

C H A P T E R

10

GENERIC ALGORITHMS

CONTENTS

Section 10.1 Overview	376
Section 10.2 A First Look at the Algorithms	378
Section 10.3 Customizing Operations	385
Section 10.4 Revisiting Iterators	401
Section 10.5 Structure of Generic Algorithms	410
Section 10.6 Container-Specific Algorithms	415
Chapter Summary	417
Defined Terms	417

The library containers define a surprisingly small set of operations. Rather than adding lots of functionality to each container, the library provides a set of algorithms, most of which are independent of any particular container type. These algorithms are *generic*: They operate on different types of containers and on elements of various types.

The generic algorithms, and a more detailed look at iterators, form the subject matter of this chapter.

The sequential containers define few operations: For the most part, we can add and remove elements, access the first or last element, determine whether a container is empty, and obtain iterators to the first or one past the last element.

We can imagine many other useful operations one might want to do: We might want to find a particular element, replace or remove a particular value, reorder the container elements, and so on.

Rather than define each of these operations as members of each container type, the standard library defines a set of **generic algorithms**: “algorithms” because they implement common classical algorithms such as sorting and searching, and “generic” because they operate on elements of differing type and across multiple container types—not only library types such as `vector` or `list`, but also the built-in array type—and, as we shall see, over other kinds of sequences as well.



10.1 Overview

Most of the algorithms are defined in the algorithm header. The library also defines a set of generic numeric algorithms that are defined in the `numeric` header.

In general, the algorithms do not work directly on a container. Instead, they operate by traversing a range of elements bounded by two iterators (§ 9.2.1, p. 331). Typically, as the algorithm traverses the range, it does something with each element. For example, suppose we have a `vector` of `ints` and we want to know if that `vector` holds a particular value. The easiest way to answer this question is to call the library `find` algorithm:

```
int val = 42; // value we'll look for
// result will denote the element we want if it's in vec, or vec.cend() if not
auto result = find(vec.cbegin(), vec.cend(), val);
// report the result
cout << "The value " << val
      << (result == vec.cend()
          ? " is not present" : " is present") << endl;
```

The first two arguments to `find` are iterators denoting a range of elements, and the third argument is a value. `find` compares each element in the given range to the given value. It returns an iterator to the first element that is equal to that value. If there is no match, `find` returns its second iterator to indicate failure. Thus, we can determine whether the element was found by comparing the return value with the second iterator argument. We do this test in the output statement, which uses the conditional operator (§ 4.7, p. 151) to report whether the value was found.

Because `find` operates in terms of iterators, we can use the same `find` function to look for values in any type of container. For example, we can use `find` to look for a value in a `list` of strings:

```
string val = "a value"; // value we'll look for
// this call to find looks through string elements in a list
auto result = find(lst.cbegin(), lst.cend(), val);
```

Similarly, because pointers act like iterators on built-in arrays, we can use `find` to look in an array:

```
int ia[] = {27, 210, 12, 47, 109, 83};
int val = 83;
int* result = find(begin(ia), end(ia), val);
```

Here we use the library `begin` and `end` functions (§ 3.5.3, p. 118) to pass a pointer to the first and one past the last elements in `ia`.

We can also look in a subrange of the sequence by passing iterators (or pointers) to the first and one past the last element of that subrange. For example, this call looks for a match in the elements `ia[1]`, `ia[2]`, and `ia[3]`:

```
// search the elements starting from ia[1] up to but not including ia[4]
auto result = find(ia + 1, ia + 4, val);
```

How the Algorithms Work

To see how the algorithms can be used on varying types of containers, let's look a bit more closely at `find`. Its job is to find a particular element in an unsorted sequence of elements. Conceptually, we can list the steps `find` must take:

1. It accesses the first element in the sequence.
2. It compares that element to the value we want.
3. If this element matches the one we want, `find` returns a value that identifies this element.
4. Otherwise, `find` advances to the next element and repeats steps 2 and 3.
5. `find` must stop when it has reached the end of the sequence.
6. If `find` gets to the end of the sequence, it needs to return a value indicating that the element was not found. This value and the one returned from step 3 must have compatible types.

None of these operations depends on the type of the container that holds the elements. So long as there is an iterator that can be used to access the elements, `find` doesn't depend in any way on the container type (or even whether the elements are stored in a container).

Iterators Make the Algorithms Container Independent, ...

All but the second step in the `find` function can be handled by iterator operations: The iterator dereference operator gives access to an element's value; if a matching element is found, `find` can return an iterator to that element; the iterator increment operator moves to the next element; the "off-the-end" iterator will indicate when `find` has reached the end of its given sequence; and `find` can return the off-the-end iterator (§ 9.2.1, p. 331) to indicate that the given value wasn't found.

...But Algorithms Do Depend on Element-Type Operations

Although iterators make the algorithms container independent, most of the algorithms use one (or more) operation(s) on the element type. For example, step 2, uses the element type's `==` operator to compare each element to the given value.

Other algorithms require that the element type have the `<` operator. However, as we'll see, most algorithms provide a way for us to supply our own operation to use in place of the default operator.

EXERCISES SECTION 10.1

Exercise 10.1: The algorithm header defines a function named `count` that, like `find`, takes a pair of iterators and a value. `count` returns a count of how often that value appears. Read a sequence of ints into a vector and print the count of how many elements have a given value.

Exercise 10.2: Repeat the previous program, but read values into a list of strings.

KEY CONCEPT: ALGORITHMS NEVER EXECUTE CONTAINER OPERATIONS

The generic algorithms do not themselves execute container operations. They operate solely in terms of iterators and iterator operations. The fact that the algorithms operate in terms of iterators and not container operations has a perhaps surprising but essential implication: Algorithms never change the size of the underlying container. Algorithms may change the values of the elements stored in the container, and they may move elements around within the container. They do not, however, ever add or remove elements directly.

As we'll see in § 10.4.1 (p. 401), there is a special class of iterator, the inserters, that do more than traverse the sequence to which they are bound. When we assign to these iterators, they execute insert operations on the underlying container. When an algorithm operates on one of these iterators, the *iterator* may have the effect of adding elements to the container. The *algorithm* itself, however, never does so.



10.2 A First Look at the Algorithms

The library provides more than 100 algorithms. Fortunately, like the containers, the algorithms have a consistent architecture. Understanding this architecture makes learning and using the algorithms easier than memorizing all 100+ of them. In this chapter, we'll illustrate how to use the algorithms, and describe the unifying principles that characterize them. Appendix A lists all the algorithms classified by how they operate.

With only a few exceptions, the algorithms operate over a range of elements. We'll refer to this range as the "input range." The algorithms that take an input range always use their first two parameters to denote that range. These parameters are iterators denoting the first and one past the last elements to process.

Although most algorithms are similar in that they operate over an input range, they differ in how they use the elements in that range. The most basic way to understand the algorithms is to know whether they read elements, write elements, or rearrange the order of the elements.

10.2.1 Read-Only Algorithms



A number of the algorithms read, but never write to, the elements in their input range. The `find` function is one such algorithm, as is the `count` function we used in the exercises for § 10.1 (p. 378).

Another read-only algorithm is `accumulate`, which is defined in the `numeric` header. The `accumulate` function takes three arguments. The first two specify a range of elements to sum. The third is an initial value for the sum. Assuming `vec` is a sequence of integers, the following

```
// sum the elements in vec starting the summation with the value 0
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
```

sets `sum` equal to the sum of the elements in `vec`, using 0 as the starting point for the summation.



The type of the third argument to `accumulate` determines which addition operator is used and is the type that `accumulate` returns.

Algorithms and Element Types

The fact that `accumulate` uses its third argument as the starting point for the summation has an important implication: It must be possible to add the element type to the type of the sum. That is, the elements in the sequence must match or be convertible to the type of the third argument. In this example, the elements in `vec` might be `ints`, or they might be `double`, or `long long`, or any other type that can be added to an `int`.

As another example, because `string` has a `+` operator, we can concatenate the elements of a vector of strings by calling `accumulate`:

```
string sum = accumulate(v.cbegin(), v.cend(), string(""));
```

This call concatenates each element in `v` onto a `string` that starts out as the empty string. Note that we explicitly create a `string` as the third parameter. Passing the empty string as a string literal would be a compile-time error:

```
// error: no + on const char*
string sum = accumulate(v.cbegin(), v.cend(), "");
```

Had we passed a string literal, the type of the object used to hold the sum would be `const char*`. That type determines which `+` operator is used. Because there is no `+` operator for type `const char*`, this call will not compile.



Ordinarily it is best to use `cbegin()` and `cend()` (§ 9.2.3, p. 334) with algorithms that read, but do not write, the elements. However, if you plan to use the iterator returned by the algorithm to change an element's value, then you need to pass `begin()` and `end()`.

Algorithms That Operate on Two Sequences



Another read-only algorithm is `equal`, which lets us determine whether two sequences hold the same values. It compares each element from the first sequence to the corresponding element in the second. It returns `true` if the corresponding elements are equal, `false` otherwise. The algorithm takes three iterators: The first two (as usual) denote the range of elements in the first sequence; the third denotes the first element in the second sequence:

```
// roster2 should have at least as many elements as roster1
equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());
```

Because `equal` operates in terms of iterators, we can call `equal` to compare elements in containers of different types. Moreover, the element types also need not be the same so long as we can use `==` to compare the element types. For example, `roster1` could be a `vector<string>` and `roster2` a `list<const char*>`.

However, `equal` makes one critically important assumption: It assumes that the second sequence is at least as big as the first. This algorithm potentially looks at every element in the first sequence. It assumes that there is a corresponding element for each of those elements in the second sequence.



WARNING

Algorithms that take a single iterator denoting a second sequence *assume* that the second sequence is at least as large as the first.

EXERCISES SECTION 10.2.1

Exercise 10.3: Use `accumulate` to sum the elements in a `vector<int>`.

Exercise 10.4: Assuming `v` is a `vector<double>`, what, if anything, is wrong with calling `accumulate(v.cbegin(), v.cend(), 0)`?

Exercise 10.5: In the call to `equal` on rosters, what would happen if both rosters held C-style strings, rather than library strings?



10.2.2 Algorithms That Write Container Elements

Some algorithms assign new values to the elements in a sequence. When we use an algorithm that assigns to elements, we must take care to ensure that the sequence into which the algorithm writes is at least as large as the number of elements we ask the algorithm to write. Remember, algorithms do not perform container operations, so they have no way themselves to change the size of a container.

Some algorithms write to elements in the input range itself. These algorithms are not inherently dangerous because they write only as many elements as are in the specified range.

As one example, the `fill` algorithm takes a pair of iterators that denote a range and a third argument that is a value. `fill` assigns the given value to each element in the input sequence:

KEY CONCEPT: ITERATOR ARGUMENTS

Some algorithms read elements from two sequences. The elements that constitute these sequences can be stored in different kinds of containers. For example, the first sequence might be stored in a `vector` and the second might be in a `list`, a `deque`, a built-in array, or some other sequence. Moreover, the element types in the two sequences are not required to match exactly. What is required is that we be able to compare elements from the two sequences. For example, in the `equal` algorithm, the element types need not be identical, but we do have to be able to use `==` to compare elements from the two sequences.

Algorithms that operate on two sequences differ as to how we pass the second sequence. Some algorithms, such as `equal`, take three iterators: The first two denote the range of the first sequence, and the third iterator denotes the first element in the second sequence. Others take four iterators: The first two denote the range of elements in the first sequence, and the second two denote the range for the second sequence.

