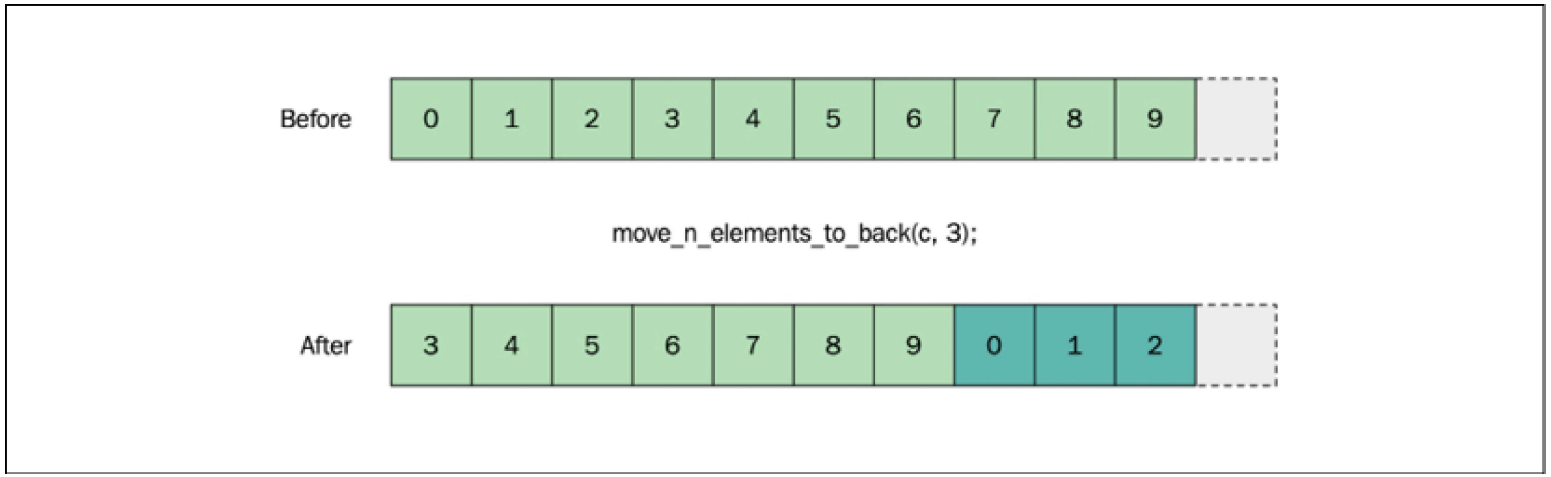


Figure 5.4: Moving the first three elements to the back of a range

Approach 1: Use a traditional for-loop

A naive approach would be to copy the first *n* elements to the back while iterating over them and then erasing the first *n* elements:

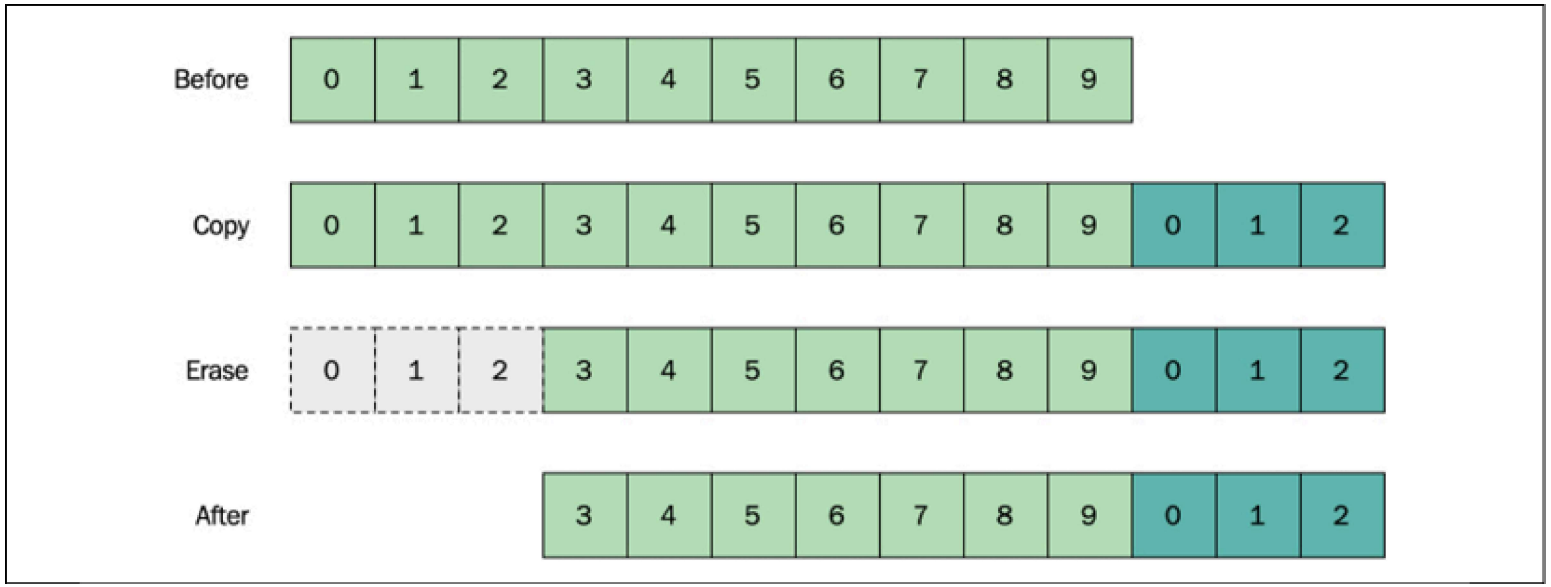


Figure 5.5: Allocating and deallocating in order to move elements to the back of a range

Here's the corresponding implementation:

```
template <typename Container>
auto move_n_elements_to_back(Container& c, std::size_t n) {
  // Copy the first n elements to the end of the container
  for (auto it = c.begin(); it != std::next(c.begin(), n); ++it) {
    c.emplace_back(std::move(*it));
  }
  // Erase the copied elements from front of container
  c.erase(c.begin(), std::next(c.begin(), n));
}
```

At first glance, it might look plausible, but inspecting it reveals a severe problem—if the container reallocates during the iteration due to `emplace_back()`, the iterator `it` will no longer be valid. As the algorithm tries to access an invalid iterator, the algorithm will go into undefined behavior and, in the best case, crash.

Approach 2: Safe for-loop (safe at the expense of performance)

As undefined behaviors are an obvious problem, we'll have to rewrite the algorithm. We are still using a handcrafted `for`-loop, but we'll utilize the index instead of the iterator:

```
template <typename Container>
auto move_n_elements_to_back(Container& c, std::size_t n) {
  for (size_t i = 0; i < n; ++i) {
    auto value = *std::next(c.begin(), i);
    c.emplace_back(std::move(value));
  }
  c.erase(c.begin(), std::next(c.begin(), n));
}
```

The solution works; it doesn't crash anymore. But now, it has a subtle performance problem. The algorithm is significantly slower on `std::list` than on `std::vector`. The reason is that `std::next(it, n)` used with `std::list::iterator` is *O(n)*, and *O(1)* on a `std::vector::iterator`. As `std::next(it, n)` is invoked in every step of the `for`-loop, this algorithm will have a time complexity of $O(n^2)$ on containers such as `std::list`. Apart from this performance limitation, the preceding code also has the following limitations:

- It doesn't work with containers of a static size, such as `std::array`, due to `emplace_back()`

- It might throw an exception, since `emplace_back()` may allocate memory and fail (this is probably rare though)

Approach 3: Find and use a suitable standard library algorithm

When we have reached this stage, we should browse through the standard library and see whether it contains a suitable algorithm to be used as a building block. Conveniently, the `<algorithm>` header provides an algorithm called `std::rotate()`, which does exactly what we are looking for while avoiding all the disadvantages mentioned before. Here is our final version using the `std::rotate()` algorithm:

```
template <typename Container>
auto move_n_elements_to_back(Container& c, std::size_t n) {
  auto new_begin = std::next(c.begin(), n);
  std::rotate(c.begin(), new_begin, c.end());
}
```

Let's have a look at the advantages of using `std::rotate()`:

- The algorithm does not throw exceptions, as it does not allocate memory (the contained object might throw exceptions though)
- It works with containers whose size cannot be changed, such as `std::array`
- Performance is *O(n)* regardless of the container it operates on
- The implementation may very well be optimized with specific hardware in mind

Maybe you find this comparison between `for`-loops and standard algorithms unfair because there are other solutions to this problem that are both elegant and efficient. Still, in the real world, it's not uncom-

mon to see implementations like the ones you just saw, when there are algorithms in the standard library just waiting to solve your problems.

Example 3: Exploiting the standard library optimizations

This last example highlights the fact that even algorithms that may seem very simple might contain optimizations you wouldn't consider. Let's have a look at `std::find()`, for example. At a glance, it seems that the obvious implementation couldn't be optimized further. Here is a possible implementation of the `std::find()` algorithm:

```
template <typename It, typename Value>
auto find_slow(It first, It last, const Value& value) {
  for (auto it = first; it != last; ++it)
    if (*it == value)
      return it;
  return last;
}
```

However, looking through the GNU libstdc++ implementation, when being used with `random_access_iterator` (in other words, `std::vector`, `std::string`, `std::deque`, and `std::array`), the libc++ implementers have unrolled the main loop into chunks of four loops at a time, resulting in the comparison (`it != last`) being executed one-fourth as many times.