Algorithms that use a single iterator to denote the second sequence *assume* that the second sequence is at least as large as the first. It is up to us to ensure that the algorithm will not attempt to access a nonexistent element in the second sequence. For example, the `equal` algorithm potentially compares every element from its first sequence to an element in the second. If the second sequence is a subset of the first, then our program has a serious error—`equal` will attempt to access elements beyond the end of the second sequence.

```
fill(vec.begin(), vec.end(), 0); // reset each element to 0
// set a subsequence of the container to 10
fill(vec.begin(), vec.begin() + vec.size()/2, 10);
```

Because `fill` writes to its given input sequence, so long as we pass a valid input sequence, the writes will be safe.

Algorithms Do Not Check Write Operations



Some algorithms take an iterator that denotes a separate destination. These algorithms assign new values to the elements of a sequence starting at the element denoted by the destination iterator. For example, the `fill_n` function takes a single iterator, a count, and a value. It assigns the given value to the specified number of elements starting at the element denoted to by the iterator. We might use `fill_n` to assign a new value to the elements in a `vector`:

```
vector<int> vec; // empty vector
// use vec giving it various values
fill_n(vec.begin(), vec.size(), 0); // reset all the elements of vec to 0
```

The `fill_n` function assumes that it is safe to write the specified number of elements. That is, for a call of the form

```
fill_n(dest, n, val)
```

`fill_n` assumes that `dest` refers to an element and that there are at least `n` elements in the sequence starting from `dest`.

It is a fairly common beginner mistake to call `fill_n` (or similar algorithms that write to elements) on a container that has no elements:

```
vector<int> vec; // empty vector
// disaster: attempts to write to ten (nonexistent) elements in vec
fill_n(vec.begin(), 10, 0);
```

This call to `fill_n` is a disaster. We specified that ten elements should be written, but there are no such elements—`vec` is empty. The result is undefined.



Algorithms that write to a destination iterator *assume* the destination is large enough to hold the number of elements being written.

Introducing `back_inserter`

One way to ensure that an algorithm has enough elements to hold the output is to use an **insert iterator**. An insert iterator is an iterator that *adds* elements to a container. Ordinarily, when we assign to a container element through an iterator, we assign to the element that iterator denotes. When we assign through an insert iterator, a new element equal to the right-hand value is added to the container.

We'll have more to say about insert iterators in § 10.4.1 (p. 401). However, in order to illustrate how to use algorithms that write to a container, we will use **`back_inserter`**, which is a function defined in the `iterator` header.

`back_inserter` takes a reference to a container and returns an insert iterator bound to that container. When we assign through that iterator, the assignment calls `push_back` to add an element with the given value to the container:

```
vector<int> vec; // empty vector
auto it = back_inserter(vec); // assigning through it adds elements to vec
*it = 42;           // vec now has one element with value 42
```

We frequently use `back_inserter` to create an iterator to use as the destination of an algorithm. For example:

```
vector<int> vec; // empty vector
// ok: back_inserter creates an insert iterator that adds elements to vec
fill_n(back_inserter(vec), 10, 0); // appends ten elements to vec
```

On each iteration, `fill_n` assigns to an element in the given sequence. Because we passed an iterator returned by `back_inserter`, each assignment will call `push_back` on `vec`. As a result, this call to `fill_n` adds ten elements to the end of `vec`, each of which has the value 0.

Copy Algorithms

The `copy` algorithm is another example of an algorithm that writes to the elements of an output sequence denoted by a destination iterator. This algorithm takes three iterators. The first two denote an input range; the third denotes the beginning of the destination sequence. This algorithm copies elements from its input range into elements in the destination. It is essential that the destination passed to `copy` be at least as large as the input range.

As one example, we can use `copy` to copy one built-in array to another:

```
int a1[] = {0,1,2,3,4,5,6,7,8,9};
int a2[sizeof(a1)/sizeof(*a1)]; // a2 has the same size as a1
// ret points just past the last element copied into a2
auto ret = copy(begin(a1), end(a1), a2); // copy a1 into a2
```

Here we define an array named `a2` and use `sizeof` to ensure that `a2` has as many elements as the array `a1` (§ 4.9, p. 157). We then call `copy` to copy `a1` into `a2`. After the call to `copy`, the elements in both arrays have the same values.

The value returned by `copy` is the (incremented) value of its destination iterator. That is, `ret` will point just past the last element copied into `a2`.

Several algorithms provide so-called “copying” versions. These algorithms compute new element values, but instead of putting them back into their input sequence, the algorithms create a new sequence to contain the results.

For example, the `replace` algorithm reads a sequence and replaces every instance of a given value with another value. This algorithm takes four parameters: two iterators denoting the input range, and two values. It replaces each element that is equal to the first value with the second:

```
// replace any element with the value 0 with 42
replace(ilst.begin(), ilst.end(), 0, 42);
```

This call replaces all instances of 0 by 42. If we want to leave the original sequence unchanged, we can call `replace_copy`. That algorithm takes a third iterator argument denoting a destination in which to write the adjusted sequence:

```
// use back_inserter to grow destination as needed
replace_copy(ilst.cbegin(), ilst.cend(),
             back_inserter(ivec), 0, 42);
```

After this call, `ilst` is unchanged, and `ivec` contains a copy of `ilst` with the exception that every element in `ilst` with the value 0 has the value 42 in `ivec`.

10.2.3 Algorithms That Reorder Container Elements



Some algorithms rearrange the order of elements within a container. An obvious example of such an algorithm is `sort`. A call to `sort` arranges the elements in the input range into sorted order using the element type’s `<` operator.

As an example, suppose we want to analyze the words used in a set of children’s stories. We’ll assume that we have a vector that holds the text of several stories. We’d like to reduce this vector so that each word appears only once, regardless of how many times that word appears in any of the given stories.

For purposes of illustration, we’ll use the following simple story as our input:

`the quick red fox jumps over the slow red turtle`

Given this input, our program should produce the following vector:

fox	jumps	over	quick	red	slow	the	turtle
-----	-------	------	-------	-----	------	-----	--------

EXERCISES SECTION 10.2.2

Exercise 10.6: Using `fill_n`, write a program to set a sequence of `int` values to 0.

Exercise 10.7: Determine if there are any errors in the following programs and, if so, correct the error(s):

```
(a) vector<int> vec; list<int> lst; int i;
    while (cin >> i)
        lst.push_back(i);
    copy(lst.cbegin(), lst.cend(), vec.begin());

(b) vector<int> vec;
    vec.reserve(10); // reserve is covered in § 9.4 (p. 356)
    fill_n(vec.begin(), 10, 0);
```

Exercise 10.8: We said that algorithms do not change the size of the containers over which they operate. Why doesn't the use of `back_inserter` invalidate this claim?

Eliminating Duplicates

To eliminate the duplicated words, we will first sort the `vector` so that duplicated words appear adjacent to each other. Once the `vector` is sorted, we can use another library algorithm, named `unique`, to reorder the `vector` so that the unique elements appear in the first part of the `vector`. Because algorithms cannot do container operations, we'll use the `erase` member of `vector` to actually remove the elements:

```
void elimDups(vector<string> &words)
{
    // sort words alphabetically so we can find the duplicates
    sort(words.begin(), words.end());
    // unique reorders the input range so that each word appears once in the
    // front portion of the range and returns an iterator one past the unique range
    auto end_unique = unique(words.begin(), words.end());
    // erase uses a vector operation to remove the nonunique elements
    words.erase(end_unique, words.end());
}
```

The `sort` algorithm takes two iterators denoting the range of elements to sort. In this call, we sort the entire `vector`. After the call to `sort`, `words` is ordered as

fox	jumps	over	quick	red	red	slow	the	the	turtle
-----	-------	------	-------	-----	-----	------	-----	-----	--------

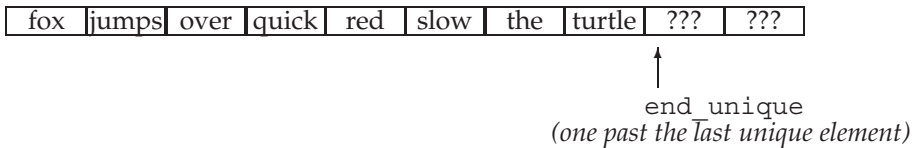
Note that the words `red` and `the` appear twice.



Using `unique`

Once `words` is sorted, we want to keep only one copy of each word. The `unique` algorithm rearranges the input range to “eliminate” adjacent duplicated entries,

and returns an iterator that denotes the end of the range of the unique values. After the call to `unique`, the vector holds



The size of words is unchanged; it still has ten elements. The order of those elements is changed—the adjacent duplicates have been “removed.” We put `remove` in quotes because `unique` doesn’t remove any elements. Instead, it overwrites adjacent duplicates so that the unique elements appear at the front of the sequence. The iterator returned by `unique` denotes one past the last unique element. The elements beyond that point still exist, but we don’t know what values they have.



The library algorithms operate on iterators, not containers. Therefore, an algorithm cannot (directly) add or remove elements.

Using Container Operations to Remove Elements



To actually remove the unused elements, we must use a container operation, which we do in the call to `erase` (§ 9.3.3, p. 349). We erase the range of elements from the one to which `end_unique` refers through the end of words. After this call, `words` contains the eight unique words from the input.

It is worth noting that this call to `erase` would be safe even if `words` has no duplicated words. In that case, `unique` would return `words.end()`. Both arguments to `erase` would have the same value: `words.end()`. The fact that the iterators are equal would mean that the range passed to `erase` would be empty. Erasing an empty range has no effect, so our program is correct even if the input has no duplicates.

EXERCISES SECTION 10.2.3

Exercise 10.9: Implement your own version of `elimDups`. Test your program by printing the vector after you read the input, after the call to `unique`, and after the call to `erase`.

Exercise 10.10: Why do you think the algorithms don’t change the size of containers?

10.3 Customizing Operations

Many of the algorithms compare elements in the input sequence. By default, such algorithms use either the element type’s `<` or `==` operator. The library also defines versions of these algorithms that let us supply our own operation to use in place of the default operator.

For example, the `sort` algorithm uses the element type's `<` operator. However, we might want to sort a sequence into a different order from that defined by `<`, or our sequence might have elements of a type (such as `Sales_data`) that does not have a `<` operator. In both cases, we need to override the default behavior of `sort`.



10.3.1 Passing a Function to an Algorithm

As one example, assume that we want to print the `vector` after we call `elimDups` (§ 10.2.3, p. 384). However, we'll also assume that we want to see the words ordered by their size, and then alphabetically within each size. To reorder the `vector` by length, we'll use a second, overloaded version of `sort`. This version of `sort` takes a third argument that is a **predicate**.

Predicates

A predicate is an expression that can be called and that returns a value that can be used as a condition. The predicates used by library algorithms are either **unary predicates** (meaning they have a single parameter) or **binary predicates** (meaning they have two parameters). The algorithms that take predicates call the given predicate on the elements in the input range. As a result, it must be possible to convert the element type to the parameter type of the predicate.

The version of `sort` that takes a binary predicate uses the given predicate in place of `<` to compare elements. The predicates that we supply to `sort` must meet the requirements that we'll describe in § 11.2.2 (p. 425). For now, what we need to know is that the operation must define a consistent order for all possible elements in the input sequence. Our `isShorter` function from § 6.2.2 (p. 211) is an example of a function that meets these requirements, so we can pass `isShorter` to `sort`. Doing so will reorder the elements by size:

```
// comparison function to be used to sort by word length
bool isShorter(const string &s1, const string &s2)
{
    return s1.size() < s2.size();
}
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
```

If `words` contains the same data as in § 10.2.3 (p. 384), this call would reorder `words` so that all the words of length 3 appear before words of length 4, which in turn are followed by words of length 5, and so on.

Sorting Algorithms

When we sort words by size, we also want to maintain alphabetic order among the elements that have the same length. To keep the words of the same length in alphabetical order we can use the `stable_sort` algorithm. A stable sort maintains the original order among equal elements.

Ordinarily, we don't care about the relative order of equal elements in a sorted sequence. After all, they're equal. However, in this case, we have defined "equal"

to mean “have the same length.” Elements that have the same length still differ from one another when we view their contents. By calling `stable_sort`, we can maintain alphabetical order among those elements that have the same length:

```
elimDups(words); // put words in alphabetical order and remove duplicates
// resort by length, maintaining alphabetical order among words of the same length
stable_sort(words.begin(), words.end(), isShorter);
for (const auto &s : words) // no need to copy the strings
    cout << s << " "; // print each element separated by a space
cout << endl;
```

Assuming `words` was in alphabetical order before this call, after the call, `words` will be sorted by element size, and the words of each length remain in alphabetical order. If we run this code on our original vector, the output will be

```
fox red the over slow jumps quick turtle
```

EXERCISES SECTION 10.3.1

Exercise 10.11: Write a program that uses `stable_sort` and `isShorter` to sort a vector passed to your version of `elimDups`. Print the vector to verify that your program is correct.

Exercise 10.12: Write a function named `compareIsbn` that compares the `isbn()` members of two `Sales_data` objects. Use that function to sort a vector that holds `Sales_data` objects.

Exercise 10.13: The library defines an algorithm named `partition` that takes a predicate and partitions the container so that values for which the predicate is `true` appear in the first part and those for which the predicate is `false` appear in the second part. The algorithm returns an iterator just past the last element for which the predicate returned `true`. Write a function that takes a string and returns a `bool` indicating whether the string has five characters or more. Use that function to partition `words`. Print the elements that have five or more characters.

10.3.2 Lambda Expressions

The predicates we pass to an algorithm must have exactly one or two parameters, depending on whether the algorithm takes a unary or binary predicate, respectively. However, sometimes we want to do processing that requires more arguments than the algorithm’s predicate allows. For example, the solution you wrote for the last exercise in the previous section had to hard-wire the size 5 into the predicate used to partition the sequence. It would be more useful to be able to partition a sequence without having to write a separate predicate for every possible size.

As a related example, we’ll revise our program from § 10.3.1 (p. 387) to report how many words are of a given size or greater. We’ll also change the output so that it prints only the words of the given length or greater.

A sketch of this function, which we’ll name `biggies`, is as follows:

```

void biggies(vector<string> &words,
            vector<string>::size_type sz)
{
    elimDups(words); // put words in alphabetical order and remove duplicates
    // resort by length, maintaining alphabetical order among words of the same length
    stable_sort(words.begin(), words.end(), isShorter);
    // get an iterator to the first element whose size() is >= sz
    // compute the number of elements with size >= sz
    // print words of the given size or longer, each one followed by a space
}

```

Our new problem is to find the first element in the `vector` that has the given size. Once we know that element, we can use its position to compute how many elements have that size or greater.

We can use the library `find_if` algorithm to find an element that has a particular size. Like `find` (§ 10.1, p. 376), the `find_if` algorithm takes a pair of iterators denoting a range. Unlike `find`, the third argument to `find_if` is a predicate. The `find_if` algorithm calls the given predicate on each element in the input range. It returns the first element for which the predicate returns a nonzero value, or its end iterator if no such element is found.

It would be easy to write a function that takes a `string` and a size and returns a `bool` indicating whether the size of a given `string` is greater than the given size. However, `find_if` takes a unary predicate—any function we pass to `find_if` must have exactly one parameter that can be called with an element from the input sequence. There is no way to pass a second argument representing the size. To solve this part of our problem we'll need to use some additional language facilities.

Introducing Lambdas

We can pass any kind of **callable object** to an algorithm. An object or expression is callable if we can apply the call operator (§ 1.5.2, p. 23) to it. That is, if `e` is a callable expression, we can write `e(args)` where `args` is a comma-separated list of zero or more arguments.

The only callables we've used so far are functions and function pointers (§ 6.7, p. 247). There are two other kinds of callables: classes that overload the function-call operator, which we'll cover in § 14.8 (p. 571), and **lambda expressions**.

C++
11

A lambda expression represents a callable unit of code. It can be thought of as an unnamed, inline function. Like any function, a lambda has a return type, a parameter list, and a function body. Unlike a function, lambdas may be defined inside a function. A lambda expression has the form

```
[capture list] (parameter list) -> return type { function body }
```

where *capture list* is an (often empty) list of local variables defined in the enclosing function; *return type*, *parameter list*, and *function body* are the same as in any ordinary function. However, unlike ordinary functions, a lambda must use a trailing `return` (§ 6.3.3, p. 229) to specify its return type.

We can omit either or both of the parameter list and return type but must always include the capture list and function body:

```
auto f = [] { return 42; };
```

Here, we've defined `f` as a callable object that takes no arguments and returns 42.

We call a lambda the same way we call a function by using the call operator:

```
cout << f() << endl; // prints 42
```

Omitting the parentheses and the parameter list in a lambda is equivalent to specifying an empty parameter list. Hence, when we call `f`, the argument list is empty. If we omit the return type, the lambda has an inferred return type that depends on the code in the function body. If the function body is just a `return` statement, the return type is inferred from the type of the expression that is returned. Otherwise, the return type is `void`.



Lambdas with function bodies that contain anything other than a single `return` statement that do not specify a return type return `void`.

Passing Arguments to a Lambda

As with an ordinary function call, the arguments in a call to a lambda are used to initialize the lambda's parameters. As usual, the argument and parameter types must match. Unlike ordinary functions, a lambda may not have default arguments (§ 6.5.1, p. 236). Therefore, a call to a lambda always has as many arguments as the lambda has parameters. Once the parameters are initialized, the function body executes.

As an example of a lambda that takes arguments, we can write a lambda that behaves like our `isShorter` function:

```
[] (const string &a, const string &b)
{ return a.size() < b.size(); }
```

The empty capture list indicates that this lambda will not use any local variables from the surrounding function. The lambda's parameters, like the parameters to `isShorter`, are references to `const string`. Again like `isShorter`, the lambda's function body compares its parameters' `size()`s and returns a `bool` that depends on the relative sizes of the given arguments.

We can rewrite our call to `stable_sort` to use this lambda as follows:

```
// sort words by size, but maintain alphabetical order for words of the same size
stable_sort(words.begin(), words.end(),
    [](const string &a, const string &b)
    { return a.size() < b.size(); });
```

When `stable_sort` needs to compare two elements, it will call the given lambda expression.

Using the Capture List

We're now ready to solve our original problem, which is to write a callable expression that we can pass to `find_if`. We want an expression that will compare the

length of each string in the input sequence with the value of the `sz` parameter in the `biggies` function.

Although a lambda may appear inside a function, it can use variables local to that function *only* if it specifies which variables it intends to use. A lambda specifies the variables it will use by including those local variables in its capture list. The capture list directs the lambda to include information needed to access those variables within the lambda itself.

In this case, our lambda will capture `sz` and will have a single string parameter. The body of our lambda will compare the given string's size with the captured value of `sz`:

```
[sz](const string &a)
{ return a.size() >= sz; };
```

Inside the `[]` that begins a lambda we can provide a comma-separated list of names defined in the surrounding function.

Because this lambda captures `sz`, the body of the lambda may use `sz`. The lambda does not capture words, and so has no access to that variable. Had we given our lambda an empty capture list, our code would not compile:

```
// error: sz not captured
[](const string &a)
{ return a.size() >= sz; };
```



A lambda may use a variable local to its surrounding function *only* if the lambda captures that variable in its capture list.

Calling `find_if`

Using this lambda, we can find the first element whose size is at least as big as `sz`:

```
// get an iterator to the first element whose size() is >= sz
auto wc = find_if(words.begin(), words.end(),
    [sz](const string &a)
    { return a.size() >= sz; });
```

The call to `find_if` returns an iterator to the first element that is at least as long as the given `sz`, or a copy of `words.end()` if no such element exists.

We can use the iterator returned from `find_if` to compute how many elements appear between that iterator and the end of `words` (§ 3.4.2, p. 111):

```
// compute the number of elements with size >= sz
auto count = words.end() - wc;
cout << count << " " << make_plural(count, "word", "s")
    << " of length " << sz << " or longer" << endl;
```

Our output statement calls `make_plural` (§ 6.3.2, p. 224) to print word or words, depending on whether that size is equal to 1.

The `for_each` Algorithm

The last part of our problem is to print the elements in `words` that have length `sz` or greater. To do so, we'll use the `for_each` algorithm. This algorithm takes a callable object and calls that object on each element in the input range:

```
// print words of the given size or longer, each one followed by a space
for_each(wc, words.end(),
        [](const string &s){cout << s << " ";});
cout << endl;
```

The capture list in this lambda is empty, yet the body uses two names: its own parameter, named `s`, and `cout`.

The capture list is empty, because we use the capture list only for (nonstatic) variables defined in the surrounding function. A lambda can use names that are defined outside the function in which the lambda appears. In this case, `cout` is not a name defined locally in `biggies`; that name is defined in the `iostream` header. So long as the `iostream` header is included in the scope in which `biggies` appears, our lambda can use `cout`.



The capture list is used for local nonstatic variables only; lambdas can use local statics and variables declared outside the function directly.

Putting It All Together

Now that we've looked at the program in detail, here is the program as a whole:

```
void biggies(vector<string> &words,
            vector<string>::size_type sz)
{
    elimDups(words); // put words in alphabetical order and remove duplicates
    // sort words by size, but maintain alphabetical order for words of the same size
    stable_sort(words.begin(), words.end(),
                [](const string &a, const string &b)
                { return a.size() < b.size(); });
    // get an iterator to the first element whose size() is >= sz
    auto wc = find_if(words.begin(), words.end(),
                     [sz](const string &a)
                     { return a.size() >= sz; });
    // compute the number of elements with size >= sz
    auto count = words.end() - wc;
    cout << count << " " << make_plural(count, "word", "s")
         << " of length " << sz << " or longer" << endl;
    // print words of the given size or longer, each one followed by a space
    for_each(wc, words.end(),
            [](const string &s){cout << s << " ";});
    cout << endl;
}
```

EXERCISES SECTION 10.3.2

Exercise 10.14: Write a lambda that takes two ints and returns their sum.

Exercise 10.15: Write a lambda that captures an int from its enclosing function and takes an int parameter. The lambda should return the sum of the captured int and the int parameter.

Exercise 10.16: Write your own version of the `biggies` function using lambdas.

Exercise 10.17: Rewrite exercise 10.12 from § 10.3.1 (p. 387) to use a lambda in the call to sort instead of the `compareIsbn` function.

Exercise 10.18: Rewrite `biggies` to use `partition` instead of `find_if`. We described the `partition` algorithm in exercise 10.13 in § 10.3.1 (p. 387).