Here is the optimized version of `std::find()` taken from the libstdc++ library:

```
template <typename It, typename Value>
auto find_fast(It first, It last, const Value& value) {
```

```
    // Main loop unrolled into chunks of four
    auto num_trips = (last - first) / 4;
    for (auto trip_count = num_trips; trip_count > 0; --trip_count) {
      if (*first == value) {return first;} ++first;
      if (*first == value) {return first;} ++first;
      if (*first == value) {return first;} ++first;
      if (*first == value) {return first;} ++first;
    }
    // Handle the remaining elements
    switch (last - first) {
      case 3: if (*first == value) {return first;} ++first;
      case 2: if (*first == value) {return first;} ++first;
      case 1: if (*first == value) {return first;} ++first;
      case 0:
      default: return last;
    }
  }
```

Note that it is actually `std::find_if()`, not `std::find()`, that utilizes this loop-unrolling optimization. But `std::find()` is implemented using `std::find_if()`.

In addition to `std::find()`, a multitude of algorithms in libstdc++ are implemented using `std::find_if()`, for example, `any_of()`, `all_of()`, `none_of()`, `find_if_not()`, `search()`, `is_partitioned()`, `remove_if()`, and `is_permutation()`, which means that all of these are slightly faster than a handcrafted `for`-loop.

And by slightly, I really mean slightly; the speedup is roughly 1.07x, as shown in the following table:

Find an integer in a `std::vector` of 10,000,000 elements

Algorithm	Time	Speedup
`find_slow()`	3.06 ms	1.00x
`find_fast()`	3.26 ms	1.07x

Table 5.5: find_fast() uses optimizations found in libstdc++. The benchmark shows that find_fast() is slightly faster than find_slow().

However, even though the benefit is almost negligible, using standard algorithms, you get it for free.

"Compare with zero" optimization

In addition to the loop unrolling, a very subtle optimization is that `trip_count` is iterated backward in order to compare with zero instead of a value. On some CPUs, comparing with zero is slightly faster than any other value, as it uses another assembly instruction (on the x86 platform, it uses `test` instead of `cmp`).

The following table shows the difference in assembly output using gcc 9.2:

Action	C++	Assembler x86

Compare with zero	`auto cmp_zero(size_t val) {`
	`  return val > 0;`
	`}`
		`test edi, edi`
		`setne al`
		`ret`

Compare with the other value	`auto cmp_val(size_t val) {`
	`  return val > 42;`
	`}`
		`cmp edi, 42`
		`setba al`
		`ret`

Table 5.6: The difference in assembly output

Even though this kind of optimization is encouraged in the standard library implementation, do not re-arrange your handmade loops in order to benefit from this optimization unless it's a (very) hot spot. Doing so will heavily reduce the readability of your code; let the algorithms handle these kinds of optimizations instead.

This was the end of my recommendations about using algorithms rather than `for` -loops. If you are not already using the standard algorithms, I hope that I have given you some arguments to convince you to give it a try. Now we will move on to my very last suggestion on using algorithms effectively.

Avoiding container copies

We will finish this chapter by highlighting a common problem when trying to combine multiple algorithms from the Algorithm library: it's hard to avoid unnecessary copies of the underlying containers.

An example will clarify what I mean here. Let's say we have some sort of `Student` class to represent a student in a particular year and with a particular exam score, like this:

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

If we want to find the student in the second year with the highest score in a big collection of students, we would probably use `max_element()` on `score_`, but as we only want to take the students from the second year into account, it gets tricky. Essentially, we want to compose a new algorithm out of a combination of `copy_if()` and `max_element()`, but composing algorithms is not possible with the Algorithm library. Instead, we would have to make a copy of all the students in the second year to a new container and then iterate the new container to find the maximum score:

```
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;
}
```

This is one of the places where it is tempting to start writing a custom algorithm from scratch without taking advantage of the standard algorithms. But as you will see in the next chapter, there is no need to abandon the standard library for tasks like this. The ability to compose algorithms is one of the key motivations for using the Ranges library, which we will cover next.

Summary

In this chapter, you learned how to use the basic concepts in the Algorithm library, the advantages of using them as building blocks instead of handwritten `for`-loops, and why using the standard Algorithm library is beneficial for optimizing your code at a later stage. We also discussed the guarantees and trade-offs of the standard algorithms, meaning that you can, from now on, use them with confidence.

By using the advantages of the algorithms instead of manual `for`-loops, your code base is well prepared for the parallelization techniques that will be discussed in the coming chapters of this book. One key feature that the standard algorithms are missing is the possibility to compose algorithms, something that was highlighted when we tried to avoid unnecessary container copies. In the next chapter, you will learn how to use views from the C++ Ranges library to overcome this limitation of standard algorithms.