Exercise 10.19: Rewrite the previous exercise to use `stable_partition`, which like `stable_sort` maintains the original element order in the partitioned sequence.

10.3.3 Lambda Captures and Returns

When we define a lambda, the compiler generates a new (unnamed) class type that corresponds to that lambda. We'll see how these classes are generated in § 14.8.1 (p. 572). For now, what's useful to understand is that when we pass a lambda to a function, we are defining both a new type and an object of that type: The argument is an unnamed object of this compiler-generated class type. Similarly, when we use `auto` to define a variable initialized by a lambda, we are defining an object of the type generated from that lambda.

By default, the class generated from a lambda contains a data member corresponding to the variables captured by the lambda. Like the data members of any class, the data members of a lambda are initialized when a lambda object is created.

Capture by Value

Similar to parameter passing, we can capture variables by value or by reference. Table 10.1 (p. 395) covers the various ways we can form a capture list. So far, our lambdas have captured variables by value. As with a parameter passed by value, it must be possible to copy such variables. Unlike parameters, the value of a captured variable is copied when the lambda is created, not when it is called:

```
void fcn1()
{
    size_t v1 = 42; // local variable
    // copies v1 into the callable object named f
    auto f = [v1] { return v1; };
    v1 = 0;
    auto j = f(); // j is 42; f stored a copy of v1 when we created it
}
```

Because the value is copied when the lambda is created, subsequent changes to a captured variable have no effect on the corresponding value inside the lambda.

Capture by Reference

We can also define lambdas that capture variables by reference. For example:

```
void fcn2()
{
    size_t v1 = 42; // local variable
    // the object f2 contains a reference to v1
    auto f2 = [&v1] { return v1; };
    v1 = 0;
    auto j = f2(); // j is 0; f2 refers to v1; it doesn't store it
}
```

The `&` before `v1` indicates that `v1` should be captured as a reference. A variable captured by reference acts like any other reference. When we use the variable inside the lambda body, we are using the object to which that reference is bound. In this case, when the lambda returns `v1`, it returns the value of the object to which `v1` refers.

Reference captures have the same problems and restrictions as reference returns (§ 6.3.2, p. 225). If we capture a variable by reference, we must be *certain* that the referenced object exists at the time the lambda is executed. The variables captured by a lambda are local variables. These variables cease to exist once the function completes. If it is possible for a lambda to be executed after the function finishes, the local variables to which the capture refers no longer exist.

Reference captures are sometimes necessary. For example, we might want our `biggies` function to take a reference to an ostream on which to write and a character to use as the separator:

```
void biggies(vector<string> &words,
            vector<string>::size_type sz,
            ostream &os = cout, char c = ' ')
{
    // code to reorder words as before
    // statement to print count revised to print to os
    for_each(words.begin(), words.end(),
             [&os, c](const string &s) { os << s << c; });
}
```

We cannot copy ostream objects (§ 8.1.1, p. 311); the only way to capture `os` is by reference (or through a pointer to `os`).

When we pass a lambda to a function, as in this call to `for_each`, the lambda executes immediately. Capturing `os` by reference is fine, because the variables in `biggies` exist while `for_each` is running.

We can also return a lambda from a function. The function might directly return a callable object or the function might return an object of a class that has a callable object as a data member. If the function returns a lambda, then—for the same reasons that a function must not return a reference to a local variable—that lambda must not contain reference captures.



WARNING

When we capture a variable by reference, we must ensure that the variable exists at the time that the lambda executes.

ADVICE: KEEP YOUR LAMBDA CAPTURES SIMPLE

A lambda capture stores information between the time the lambda is created (i.e., when the code that defines the lambda is executed) and the time (or times) the lambda itself is executed. It is the programmer's responsibility to ensure that whatever information is captured has the intended meaning each time the lambda is executed.

Capturing an ordinary variable—an `int`, a `string`, or other nonpointer type—by value is usually straightforward. In this case, we only need to care whether the variable has the value we need when we capture it.

If we capture a pointer or iterator, or capture a variable by reference, we must ensure that the object bound to that iterator, pointer, or reference still exists, whenever the lambda *executes*. Moreover, we need to ensure that the object has the intended value. Code that executes between when the lambda is created and when it executes might change the value of the object to which the lambda capture points (or refers). The value of the object at the time the pointer (or reference) was captured might have been what we wanted. The value of that object when the lambda executes might be quite different.

As a rule, we can avoid potential problems with captures by minimizing the data we capture. Moreover, if possible, avoid capturing pointers or references.

Implicit Captures

Rather than explicitly listing the variables we want to use from the enclosing function, we can let the compiler infer which variables we use from the code in the lambda's body. To direct the compiler to infer the capture list, we use an `&` or `=` in the capture list. The `&` tells the compiler to capture by reference, and the `=` says the values are captured by value. For example, we can rewrite the lambda that we passed to `find_if` as

```
//  sz implicitly captured by value
wc = find_if(words.begin(), words.end(),
             [=](const string &s)
               { return s.size() >= sz; });
```

If we want to capture some variables by value and others by reference, we can mix implicit and explicit captures:

```
void biggies(vector<string> &words,
             vector<string>::size_type sz,
             ostream &os = cout, char c = ' ')
{
    //  other processing as before
    //  os implicitly captured by reference; c explicitly captured by value
    for_each(words.begin(), words.end(),
              [&, c](const string &s) { os << s << c; });
    //  os explicitly captured by reference; c implicitly captured by value
    for_each(words.begin(), words.end(),
              [=, &os](const string &s) { os << s << c; });
}
```

When we mix implicit and explicit captures, the first item in the capture list must be an `&` or `=`. That symbol sets the default capture mode as by reference or by value, respectively.

When we mix implicit and explicit captures, the explicitly captured variables must use the alternate form. That is, if the implicit capture is by reference (using `&`), then the explicitly named variables must be captured by value; hence their names may not be preceded by an `&`. Alternatively, if the implicit capture is by value (using `=`), then the explicitly named variables must be preceded by an `&` to indicate that they are to be captured by reference.

Table 10.1: Lambda Capture List

<code>[]</code>	Empty capture list. The lambda may not use variables from the enclosing function. A lambda may use local variables only if it captures them.
<code>[names]</code>	<i>names</i> is a comma-separated list of names local to the enclosing function. By default, variables in the capture list are copied. A name preceded by <code>&</code> is captured by reference.
<code>[&]</code>	Implicit by reference capture list. Entities from the enclosing function used in the lambda body are used by reference.
<code>[=]</code>	Implicit by value capture list. Entities from the enclosing function used in the lambda body are copied into the lambda body.
<code>[&, identifier_list]</code>	<i>identifier_list</i> is a comma-separated list of zero or more variables from the enclosing function. These variables are captured by value; any implicitly captured variables are captured by reference. The names in <i>identifier_list</i> must not be preceded by an <code>&</code> .
<code>[=, reference_list]</code>	Variables included in the <i>reference_list</i> are captured by reference; any implicitly captured variables are captured by value. The names in <i>reference_list</i> may not include <code>this</code> and must be preceded by an <code>&</code> .

Mutable Lambdas

By default, a lambda may not change the value of a variable that it copies by value. If we want to be able to change the value of a captured variable, we must follow the parameter list with the keyword `mutable`. Lambdas that are mutable may not omit the parameter list:

```
void fcn3()
{
    size_t v1 = 42; // local variable
    // f can change the value of the variables it captures
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j is 43
}
```

Whether a variable captured by reference can be changed (as usual) depends only on whether that reference refers to a `const` or `nonconst` type:

```

void fcn4()
{
    size_t v1 = 42; // local variable
    // v1 is a reference to a nonconst variable
    // we can change that variable through the reference inside f2
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j is 1
}

```

Specifying the Lambda Return Type

The lambdas we've written so far contain only a single return statement. As a result, we haven't had to specify the return type. By default, if a lambda body contains any statements other than a return, that lambda is assumed to return void. Like other functions that return void, lambdas inferred to return void may not return a value.

As a simple example, we might use the library transform algorithm and a lambda to replace each negative value in a sequence with its absolute value:

```

transform(vi.begin(), vi.end(), vi.begin(),
    [](int i) { return i < 0 ? -i : i; });

```

The transform function takes three iterators and a callable. The first two iterators denote an input sequence and the third is a destination. The algorithm calls the given callable on each element in the input sequence and writes the result to the destination. As in this call, the destination iterator can be the same as the iterator denoting the start of the input. When the input iterator and the destination iterator are the same, transform replaces each element in the input range with the result of calling the callable on that element.

In this call, we passed a lambda that returns the absolute value of its parameter. The lambda body is a single return statement that returns the result of a conditional expression. We need not specify the return type, because that type can be inferred from the type of the conditional operator.

However, if we write the seemingly equivalent program using an if statement, our code won't compile:

```

// error: cannot deduce the return type for the lambda
transform(vi.begin(), vi.end(), vi.begin(),
    [](int i) { if (i < 0) return -i; else return i; });

```

This version of our lambda infers the return type as void but we returned a value.

**C++
11**

When we need to define a return type for a lambda, we must use a trailing return type (§ 6.3.3, p. 229):

```

transform(vi.begin(), vi.end(), vi.begin(),
    [](int i) -> int
    { if (i < 0) return -i; else return i; });

```

In this case, the fourth argument to transform is a lambda with an empty capture list, which takes a single parameter of type int and returns a value of type int. Its function body is an if statement that returns the absolute value of its parameter.

EXERCISES SECTION 10.3.3

Exercise 10.20: The library defines an algorithm named `count_if`. Like `find_if`, this function takes a pair of iterators denoting an input range and a predicate that it applies to each element in the given range. `count_if` returns a count of how often the predicate is true. Use `count_if` to rewrite the portion of our program that counted how many words are greater than length 6.

Exercise 10.21: Write a lambda that captures a local `int` variable and decrements that variable until it reaches 0. Once the variable is 0 additional calls should no longer decrement the variable. The lambda should return a `bool` that indicates whether the captured variable is 0.

10.3.4 Binding Arguments



Lambda expressions are most useful for simple operations that we do not need to use in more than one or two places. If we need to do the same operation in many places, we should usually define a function rather than writing the same lambda expression multiple times. Similarly, if an operation requires many statements, it is ordinarily better to use a function.

It is usually straightforward to use a function in place of a lambda that has an empty capture list. As we've seen, we can use either a lambda or our `isShorter` function to order the `vector` on word length. Similarly, it would be easy to replace the lambda that printed the contents of our `vector` by writing a function that takes a string and prints the given string to the standard output.

However, it is not so easy to write a function to replace a lambda that captures local variables. For example, the lambda that we used in the call to `find_if` compared a string with a given size. We can easily write a function to do the same work:

```
bool check_size(const string &s, string::size_type sz)
{
    return s.size() >= sz;
}
```

However, we can't use this function as an argument to `find_if`. As we've seen, `find_if` takes a unary predicate, so the callable passed to `find_if` must take a single argument. The lambda that `biggies` passed to `find_if` used its capture list to store `sz`. In order to use `check_size` in place of that lambda, we have to figure out how to pass an argument to the `sz` parameter.

The Library `bind` Function

We can solve the problem of passing a size argument to `check_size` by using a new library function named `bind`, which is defined in the functional header. The `bind` function can be thought of as a general-purpose function adaptor (§ 9.6, p. 368). It takes a callable object and generates a new callable that “adapts” the parameter list of the original object.

**C++
11**

The general form of a call to `bind` is:

```
auto newCallable = bind(callable, arg_list);
```

where *newCallable* is itself a callable object and *arg_list* is a comma-separated list of arguments that correspond to the parameters of the given *callable*. That is, when we call *newCallable*, *newCallable* calls *callable*, passing the arguments in *arg_list*.

The arguments in *arg_list* may include names of the form *_n*, where *n* is an integer. These arguments are “placeholders” representing the parameters of *newCallable*. They stand “in place of” the arguments that will be passed to *newCallable*. The number *n* is the position of the parameter in the generated callable: *_1* is the first parameter in *newCallable*, *_2* is the second, and so forth.

Binding the `sz` Parameter of `check_size`

As a simple example, we’ll use `bind` to generate an object that calls `check_size` with a fixed value for its size parameter as follows:

```
// check6 is a callable object that takes one argument of type string
// and calls check_size on its given string and the value 6
auto check6 = bind(check_size, _1, 6);
```

This call to `bind` has only one placeholder, which means that `check6` takes a single argument. The placeholder appears first in *arg_list*, which means that the parameter in `check6` corresponds to the first parameter of `check_size`. That parameter is a `const string&`, which means that the parameter in `check6` is also a `const string&`. Thus, a call to `check6` must pass an argument of type `string`, which `check6` will pass as the first argument to `check_size`.

The second argument in *arg_list* (i.e., the third argument to `bind`) is the value 6. That value is bound to the second parameter of `check_size`. Whenever we call `check6`, it will pass 6 as the second argument to `check_size`:

```
string s = "hello";
bool b1 = check6(s); // check6(s) calls check_size(s, 6)
```

Using `bind`, we can replace our original lambda-based call to `find_if`

```
auto wc = find_if(words.begin(), words.end(),
    [sz](const string &a)
```

with a version that uses `check_size`:

```
auto wc = find_if(words.begin(), words.end(),
    bind(check_size, _1, sz));
```

This call to `bind` generates a callable object that binds the second argument of `check_size` to the value of `sz`. When `find_if` calls this object on the strings in `words` those calls will in turn call `check_size` passing the given string and `sz`. So, `find_if` (effectively) will call `check_size` on each string in the input range and compare the size of that string to `sz`.

Using placeholders Names

The `_n` names are defined in a namespace named `placeholders`. That namespace is itself defined inside the `std` namespace (§ 3.1, p. 82). To use these names, we must supply the names of both namespaces. As with our other examples, our calls to `bind` assume the existence of appropriate `using` declarations. For example, the `using` declaration for `_1` is

```
using std::placeholders::_1;
```

This declaration says we're using the name `_1`, which is defined in the namespace `placeholders`, which is itself defined in the namespace `std`.

We must provide a separate `using` declaration for each placeholder name that we use. Writing such declarations can be tedious and error-prone. Rather than separately declaring each placeholder, we can use a different form of `using` that we will cover in more detail in § 18.2.2 (p. 793). This form:

```
using namespace namespace_name;
```

says that we want to make all the names from `namespace_name` accessible to our program. For example:

```
using namespace std::placeholders;
```

makes all the names defined by `placeholders` usable. Like the `bind` function, the `placeholders` namespace is defined in the functional header.

Arguments to `bind`

As we've seen, we can use `bind` to fix the value of a parameter. More generally, we can use `bind` to bind or rearrange the parameters in the given callable. For example, assuming `f` is a callable object that has five parameters, the following call to `bind`:

```
// g is a callable object that takes two arguments
auto g = bind(f, a, b, _2, c, _1);
```

generates a new callable that takes two arguments, represented by the placeholders `_2` and `_1`. The new callable will pass its own arguments as the third and fifth arguments to `f`. The first, second, and fourth arguments to `f` are bound to the given values, `a`, `b`, and `c`, respectively.

The arguments to `g` are bound positionally to the placeholders. That is, the first argument to `g` is bound to `_1`, and the second argument is bound to `_2`. Thus, when we call `g`, the first argument to `g` will be passed as the last argument to `f`; the second argument to `g` will be passed as `f`'s third argument. In effect, this call to `bind` maps

```
g(_1, _2)
```

to

```
f(a, b, _2, c, _1)
```

That is, calling `g` calls `f` using `g`'s arguments for the placeholders along with the bound arguments, `a`, `b`, and `c`. For example, calling `g(X, Y)` calls

```
f(a, b, Y, c, X)
```

Using to bind to Reorder Parameters

As a more concrete example of using `bind` to reorder arguments, we can use `bind` to invert the meaning of `isShorter` by writing

```
// sort on word length, shortest to longest
sort(words.begin(), words.end(), isShorter);
// sort on word length, longest to shortest
sort(words.begin(), words.end(), bind(isShorter, _2, _1));
```

In the first call, when `sort` needs to compare two elements, `A` and `B`, it will call `isShorter(A, B)`. In the second call to `sort`, the arguments to `isShorter` are swapped. In this case, when `sort` compares elements, it will be as if `sort` called `isShorter(B, A)`.

Binding Reference Parameters

By default, the arguments to `bind` that are not placeholders are copied into the callable object that `bind` returns. However, as with lambdas, sometimes we have arguments that we want to bind but that we want to pass by reference or we might want to bind an argument that has a type that we cannot copy.

For example, to replace the lambda that captured an `ostream` by reference:

```
// os is a local variable referring to an output stream
// c is a local variable of type char
for_each(words.begin(), words.end(),
          [&os, c](const string &s) { os << s << c; });
```

We can easily write a function to do the same job:

```
ostream &print(ostream &os, const string &s, char c)
{
    return os << s << c;
}
```

However, we can't use `bind` directly to replace the capture of `os`:

```
// error: cannot copy os
for_each(words.begin(), words.end(), bind(print, os, _1, ' '));
```

because `bind` copies its arguments and we cannot copy an `ostream`. If we want to pass an object to `bind` without copying it, we must use the library **ref** function:

```
for_each(words.begin(), words.end(),
          bind(print, ref(os), _1, ' '));
```

The `ref` function returns an object that contains the given reference and that is itself copyable. There is also a **cref** function that generates a class that holds a reference to `const`. Like `bind`, the `ref` and `cref` functions are defined in the functional header.

BACKWARD COMPATIBILITY: BINDING ARGUMENTS

Older versions of C++ provided a much more limited, yet more complicated, set of facilities to bind arguments to functions. The library defined two functions named `bind1st` and `bind2nd`. Like `bind`, these functions take a function and generate a new callable object that calls the given function with one of its parameters bound to a given value. However, these functions can bind only the first or second parameter, respectively. Because they are of much more limited utility, they have been *deprecated* in the new standard. A deprecated feature is one that may not be supported in future releases. Modern C++ programs should use `bind`.

EXERCISES SECTION 10.3.4

Exercise 10.22: Rewrite the program to count words of size 6 or less using functions in place of the lambdas.

Exercise 10.23: How many arguments does `bind` take?

Exercise 10.24: Use `bind` and `check_size` to find the first element in a vector of ints that has a value greater than the length of a specified string value.

Exercise 10.25: In the exercises for § 10.3.2 (p. 392) you wrote a version of `biggies` that uses `partition`. Rewrite that function to use `check_size` and `bind`.

10.4 Revisiting Iterators

In addition to the iterators that are defined for each of the containers, the library defines several additional kinds of iterators in the `iterator` header. These iterators include

- **Insert iterators:** These iterators are bound to a container and can be used to insert elements into the container.
- **Stream iterators:** These iterators are bound to input or output streams and can be used to iterate through the associated IO stream.
- **Reverse iterators:** These iterators move backward, rather than forward. The library containers, other than `forward_list`, have reverse iterators.
- **Move iterators:** These special-purpose iterators move rather than copy their elements. We'll cover move iterators in § 13.6.2 (p. 543).

10.4.1 Insert Iterators

An inserter is an iterator adaptor (§ 9.6, p. 368) that takes a container and yields an iterator that adds elements to the specified container. When we assign a value through an insert iterator, the iterator calls a container operation to add an element at a specified position in the given container. The operations these iterators support are listed in Table 10.2 (overleaf).



There are three kinds of inserters. Each differs from the others as to where elements are inserted:

- `back_inserter` (§ 10.2.2, p. 382) creates an iterator that uses `push_back`.
- `front_inserter` creates an iterator that uses `push_front`.
- `inserter` creates an iterator that uses `insert`. This function takes a second argument, which must be an iterator into the given container. Elements are inserted ahead of the element denoted by the given iterator.



We can use `front_inserter` *only* if the container has `push_front`. Similarly, we can use `back_inserter` *only* if it has `push_back`.

Table 10.2: Insert Iterator Operations

<code>it = t</code>	Inserts the value <code>t</code> at the current position denoted by <code>it</code> . Depending on the kind of insert iterator, and assuming <code>c</code> is the container to which it is bound, calls <code>c.push_back(t)</code> , <code>c.push_front(t)</code> , or <code>c.insert(t, p)</code> , where <code>p</code> is the iterator position given to <code>inserter</code> .
<code>*it, ++it, it++</code>	These operations exist but do nothing to <code>it</code> . Each operator returns <code>it</code> .

It is important to understand that when we call `inserter(c, iter)`, we get an iterator that, when used successively, inserts elements ahead of the element originally denoted by `iter`. That is, if `it` is an iterator generated by `inserter`, then an assignment such as

```
*it = val;
```

behaves as

```
it = c.insert(it, val); // it points to the newly added element
++it; // increment it so that it denotes the same element as before
```

The iterator generated by `front_inserter` behaves quite differently from the one created by `inserter`. When we use `front_inserter`, elements are always inserted ahead of the then first element in the container. Even if the position we pass to `inserter` initially denotes the first element, as soon as we insert an element in front of that element, that element is no longer the one at the beginning of the container:

```
list<int> lst = {1,2,3,4};
list<int> lst2, lst3; // empty lists
// after copy completes, lst2 contains 4 3 2 1
copy(lst.cbegin(), lst.cend(), front_inserter(lst2));
// after copy completes, lst3 contains 1 2 3 4
copy(lst.cbegin(), lst.cend(), inserter(lst3, lst3.begin()));
```

When we call `front_inserter(c)`, we get an insert iterator that successively calls `push_front`. As each element is inserted, it becomes the new first element in `c`. Therefore, `front_inserter` yields an iterator that reverses the order of the sequence that it inserts; `inserter` and `back_inserter` don't.

EXERCISES SECTION 10.4.1

Exercise 10.26: Explain the differences among the three kinds of insert iterators.

Exercise 10.27: In addition to `unique` (§ 10.2.3, p. 384), the library defines function named `unique_copy` that takes a third iterator denoting a destination into which to copy the unique elements. Write a program that uses `unique_copy` to copy the unique elements from a vector into an initially empty list.

Exercise 10.28: Copy a vector that holds the values from 1 to 9 inclusive, into three other containers. Use an `inserter`, a `back_inserter`, and a `front_inserter`, respectively to add elements to these containers. Predict how the output sequence varies by the kind of inserter and verify your predictions by running your programs.

10.4.2 `iostream` Iterators

Even though the `iostream` types are not containers, there are iterators that can be used with objects of the IO types (§ 8.1, p. 310). An `istream_iterator` (Table 10.3 (overleaf)) reads an input stream, and an `ostream_iterator` (Table 10.4 (p. 405)) writes an output stream. These iterators treat their corresponding stream as a sequence of elements of a specified type. Using a stream iterator, we can use the generic algorithms to read data from or write data to stream objects.

Operations on `istream_iterator`s

When we create a stream iterator, we must specify the type of objects that the iterator will read or write. An `istream_iterator` uses `>>` to read a stream. Therefore, the type that an `istream_iterator` reads must have an input operator defined. When we create an `istream_iterator`, we can bind it to a stream. Alternatively, we can default initialize the iterator, which creates an iterator that we can use as the off-the-end value.

```
istream_iterator<int> int_it(cin);    // reads ints from cin
istream_iterator<int> int_eof;       // end iterator value
ifstream in("afile");
istream_iterator<string> str_it(in); // reads strings from "afile"
```

As an example, we can use an `istream_iterator` to read the standard input into a vector:

```
istream_iterator<int> in_iter(cin); // read ints from cin
istream_iterator<int> eof;          // istream "end" iterator
while (in_iter != eof) // while there's valid input to read
    // postfix increment reads the stream and returns the old value of the iterator
    // we dereference that iterator to get the previous value read from the stream
    vec.push_back(*in_iter++);
```

This loop reads ints from `cin`, storing what was read in `vec`. On each iteration, the loop checks whether `in_iter` is the same as `eof`. That iterator was defined as the empty `istream_iterator`, which is used as the end iterator. An iterator

bound to a stream is equal to the end iterator once its associated stream hits end-of-file or encounters an IO error.

The hardest part of this program is the argument to `push_back`, which uses the dereference and postfix increment operators. This expression works just like others we’ve written that combined dereference with postfix increment (§ 4.5, p. 148). The postfix increment advances the stream by reading the next value but returns the *old* value of the iterator. That old value contains the previous value read from the stream. We dereference that iterator to obtain that value.

What is more useful is that we can rewrite this program as

```
istream_iterator<int> in_iter(cin), eof; // read ints from cin
vector<int> vec(in_iter, eof); // construct vec from an iterator range
```

Here we construct `vec` from a pair of iterators that denote a range of elements. Those iterators are `istream_iterator`s, which means that the range is obtained by reading the associated stream. This constructor reads `cin` until it hits end-of-file or encounters an input that is not an `int`. The elements that are read are used to construct `vec`.

Table 10.3: `istream_iterator` Operations

<code>istream_iterator<T> in(is);</code> <code>in</code> reads values of type <code>T</code> from input stream <code>is</code> .	
<code>istream_iterator<T> end;</code>	Off-the-end iterator for an <code>istream_iterator</code> that reads values of type <code>T</code> .
<code>in1 == in2</code>	<code>in1</code> and <code>in2</code> must read the same type. They are equal if they are both the end value or are bound to the same input stream.
<code>in1 != in2</code>	
<code>*in</code>	Returns the value read from the stream.
<code>in->mem</code>	Synonym for <code>(*in).mem</code> .
<code>++in, in++</code>	Reads the next value from the input stream using the <code>>></code> operator for the element type. As usual, the prefix version returns a reference to the incremented iterator. The postfix version returns the old value.

Using Stream Iterators with the Algorithms

Because algorithms operate in terms of iterator operations, and the stream iterators support at least some iterator operations, we can use stream iterators with at least some of the algorithms. We’ll see in § 10.5.1 (p. 410) how to tell which algorithms can be used with the stream iterators. As one example, we can call `accumulate` with a pair of `istream_iterator`s:

```
istream_iterator<int> in(cin), eof;
cout << accumulate(in, eof, 0) << endl;
```

This call will generate the sum of values read from the standard input. If the input to this program is

```
23 109 45 89 6 34 12 90 34 23 56 23 8 89 23
```

then the output will be 664.

istream_iterators Are Permitted to Use Lazy Evaluation

When we bind an `istream_iterator` to a stream, we are not guaranteed that it will read the stream immediately. The implementation is permitted to delay reading the stream until we use the iterator. We are guaranteed that before we dereference the iterator for the first time, the stream will have been read. For most programs, whether the read is immediate or delayed makes no difference. However, if we create an `istream_iterator` that we destroy without using or if we are synchronizing reads to the same stream from two different objects, then we might care a great deal when the read happens.

Operations on ostream_iterators

An `ostream_iterator` can be defined for any type that has an output operator (the `<<` operator). When we create an `ostream_iterator`, we may (optionally) provide a second argument that specifies a character string to print following each element. That string must be a C-style character string (i.e., a string literal or a pointer to a null-terminated array). We must bind an `ostream_iterator` to a specific stream. There is no empty or off-the-end `ostream_iterator`.

Table 10.4: ostream_iterator Operations

<code>ostream_iterator<T> out(os);</code>	out writes values of type T to output stream os.
<code>ostream_iterator<T> out(os, d);</code>	out writes values of type T followed by d to output stream os. d points to a null-terminated character array.
<code>out = val</code>	Writes val to the ostream to which out is bound using the <code><<</code> operator. val must have a type that is compatible with the type that out can write.
<code>*out, ++out, out++</code>	These operations exist but do nothing to out. Each operator returns out.

We can use an `ostream_iterator` to write a sequence of values:

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
    *out_iter++ = e; // the assignment writes this element to cout
cout << endl;
```

This program writes each element from `vec` onto `cout` following each element with a space. Each time we assign a value to `out_iter`, the write is committed.

It is worth noting that we can omit the dereference and the increment when we assign to `out_iter`. That is, we can write this loop equivalently as

```
for (auto e : vec)
    out_iter = e; // the assignment writes this element to cout
cout << endl;
```

The `*` and `++` operators do nothing on an `ostream_iterator`, so omitting them has no effect on our program. However, we prefer to write the loop as first presented. That loop uses the iterator consistently with how we use other iterator

types. We can easily change this loop to execute on another iterator type. Moreover, the behavior of this loop will be clearer to readers of our code.

Rather than writing the loop ourselves, we can more easily print the elements in `vec` by calling `copy`:

```
copy(vec.begin(), vec.end(), out_iter);
cout << endl;
```

Using Stream Iterators with Class Types

We can create an `istream_iterator` for any type that has an input operator (`>>`). Similarly, we can define an `ostream_iterator` so long as the type has an output operator (`<<`). Because `Sales_item` has both input and output operators, we can use IO iterators to rewrite the bookstore program from § 1.6 (p. 24):

```
istream_iterator<Sales_item> item_iter(cin), eof;
ostream_iterator<Sales_item> out_iter(cout, "\n");
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
while (item_iter != eof) {
    // if the current transaction (which is stored in item_iter) has the same ISBN
    if (item_iter->isbn() == sum.isbn())
        sum += *item_iter++; // add it to sum and read the next transaction
    else {
        out_iter = sum;      // write the current sum
        sum = *item_iter++;  // read the next transaction
    }
}
out_iter = sum; // remember to print the last set of records
```

This program uses `item_iter` to read `Sales_item` transactions from `cin`. It uses `out_iter` to write the resulting sums to `cout`, following each output with a newline. Having defined our iterators, we use `item_iter` to initialize `sum` with the value of the first transaction:

```
// store the first transaction in sum and read the next record
Sales_item sum = *item_iter++;
```

Here, we dereference the result of the postfix increment on `item_iter`. This expression reads the next transaction, and initializes `sum` from the value previously stored in `item_iter`.

The `while` loop executes until we hit end-of-file on `cin`. Inside the `while`, we check whether `sum` and the record we just read refer to the same book. If so, we add the most recently read `Sales_item` into `sum`. If the ISBNs differ, we assign `sum` to `out_iter`, which prints the current value of `sum` followed by a newline. Having printed the sum for the previous book, we assign `sum` a copy of the most recently read transaction and increment the iterator, which reads the next transaction. The loop continues until an error or end-of-file is encountered. Before exiting, we remember to print the values associated with the last book in the input.

EXERCISES SECTION 10.4.2

Exercise 10.29: Write a program using stream iterators to read a text file into a vector of strings.

Exercise 10.30: Use stream iterators, `sort`, and `copy` to read a sequence of integers from the standard input, sort them, and then write them back to the standard output.

Exercise 10.31: Update the program from the previous exercise so that it prints only the unique elements. Your program should use `unique_copy` (§ 10.4.1, p. 403).

Exercise 10.32: Rewrite the bookstore problem from § 1.6 (p. 24) using a vector to hold the transactions and various algorithms to do the processing. Use `sort` with your `compareIsbn` function from § 10.3.1 (p. 387) to arrange the transactions in order, and then use `find` and `accumulate` to do the sum.

Exercise 10.33: Write a program that takes the names of an input file and two output files. The input file should hold integers. Using an `istream_iterator` read the input file. Using `ostream_iterators`, write the odd numbers into the first output file. Each value should be followed by a space. Write the even numbers into the second file. Each of these values should be placed on a separate line.

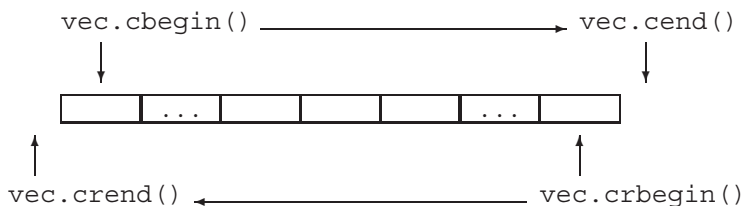
10.4.3 Reverse Iterators

A reverse iterator is an iterator that traverses a container backward, from the last element toward the first. A reverse iterator inverts the meaning of increment (and decrement). Incrementing (`++it`) a reverse iterator moves the iterator to the previous element; decrementing (`--it`) moves the iterator to the next element.

The containers, aside from `forward_list`, all have reverse iterators. We obtain a reverse iterator by calling the `rbegin`, `rend`, `crbegin`, and `crend` members. These members return reverse iterators to the last element in the container and one “past” (i.e., one before) the beginning of the container. As with ordinary iterators, there are both `const` and `nonconst` reverse iterators.

Figure 10.1 illustrates the relationship between these four iterators on a hypothetical vector named `vec`.

Figure 10.1: Comparing `begin/cend` and `rbegin/crend` Iterators



As an example, the following loop prints the elements of `vec` in reverse order:

```
vector<int> vec = {0,1,2,3,4,5,6,7,8,9};
// reverse iterator of vector from back to front
for (auto r_iter = vec.crbegin(); // binds r_iter to the last element
     r_iter != vec.crend(); // crend refers 1 before 1st element
     ++r_iter) // decrements the iterator one element
    cout << *r_iter << endl; // prints 9, 8, 7, ... 0
```

Although it may seem confusing to have the meaning of the increment and decrement operators reversed, doing so lets us use the algorithms transparently to process a container forward or backward. For example, we can sort our vector in descending order by passing `sort` a pair of reverse iterators:

```
sort(vec.begin(), vec.end()); // sorts vec in "normal" order
// sorts in reverse: puts the smallest element at the end of vec
sort(vec.rbegin(), vec.rend());
```

Reverse Iterators Require Decrement Operators

Not surprisingly, we can define a reverse iterator only from an iterator that supports `--` as well as `++`. After all, the purpose of a reverse iterator is to move the iterator backward through the sequence. Aside from `forward_list`, the iterators on the standard containers all support decrement as well as increment. However, the stream iterators do not, because it is not possible to move backward through a stream. Therefore, it is not possible to create a reverse iterator from a `forward_list` or a stream iterator.



Relationship between Reverse Iterators and Other Iterators

Suppose we have a string named `line` that contains a comma-separated list of words, and we want to print the first word in `line`. Using `find`, this task is easy:

```
// find the first element in a comma-separated list
auto comma = find(line.cbegin(), line.cend(), ',');
cout << string(line.cbegin(), comma) << endl;
```

If there is a comma in `line`, then `comma` refers to that comma; otherwise it is `line.cend()`. When we print the string from `line.cbegin()` to `comma`, we print characters up to the comma, or the entire string if there is no comma.

If we wanted the last word, we can use reverse iterators instead:

```
// find the last element in a comma-separated list
auto rcomma = find(line.crbegin(), line.crend(), ',');
```

Because we pass `crbegin()` and `crend()`, this call starts with the last character in `line` and searches backward. When `find` completes, if there is a comma, then `rcomma` refers to the last comma in `line`—that is, it refers to the first comma found in the backward search. If there is no comma, then `rcomma` is `line.crend()`.

The interesting part comes when we try to print the word we found. The seemingly obvious way

```
// WRONG: will generate the word in reverse order
cout << string(line.crbegin(), rcomma) << endl;
```

generates bogus output. For example, had our input been

```
FIRST,MIDDLE, LAST
```

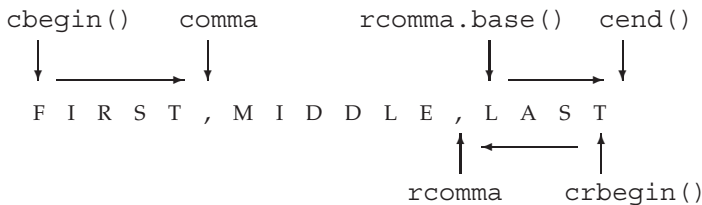
then this statement would print TSAL!

Figure 10.2 illustrates the problem: We are using reverse iterators, which process the string backward. Therefore, our output statement prints from `crbegin` backward through `line`. Instead, we want to print from `rcomma` forward to the end of `line`. However, we can't use `rcomma` directly. That iterator is a reverse iterator, which means that it goes backward toward the beginning of the string. What we need to do is transform `rcomma` back into an ordinary iterator that will go forward through `line`. We can do so by calling the `reverse_iterator`'s base member, which gives us its corresponding ordinary iterator:

```
// ok: get a forward iterator and read to the end of line
cout << string(rcomma.base(), line.cend()) << endl;
```

Given the same preceding input, this statement prints `LAST` as expected.

Figure 10.2: Relationship between Reverse and Ordinary Iterators



The objects shown in Figure 10.2 illustrate the relationship between ordinary and reverse iterators. For example, `rcomma` and `rcomma.base()` refer to different elements, as do `line.crbegin()` and `line.cend()`. These differences are needed to ensure that the *range* of elements, whether processed forward or backward, is the same.

Technically speaking, the relationship between normal and reverse iterators accommodates the properties of a left-inclusive range (§ 9.2.1, p. 331). The point is that `[line.crbegin(), rcomma)` and `[rcomma.base(), line.cend())` refer to the same elements in `line`. In order for that to happen, `rcomma` and `rcomma.base()` must yield adjacent positions, rather than the same position, as must `crbegin()` and `cend()`.



The fact that reverse iterators are intended to represent ranges and that these ranges are asymmetric has an important consequence: When we initialize or assign a reverse iterator from a plain iterator, the resulting iterator does not refer to the same element as the original.

EXERCISES SECTION 10.4.3

Exercise 10.34: Use `reverse_iterators` to print a vector in reverse order.

Exercise 10.35: Now print the elements in reverse order using ordinary iterators.

Exercise 10.36: Use `find` to find the last element in a list of ints with value 0.

Exercise 10.37: Given a vector that has ten elements, copy the elements from positions 3 through 7 in reverse order to a list.



10.5 Structure of Generic Algorithms

The most fundamental property of any algorithm is the list of operations it requires from its iterator(s). Some algorithms, such as `find`, require only the ability to access an element through the iterator, to increment the iterator, and to compare two iterators for equality. Others, such as `sort`, require the ability to read, write, and randomly access elements. The iterator operations required by the algorithms are grouped into five **iterator categories** listed in Table 10.5. Each algorithm specifies what kind of iterator must be supplied for each of its iterator parameters.

A second way is to classify the algorithms (as we did in the beginning of this chapter) is by whether they read, write, or reorder the elements in the sequence. Appendix A covers all the algorithms according to this classification.

The algorithms also share a set of parameter-passing conventions and a set of naming conventions, which we shall cover after looking at iterator categories.

Table 10.5: Iterator Categories	
Input iterator	Read, but not write; single-pass, increment only
Output iterator	Write, but not read; single-pass, increment only
Forward iterator	Read and write; multi-pass, increment only
Bidirectional iterator	Read and write; multi-pass, increment and decrement
Random-access iterator	Read and write; multi-pass, full iterator arithmetic



10.5.1 The Five Iterator Categories

Like the containers, iterators define a common set of operations. Some operations are provided by all iterators; other operations are supported by only specific kinds of iterators. For example, `ostream_iterators` have only increment, dereference, and assignment. Iterators on `vector`, `strings`, and `deques` support these operations and the decrement, relational, and arithmetic operators.

Iterators are categorized by the operations they provide and the categories form a sort of hierarchy. With the exception of output iterators, an iterator of a higher category provides all the operations of the iterators of a lower categories.

The standard specifies the minimum category for each iterator parameter of the

generic and numeric algorithms. For example, `find`—which implements a one-pass, read-only traversal over a sequence—minimally requires an input iterator. The `replace` function requires a pair of iterators that are at least forward iterators. Similarly, `replace_copy` requires forward iterators for its first two iterators. Its third iterator, which represents a destination, must be at least an output iterator, and so on. For each parameter, the iterator must be at least as powerful as the stipulated minimum. Passing an iterator of a lesser power is an error.



Many compilers will not complain when we pass the wrong category of iterator to an algorithm.

The Iterator Categories

Input iterators: can read elements in a sequence. An input iterator must provide

- Equality and inequality operators (`==`, `!=`) to compare two iterators
- Prefix and postfix increment (`++`) to advance the iterator
- Dereference operator (`*`) to read an element; dereference may appear only on the right-hand side of an assignment
- The arrow operator (`->`) as a synonym for `(*it).member`—that is, dereference the iterator and fetch a member from the underlying object

Input iterators may be used only sequentially. We are guaranteed that `*it++` is valid, but incrementing an input iterator may invalidate all other iterators into the stream. As a result, there is no guarantee that we can save the state of an input iterator and examine an element through that saved iterator. Input iterators, therefore, may be used only for single-pass algorithms. The `find` and `accumulate` algorithms require input iterators; `istream_iterators` are input iterators.

Output iterators: can be thought of as having complementary functionality to input iterators; they write rather than read elements. Output iterators must provide

- Prefix and postfix increment (`++`) to advance the iterator
- Dereference (`*`), which may appear only as the left-hand side of an assignment (Assigning to a dereferenced output iterator writes to the underlying element.)

We may assign to a given value of an output iterator only once. Like input iterators, output iterators may be used only for single-pass algorithms. Iterators used as a destination are typically output iterators. For example, the third parameter to `copy` is an output iterator. The `ostream_iterator` type is an output iterator.

Forward iterators: can read and write a given sequence. They move in only one direction through the sequence. Forward iterators support all the operations of both input iterators and output iterators. Moreover, they can read or write the same element multiple times. Therefore, we can use the saved state of a forward iterator. Hence, algorithms that use forward iterators may make multiple passes through the sequence. The `replace` algorithm requires a forward iterator; iterators on `forward_list` are forward iterators.

Bidirectional iterators: can read and write a sequence forward or backward. In addition to supporting all the operations of a forward iterator, a bidirectional iterator also supports the prefix and postfix decrement (`--`) operators. The `reverse` algorithm requires bidirectional iterators, and aside from `forward_list`, the library containers supply iterators that meet the requirements for a bidirectional iterator.

Random-access iterators: provide constant-time access to any position in the sequence. These iterators support all the functionality of bidirectional iterators. In addition, random-access iterators support the operations from Table 3.7 (p. 111):

- The relational operators (`<`, `<=`, `>`, and `>=`) to compare the relative positions of two iterators.
- Addition and subtraction operators (`+`, `+=`, `-`, and `-=`) on an iterator and an integral value. The result is the iterator advanced (or retreated) the integral number of elements within the sequence.
- The subtraction operator (`-`) when applied to two iterators, which yields the distance between two iterators.
- The subscript operator (`iter[n]`) as a synonym for `*(iter + n)`.

The sort algorithms require random-access iterators. Iterators for `array`, `deque`, `string`, and `vector` are random-access iterators, as are pointers when used to access elements of a built-in array.

EXERCISES SECTION 10.5.1

Exercise 10.38: List the five iterator categories and the operations that each supports.

Exercise 10.39: What kind of iterator does a `list` have? What about a `vector`?

Exercise 10.40: What kinds of iterators do you think `copy` requires? What about `reverse` or `unique`?



10.5.2 Algorithm Parameter Patterns

Superimposed on any other classification of the algorithms is a set of parameter conventions. Understanding these parameter conventions can aid in learning new algorithms—by knowing what the parameters mean, you can concentrate on understanding the operation the algorithm performs. Most of the algorithms have one of the following four forms:

```
alg (beg, end, other args) ;
alg (beg, end, dest, other args) ;
alg (beg, end, beg2, other args) ;
alg (beg, end, beg2, end2, other args) ;
```

where `alg` is the name of the algorithm, and `beg` and `end` denote the input range on which the algorithm operates. Although nearly all algorithms take an input

range, the presence of the other parameters depends on the work being performed. The common ones listed here—`dest`, `beg2`, and `end2`—are all iterators. When used, these iterators fill similar roles. In addition to these iterator parameters, some algorithms take additional, noniterator parameters that are algorithm specific.

Algorithms with a Single Destination Iterator

A `dest` parameter is an iterator that denotes a destination in which the algorithm can write its output. Algorithms *assume* that it is safe to write as many elements as needed.



Algorithms that write to an output iterator assume the destination is large enough to hold the output.

If `dest` is an iterator that refers directly to a container, then the algorithm writes its output to existing elements within the container. More commonly, `dest` is bound to an insert iterator (§ 10.4.1, p. 401) or an `ostream_iterator` (§ 10.4.2, p. 403). An insert iterator adds new elements to the container, thereby ensuring that there is enough space. An `ostream_iterator` writes to an output stream, again presenting no problem regardless of how many elements are written.

Algorithms with a Second Input Sequence

Algorithms that take either `beg2` alone or `beg2` and `end2` use those iterators to denote a second input range. These algorithms typically use the elements from the second range in combination with the input range to perform a computation.

When an algorithm takes both `beg2` and `end2`, these iterators denote a second range. Such algorithms take two completely specified ranges: the input range denoted by `[beg, end)`, and a second input range denoted by `[beg2, end2)`.

Algorithms that take only `beg2` (and not `end2`) treat `beg2` as the first element in a second input range. The end of this range is not specified. Instead, these algorithms *assume* that the range starting at `beg2` is at least as large as the one denoted by `beg, end`.



Algorithms that take `beg2` alone *assume* that the sequence beginning at `beg2` is as large as the range denoted by `beg` and `end`.

10.5.3 Algorithm Naming Conventions



Separate from the parameter conventions, the algorithms also conform to a set of naming and overload conventions. These conventions deal with how we supply an operation to use in place of the default `<` or `==` operator and with whether the algorithm writes to its input sequence or to a separate destination.

Some Algorithms Use Overloading to Pass a Predicate

Algorithms that take a predicate to use in place of the `<` or `==` operator, and that do not take other arguments, typically are overloaded. One version of the function

uses the element type's operator to compare elements; the second takes an extra parameter that is a predicate to use in place of `<` or `==`:

```
unique(beg, end);           // uses the == operator to compare the elements
unique(beg, end, comp);    // uses comp to compare the elements
```

Both calls reorder the given sequence by removing adjacent duplicated elements. The first uses the element type's `==` operator to check for duplicates; the second calls `comp` to decide whether two elements are equal. Because the two versions of the function differ as to the number of arguments, there is no possible ambiguity (§ 6.4, p. 233) as to which function is being called.

Algorithms with `_if` Versions

Algorithms that take an element value typically have a second named (not overloaded) version that takes a predicate (§ 10.3.1, p. 386) in place of the value. The algorithms that take a predicate have the suffix `_if` appended:

```
find(beg, end, val);        // find the first instance of val in the input range
find_if(beg, end, pred);    // find the first instance for which pred is true
```

These algorithms both find the first instance of a specific element in the input range. The `find` algorithm looks for a specific value; the `find_if` algorithm looks for a value for which `pred` returns a nonzero value.

These algorithms provide a named version rather than an overloaded one because both versions of the algorithm take the same number of arguments. Overloading ambiguities would therefore be possible, albeit rare. To avoid any possible ambiguities, the library provides separate named versions for these algorithms.

Distinguishing Versions That Copy from Those That Do Not

By default, algorithms that rearrange elements write the rearranged elements back into the given input range. These algorithms provide a second version that writes to a specified output destination. As we've seen, algorithms that write to a destination append `_copy` to their names (§ 10.2.2, p. 383):

```
reverse(beg, end);          // reverse the elements in the input range
reverse_copy(beg, end, dest); // copy elements in reverse order into dest
```

Some algorithms provide both `_copy` and `_if` versions. These versions take a destination iterator and a predicate:

```
// removes the odd elements from v1
remove_if(v1.begin(), v1.end(),
          [](int i) { return i % 2; });
// copies only the even elements from v1 into v2; v1 is unchanged
remove_copy_if(v1.begin(), v1.end(), back_inserter(v2),
               [](int i) { return i % 2; });
```

Both calls use a lambda (§ 10.3.2, p. 388) to determine whether an element is odd. In the first case, we remove the odd elements from the input sequence itself. In the second, we copy the non-odd (aka even) elements from the input range into `v2`.

EXERCISES SECTION 10.5.3

Exercise 10.41: Based only on the algorithm and argument names, describe the operation that each of the following library algorithms performs:


```
replace(beg, end, old_val, new_val);
replace_if(beg, end, pred, new_val);
replace_copy(beg, end, dest, old_val, new_val);
replace_copy_if(beg, end, dest, pred, new_val);
```

10.6 Container-Specific Algorithms

Unlike the other containers, `list` and `forward_list` define several algorithms as members. In particular, the list types define their own versions of `sort`, `merge`, `remove`, `reverse`, and `unique`. The generic version of `sort` requires random-access iterators. As a result, `sort` cannot be used with `list` and `forward_list` because these types offer bidirectional and forward iterators, respectively.

The generic versions of the other algorithms that the list types define can be used with lists, but at a cost in performance. These algorithms swap elements in the input sequence. A list can “swap” its elements by changing the links among its elements rather than swapping the values of those elements. As a result, the list-specific versions of these algorithms can achieve much better performance than the corresponding generic versions.

These list-specific operations are described in Table 10.6. Generic algorithms not listed in the table that take appropriate iterators execute equally efficiently on lists and `forward_list`s as on other containers.



The list member versions should be used in preference to the generic algorithms for lists and `forward_list`s.

Table 10.6: Algorithms That are Members of <code>list</code> and <code>forward_list</code>	
	These operations return <code>void</code> .
<code>lst.merge(lst2)</code>	Merges elements from <code>lst2</code> onto <code>lst</code> . Both <code>lst</code> and <code>lst2</code> must be sorted. Elements are removed from <code>lst2</code> . After the merge, <code>lst2</code> is empty. The first version uses the <code><</code> operator; the second version uses the given comparison operation.
<code>lst.merge(lst2, comp)</code>	
<code>lst.remove(val)</code>	Calls <code>erase</code> to remove each element that is <code>==</code> to the given value or for which the given unary predicate succeeds.
<code>lst.remove_if(pred)</code>	
<code>lst.reverse()</code>	Reverses the order of the elements in <code>lst</code> .
<code>lst.sort()</code>	Sorts the elements of <code>lst</code> using <code><</code> or the given comparison operation.
<code>lst.sort(comp)</code>	
<code>lst.unique()</code>	Calls <code>erase</code> to remove consecutive copies of the same value. The first version uses <code>==</code> ; the second uses the given binary predicate.
<code>lst.unique(pred)</code>	



The splice Members

The list types also define a `splice` algorithm, which is described in Table 10.7. This algorithm is particular to list data structures. Hence a generic version of this algorithm is not needed.

Table 10.7: Arguments to the `list` and `forward_list` `splice` Members

<code>lst.splice(args)</code> or <code>flst.splice_after(args)</code>	
<code>(p, lst2)</code>	<code>p</code> is an iterator to an element in <code>lst</code> or an iterator just before an element in <code>flst</code> . Moves all the element(s) from <code>lst2</code> into <code>lst</code> just before <code>p</code> or into <code>flst</code> just after <code>p</code> . Removes the element(s) from <code>lst2</code> . <code>lst2</code> must have the same type as <code>lst</code> or <code>flst</code> and may not be the same list.
<code>(p, lst2, p2)</code>	<code>p2</code> is a valid iterator into <code>lst2</code> . Moves the element denoted by <code>p2</code> into <code>lst</code> or moves the element just after <code>p2</code> into <code>flst</code> . <code>lst2</code> can be the same list as <code>lst</code> or <code>flst</code> .
<code>(p, lst2, b, e)</code>	<code>b</code> and <code>e</code> must denote a valid range in <code>lst2</code> . Moves the elements in the given range from <code>lst2</code> . <code>lst2</code> and <code>lst</code> (or <code>flst</code>) can be the same list but <code>p</code> must not denote an element in the given range.

The List-Specific Operations Do Change the Containers

Most of the list-specific algorithms are similar—but not identical—to their generic counterparts. However, a crucially important difference between the list-specific and the generic versions is that the list versions change the underlying container. For example, the list version of `remove` removes the indicated elements. The list version of `unique` removes the second and subsequent duplicate elements.

Similarly, `merge` and `splice` are destructive on their arguments. For example, the generic version of `merge` writes the merged sequence to a given destination iterator; the two input sequences are unchanged. The list `merge` function destroys the given list—elements are removed from the argument list as they are merged into the object on which `merge` was called. After a `merge`, the elements from both lists continue to exist, but they are all elements of the same list.

EXERCISES SECTION 10.6

Exercise 10.42: Reimplement the program that eliminated duplicate words that we wrote in § 10.2.3 (p. 383) to use a `list` instead of a `vector`.

CHAPTER SUMMARY

The standard library defines about 100 type-independent algorithms that operate on sequences. Sequences can be elements in a library container type, a built-in array, or generated (for example) by reading or writing to a stream. Algorithms achieve their type independence by operating in terms of iterators. Most algorithms take as their first two arguments a pair of iterators denoting a range of elements. Additional iterator arguments might include an output iterator denoting a destination, or another iterator or iterator pair denoting a second input sequence.

Iterators are categorized into one of five categories depending on the operations they support. The iterator categories are input, output, forward, bidirectional, and random access. An iterator belongs to a particular category if it supports the operations required for that iterator category.

Just as iterators are categorized by their operations, iterator parameters to the algorithms are categorized by the iterator operations they require. Algorithms that only read their sequences require only input iterator operations. Those that write to a destination iterator require only the actions of an output iterator, and so on.

Algorithms *never* directly change the size of the sequences on which they operate. They may copy elements from one position to another but cannot directly add or remove elements.

Although algorithms cannot add elements to a sequence, an insert iterator may do so. An insert iterator is bound to a container. When we assign a value of the container's element type to an insert iterator, the iterator adds the given element to the container.

The `forward_list` and `list` containers define their own versions of some of the generic algorithms. Unlike the generic algorithms, these list-specific versions modify the given lists.

DEFINED TERMS

back_inserter Iterator adaptor that takes a reference to a container and generates an insert iterator that uses `push_back` to add elements to the specified container.

bidirectional iterator Same operations as forward iterators plus the ability to use `--` to move backward through the sequence.

binary predicate Predicate that has two parameters.

bind Library function that binds one or more arguments to a callable expression. `bind` is defined in the `functional` header.

callable object Object that can appear as the left-hand operand of the call operator. Pointers to functions, lambdas, and objects

of a class that defines an overloaded function call operator are all callable objects.

capture list Portion of a lambda expression that specifies which variables from the surrounding context the lambda expression may access.

cref Library function that returns a copyable object that holds a reference to a `const` object of a type that cannot be copied.

forward iterator Iterator that can read and write elements but is not required to support `--`.

front_inserter Iterator adaptor that, given a container, generates an insert iterator that

uses `push_front` to add elements to the beginning of that container.

generic algorithms Type-independent algorithms.

input iterator Iterator that can read, but not write, elements of a sequence.

insert iterator Iterator adaptor that generates an iterator that uses a container operation to add elements to a given container.

inserter Iterator adaptor that takes an iterator and a reference to a container and generates an insert iterator that uses `insert` to add elements just ahead of the element referred to by the given iterator.

istream_iterator Stream iterator that reads an input stream.

iterator categories Conceptual organization of iterators based on the operations that an iterator supports. Iterator categories form a hierarchy, in which the more powerful categories offer the same operations as the lesser categories. The algorithms use iterator categories to specify what operations the iterator arguments must support. As long as the iterator provides at least that level of operation, it can be used. For example, some algorithms require only input iterators. Such algorithms can be called on any iterator other than one that meets only the output iterator requirements. Algorithms that require random-access iterators can be used only on iterators that support random-access operations.

lambda expression Callable unit of code. A lambda is somewhat like an unnamed, inline function. A lambda starts with a capture list, which allows the lambda to access variables in the enclosing function. Like a

function, it has a (possibly empty) parameter list, a return type, and a function body. A lambda can omit the return type. If the function body is a single `return` statement, the return type is inferred from the type of the object that is returned. Otherwise, an omitted return type defaults to `void`.

move iterator Iterator adaptor that generates an iterator that moves elements instead of copying them. Move iterators are covered in Chapter 13.

ostream_iterator Iterator that writes to an output stream.

output iterator Iterator that can write, but not necessarily read, elements.

predicate Function that returns a type that can be converted to `bool`. Often used by the generic algorithms to test elements. Predicates used by the library are either unary (taking one argument) or binary (taking two).

random-access iterator Same operations as bidirectional iterators plus the relational operators to compare iterator values, and the subscript operator and arithmetic operations on iterators, thus supporting random access to elements.

ref Library function that generates a copyable object from a reference to an object of a type that cannot be copied.

reverse iterator Iterator that moves backward through a sequence. These iterators exchange the meaning of `++` and `--`.

stream iterator Iterator that can be bound to a stream.

unary predicate Predicate that has one parameter.